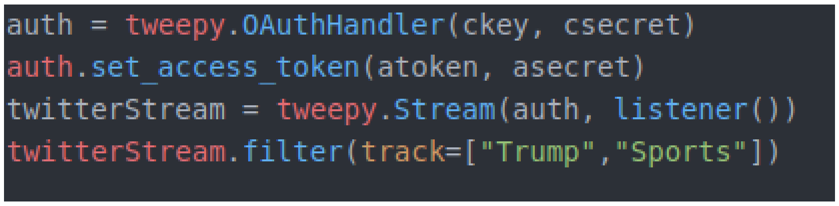
CS 432 Assignment #2

Larry Montgomery 01093132

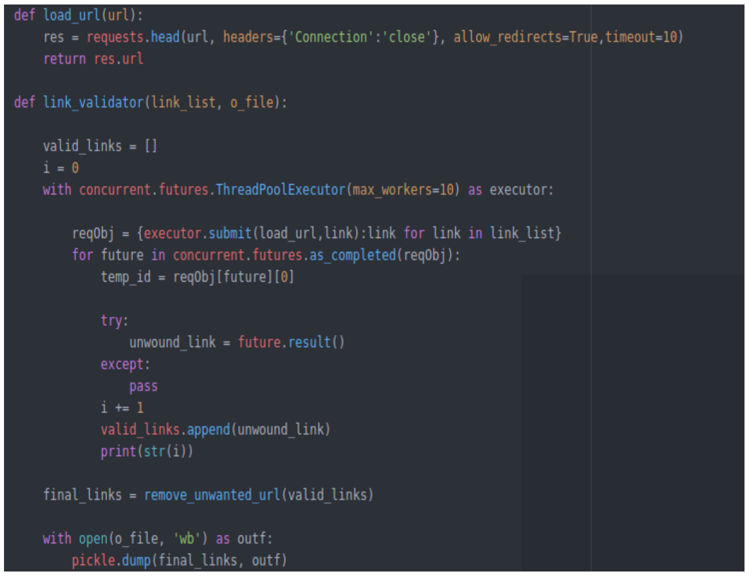
Part 1: Python Program for Extracting Twitter Links

To accomplish the task of collecting 1000 unique links from Twitter I made use of the built in os and pickle libraries and a secondary library, tweepy , that I downloaded with pip. The os module was used to fork the process for writing the twitter response objects to a file. I chose to collect 5000 responses from the Twitter API at a time and write them to a file to guard against the loss of data in the event of a program failure. I used the pickle library to write the data as a binary stream, that I could load again after collection was complete and process the Twitter API responses. I choose to use the Twitter Streaming API, and the tweepy library was very helpful in setting up a streaming object that would maintain a persistent connection with the Twitter API endpoint. To use this API Twitter required the use of authentication keys and OAuth service validation. Those keys were stored in the script file for easy access, but I would take make a more secure route if using them in production code. The code below shows the base setup of the stream object and the keywords I used to filter the Twitter responses I received.

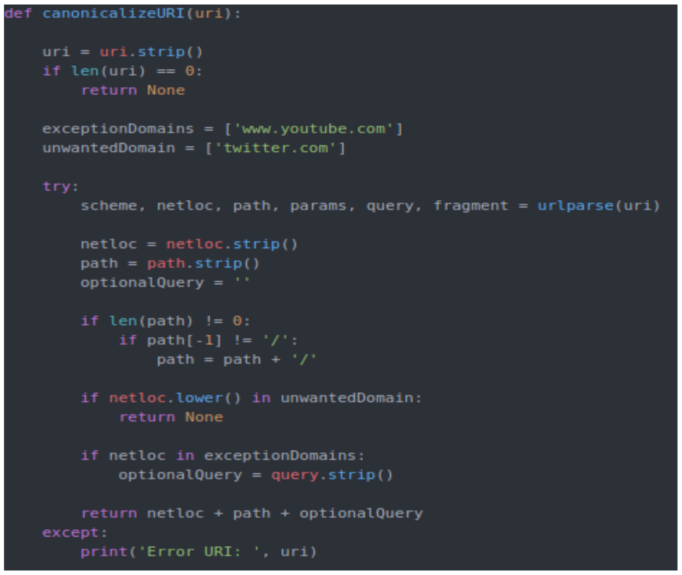


```python
auth = tweepy.OAuthHandler(ckey, csecret)
auth.set_access_token(atoken, asecret)
twitterStream = tweepy.Stream(auth, listener())
twitterStream.filter(track=["Trump","Sports"])
```

After the twitter responses were collected I had to unwind the links given in the Twitter Json response body. I used the requests library to follow redirects until a 200 was resolved and stored all these links in a list. Then I performed canonization of the links by using a slightly modified version of the code Professor Anwala provided to us. This provided me with over 1000 unique links, 1242 to be more precise. I did find that a large amount of the Twitter API responses were links to retweets and had to be discarded. This required me to collect data twice as I underestimated the amount of links I would be able to collect. Now I moved on to collecting the TimeMaps and Creation Dates, if they existed.



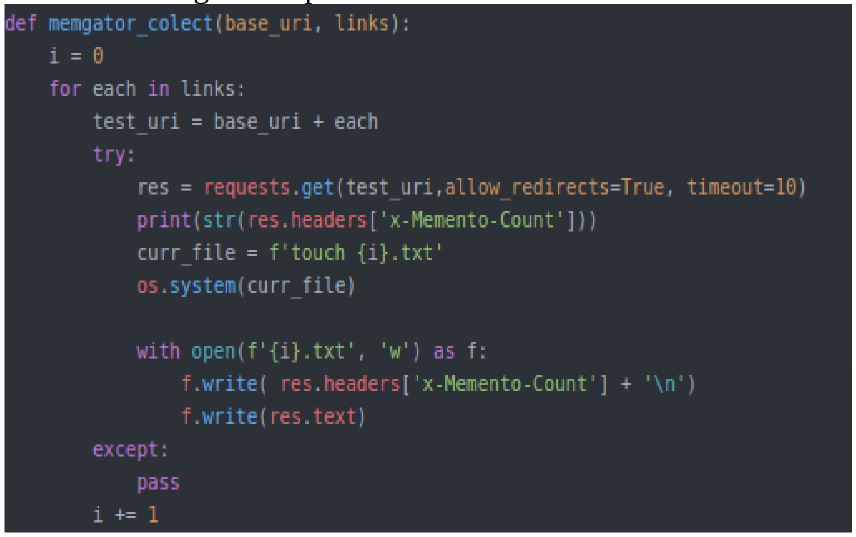
```python
def load_url(url):
    res = requests.head(url, headers={'Connection':'close'}, allow_redirects=True,timeout=10)
    return res.url

def link_validator(link_list, o_file):

    valid_links = []
    i = 0
    with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:

        reqObj = {executor.submit(load_url,link):link for link in link_list}
        for future in concurrent.futures.as_completed(reqObj):
            temp_id = reqObj[future][0]

            try:
                unwound_link = future.result()
            except:
                pass
            i += 1
            valid_links.append(unwound_link)
            print(str(i))

    final_links = remove_unwanted_url(valid_links)

    with open(o_file, 'wb') as outf:
        pickle.dump(final_links, outf)
```



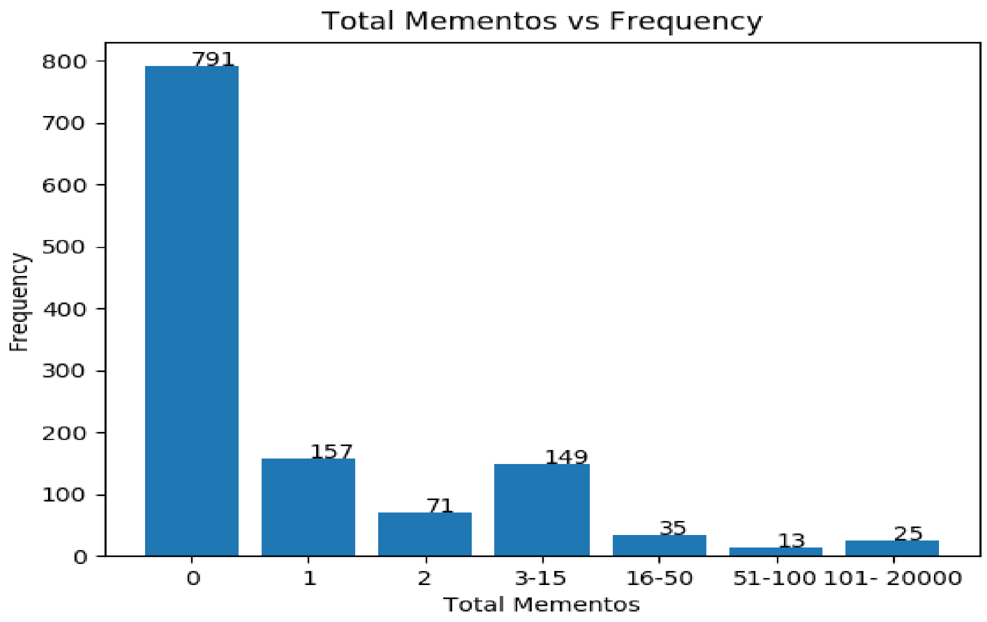
```python
def canonicalizeURI(uri):

    uri = uri.strip()
    if len(uri) == 0:
        return None

    exceptionDomains = ['www.youtube.com']
    unwantedDomain = ['twitter.com']

    try:
        scheme, netloc, path, params, query, fragment = urlparse(uri)

        netloc = netloc.strip()
        path = path.strip()
        optionalQuery = ''

        if len(path) != 0:
            if path[-1] != '/':
                path = path + '/'

        if netloc.lower() in unwantedDomain:
            return None

        if netloc in exceptionDomains:
            optionalQuery = query.strip()

        return netloc + path + optionalQuery
    except:
        print('Error URI: ', uri)
```

Part 2: TimeMap Collection using Memgator

I found Memgator to be a very intuitive server based application, even though I had some issues running the docker container. I ended up following the other avenues provided in the GitHub documentation. To be more specific I downloaded the binary and changed the execution privileges. This ran fine for me and I was able to collect the time maps on the list of URLs. I made use of the header ‘x-Memento-Count’ to resolve how many mementos were associated with any URL. I saved the header response in a text file along with the TimeMap json response. I used this later in the production of the histogram showing the distribution of mementos per URL. To generate the graph I used the Matplotlib library and its bar graph method. I had to sort the URLs into unique bins based on the number of mementos. I grouped the URLs with large memento counts into a few bins with very large ranges because there were far fewer of them and this made the graph illegible. The code below shows the script used to collect the memgator response from the server.

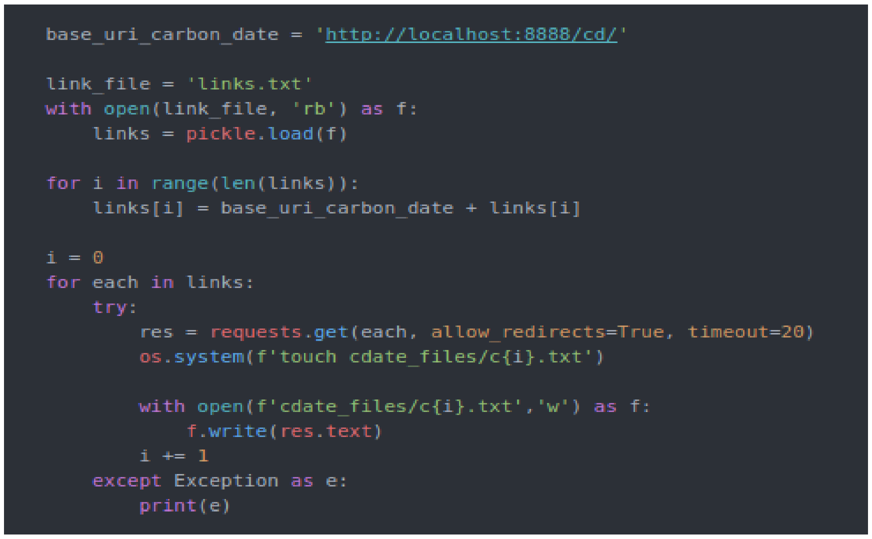


```python
def memgator_colect(base_uri, links):
    i = 0
    for each in links:
        test_uri = base_uri + each
        try:
            res = requests.get(test_uri,allow_redirects=True, timeout=10)
            print(str(res.headers['x-Memento-Count']))
            curr_file = f'touch {i}.txt'
            os.system(curr_file)

            with open(f'{i}.txt', 'w') as f:
                f.write( res.headers['x-Memento-Count'] + '\n')
                f.write(res.text)
        except:
            pass
        i += 1
```



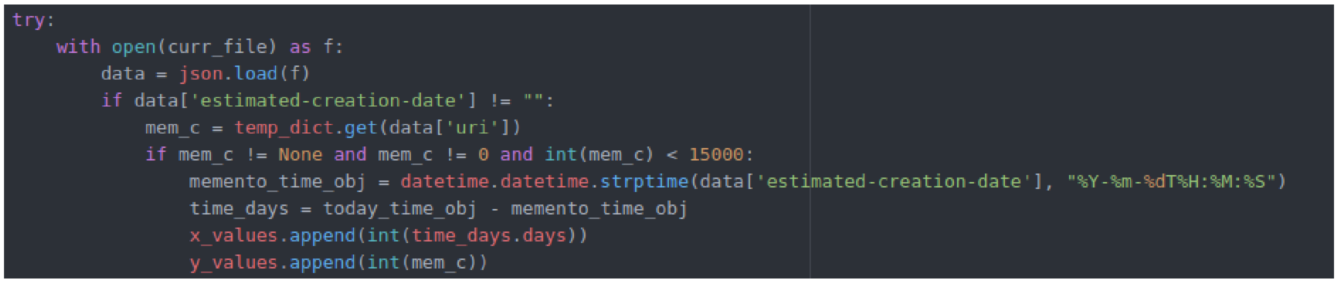
Part 3: Estimate Creation Time with Carbon Date Tool

The Carbon Date server application I had no issues in running in the Docker container as the GitHub instruction. I did run into a little problem when setting the path for the request to the server. I found after a looking through the web application that the path I was attempting to use lacked the ‘cd’ directory in the file path. I generated a file for each response just as I did with the memgator responses. The code below shows the script used to collect responses from the Carbon Date server.

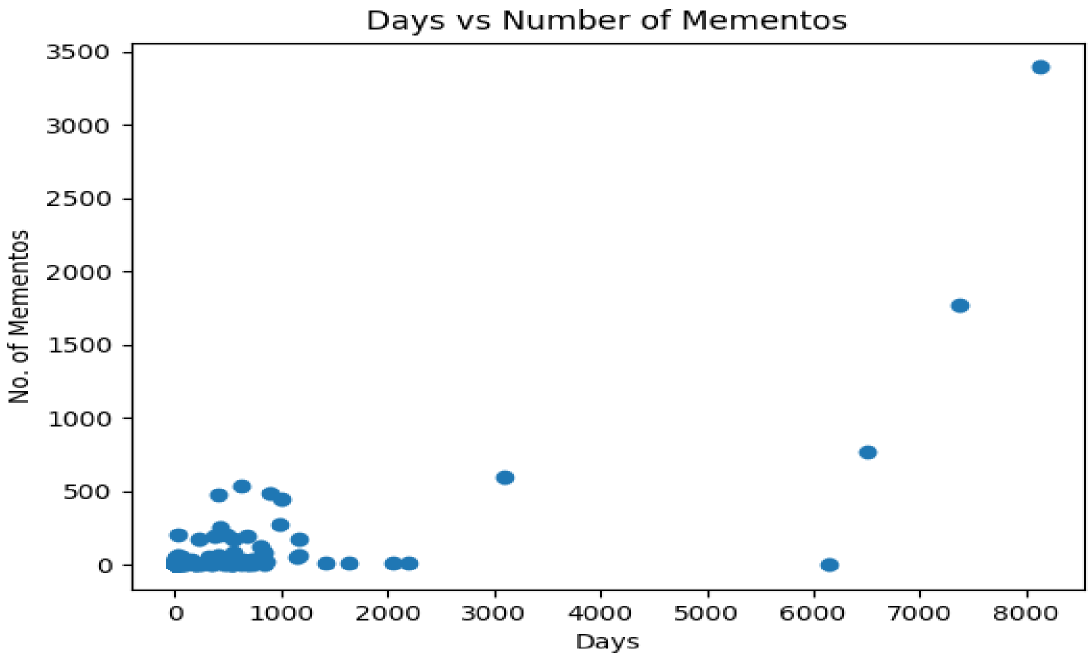


```python
base_uri_carbon_date = 'http://localhost:8888/cd/'

link_file = 'links.txt'
with open(link_file, 'rb') as f:
    links = pickle.load(f)

for i in range(len(links)):
    links[i] = base_uri_carbon_date + links[i]

i = 0
for each in links:
    try:
        res = requests.get(each, allow_redirects=True, timeout=20)
        os.system(f'touch cdate_files/c{i}.txt')

        with open(f'cdate_files/c{i}.txt','w') as f:
            f.write(res.text)
        i += 1
    except Exception as e:
        print(e)
```

I found that almost all of the URLs had a creation date. After opening a few files to check out how, I saw most of the files I sampled had a creation date that was retrieved from the web page itself and not from the archives. After all the Carbon Date responses were gathered, I loaded URLs that had a memento count greater than zero in a dictionary. I used this to lookup each URL in the Carbon Date responses, to make sure I would only graph the URLs that had both memento count greater than zero and a creation date. I made use of the Python datetime library to convert the creation time in the Carbon Date responses to a datetime object that I could use to find the number of days since creation. The datetime library also allowed me to get the current date, which was very helpful in generating the number of days since creation. The code below is the method to generate the days and memento counts for the graph.



```python
try:
    with open(curr_file) as f:
        data = json.load(f)
        if data['estimated-creation-date'] != "":
            mem_c = temp_dict.get(data['uri'])
            if mem_c != None and mem_c != 0 and int(mem_c) < 15000:
                memento_time_obj = datetime.datetime.strptime(data['estimated-creation-date'], "%Y-%m-%dT%H:%M:%S")
                time_days = today_time_obj - memento_time_obj
                x_values.append(int(time_days.days))
                y_values.append(int(mem_c))
```


Days vs Number of Mementos
No. of Mementos
Days
0
500
1000
1500
2000
2500
3000
3500
0
1000
2000
3000
4000
5000
6000
7000
8000