Exercise 1:

For this exercise we take as input parameters the URL of a website we want to collect our links from and we print out the relevant information regarding the input such as TLD, hostname, domain name as well as links found on the page.

Example:
`$python exercise1.py http://example.com/xyz`

Due to the fact that some pages may be generated dynamically, this exercise is implemented by using **Selenium** in conjunction with **BeautifulSoup4** to scrape the html page loaded. An initial approach with the requests library worked only for static pages.

Note: The current code is written to use the chromedriver for windows version 80. If you are using a different operative system or version of chrome another version of chromedriver needs to be used.

The basic components from the URL page are extracted via analyzing the string passed as argument.

```
s = self.URL.split('/')
# this part will collect the TLD, hostname, domain as well as current path.
for entry in s:
       if entry.lower() not in protocol:
       # we enter in this condition if we are visiting the array for the first time.
              if self.TLD == '':
                     url = entry.split('.')
                     self.TLD = url[-1]
                      self.hostname = entry
                      self.domain = '.'.join(url[1:]) if len(url) > 2 else entry
              else:
                      self.path += '/' + entry
```

Note that the string only consider
A 3-dimensional variable self.links will store the links found on the loaded webpage according to their classification (same hostname, same domain or external links).

We then proceed to load the pages using Selenium in headless mode.

```
   options = webdriver.ChromeOptions()
        options.add_argument('incognito')
        options.add_argument('headless')
        browser = webdriver.Chrome(executable_path='../chromedriver/chromedriver.exe',
options=options)
        browser.get(self.URL)
        time.sleep(SLEEP)
        soup = BeautifulSoup(browser.page_source, 'html.parser')
        browser.close()
```

After loading the webpage we check for all the anchor elements and collect href attributes. Some href values are of no interest for us and therefore we filter them out using the ignore_list variable.
The URLs collected may fall in 2 different categories:

1. fully qualified URLs or;
2. not fully qualified URLs.

If the URLs are fully qualified we remove fragmented identifiers or queries that may exist in the href element so we do not evaluate the URL incorrectly. For example the url:
http://www.externalurl.com/xqe?redirect=example.com/abc
Would return as being the same hostname if we checked the full URL while it was in fact an external link.
After analyzing the filtered URL we determine if it belongs to the same domain, hostname or it is an external link.
Another case is of href elements that start with double slash ('//') but do not contain a protocol. These hyperlinks refer to other links that inherit the same protocol as the current page. When we encounter these hyperlinks we treat them the same way as fully qualified URLs.
Finally for non-fully qualified URLs, we have two different circumstances. If they start with a slash '/' the page is redirecting to the top level domain, otherwise it is a child page.

After sorting out all the href attributes found in the page we clean out our data for duplicates and print the value as requested.

Example output:

```
TLD: com
DOMAIN: bbc.com
HOSTNAME: www.bbc.com
PATH: /news
LINKS:

    Same Hostname:
        https://www.bbc.com/news/news/england
        https://www.bbc.com/news/news/wales
        http://www.bbc.com/earth
        ...
        https://www.bbc.com/news/news/have_your_say
        https://www.bbc.com/news/news

    Same domain:
        http://pages.email.bbc.com/subscribe
        ...

    Different domain:
        https://twitter.com/BBC_HaveYourSay
        https://www.bbc.co.uk/news/entertainment-arts-51745702
        https://www.facebook.com/bbcnews
        https://www.bbc.co.uk/sounds
        ...
        https://www.bbc.co.uk/aboutthebbc
        https://www.bbc.co.uk/usingthebbc/cookies/
        http://www.bbc.co.uk/news/help-41670342
```

```
https://www.bbc.co.uk/contact
```

Exercise 2:

This exercise, split in 3 different parts, the objective is to read from a json file and save its contents to a csv file (part 1), generate statistics and save to a json file (part 2) and finally propose an approach when dealing with several files. The parts 1 and 2 are solved under the script exercise2.py whilst the part 3 is solved under the script exercise2.3.py.

Exercise 2 - part 1 and 2

In this part we read the contents of the provided json file to a **pandas** dataframe since pandas is faster when handling data than python native data structures.

```
df = pd.read_json('data.json', orient='records')
# The orientation needs to be changed so we transpose the DataFrame
df = df.T.reset_index()
df = df.rename(columns={'index': 'fullname'})
df.to_csv('data_csv.csv', index=False)
make_statistics(df, 'output')
```

Upon reading the json file we manipulate the dataframe by transposing it and resetting the index before we rename the first column to ‘fullname’ as requested in the exercise description. The file then is saved using the pandas `to_csv()` function without saving the index.

For the second part, we call the function make_statistics where we pass as input arguments the dataframe and the name of the file we want it to be saved as.
The requested exercise for this part was to generate statistics based on the last name of all the persons on the json file. This is achieved by first filtering out all the last names in the dataframe:

```
last_names = data.fullname.map(lambda x: x.split(' '), na_action='ignore').str[1].unique()
```

Afterwards, for each last_name found we create the statistics via dictionary comprehension on the dataframe before we finally create an entry to be added to the output file.

```
filtered_df = data[data.fullname.str.contains(last_name)]
count = filtered_df.fullname.count()
ages = {str(x): filtered_df[filtered_df.age == x].age.count() for x in
filtered_df.age.unique()}
addresses = {x: filtered_df[filtered_df.address == x].address.count() for x in
filtered_df.address.unique()}
occupations = {x: filtered_df[filtered_df.occupation == x].occupation.count() for x in
filtered_df.occupation.unique()}
d[last_name] = {'count': count, 'age': ages, 'address': addresses, 'occupation': occupations}
```

Finally, we save the dataframe with the statistics to the json file using pretty print (indent value set) as it is shown on the exercise description.

Exercise 2 - part 3

In this part, my proposal is that to decrease the runtime of the code for N scripts, we should first collect all the data to a single dataframe **considering we have memory capacity**. When dealing with files, much of the processing time is spent in opening and closing these therefore if all the data can be collected to a single dataframe before being processed this method is preferred. Reading the whole data to a single dataframe has also the benefit that we can filter out undesired and repeated information that may exist across different files.

The example script runs two different simulations:

1. Open N files to a single dataframe and perform the data analysis tasks
2. Open each file individually and save to an individual file.

The additional data was generated using the Faker library. The extra datasets are stored under the faker directory of the package provided. In addition, we create another directory inside of the faker directory to store the statistics files from each test.

Below is the implementation for each test.

Read each file one at a time and save the results to a statistics file.

```
def read_single():
    """
    Read each file to a DataFrame using pd.read_json() process each file individually
    :return: None
    """
    df = pd.DataFrame()
    for n in range(n_files):
        part_df = pd.read_json(f'{SAVE_DIR}data{n+1}.json', orient='records')
        part_df = part_df.T.reset_index()
        part_df = part_df.rename(columns={'index': 'fullname'})
        df = df.append(part_df)
        exercise2.make_statistics(df,
f'{SAVE_DIR}{STATISTICS_DIR}single_keep_duplicates_output{n}')
```

Read multiple files and process them all at once.

```
def read_multi():
    """
    Open all the files using json.load function instead of calling pandas pd.read_json
    Gather all data to a single DataFrame and save the statistics to only one json file.
    :return: None
    """

    df = pd.DataFrame()
    for n in range(n_files):
        with open(f'{SAVE_DIR}data{n+1}.json') as f:
            json_dict = json.load(f)
            part_df = pd.DataFrame(data=json_dict).T.reset_index()
            part_df = part_df.rename(columns={'index': 'fullname'})
            df = df.append(part_df)
    # to reduce the runtime for the make_statistics function we can drop the duplicates
    # if they may occur across different files
```

```
    if REMOVE_DUPLICATES:
        df.drop_duplicates(inplace=True, keep=False)

    exercise2.make_statistics(df, f'{SAVE_DIR}{STATISTICS_DIR}multi_file_output')
```

The resulting average runtime for each test is (using timeit for 5 runs):

```
Average runtime for combining multiple files to one DataFrame: 1.2599293 sec.
Average runtime for executing statistics on each file individually: 15.430750960000001 sec.
```

As we can see, processing all the data frames at once is much quicker than doing one at a time.

However, as mentioned above this scenario is considering that we do not have memory limitations. If the datasets are too large to be stored in a single dataframe the preferred approach would be accessing each file individually and then appending the results to the statistics file.

Exercise 3

List of websites - “gambling.txt”

In this exercise the objective is collect information from different websites and perform content classification. To achieve this, we first collect a list of gambling websites where all of the content is in english language. The list of websites “gambling.txt” can be found under the exercise3 directory.
This list was compiled using websites from different countries in order to avoid skewing the wording and idiomatic nuances. A total of 150 different webpages were added to the text file.

Exercise3.1 script and dictionary creation.

The dictionary was created by visiting the websites on the list created. In order to achieve this we collect use the libraries Selenium in order to automate the visits since some websites are dynamically generated and the requests library would not produce good results. In order to collect the words we use the library BeautifulSoup.

Note: The current code is written to use the chromedriver for windows version 80. If you are using a different operative system or version of chrome another version of chromedriver needs to be used.

As mentioned above, this script uses the chromewebdriver to visit the pages in headless mode. Once we load the pages we scrape the data with BeautifulSoup and then analyze relevant elements where text is contained.

Setting up Chrome webdriver.

```
 # setup selenium using Chrome driver
    options = webdriver.ChromeOptions()
    options.add_argument('incognito')
```

```
    options.add_argument('headless')
    # location of chrome driver using chrome on windows v.80
    browser = webdriver.Chrome(executable_path='../chromedriver/chromedriver.exe',
options=options)
```

Collecting the data with bs4:

```
    for i, website in enumerate(websites):
        if SHOW:
            print(f'website {i+1} of {len(websites)}: {website}')

        # using Selenium to call Chrome and parse the text using bs4
        browser.get(website)
        soup = BeautifulSoup(browser.page_source, 'html.parser')
        text = soup.find_all(text=True)
        # put all the words found on the website to a string
        output = ''
        for t in text:
            # work with a whitelist of elements to read
            if t.parent.name in whitelist:
                t = t.strip('\t\r\n')
                if t != '':
                    output += t+' '
        # Convert the collected data to lowercase and split on spaces
        output = output.lower().split()
        for word in output:
            texts.append(word)
```

For each page we load we analyze elements only present on a whitelist. We then extract the list of words to the variable text and perform an initial cleanup to remove extra spacing as well as new line and tabulations. We also convert all the words to lower case to correct possible typos and avoid having the same words mismatched, e.g. (Day, DAy, day are all the same word). The text is then added to a list called texts.

Once we collect the text data of all the pages we run the `function tokenize_and_stem()` which will allow us to further clean our data and obtain relevant information from each page visited. The implementation of the function is as follows:

```
def tokenize_and_stem(text_list):
    """
    :param text_list: list of input texts from the webpages
    :return: stemmed and tokenized word list
    """
    stop_words = set(stopwords.words('english'))
    words = []
    for e in text_list:
        words += word_tokenize(e)
    # remove punctuations, stopwords, irrelevant terms (see above exclude_list) and numbers
    words = [w for w in words if w.isalpha()]
    words = [w for w in words if w not in stop_words]
    words = [w for w in words if w not in exclude_list]
    words = [w for w in words if not w.isdigit()]
    # stem the words using the Snowball stemmer
    stemmer = SnowballStemmer('english')
    stemmed = [stemmer.stem(w) for w in words]
```

```
    return stemmed
```

The function will first remove characters that are not alphabetic. This will allow us to remove punctuations however at a cost of removing words that are hyphenated. Further we remove the stop words which are words that are not relevant in any context, be them on a gambling site or not. Words such as “I”, “the”, “a” are all considered stop words. We also remove words that are also commonly found on websites such as cookies, http, or even tags (for example angular tags) that will escape the filtering process.
Finally we remove the numbers since they do not add any relevant information. The function will then return a list of stemmed and tokenized words that we can use to create our dictionary.

The most frequent entries are found using the nltk library function `nltk.FreqDist` and finally we store the 1000 most common tokenized and stemmed entries from all the websites visited.

The values are written to a csv file called “dictionary.csv”

```
    # write to a csv file
    csv = 'dictionary.csv'
    with open(csv, 'w') as csv_file:
        csv_file.write('entry,occurrences\n')
        for key, value in features:
            csv_file.write(f'{key},{value}\n')
```

Exercise3.2 script and classification

For this exercise we use the dictionary created in 3.1. And classify any given website based on the dictionary we created.

The implementation of this exercise follows the same pattern of the exercise 3.1 to collect the relevant information from the provided website. We also create a list of tokens and stems for the website visited.

There are a number of classification methods available, however most rely on comparing two or more different labels. When using two different classes we can create machine algorithms such as Naive Bayes or Logistic regression to classify the websites we can use as the training data information that we know, belongs to either being a gambling website or not. However, since we do not have this information the limitations of this classification method are increased.

In order to classify the websites in this task we search in the list of the words if the word stem is found on the dictionary. If so we increase the score of confidence. If the score hits a threshold of 2/3s of all the words found on the list or more we classify the website as being a gambling site, otherwise we do not consider it a gambling website.

```
    # scoring of entries.
    score = 0
    total_score = len(aggregate.keys())
    for word in aggregate.keys():
        if word in dictionary.entry.tolist():
            score += 1
    # if more than 2/3 of the words are found returns as a Gambling website
    x = 'Gambling Site' if score/total_score > 0.66 else 'Not Gambling site'
```

This method while finding some success, has limitations. Possible improvements would include collecting data from websites we know that are not gambling and create a list of non-gambling terms, training the data and fitting to:

1. a logistic classifier;
2. A naive bayes implementation;
3. A support vector machine;
4. Using neural networks etc.

Finally, since not all the methods are 100% foolproof, we can use a voting based system to classify our results before finally providing an output. The voting system would allow us to decrease the number of false-positives since a result that may be given as positive by one method may not correspond to the truth.