website-information-scraper

April 10, 2024

0.0.1 Test URLS

https://sunstonepartners.com
https://www.appliedlearning.com
https://www.forsalebyowner.com login
https://ecmins.com/
http://www.iconnect-corp.com
https://cessco.ca/ ROBOT
http://www.ticss.net
https://www.tyremarket.com/Car-Tyres
https://www.dentalxchange.com/

TODOList:

-Improve URL relevance check (exclude /#, /login, /sign-up)

-Never return empty nav object, instead string. If after try bs4 and sel, fails

-script defers to selenium if bs4 does nav_scrape but nav still empty (ex: https://www.forsalebyowner.com)

-File output with all columns

-Enhance href relevance function (both contain base_url)

-Four columns of information for each website

-Improve speed of sel nav_tree recursion

-retry if page_result is empty after page scrape

-page scrape for bs4 (page scrape working for sel)

-assess whitespace split to help headers

-requests 200 requirement for first href selection

-account for more options in 'assess' functions

-add website_url parameter into sel_nav_scrape for consistency

-FIX nav scrape for sel (nav scrape working for bs4). Specifically, first_href - not critical because very slow.

Figure out alternative when no nav (all a tags' hrefs or first relevant href?) (ex: https://www.forsalebyowner.com)

> Make sure that first_href returned is a URL in both bs4_nav_scrape() and sel_nav_scrape()

-first text on page (home/about)

-mistral given as many pages as possible (via nav) -> really slow (~15 mins) save until next step

-utilize irrelevant in asses_href

0.0.2 All imports

[ ]:
```
import requests
import pandas as pd
import validators
import time
from openai import OpenAI
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
from selenium_stealth import stealth
from selenium import webdriver
import re
import urllib.parse as up
from bs4 import BeautifulSoup

import asyncio
from urllib.parse import urlparse
import aiohttp
import nest_asyncio
import os
```

0.0.3 Important meta tags

[ ]:
```
def get_meta_tags(url): #: str) -> dict[str,str]:
    api_key = os.getenv('JSON_LINKS_KEY')

#     url = 'https://cessco.ca/'
    # url = 'https://www.appliedlearning.com'

    params = {'url': url, 'api_key': api_key}

    # Free JsonLink limit is 30 req/minute so wait 3 seconds just in case
    time.sleep(3)

    response = requests.get('https://jsonlink.io/api/extract', params=params)

    if response.status_code == 200:
        data = response.json()
#         print(data)
```

```
        print('Title: ', data['title'], '\nDescription: ', data['description'],␣
↪'\nDomain: ', data['domain'])
        return {'metadata':{k: data.get(k,None) for k in ('title',␣
↪'description', 'domain')}}
    else:
        print(f'JSONLink Error for {url}: {response.status_code} - {response.
↪text}')
        return {'metadata': 'Metadata unavailable'}
```

[ ]:
```
get_meta_tags('https://www.dentalxchange.com/')
```

0.0.4 Handle navigation

Get all a tags (get best nav)

1. test if href valid url

2. test if url + (optional /) + href valid url

3. likely a bust

Selenium

[ ]:
```
def sel_normalize_whitespace(text):
    # Replace one or more whitespace characters (including spaces, tabs, and␣
↪newlines) with a single space
    return re.sub(r'\s+', ' ', text).strip()

def sel_assess_href(base_url: str, href: str) -> str:
    if not validators.url(href):
        href = up.urljoin(base_url,href)
    # Add functionality here to compare if one is contained in the other
    return [href, 'relevant' if up.urlparse(href).netloc == up.
↪urlparse(base_url).netloc else 'irrelevant']

def sel_find_relevant_hrefs(driver):
    atags = driver.find_elements("xpath","//a")
    relevant_hrefs = []
    for a in atags:
        text, href = sel_normalize_whitespace(a.get_attribute('textContent')),a.
↪get_attribute('href')
        website_url = driver.current_url
        assessed_href = sel_assess_href(website_url, href)
        if assessed_href[1] == 'relevant' and (assessed_href[0] != website_url␣
↪and text != 'Skip to content'):
            relevant_hrefs += [(text, assessed_href[0])]
    return relevant_hrefs
```

```
def sel_find_first_href(home_page_url, nested_list) -> str:
    for item in nested_list:
        if isinstance(item, list):
                    # Recursively search within the list
                    result = sel_find_first_href(home_page_url, item)
                    if result:  # If a valid URL is found in the recursion,␣
 ↪return it
                        return result
        elif isinstance(item, tuple) and len(item) == 2:
            # If the item is a tuple with 2 elements, check the second element␣
 ↪for a relevant URL
            if validators.url(item[1]) and item[1] != home_page_url:
                return item[1]  # Return the URL if it's valid
    return 'No href found'

def sel_build_tree(base_url,element):
    # Initialize the node with tag name and text content
    node_contents = {'text': sel_normalize_whitespace(element.
 ↪get_attribute('textContent')),
                     'href': sel_assess_href(base_url,element.
 ↪get_attribute('href'))[0]} if element.tag_name == 'a' else {}
    node = {
        **node_contents,
        'children': []
    }

    # Recursively build the tree for each child element
    children = element.find_elements(By.XPATH, "./*")  # Only direct children
    for child in children:
        node['children'].append(sel_build_tree(base_url,child))

    if not node['children'] or all(not obj for obj in node['children']):
        del node['children']

    return node

def sel_convert_tree(root) -> list[list,int]:
    ans, total_hrefs = [], 0
    if 'children' not in root:
        if 'text' in root and 'href' in root:
            return (root['text'], root['href'], 1)
    else:
        for child in root['children']:
            links = sel_convert_tree(child)
            if links:
                total_hrefs += links[-1]
                ans += [links[:-1]]
    return [*list(filter(None, ans)), total_hrefs]

# TODO: Fix this - first_href must be a URL (even if no nav, find first␣
 ↪relevant href but need to specify def of relevant)
# Returns the nav tree (either advanced nested or basic list of hrefs) and␣
 ↪first href
def sel_nav_scrape(driver) -> list[list[tuple[str,str]],str]:
    sel_nav_return, nav_trees = [], []
    home_page_url = driver.current_url

    # Find navs, construct trees and find max
    navs = driver.find_elements("xpath","//nav")
    for nav in navs:
        nav_trees.append(sel_build_tree(home_page_url, nav))
    max_nav = ({}, 0)
    for tree in nav_trees:
        converted = sel_convert_tree(tree)
        if converted[-1] > max_nav[-1]:
            max_nav = converted
    # [:-1] to account for nested tree
    max_nav_tree = max_nav[:-1]

    # Construct return
    if not max_nav[0]:
    # if max_nav == ({}, 0): #or len(max_nav[-1]) < x:
        # If no/not enough navs, find all relevant atags
        relevant_hrefs = sel_find_relevant_hrefs(driver)
        sel_nav_return.append(relevant_hrefs)
        first_href = relevant_hrefs[0][1] if relevant_hrefs else 'No href found'
    else:
        sel_nav_return.append(max_nav_tree)
        first_href = sel_find_first_href(home_page_url, max_nav)

    sel_nav_return.append(first_href)
    return sel_nav_return
```

Example

[ ]:
```
nav_driver = webdriver.Chrome()
nav_driver.get('https://forsalebyowner.com/')
sel_nav = sel_nav_scrape(nav_driver)
nav_driver.close()
sel_nav
```

bs4

```
[ ]: def bs4_build_tree(base_url, element):
         # Initialize the node with tag name and text content
         node_text = {'text': element.get_text(strip=True)} if element.name == 'a'␣
      ↪else {}
         node = {
             **node_text,
             'children': []
         }

         # If it's an <a> tag, include the href attribute
         if element.name == 'a':
             node['href'] = bs4_assess_href(base_url,element.get('href'))[0]

         # Recursively build the tree for each child element
         for child in element.find_all(recursive=False):  # Only direct children
             node['children'].append(bs4_build_tree(base_url, child))

         if not node['children'] or all(not obj for obj in node['children']):
             del node['children']

         return node

     def bs4_convert_tree(root):
         ans, total_hrefs = [], 0
         if 'children' not in root:
             if 'text' in root and 'href' in root:
                 return (root['text'],'' if root['href'] == 'javascript:void(0);'␣
      ↪else root['href'],1)
                 # Aesthetic output
                 # return (f"root['text']}-> {root['href']}",1)
             # else:
             #     return ['','',0]
         else:
             for child in root['children']:
                 # print(child)
                 links = bs4_convert_tree(child)
                 if links:
                     # print(links)
                     total_hrefs += links[-1]
                     ans += [links[:-1]]

         return [*list(filter(None,ans)),total_hrefs]

     # Need to handle javascript:void(0); case
     def bs4_assess_href(base_url, href) -> str:
         if not validators.url(href):
             href = up.urljoin(base_url,href)
```

```
    return [href, 'relevant' if up.urlparse(href).netloc == up.
↪urlparse(base_url).netloc else 'irrelevant']

def bs4_find_relevant_hrefs(soup, website_url: str) -> list[tuple[str, str]]:
    atags = soup.find_all('a')
    relevant_hrefs = []
    for a in atags:
        text, href = a.get_text(strip=True),a.get('href')
        assessed_href = bs4_assess_href(website_url, href)
        if assessed_href[1] == 'relevant' and (assessed_href[0] != website_url␣
↪and text != 'Skip to content'):
            relevant_hrefs += [(text, assessed_href[0])]
    return relevant_hrefs

def bs4_find_first_href(home_page_url, nested_list) -> str:
    for item in nested_list:
        if isinstance(item, list):
                    # Recursively search within the list
                    result = bs4_find_first_href(home_page_url, item)
                    if result:  # If a valid URL is found in the recursion,␣
↪return it
                        return result
        elif isinstance(item, tuple) and len(item) == 2:
            if validators.url(item[1]) and item[1] != home_page_url:
                return item[1]  # Return the URL if it's valid
    return 'No href found'

def bs4_nav_scrape(website_url: str, soup) -> list[list[tuple[str,str]],str]:
    bs4_nav_return, nav_trees = [], []

    # Find navs, construct trees, find max
    navs = soup.find_all('nav')
    for nav in navs:
        nav_trees.append(bs4_build_tree(website_url, nav))
    max_nav = ({},0)
    for tree in nav_trees:
        converted = bs4_convert_tree(tree)
        if converted[-1] > max_nav[-1]:
            max_nav = converted
    # [:-1] to account for nested tree
    bs4_max_nav_tree = max_nav[:-1]

    # Construct return
    if not max_nav[0]: #or len(max_nav[-1]) < x:
        # If no/not enough navs, find all relevant atags
        relevant_hrefs = bs4_find_relevant_hrefs(soup, website_url)
        bs4_nav_return.append(relevant_hrefs)
        first_href = relevant_hrefs[0][1] if relevant_hrefs else ''
    else:
        bs4_nav_return.append(bs4_max_nav_tree)
        first_href = bs4_find_first_href(website_url, max_nav)

    bs4_nav_return.append(first_href)
    return bs4_nav_return

# url = 'https://www.dentalxchange.com/'
# url = 'https://ecmins.com/'
# url = 'https://iquartic.com/' # blocked on requests
# url = 'https://www.ripoffreportremovalhelp.com/' # blocked on requests
# url = 'https://pulseca.com/'
url_test = 'https://www.scorpion.co/'

html = requests.get(url_test).content
soupt = BeautifulSoup(html, 'html.parser')

# url_test = 'https://www.pavestone.com/'
response = requests.get(url_test)
soupy = BeautifulSoup(response.text, 'html.parser')

# response = requests.get(url_test).content
# soupr = BeautifulSoup(response, 'html.parser')
bs4_nav_scrape(url_test, soupy)
# soupy.find_all('a')
# print(soupr.find('h2'))
```

Report: it seems as though the javascript:void(0); case is handled because validators.url thinks it's valid, but the netloc's are not the same, so it's labelled irrelevant.
TODO: Need to figure out nav name (the text only in the nav element, not in the contained a's, create tree-like structure.

0.0.5 Scraping Methods

[ ]:
```
def word_count(seg):
    count = 0
    for i in seg:
        if i == ' ':
            count += 1
    return count+1
```

bs4

[ ]:
```
def bs4_pages_scrape(urls: list[str]) -> list[dict]:
    pages = []
    for url in urls:
        if url and validators.url(url):
            try:
                response = requests.get(url).text
            except Exception as e:
                print(f'HTTPRequest error: {e}')
                pages.append({'headers':['Page not available']})
                return pages
            soup = BeautifulSoup(response, 'html.parser')
            # Split on any whitespace (\n and \t) -> maybe this is causing␣
 ↪weird headers
            page_text = soup.get_text("|",strip=True).split("|")
            # Extract the first two pieces of text with more than (7) words ->␣
 ↪to be tested
            first_relevant = {'first_relevant': [i for i in page_text if␣
 ↪word_count(i) > 7][:2]}
            # Two longest pieces of text on the page. Test if this produces␣
 ↪relevant results
            two_longest = {'two_longest': sorted(page_text,key=len)[-2:]}
            # Find all h1s and h2s
            h1s = soup.find_all('h1')
            h2s = soup.find_all('h2')
            h1_texts = [h1.get_text(strip=True) for h1 in h1s]
            h2_texts = [h2.get_text(strip=True) for h2 in h2s]
            headers = {'headers': list(filter(None,h1_texts+h2_texts))}
            pages.append({**first_relevant, **two_longest,**headers})
        else:
            pages.append({'headers':['Page not available']})

    return pages

# Takes response_text instead of a URL since the request is required to␣
 ↪determine bs4/sel
def bs4_scrape(website_url: str, response_text: str) ->␣
 ↪dict[str,str|dict[str,str]]:
    soup = BeautifulSoup(response_text,'html.parser')
    url_results = {}

    #Scrape nav
    nav_list = bs4_nav_scrape(website_url, soup)
    if not nav_list[0]: return 'BS4 Nav list unavailable'
    url_results['nav'] = nav_list

    # Scrape home page and if there, first page
    urls = [website_url]
    if nav_list[1] and validators.url(nav_list[1]):
        urls.append(nav_list[1])
    pages = bs4_pages_scrape(urls)
    home_page_obj = pages[0]
    first_page_obj = pages[1] if len(pages) > 1 else {'headers':['First page␣
 ↪unavailable']}
    url_results['home_page'], url_results['first_page'] = home_page_obj,␣
 ↪first_page_obj

    # Extract headers
    url_results['headers'] = home_page_obj['headers'] +␣
 ↪first_page_obj['headers']

    return url_results

urler = 'https://www.forsalebyowner.com/'
response = requests.get(urler)
print(response.url)
souper = BeautifulSoup(response.content, 'html.parser')
bs4_scrape(urler, response.text)
```

Selenium

```
[ ]: # Gathers first two relevant chunks of texts, two longest chunks of text and␣
 ↪all h1s and h2s from every url in list then closes stealth driver fed in
def sel_pages_scrape(driver, urls: list[str]) -> dict:
    pages = []
    for url in urls:
        if url and validators.url(url):
            driver.get(url)
            time.sleep(2)
            page_text = driver.find_element("xpath","/html/body").text
            # Split on any whitespace (\n and \t)
            page_array = re.split(r'[\n\t]+',page_text)
            # Extract the first two pieces of text with more than (7) words ->␣
 ↪to be tested
            first_relevant = {'first_relevant': [i for i in page_array if␣
 ↪word_count(i) > 7][:2]}
            # Two longest pieces of text on the page. Test if this produces␣
 ↪relevant results
            two_longest = {'two_longest': sorted(page_array,key=len)[-2:]}
            h1s = driver.find_elements("xpath","//h1")
            h2s = driver.find_elements("xpath","//h2")
            h1_texts = [h1.text for h1 in h1s if h1]
            h2_texts = [h2.text for h2 in h2s if h2]
            headers = {'headers':list(filter(None,h1_texts+h2_texts))}
            pages.append({**first_relevant, **two_longest, **headers})
        else:
            pages.append({'headers':['First page unavailable']})
    # driver.close()
    return pages

def sel_scrape(url: str) -> dict[str,str|dict[str,str]]:
    print(f'Selenium scraping {url}')
    url_results = {}

    #Scrape nav
    driver = webdriver.Chrome()
    driver.get(url)
    nav_list = sel_nav_scrape(driver)
    driver.close()
    url_results['nav'] = nav_list
    print(f'{up.urlparse(url).netloc} sel naver', nav_list)

    # Configure driver to be passed throughout
    options = webdriver.ChromeOptions()
    options.add_argument("--start-maximized")
    stealth_driver = webdriver.Chrome(options=options)
    stealth(stealth_driver,
            languages=["en-US", "en"],
            vendor="Google Inc.",
            platform="Win32",
            webgl_vendor="Intel Inc.",
            renderer="Intel Iris OpenGL Engine",
            fix_hairline=True,
            )
    stealth_driver.set_window_size(1100, 720)
    # stealth_driver.get(url)

    # Scrape home and first pages (requires both of these to have urls).
    home_page_obj, first_page_obj = sel_pages_scrape(stealth_driver, [url,␣
 ↪nav_list[1]])
    url_results['home_page'], url_results['first_page'] = home_page_obj,␣
 ↪first_page_obj

    # Extract headers
    url_results['headers'] = home_page_obj['headers'] +␣
 ↪first_page_obj['headers']

    # stealth_driver.close()

    return url_results
```

Example

```
[ ]: # https://www.appliedlearning.com
     # https://sunstonepartners.com
     # https://ecmins.com
     # https://www.dentalxchange.com/
     # https://pulseca.com/
     # https://cessco.ca/
     # sel_scrape('https://www.dentalxchange.com/')
     # options = Options()

     nav_driver = webdriver.Chrome()
     sel_pages_scrape(nav_driver,['https://www.forsalebyowner.com/','https://www.
      ↪forsalebyowner.com/sellyourhome/package'])
     nav_driver.quit()
```

Ancillary Functions

```
[ ]: def initial_processing(url):
         if not url or url != url or pd.isna(url):
             return ''

         # Sanitize URL
         corrected_url = sanitize_url(url)
         return corrected_url

     # Function to sanitize/correct URLs missing pieces
     def sanitize_url(url):
         # Parse URL to correct any issues then reconstruct
         parsed_url = urlparse(url)

         if not parsed_url.scheme:
         # Assume http scheme
             corrected_url = 'http://'+parsed_url.netloc + parsed_url.path +␣
      ↪parsed_url.params + parsed_url.query + parsed_url.fragment
         else:
             corrected_url = parsed_url.geturl()

         return corrected_url

     async def check_url(session, url, semaphore):
         async with semaphore:
             try:
                 async with session.head(url, allow_redirects=True, timeout=100) as␣
      ↪response:
                     return str(response.url) # Return final URL as string
             # Catch errors
             except asyncio.TimeoutError as te:
                 return 'Timeout Error'
```

```
        except aiohttp.ClientError as ce:
            return 'Client Error'
        except ValueError as ve:
            return 'Value Error'

async def capture_url_redirects(urls, MAX_CONCURRENT_REQUESTS):
    print(f"processing {len(urls)} urls")
    semaphore = asyncio.Semaphore(MAX_CONCURRENT_REQUESTS)
    async with aiohttp.ClientSession() as session:
        tasks = [check_url(session, url, semaphore) for url in urls]
        results = await asyncio.gather(*tasks)
        return results

async def return_invalid_url_object(url) -> dict[str, str | dict | list]:
    return {'website_redirect': url, 'nav': 'Invalid_URL','home_page':
 ↪{},'first_page':{},'headers':[]}

# # Specify the desired Chromium version
# os.environ['PYPPETEER_CHROMIUM_REVISION'] = '1263111'
# chromium_revision = os.getenv('PYPPETEER_CHROMIUM_REVISION', 'Environment␣
 ↪variable not set')
# print('sdfsdf',chromium_revision)
# print(pyppeteer.__chromium_revision__)
async def pypp_scrape(url):
    print(f'pypp_scrape {url}')
    browser = await launch(headless=True, executablePath='C:/Users/leogr/
 ↪AppData/Local/pyppeteer/pyppeteer/chrome-win/chrome-win/chrome.exe')
    page = await browser.newPage()
    # start_time = time.time()
    await page.goto(url)
    # elapsed_time = time.time() - start_time
    cookies = await page.cookies()
    # print(f"Page loaded in {elapsed_time} seconds.")
    await browser.close()
    return cookies

# async def download_chromium():
#     browser = await launch()
#     await browser.close()

# print(pyppeteer.__chromium_revision__)

# asyncio.run(download_chromium())

# url = 'https://cessco.ca/'
# # url = 'https://ecmins.com/'
# asyncio.run(load_page(url))
```

```
def update_redirect_urls(file_path, index_range, redirect_urls):
    df = pd.read_csv(file_path, low_memory=False)
    new_list = list(df['Website Redirect'][:index_range.start]) + redirect_urls␣
 ↪+ list(df['Website Redirect'][index_range.stop:])
    df['Website Redirect'] = new_list
    df.to_csv(file_path, index=False)

def construct_df_col(df, col_name: str, scrape_col: list, index_range: slice,␣
 ↪col_exists: bool):
    if col_exists:
        return list(df[col_name][:index_range.start]) + scrape_col +␣
 ↪list(df[col_name][index_range.stop:])
    else:
        return ['']*index_range.start + scrape_col + ['']*(len(df)-index_range.
 ↪stop)

def update_scrape_results(file_path: str, scrape_results: list[dict],␣
 ↪index_range: slice):
    df = pd.read_csv(file_path, low_memory=False)
    print('TYPE',type(scrape_results),type(scrape_results[0]))
    # TODO: Add 'Metadata' here when ready
    for column in ['Website Redirect','Nav','Headers','Home Page','First Page']:
        isolated_col = [result['_'.join(column.lower().split(' '))] for result␣
 ↪in scrape_results]
        df[column] = construct_df_col(df, column, isolated_col, index_range,␣
 ↪column in df)
    df.to_csv(file_path, index=False)
```

0.0.6 Threaded Main

[ ]:
```
import asyncio
import aiohttp
import pandas as pd
from urllib.parse import urlparse
import validators
import time
from concurrent.futures import ThreadPoolExecutor
from pyppeteer import launch
import pyppeteer
import os

nest_asyncio.apply()

# Initialize Selenium drivers for each type of task
async def init_driver_pool(size):
    queue = asyncio.Queue(maxsize=size)
    for _ in range(size):
        options = webdriver.ChromeOptions()
        # options.add_argument("--start-maximized")
        stealth_driver = webdriver.Chrome(options=options)
        stealth(stealth_driver,
                languages=["en-US", "en"],
                vendor="Google Inc.",
                platform="Win32",
                webgl_vendor="Intel Inc.",
                renderer="Intel Iris OpenGL Engine",
                fix_hairline=True,
                )
        # stealth_driver.set_window_size(1100, 720)
        await queue.put(stealth_driver)
    return queue

# Close all drivers in the pool
async def close_driver_pool(driver_pool):
    while not driver_pool.empty():
        driver = await driver_pool.get()
        driver.quit()
        driver_pool.task_done()

async def capture_redirect(session, url, headers, semaphore, executor) ->␣
 ↪list[str,str]:
    async with semaphore:
        try:
            # First, attempt to scrape using aiohttp
            async with session.get(url, allow_redirects=True, headers=headers,␣
 ↪timeout=150) as response:
                if response.status//100 == 2:
                    response_text = await response.text()
                    bs4_result = bs4_scrape(url, response_text)
                    if bs4_result == 'BS4 Nav list unavailable':
                        # Define no BS4 nav as 600 error
                        raise Exception('BS4 doesn\'t know where to go -> 600')
                    return [response.url, 'bs4',bs4_result]
                else:
                    raise Exception(f"Non-200 response -> {response.status}")
        # TODO: Catch errors better (cessco)
        except asyncio.TimeoutError as te:
            # TODO: go back and selenium all of these with longer timeout,␣
 ↪returning invalid to get through
            return ['Timeout_Error', 'invalid']
        except aiohttp.ClientError as ce:
            return ['Client_Error', 'invalid']
        except ValueError as ve:
            return ['Value_Error', 'invalid']
        except Exception as e:
            print(f'Error with {url}: {e}')
            try:
                error_code = int(str(e).split(' ')[-1])
                # TODO: figure out 464 error for hellohero
                if type(error_code == int) and (error_code // 100 == 5 or␣
↪error_code == 404):
                    return ['Invalid_URL', 'invalid']
            except Exception as e:
                raise Exception(f'Error with exception: {e}')
            # Fallback to Selenium scraping within the thread pool executor
            return ['Pyppeteer','pyppeteer']
            # return [url, 'selenium']

async def nav_scrape(final_url, session, semaphore, executor, driver_pool) ->␣
↪list[list[tuple[str, str]],str]:
    nav_driver = await driver_pool.get()
    nav_driver.get(final_url)
    ret = sel_nav_scrape(nav_driver)
    await driver_pool.put(nav_driver)
    return ret

async def home_page_scrape(final_url, session, semaphore, executor,␣
↪driver_pool):
    home_driver = await driver_pool.get()
    home_driver.get(final_url)
    ret = sel_pages_scrape(home_driver, [final_url])[0]
    await driver_pool.put(home_driver)
    return ret

async def first_page_scrape(first_url, session, semaphore, executor,␣
↪driver_pool):
    first_driver = await driver_pool.get()
    if validators.url(first_url):
        first_driver.get(first_url)
    ret = sel_pages_scrape(first_driver, [first_url])[0]
    await driver_pool.put(first_driver)
    return ret

# Coordination point of website scraping
async def scrape_url_async(session, url, driver_pool):
    print(f'Starting {url} scrape.')
    MAX_CONCURRENT_REQUESTS, MAX_WORKERS = 1000, 3
    semaphore = asyncio.Semaphore(MAX_CONCURRENT_REQUESTS)
    executor = ThreadPoolExecutor(max_workers=MAX_WORKERS)
    # Assemble headers
    headers = {
        'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.
↪36 (KHTML, like Gecko) Chrome/122.0.0.0 Safari/537.36 Edg/122.0.0.0',
        'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/
↪avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
        'Referer':url
    }

    redirect_task = asyncio.create_task(capture_redirect(session, url, headers,␣
↪semaphore, executor))
    redirect_return = await redirect_task
    final_url, scrape_type = redirect_return[:2]
    print('stype',scrape_type)
    # bs4_result acquired
    if scrape_type == 'bs4':
        print(f'{url} processed by bs4.')
        return {'website_redirect': final_url, **redirect_return[2]}
    elif scrape_type == 'invalid':
        print(f'{url} processed. Invalid: {final_url}.')
        return await return_invalid_url_object(final_url)

    # TODO: What are the repsonses when it's invalid vs. pypp?

    # TODO: pypp gets home page text?
    pypp_task = asyncio.create_task(pypp_scrape(url))
    pypp_return = await pypp_task
    # headers['Cookie'] = asemble_relevant_cookie(pypp_return)
    headers['Cookie'] =␣
↪'sd_fw_data=3f877dcf6ce2b0cd5ff8421da7101cb0|1|IN78Nl9dz9599|V2luMzJ8ZmFsc2V8ZW4tVVN8NS4wIC

    redirect_retask = asyncio.create_task(capture_redirect(session, url,␣
↪headers, semaphore, executor))
    redirect_rereturn = await redirect_retask
    final_url, scrape_type = redirect_rereturn[:2]
    print('wht',final_url,scrape_type)
    # bs4_result acquired
    if scrape_type == 'bs4':
        print(f'{url} processed by bs4.')
        return {'website_redirect': final_url, **redirect_return[2]}
    elif scrape_type == 'invalid':
        print(f'{url} processed. Invalid: {final_url}.')
        return await return_invalid_url_object(final_url)
    print('pypper',pypp_return)
    return await return_invalid_url_object(url)


    loop = asyncio.get_running_loop()
    nav_task = asyncio.create_task(nav_scrape(final_url, session, semaphore,␣
↪executor, driver_pool))
    # nav_task = loop.run_in_executor(executor, nav_scrape, final_url, session,␣
↪semaphore, executor, driver_pool)
    home_task = loop.run_in_executor(executor, home_page_scrape, final_url,␣
↪session, semaphore, executor,driver_pool)

    # Handle first page scrape
    # Await nav_task to ensure nav_info is available for first_page_scrape
    nav_info = await nav_task
    first_page_url = nav_info[-1]
    first_page_data, home_page_data = {}, {}
    if validators.url(first_page_url):
        # first_page_task = asyncio.create_task(first_page_scrape(nav_info[-1],␣
↪session, semaphore, executor, driver_pool))
        first_page_task = loop.run_in_executor(executor, first_page_scrape,␣
↪nav_info[-1], session, semaphore, executor, driver_pool)

        # Await all tasks and collect results
        home_page_data, first_page_data = await asyncio.gather(await home_task,␣
↪await first_page_task)
    else:
        first_page_data = {'headers':'No first page found'}
    # TODO: unnecessary because the data is already there?
    # headers = home_page_data['headers'] + first_page_data['headers']
    print(f'{url} processed by sel.')
    return {'website_redirect': final_url,'nav':nav_info[:-1], 'home_page':
↪home_page_data, 'first_page': first_page_data, 'headers':[]} #, 'headers':
↪headers}

async def threaded_main(start, stop):
    start_time = time.time()
    MAX_DRIVERS = 12

    # Excel index = 2 + this index
    # start, stop = 150,200
    if stop <= start:
        print('Start must be strictly less than stop')
        return 1
    index_range = slice(start, stop)
    # Load your URLs from a file or list
    file_path = './Excel_Sheets/Website_Redirects_230919.csv'
    df = pd.read_csv(file_path, low_memory=False)
    raw_urls = df['Website'][index_range].tolist()
    if 'Website Redirect' in df:
        redirect_urls = df.get('Website Redirect', pd.Series(dtype=str)).
↪tolist()[index_range]

    print(raw_urls, redirect_urls, redirect_urls[0] == True)

    # Check if 'Website Redirect' column is already populated (with valid URL)
    for i, redirect_url in enumerate(redirect_urls):
        if redirect_url:
            print('plp',i, redirect_url == False, redirect_url == True,␣
↪str(redirect_url) == True)
        if redirect_url and validators.url(redirect_url):
            raw_urls[i] = redirect_url

    sanitized_urls = [initial_processing(url) for url in raw_urls]
    valid_urls = [url if validators.url(url) else '' for url in sanitized_urls]

    scrape_tasks = []
    # driver_pool = await init_driver_pool(MAX_DRIVERS)
    driver_pool = []

    async with aiohttp.ClientSession() as session:
        scrape_tasks = [scrape_url_async(session, url, driver_pool) for url in␣
↪valid_urls]
        scrape_results = await asyncio.gather(*scrape_tasks)

        # returns single scrape_result
    #     scrape_task = asyncio.create_task(scrape_url_async(session, url,␣
↪driver_pool))
    #     scrape_tasks.append(scrape_task)
    # scrape_results = await asyncio.gather(*scrape_tasks)

    print('closing pool')
    count = 0

    # await close_driver_pool(driver_pool)
    for i in scrape_results:
        if i['nav'] == 'Invalid_URL' or (type(i['website_redirect']) == str and␣
↪'Error' in i['website_redirect']):
            count+=1
    print('scrrr',count, type(scrape_results),type(scrape_results[0]),␣
↪scrape_results)
```

```
    # loop = asyncio.get_event_loop()
    # scrape_results = loop.run_until_complete(main_async(valid_urls))

    update_scrape_results(file_path, scrape_results, index_range)

    print(f"Completed in {time.time() - start_time} seconds. Excel␣
 ↪{index_range} updated.")

# loop = asyncio.get_event_loop()
asyncio.run(threaded_main(239,240))
```

```
[ ]: for i in range(9,10):
         asyncio.run(threaded_main(i*50,(i+1)*50))
```

0.0.7 Handle SINGULARITIES

```
[ ]: sel_scrape('https://www.cosmonetsolutions.com')
```

```
[ ]: # driver = webdriver.Chrome()
     # driver.get('https://hellohero.com')
     x,y = 9,10
     tost_pg_res = [{'website_redirect': 'https://21stsoft.com', **sel_scrape('https:
      ↪//21stsoft.com')}]
     file_path = './Excel_Sheets/Website_Redirects_230919.csv'
     update_scrape_results(file_path,tost_pg_res, slice(x,y))

     # driver.close()
```

aiohttp response different:

```
[ ]: requests.get('https://www.cosmonetsolutions.com')
```

Pyppeteer/Playwright

```
[ ]: from pyppeteer import launch
     import pyppeteer
     import asyncio
     import time
     import os

     # Specify the desired Chromium version
     os.environ['PYPPETEER_CHROMIUM_REVISION'] = '1263111'
     chromium_revision = os.getenv('PYPPETEER_CHROMIUM_REVISION', 'Environment␣
      ↪variable not set')
     print('sdfsdf',chromium_revision)
     print(pyppeteer.__chromium_revision__)

     nest_asyncio.apply()

     async def load_page(url):
         browser = await launch(headless=True, executablePath='C:/Users/leogr/
      ↪AppData/Local/pyppeteer/pyppeteer/chrome-win/chrome-win/chrome.exe')
         page = await browser.newPage()
         start_time = time.time()
         await page.goto(url)
         elapsed_time = time.time() - start_time
         cookies = await page.cookies()
         print(f"Page loaded in {elapsed_time} seconds.")
         await browser.close()
         return cookies

     async def download_chromium():
         browser = await launch()
         await browser.close()

     # print(pyppeteer.__chromium_revision__)

     # asyncio.run(download_chromium())

     url = 'https://cessco.ca/'
     # url = 'https://ecmins.com/'
     asyncio.run(load_page(url))
```

```
[ ]: from playwright.async_api import async_playwright

async def tester():
    with async_playwright() as p:
        print(p)

t = await tester()
t
```

```
[ ]: urlh = 'https://cessco.ca/'

async with aiohttp.ClientSession() as session:
     async with session.get(urlh, allow_redirects=True, timeout=50) as response:
        print('sta',response.status)

        # print('STATUS', response.status,type(response.text()),type(response.
 ↪text),type(response))
        if response.status == 200:
            # bs4_soup = BeautifulSoup(response.text(), 'html.parser')
            # print('BS4 result',bs4_soup)
            # response.text() coroutine for asynchroneity
            response_text = await response.text()
            print('r',type(response_text), response_text)
            bs4_result = bs4_scrape(urlh, response_text)

        try:
            response = requests.get(urlh)
            # res = bs4_scrape(urlh,response.text)
            soup1 = BeautifulSoup(response.text, 'html.parser')
            # soup2 = BeautifulSoup(await response.text(), 'html.parser')
            bs5_result = bs4_scrape(urlh, response.text)
            # print(res)
        except Exception as e:
            print('fail', e)
```

```
[ ]: # urlh = 'https://www.cessco.ca/'
     # urlh = 'https://hellohero.com'
     urlh = 'https://www.cosmonetsolutions.com'

     async with aiohttp.ClientSession() as session:
          async with session.get(urlh, allow_redirects=True, timeout=50) as response:
             print('sta',response.status)

             # print('STATUS', response.status,type(response.text()),type(response.
       ↪text),type(response))
             if response.status == 200:
                 # bs4_soup = BeautifulSoup(response.text(), 'html.parser')
                 # print('BS4 result',bs4_soup)
                 # response.text() coroutine for asynchroneity
                 response_text = await response.text()
                 print('r',type(response_text), response_text)
                 bs4_result = bs4_scrape(urlh, response_text)

             try:
                 response = requests.get(urlh)
                 # res = bs4_scrape(urlh,response.text)
                 soup1 = BeautifulSoup(response.text, 'html.parser')
                 # soup2 = BeautifulSoup(await response.text(), 'html.parser')
                 bs5_result = bs4_scrape(urlh, response.text)
                 # print(res)
             except Exception as e:
                 print('fail', e)

     print('res',bs4_result, bs5_result)

     # sel_scrape('http://www.academicresourcesolutions.com')
```

[ ]:

```
a = {'b':3,'c': 4}
d = {'redirect':'asdf',**a}
d
requests.get('http://quickcarepharmacy.com')
```

[ ]:

```

```