CitySearch

Webscraping

Libraries

In [ ]:
```
import re
import time
import asyncio
import pandas as pd
from bs4 import BeautifulSoup
from requests_html import AsyncHTMLSession

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException, \
    NoSuchElementException, WebDriverException, StaleElementReferenceException
```

Global Variables

In [ ]:
```
us_state_to_abbrev = {
    "Alabama": "AL",
    "Alaska": "AK",
    "Arizona": "AZ",
    "Arkansas": "AR",
    "California": "CA",
    "Colorado": "CO",
    "Connecticut": "CT",
    "Delaware": "DE",
    "Florida": "FL",
    "Georgia": "GA",
    "Hawaii": "HI",
    "Idaho": "ID",
    "Illinois": "IL",
    "Indiana": "IN",
    "Iowa": "IA",
    "Kansas": "KS",
    "Kentucky": "KY",
    "Louisiana": "LA",
    "Maine": "ME",
    "Maryland": "MD",
    "Massachusetts": "MA",
    "Michigan": "MI",
    "Minnesota": "MN",
    "Mississippi": "MS",
    "Missouri": "MO",
    "Montana": "MT",
```

```
    "Nebraska": "NE",
    "Nevada": "NV",
    "New Hampshire": "NH",
    "New Jersey": "NJ",
    "New Mexico": "NM",
    "New York": "NY",
    "North Carolina": "NC",
    "North Dakota": "ND",
    "Ohio": "OH",
    "Oklahoma": "OK",
    "Oregon": "OR",
    "Pennsylvania": "PA",
    "Rhode Island": "RI",
    "South Carolina": "SC",
    "South Dakota": "SD",
    "Tennessee": "TN",
    "Texas": "TX",
    "Utah": "UT",
    "Vermont": "VT",
    "Virginia": "VA",
    "Washington": "WA",
    "West Virginia": "WV",
    "Wisconsin": "WI",
    "Wyoming": "WY",
    "District of Columbia": "DC",
    "American Samoa": "AS",
    "Guam": "GU",
    "Northern Mariana Islands": "MP",
    "Puerto Rico": "PR",
    "United States Minor Outlying Islands": "UM",
    "U.S. Virgin Islands": "VI",
}

industries = [
    {
        "Construction": [
            "Carpentry",
            "Plumbing",
            "Electrical work"
        ]
    },

    {
        "Manufacturing": [
            "Welding",
            "Machine operation",
            "Assembly line work"
        ]
    },

    {
        "Transportation": [
            "Truck driving",
            "Warehouse operations",
            "Forklift operation"
        ]
```

```
    },

    {
        "Logistics": []
    },

    {
        "Automotive": [
            "Automotive repair",
            "Auto maintenance",
            "Auto Bodywork",
            "Tire services"
        ]
    },

    {
        "Maintenance and Repair": [
            "HVAC",
            "Appliance repair",
            "General maintenance"
        ]
    },

    {
        "Retail": [
            "Boutiques",
            "Specialty stores",
            "Online shops"
        ]
    },

    {
        "Food and Beverage": [
            "Restaurants",
            "Cafes",
            "Food trucks"
        ]
    },

    {
        "Personal Services": [
            "Hair salons",
            "Barber shops"
        ]
    }
]

states_of_interest = ["California", "New Jersey", "New York", "Texas"]
```

Functions Used

There's a bit going on, so I'll try my best to explain what each function does in the order they are called. I hope it helps understand the main implementation better.

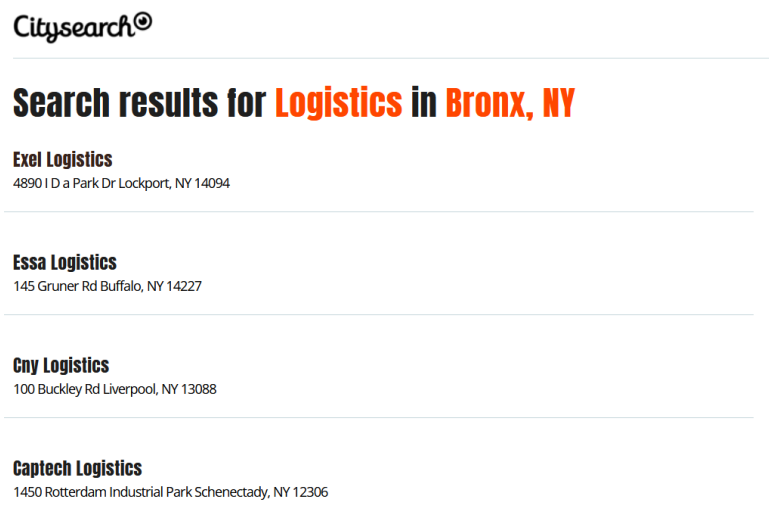
switch()

The switch funciton is simply a quality-of-life function to switch the orders of an output.

In [ ]:
```
def switch(el):
    pos1, pos2 = el.split("/")
    return pos2 + ",%20" + pos1
```

get_job_cards_links()

In CitySearch, per industry and location, there's a list of companies. Here we're locating and saving the links to each company profile. Handling exceptions at this stage wasn't too much of an issue, but on rare occasions there were no companies.

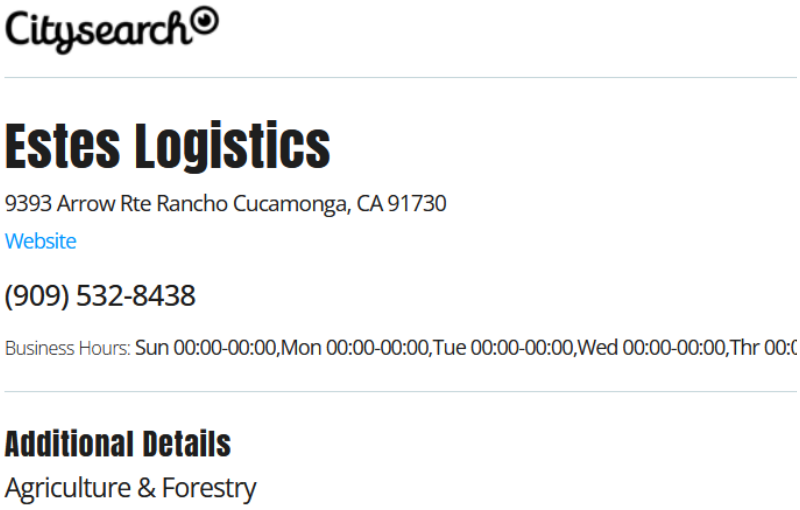


In [ ]:
```
def get_job_cards_links(driver):
    try:
        print('looking for job card list')

        job_cards = WebDriverWait(driver, 10).until(
            EC.presence_of_all_elements_located(
                (By.CSS_SELECTOR, "div.list-container > div.card > a")
            )
        )

        job_cards_links = [job.get_attribute("href") for job in job_cards]

    except (NoSuchElementException, TimeoutException):
        print("Error: Timed out waiting for page to load. \
              Most likely no job listing in this category")

        job_cards_links = []

    return job_cards_links
```

business_details_to_dict()

business_details_to_dict receives a link from the list created above and scrapes business

details, such as name, address, phone number, link to webiste, business hours, and additional details. However, not all businesses had a link to their website. Lastly, I decided to include additional details to get a better idea of what the company actually does. Although the information isn't always useful, it provides important insight especially when it isn't clear whether a company belongs in a specific industry.



In [ ]:
```python
def business_details_to_dict(driver, industry):
    business_details_dict = {'industry': industry}

    try:
        print('looking for business details')
        business_details = WebDriverWait(driver, 5).until(
            EC.presence_of_all_elements_located(
                (By.CSS_SELECTOR, "div.business-details > *")
            )
        )

    except (NoSuchElementException, TimeoutException, StaleElementReferenceExceptio
        print("Timed out waiting for business details to load")
        return {
                    "business-name": "",
                    "address": "",
                    "external-links-container": "",
                    "phone-trigger": "",
                    "industry": "",
                    "additional_info": "",
                    "emails": []
                }

    for entry in business_details:
        try:
            class_name = entry.get_attribute("class")

            if class_name == "external-links-container":
                try:
                    print('looking for external links')
```

```
                    elem = WebDriverWait(driver, 3).until(
                        EC.presence_of_element_located(
                            (
                                By.CSS_SELECTOR,
                                "div.external-links-container a"
                            )
                        )
                    )

                    business_details_dict[class_name] = elem.get_attribute("href")

                except (NoSuchElementException, TimeoutException):
                    print("Timed out waiting for external link: \
                          No link to website")
                    business_details_dict[class_name] = ""

            else:
                business_details_dict[class_name] = entry.text

        except StaleElementReferenceException:
            print("Error: Stale reference exception")

    return business_details_dict
```

get_email() & html_to_string()

Below, I've grouped get_email and html_to_string together because they're always called togehter.

If business_details_to_dict returns a dictionary that includes a company's website, get_email is called and inside get_email, html_to_string is called.

get_email receives the link and send the link to html_to_string. html_to_string parses the page, replaces all whitespace (i.e. \n, \t, etc.) with a space and returns a stringified version of the page. get_email then uses regex to detect all emails within the current page.

The process of replacing whitespace and returning a string was mainly done to avoid emails being scraped incorrectly.

In [ ]:
```
async def get_email(url, email_set):
    print("getting emails")

    pattern = "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"
    body = await html_to_string(url)

    if body is None:
        return

    emails = re.findall(pattern, body)
    email_set.update(emails)

async def html_to_string(url):
```

```
    session = AsyncHTMLSession()

    try:
        r = await session.get(url, verify=False)
        soup = BeautifulSoup(r.html.raw_html, "html.parser")
        body = soup.find('body')
        return body.get_text(separator=" ").strip()

    except:
        print("Error: Website does not exist")
        return None
```

get_email_from_contact()

get_email_from_contact also uses the external link provided by business_details_to_dict. From a companies main page, it looks for the existence of a contact page (i.e. /contact, /Contact, /CONTACT, /contact-us, /Contact-Us, /CONTACT-US). If a contact page is found, it calls get_email and follows the same process as above to extract email addresses.

In [ ]:
```
async def get_email_from_contact(driver, url, email_set):
    try:
        print('looking for email from contact')
        driver.get(url)

        contact_button = WebDriverWait(driver, 5).until(
            EC.presence_of_element_located(
                (
                    By.CSS_SELECTOR,
                    "a[href*='contact'], \
                    a[href*='Contact'], \
                    a[href*='CONTACT']"
                )
            )
        )

        driver.get(contact_button.get_attribute("href"))
        url_to_scrape = driver.current_url

        await get_email(url_to_scrape, email_set)

    except (NoSuchElementException, TimeoutException, WebDriverException):
        print("Error: No contact page")
```

save_to_csv()

This is the last step of each loop. It simply exports all the information gathered into a .csv file.

In [ ]:
```
def save_to_csv(business_list, where_param):
    df = pd.DataFrame.from_dict(business_list)

    df.rename(columns={
        "business-name": "business name",
        "external-links-container": "external link",
```

```
        "phone-trigger": "phone number",
        "business-hours": "business hours"
    }, inplace=True)

    df.to_csv(f'./{where_param.replace(",%20", "_").replace("%20", "_")}.csv',
              index=False)
```

Implementation

In [ ]:

```
##########################################################################################
#------iterating over all the states, then cities, then industries and scarping bus
async def main():
    ###########################################
    #------opening and cleaning xlsx file------#
    df = pd.read_excel("assets/google_maps_keywords.xlsx")
    df.loc[:, ["Country", "State"]] = df.loc[:, ["Country", "State"]].ffill()

    ###########################################################
    #------grouping dataframe by country then by state------#
    grouped_df = df.groupby("Country")
    grouped_countries = grouped_df.get_group("United States")
    grouped_states = grouped_countries.groupby("State")
    states = grouped_states.groups.keys()

    ###########################################################
    #------navigating to the front page of CitySearch------#
    chrome_options = webdriver.ChromeOptions()
    chrome_options.add_argument("--window-size=1920,1080")
    chrome_options.add_argument("--disable-extensions")
    chrome_options.add_argument("--proxy-server='direct://'")
    chrome_options.add_argument("--proxy-bypass-list=*")
    chrome_options.add_argument("--start-maximized")
    chrome_options.add_argument('--headless')
    chrome_options.add_argument('--disable-gpu')
    chrome_options.add_argument('--disable-dev-shm-usage')
    chrome_options.add_argument('--no-sandbox')
    chrome_options.add_argument('--ignore-certificate-errors')

    driver = webdriver.Chrome(options=chrome_options)
    driver.get("https://www.citysearch.com/")

    ###########################################################
    #------extracting the links to individual cities------#
    container = driver.find_element(
        By.CSS_SELECTOR,
        "div.cities-container"
    )
    cities = container.find_elements(
        By.CSS_SELECTOR,
        "li:not([class*='state']) > a"
    )
    city_links = [city.get_attribute("href") for city in cities]

    for state in states:
        visited = set() # skipping business already scraped
```

```
        pattern = re.compile(f".*/{us_state_to_abbrev[state]}/.*", re.IGNORECASE)
        where_params = [
            switch(param)
            for param in [
                link.replace("https://www.citysearch.com/", "")
                for link in city_links
                if bool(pattern.match(link))
            ]
        ]

        for where_param in where_params:
            business_list = []

            for industry in [list(industry.keys())[0] for industry in industries]:
                url = f"https://www.citysearch.com/results?term={industry.strip().r

                print("--------------------state, params, industry----------------
                print("-------------------------", url, "-------------------------
                print(state, where_param, industry)

                driver.get(url)
                job_cards_links = get_job_cards_links(driver)

                if len(job_cards_links) == 0:
                    continue

                # visiting each job link for the current \
                # industry and scraping information
                for job_cards_link in job_cards_links:
                    if job_cards_link in visited:
                        print('already visited skipping')
                        continue

                    visited.add(job_cards_link)

                    print("------visiting profile: ", job_cards_link, "------")
                    driver.get(job_cards_link)

                    business_details_dict = business_details_to_dict(driver,
                                                                     industry)


                    try:
                        print('looking for additional details')
                        additional_info = WebDriverWait(driver, 5).until(
                            EC.presence_of_element_located(
                                (
                                    By.CSS_SELECTOR,
                                    'div.panel-container \
                                        > div.panel-details'
                                )
                            )
                        )

                        business_details_dict['additional_info'] = \
```

```
                            additional_info.text

                    except (NoSuchElementException, TimeoutException):
                        print("No additional info container")
                        print("stopped at: ", business_details_dict)
                        business_details_dict['additional_info'] = ''

                    emails = set()

                    external_link = business_details_dict['external-links-container

                    if external_link != '':
                        await get_email(external_link, emails)
                        await get_email_from_contact(driver, external_link, emails)
                        print("emails have been updated these are emails", emails)

                    business_details_dict['emails'] = str(list(emails))

                    print(business_details_dict)
                    business_list.append(business_details_dict)
                    # there's a lot of waiting in between,\
                    # don't think we need a long wait
                    time.sleep(1)

            save_to_csv(business_list, where_param)

    driver.quit()
```

In [ ]:
```
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

Data Cleanup

Libraries

In [ ]:
```
import os
import re
import functools as ft
import pandas as pd
from selenium import webdriver
from selenium.webdriver.common.by import By
```

Keys

In [ ]:
```
state_dict = {
                'California': 'ca',
                'New Jersey': 'nj',
                'New York': 'ny',
                'Texas': 'tx'
            }
```

Implementation

Step 1) Using https://www.zipcodestogo.com gathering zip code of every city for the current state and converting it to a dataframe

Step 2) During scraping, each city, state pair was exported to a separate .csv file. Here, we are combining all the results for each state into a dataframe, dropping rows that don't have a business name and phone number and dropping rows that don't have an address

Step 3) From the dataframe created in step 2, extracting zip code from the values in the address column into a column call 'zip_code'

Step 4) Merging dataframe from step 1 and step 3 by zip_code

Step 5) Highlighting non-unique addresses and exporting the final result to a .xlxs file

In [ ]:
```
for state in state_dict.keys():
    ##############################
    #------getting zipcodes------#
    driver = webdriver.Chrome()
    driver.get(f'https://www.zipcodestogo.com/{state}/')

    table_rows = driver.find_elements(
        By.CSS_SELECTOR,
        'table.inner_table > tbody > tr'
    )

    zip_list = []
    temp_key = {0: 'zip_code', 1: 'city', 2: 'county'}

    for row in table_rows:
        cols = row.find_elements(By.CSS_SELECTOR, 'td')
        zip_dict = {'zip_code':'', 'city':'', 'county':''}

        for idx, col in enumerate(cols[0:3]):
            zip_dict[temp_key[idx]] = col.text

        zip_list.append(zip_dict)

    ny_zip_df = pd.DataFrame.from_dict(zip_list)
    ny_zip_df = ny_zip_df.iloc[2:, :]
    ny_zip_df = ny_zip_df.astype({'zip_code': str})
    ny_zip_df.dtypes

    driver.close()


    ####################################
    #------combining state results------#
    # path = './results'
    path = './' #path to where .csv file separated by city are
    os.listdir(path)
    csv_list = [
```

```
    file
    for file in os.listdir(path)
    if file.endswith(
        f"_{state_dict[state]}.csv"
    )
]
state_dfs = [
    pd.read_csv(path + '/' + csv_name)
    for csv_name in csv_list
    if os.stat(
        path + '/' + csv_name
    ).st_size > 2
]
state_df = pd.concat(state_dfs)
state_df.reset_index(drop=True ,inplace=True)
state_df = state_df[
    [
        'industry',
        'business name',
        'address',
        'external link',
        'phone number',
        'additional_info',
        'emails',
        'business hours'
    ]
]
state_df.dropna(
    subset=['business name', 'phone number'],
    how='all',
    inplace=True
)
state_df.dropna(subset=['address'], inplace=True)


##################################
#------extracting zip code------#
zip_code_pattern = re.compile("\d{5}(-\d{4})?$")

state_df['zip_code'] = state_df.apply(
    lambda row: str(re.search(zip_code_pattern, row['address']).group(0))
    if not pd.isna(row['address'])
    else row['address'], axis=1
)

# checking to see proper zip codes
state_df[~state_df['zip_code'].apply(
    lambda x: str(x).startswith('9') and len(str(x)) == 5
)]

state_df = state_df.astype({'zip_code': str})
state_df.dtypes
# checking to see all zipcodes are accounted for
state_df['zip_code'].isna().sum()
```

```
##################################
#------merging by zip code------#
merge_list = [state_df, ny_zip_df]
df_final = ft.reduce(lambda left, right: pd.merge(left, right, on='zip_code'),
                     merge_list)


##################################
#------finding duplicates------#
# finding duplicates
df_final['address'].value_counts()
rows = df_final.loc[df_final.duplicated(subset=['address'], keep=False)]
# index of duplicates
duplicates_mask = df_final['address'].duplicated(keep=False)
# Get indexes of non-unique values
non_unique_indexes = df_final[duplicates_mask].index.tolist()
repeats = df_final.iloc[non_unique_indexes]

def highlight_high_score(row):
    return [
                'background-color: yellow'
                if row.name in non_unique_indexes
                else ''
                for _ in row
            ]

# Highlighting non-unique addresses
df_final.sort_values(by='address', inplace=True)
styled_df = df_final.style.apply(highlight_high_score, axis=1)
styled_df.to_excel(
    f'{state_dict[state]}_final.xlsx',
    engine='openpyxl',
    index=False
)
```