Github repo link: https://github.com/gandhiz21/si_507_final

Data sources:

Yelp Fusion (Web API you haven't used before that requires API key, 4)

- Site origin: https://api.yelp.com/v3/businesses/search
  Requires an API key, which is in the file checkpoint.py in the github repo
- Format: data is first stored in a dictionary called searchname_dict within the python file, then cached into a JSON file, called yelp_dict.json in the github repo (CSV or JSON file you haven't used before with > 1000 records, 2)
  Currently has a dictionary with 22 keys/values and 1,039 search entries within all those values
- Data access: I accessed the data using the requests library and the API key
  I used requests functions as available on the Yelp Fusion Github, https://github.com/Yelp/yelp-fusion/blob/master/fusion/python/sample.py *(citation needed?)*
  I did use caching (details are in the caching section)
- Summary of data: A single searchterm of a city (ex: Ann Arbor, MI) can yield hundreds of business results. However, the API has a limit of 50 results per search. So I was able to retrieve a maximum of 50 results for each searchterm I used. Currently, I retrieved 1,039 results.
  Each result is a dictionary that consists of information available on the Yelp page for that search result. It includes a Yelp ID (which can be used for further searches), business name, business picture, number of reviews, address and phone number. Additional information such as reviews, hours of operation, and photos can be obtained separately using other request functions (ex: get_business, get_business_review). My data is stored in a dictionary, where each keyword (ex: "Exercise", "Park") has a value of a list of dictionaries of each search result.
- Caching: I used caching to store the results of my Yelp API search. The cache is stored in yelp_dict.json, and it has the structure {keyword1: [{search1_result}, {search2_result}, …], keyword2: …}. If there is no cache, it uses the API and builds a cache after retrieving the data. If there is a cache, but it was not made today, it deletes the previous cache and updates it with a new cache that accounts for daily updates on Yelp. If there is a cache, and it was made today, it takes the information from the cache and converts it into a usable dictionary searchname_dict.

```
# Saving the cache daily
from datetime import date
from pathlib import Path
path = Path(CACHE_FILENAME)
timestamp = date.fromtimestamp(path.stat().st_mtime) # records today's date

if os.path.exists(CACHE_FILENAME) == True and date.today() == timestamp: # if
    print("Cache was used.") # use cache
    print("Time to build graph with cache:", (t4 - t3) * 1000, "ms")
elif os.path.exists(CACHE_FILENAME) == True and date.today() != timestamp: #
    os.remove(path) # remove old cache
    print("Building a new cache...")
    print("Time to build graph with cache:", (t4 - t3) * 1000, "ms")
    for searchterm in searchterms:
        YELP_CACHE[searchterm] = searchname_dict[searchterm]
        save_cache(YELP_CACHE)
else: # if there's no cache at all
    print("Building cache...")
    print("Time to build graph without cache:", (t2 - t1) * 1000, "ms")
    for searchterm in searchterms:
        YELP_CACHE[searchterm] = searchname_dict[searchterm]
        save_cache(YELP_CACHE)
```

```
# Making the Graph
if os.path.exists(CACHE_FILENAME) == False: # if no cache
    t1 = datetime.datetime.now().timestamp()
    for searchterm in searchterms: # adding all the API results to th
        searchname = search(API_Key, searchterm, "Ann Arbor, MI")
        options = searchname["businesses"]
        print(len(options))

        searchname_dict[searchterm] = options # appending results to
```

```
else: # if cache exists
    t3 = datetime.datetime.now().timestamp()
    for searchterm in searchterms:
        searchname_dict[searchterm] = YELP_CACHE[searchterm] # building searchn
```

```
Building cache...
Time to build graph without cache: 14042.05298423767 ms
```

```
Cache was used.
Time to build graph with cache: 13.960838317871094 ms
```

TripAdvisor (Scraping a new single page, 4)

- I still have to do some work here. Basically, after users find a business/activity they are interested in, I will ask them if they want a "second opinion" from a different website, TripAdvisor. I will scrape TripAdvisor to see if they have that business, and print the rating and top reviews from that business. There will not be any caching because the information will be scraped on a case-by-case basis.

Data structure:

Graph (using the networkx graph class)

- Description: I have made one graph for this project. The graph has nodes for every searchterm/keyword (“Exercise”, “Nature”, etc.) and the name of every search result (“Gemstone Nails”, “Lillie Park”, etc.). The graph has edges between the search result and the keyword that was used to search it. Often, multiple keywords can yield the same search results (ex: “Lillie Park” is a result from searching “Nature” and “Park”), so there are fewer nodes than edges. Each keyword is connected to >50 results, and each result is connected to >=1 keyword. Also, because I can only retrieve the first 50 results, I acknowledge that there may be some businesses that should be connected to multiple keywords, but weren’t within the 50 that were retrieved, so that is a caveat. I also used the attributes in networkx to add other information to the search nodes. This will make it easier to get details like rating, url, address, etc. when a user asks for it.
- Description of graph assembly:
    1) Make a for loop that does an API call for a searchterm in a list of 22 searchterms
        a) Add a node for the searchterm/keyword (ex: “Park”)
        b) Add a node for the result using the result dict keyword ‘name’ (ex: “Lillie Park”)
        c) Add an edge between the searchterm and result
        d) Assign attributes (values) to the ‘name’ (keys) in attribute dictionaries
    2) Outside of the for loop, assign the dictionary attributes in 1d to the nodes using networkx
    3) Print the graph information (`print(nx.info(G))`)
    4) There are some additional loops I have for checks (see checkpoint.py)
        a) Print result node and node’s rating (from the attribute dictionary)
        b) For result nodes with edges to more than one keyword node, print those edges

```
# dictionaries of attributes that I want to be accessible within the graph
yelp_id = {} # id for further testing
image_url = {} # display picture
url = {} # url of yelp page
review_count = {} # number of reviews
rating = {} # rating
display_address = {} # full address
display_phone = {} # phone number
G = nx.Graph() # Initialize a Graph object
nodes = [] # set to empty list
```

```
# Assigning dictionary attributes to the nodes themselves
nx.set_node_attributes(G, yelp_id, "yelp_id")
nx.set_node_attributes(G, image_url, "image_url")
nx.set_node_attributes(G, url, "url")
nx.set_node_attributes(G, review_count, "review_count")
nx.set_node_attributes(G, rating, "rating")
nx.set_node_attributes(G, display_address, "display_address")
nx.set_node_attributes(G, display_phone, "display_phone")
```

```
G.add_node(searchterm) # adding "Exercise", "Nature", etc. as nodes in the Gra

for entry in searchname_dict[searchterm]:
    nodes.append(entry["name"])
G.add_nodes_from(nodes)
for node in nodes:
    G.add_edge(searchterm, node)

nodes = [] # resetting to empty list for the next searchterm

for entry in searchname_dict[searchterm]: # assigning attributes (values) to t
    yelp_id[entry["name"]] = entry['id']
    image_url[entry["name"]] = entry["image_url"]
    url[entry["name"]] = entry["url"]
    review_count[entry["name"]] = entry["review_count"]
    rating[entry["name"]] = entry["rating"]
    display_address[entry["name"]] = entry["location"]["display_address"][0]
    display_phone[entry["name"]] = entry["display_phone"]
```

```
print(nx.info(G)) # Print information about the Graph
Graph with 750 nodes and 1031 edges
```

Interaction and Presentation:

User-input capabilities

1) Prompt the user for the keywords they would like to use to narrow down the search. A list of 22 possible keywords will be provided. At least 1 keyword must be entered
    - In case there are no results that are connected to all the keyword(s) entered, they will be prompted for a new search
2) The program’s response will be the top 5 rated activities that are connected to the keyword(s)
3) Then, the user will be able to choose what other information they would like to see:

- Select 1 of the top 5 activities and get the remaining information (the contents of the attribute dictionary: yelp id, image url, url, # of reviews, rating, address, phone)
- Select 1 of the top 5 activities and get 3 reviews (this will require a new API call)
- Compare information between 2 activities
- Get ALL the activities that are connected to the keyword(s), assuming more than 5
- Get a second opinion: reviews from TripAdvisor
- Be redirected to a url
- View one of the presentation options below…
- Do a new search

Presentation Technologies

- I plan to use Plotly and/or Matplotlib to make these figures
    1) A scale (plotly.graph_objects.Scatter or matplotlib.pyplot.scatter) from 0 to 5, with points on the scale to represent the activities' ratings. Every node has a keyword "rating" in its attribute dictionary, so I plan to plot those values for each node. From there, they will be prompted to type the activity/node they are interested in, and they can then get more information.
    2) A table (plotly.graph_objects.Table or matplotlib.pyplot.table) that shows the activities' hours of operation. The result dictionary has a key called "is_closed" that indicates if the business is closed, so I plan to use the value True or False to show red (closed) or green (open). I can do an additional API call to get the hours of operation, and display them for each day of the week. Users can compare times/days across the top 5 rated activities.