STA326 Assignment 1: Data Collection

This is an assignment that is openly available for the Data Science Practice (STA326).

The assignment encapsulates a holistic approach towards data collection and analysis, covering a spectrum of data formats and sources. Our objective is to amass, process, and scrutinize data to unearth significant insights. The methodology is sectioned into four pivotal tasks:

- Web scraping
- JSON file analysis
- Working with CSV files
- Data Cleaning

In [1]:
```
# Imports
import requests  # send request
from bs4 import BeautifulSoup  # parse web pages
import pandas as pd  # csv
from time import sleep  # wait
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import json
```

Part 1: Web Scraping

In this assignment, we will explore web scraping, which can often include diverse information from website, and also use the data for simple analysis. We take douban as the target website in this assignment.

Scraping Rules

1. If you are using another organization's website for scraping, make sure to check the website's terms & conditions.

2. Do not request data from the website too aggressively (quickly) with your program (also known as spamming), as this may break the website. Make sure your program behaves in a reasonable manner (i.e. acts like a human). One request for one webpage per second is good practice.

3. The layout of a website may change from time to time. Because of this, if you're scraping a website, make sure to revisit the site and rewrite your code as needed.

1a) Web Scrape

In order to extract the data we want, we'll start with extracting the whole web.

In [2]:
```
# Define a request header (to prevent anti-scraping)
headers = {
    'authority': 'curlconverter.com',
    'accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,app
    'accept-language': 'zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6',
    'cache-control': 'max-age=0',
    'if-modified-since': 'Fri, 15 Jul 2022 21:44:42 GMT',
    'if-none-match': 'W/"62d1dfca-3a58"',
    'referer': 'https://curlconverter.com/',
    'sec-ch-ua': '" Not A;Brand";v="99", "Chromium";v="102", "Microsoft Edge";v="102"',
    'sec-ch-ua-mobile': '?0',
    'sec-ch-ua-platform': '"Linux"',
    'sec-fetch-dest': 'document',
    'sec-fetch-mode': 'navigate',
    'sec-fetch-site': 'cross-site',
    'sec-fetch-user': '?1',
    'upgrade-insecure-requests': '1',
```

```
    'user-agent': 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/102.0.5
}
```

This process can be split into three steps:

1. Make a variable called `url`, that stores the following URL (as a string): https://movie.douban.com/top250?start=0
2. Now, to open the URL, use `requests.get()` and provide `url` and `headers` as its input. Store this in a variable called `page`.
3. After that, make a variable called `soup` to parse the HTML using `BeautifulSoup`. Consider that there will be a method from `BeautifulSoup` that you'll need to call on to get the content from the page.

In [3]:
```
# YOUR CODE HERE
# Step 1: Define the URL
url = "https://movie.douban.com/top250?start=0"

# Step 2: Open the URL using requests.get() with headers
page = requests.get(url, headers=headers)

# Step 3: Parse the HTML content using BeautifulSoup
soup = BeautifulSoup(page.content, 'lxml')
```

In [4]:
```
assert url
assert page
assert soup
```

1b) Data Extraction

Extract the data (`name` and `star`) from the page and save it in the corresponding list like `movie_name` and `movie_star`.

Make sure you extract it as a string.

To do so, you have to use the soup object created in the above cell.

Hint: from your soup variable, you can access this with `soup.select()`

In [5]:
```
movie_name = []
movie_star = []

for i in range(0, 5):
    url = "https://movie.douban.com/top250?start=" + str(i * 25)
    page = requests.get(url, headers=headers)
    soup = BeautifulSoup(page.content, 'lxml')
    for li in soup.ol.find_all('li'):
        movie_name.append(li.a.img['alt'])
    for li in soup.ol.find_all('span', class_='rating_num'):
        movie_star.append(li.text)
```

1c) Collecting into a dataframe

Create a dataframe `movie_df` and add the data from the lists above to it.

- `movie_name` is the movie name. Set the column name as `movie name`
- `movie_star` is the population estimate via star. Add it to the dataframe, and set the column name as `movie star`

Make sure to check the head of your dataframe to see that everything looks right! ie: movie_df.head()

Finally, you should save the DataFrame to a csv file under this folder `'./output'`.

In [6]:
```
csv_name = "MovieDouban.csv"
csv_dir = "./output"
file_path = f"{csv_dir}/{csv_name}"

movie_df = pd.DataFrame({
```

```
    'movie name': movie_name,
    'movie star': movie_star
})
# Checking the head of the dataframe
print(movie_df.head())

# Save DataFrame to CSV
movie_df.to_csv(file_path, index=False)

print(f"DataFrame saved to {file_path}")
```

```
  movie name movie star
0     肖申克的救赎        9.7
1       霸王别姬        9.6
2       阿甘正传        9.5
3      泰坦尼克号        9.5
4       千与千寻        9.4
DataFrame saved to ./output/MovieDouban.csv
```

Part 2: JSON File Analysis

After the initial phase of web scraping, we transition to analyzing pre-collected data, which is often stored in accessible and structured formats like JSON and CSV. This approach allows us to bypass the time-consuming process of data collection through web scraping for certain datasets that are already available, enabling us to dive directly into data analysis.

Overview

In the section, you will first be working with a file called 'anon_user_dat.json'. You can find the given data under the folder `'./data/data_identifying'`. This file contains information about some (fake) Tinder users. When creating an account, each Tinder user was asked to provide their first name, last name, work email (to verify the disclosed workplace), age, gender, phone # and zip code. Before releasing this data, a data scientist cleaned the data to protect the privacy of Tinder's users by removing the obvious personal identifiers: phone #, zip code, and IP address. However, the data scientist chose to keep each users' email addresses because when they visually skimmed a couple of the email addresses none of them seemed to have any of the users' actual names in them. This is where the data scientist made a huge mistake!

Data Files:

- anon_user_dat.json
- employee_info.json

We will take advantage of having the work email addresses by finding the employee information of different companies and matching that employee information with the information we have, in order to identify the names of the secret Tinder users!

2a) Load data from JSON file

Load the `anon_user_dat.json` json file into a pandas dataframe. Call it `df_personal`.

In [7]:
```
df_personal = pd.read_json("./data/data_identifying/anon_user_dat.json")
df_personal
```

Out[7]:

	age	email	gender
0	60	gshoreson0@seattletimes.com	Male
1	47	eweaben1@salon.com	Female
2	27	akillerby2@gravatar.com	Male
3	46	gsainz3@zdnet.com	Male
4	72	bdanilewicz4@4shared.com	Male
...	...	...	...
995	3	pstroulgerrn@time.com	Female
996	49	kbasnettro@seattletimes.com	Female
997	75	pmortlockrp@liveinternet.ru	Male
998	81	sphetterq@toplist.cz	Male
999	70	jtyresrr@slashdot.org	Male

1000 rows × 3 columns

In [8]:
```
assert isinstance(df_personal, pd.DataFrame)
```

2b) Check the first 10 emails

Save the first 10 emails to a Series, and call it `sample_emails`. You should then print out this Series. (Use print())

The purpose of this is to get a sense of how these work emails are structured and how we could possibly extract where each anonymous user seems to work.

In [9]:
```
# Extract the first 10 emails
sample_emails = df_personal['email'][:10]
# Print out the Series
print(sample_emails)
```

```
0    gshoreson0@seattletimes.com
1             eweaben1@salon.com
2        akillerby2@gravatar.com
3              gsainz3@zdnet.com
4       bdanilewicz4@4shared.com
5      sdeerness5@wikispaces.com
6          jstillwell6@ustream.tv
7          mpriestland7@opera.com
8       nerickssen8@hatena.ne.jp
9             hparsell9@xing.com
Name: email, dtype: object
```

In [10]:
```
assert isinstance(sample_emails, pd.Series)
```

2c) Extract the Company Name From the Email

Create a function with the following specifications:

- Function Name: extract_company
- Purpose: to extract the company of the email (i.e., everything after the @ sign but before the first .)
- Parameter(s): email (string)
- Returns: The extracted part of the email (string)
- Hint: This should take 1 line of code. Look into the find('') method.

You can start with this outline:

```
def extract_company(email):
    return
```

Example Usage:

- extract_company("larhe@uber.com") should return "uber"

- extract_company("ds@cogs.edu") should return "cogs"

In [11]:
```
def extract_company(email):
    return email.split('@')[1].split('.')[0]


# 测试函数
print(extract_company("larhe@uber.com"))  # 输出: "uber"
print(extract_company("ds@cogs.edu"))  # 输出: "cogs"
```

```
uber
cogs
```

In [12]:
```
assert extract_company("gshoreson0@seattletimes.com") == "seattletimes"
assert extract_company("amcgeffen1d@goo.ne.jp") == 'goo'
```

With a little bit of basic sleuthing (aka googling) and web-scraping (aka selectively reading in html code) it turns out that you've been able to collect information about all the present employees/interns of the companies you are interested in. Specifically, on each company website, you have found the name, gender, and age of its employees. You have saved that info in `employee_info.json` and plan to see if, using this new information, you can match the Tinder accounts to actual names.

2d) Load in employee data

Load the json file into a pandas dataframe. Call it `df_employee`.

In [13]:
```
df_employee = pd.read_json("./data/data_identifying/employee_info.json")
df_employee
```

Out[13]:

	company	first_name	last_name	gender	age
0	123-reg	Inglebert	Falconer	Male	42
1	163	Rafael	Bedenham	Male	14
2	163	Lemuel	Lind	Male	31
3	163	Penny	Pennone	Female	45
4	163	Elva	Crighton	Female	52
...	...	...	...	...	...
995	zdnet	Guido	Comfort	Male	46
996	zdnet	Biron	Malkinson	Male	48
997	zimbio	Becka	Waryk	Female	27
998	zimbio	Andreana	Ladewig	Female	34
999	zimbio	Jobyna	Busek	Female	75

1000 rows × 5 columns

In [14]:
```
assert isinstance(df_employee, pd.DataFrame)
```

2e) Match the employee name with company, age, gender

Create a function with the following specifications:

- Function name: employee_matcher
- Purpose: to match the employee name with the provided company, age, and gender
- Parameter(s): company (string), age (int), gender (string)
- Returns: The employee first_name and last_name like this: return first_name, last_name
- Note: If there are multiple employees that fit the same description, first_name and last_name should return a list of all possible first names and last names i.e., ['Desmund', 'Kelby'], ['Shepley', 'Tichner']. Note that the names of the individuals that would produce this output are 'Desmund Shepley' and 'Kelby Tichner'.

Hint: There are many different ways to code this. An inelegant solution is to loop through `df_employee` and for each data item see if the company, age, and gender match i.e.,

```
for i in range(0, len(df_employee)):
    if (company == df_employee.loc[i,'company']):
        print(1)
```

However! The solution above is very inefficient and long, so you should try to look into this: Google the df.loc method: It extracts pieces of the dataframe if it fulfills a certain condition. i.e.,

```
df_employee.loc[df_employee['company'] == company]
```

If you need to convert your pandas data series into a list, you can do `list(result)` where result is a pandas "series"

You can start with this outline:

```
def employee_matcher(company, age, gender):
    return first_name, last_name
```

In [15]:
```
def employee_matcher(company, age, gender):
    matched_employees = df_employee.loc[(df_employee['company'] == company) &
                                        (df_employee['age'] == age) &
                                        (df_employee['gender'] == gender)]

    first_names = list(matched_employees['first_name'])
    last_names = list(matched_employees['last_name'])

    return first_names, last_names
```

In [16]:
```
assert employee_matcher("google", 41, "Male") == (['Maxwell'], ['Jorio'])
assert employee_matcher("salon", 47, "Female") == (['Elenore'], ['Gravett'])
assert employee_matcher("webmd", 28, "Nonbinary") == (['Zaccaria'], ['Bartosiak'])
```

2f) Extract all the private data

- Create 2 empty lists called `first_names` and `last_names`
- Loop through all the people we are trying to identify in df_personal
- Call the `extract_company` function (i.e., `extract_company(df_personal.loc[i, 'email'])`)
- Call the `employee_matcher` function
- Append the results of `employee_matcher` to the appropriate lists (`first_names` and `last_names`)

In [17]:
```
# Create empty lists
first_names = []
last_names = []

# Loop through the people in df_personal
for i in range(len(df_personal)):
    # Extract company from email using extract_company function
    company = extract_company(df_personal.loc[i, 'email'])
    # Call employee_matcher function to match employee
    matched_first_names, matched_last_names = employee_matcher(company, df_personal.loc[i, 'age'],
                                                               df_personal.loc[i, 'gender'])

    # Append results to first_names and last_names lists
    first_names.append(matched_first_names)
    last_names.append(matched_last_names)

# Print the results
# first_names, last_names
```

In [18]:
```
assert first_names[45:50] == [['Justino'], ['Tadio'], ['Kennith'], ['Cedric'], ['Amargo']]
assert last_names[45:50] == [['Corro'], ['Blackford'], ['Milton'], ['Yggo'], ['Grigor']]
```

2g) Add the names to the original 'secure' dataset!

We have done this last step for you below, all you need to do is run this cell.

For your own personal enjoyment, you should also print out the new `df_personal` with the identified people.

In [19]:
```
df_personal['first_name'] = first_names
df_personal['last_name'] = last_names
df_personal
```

Out[19]:

	age	email	gender	first_name	last_name
0	60	gshoreson0@seattletimes.com	Male	[Gordon]	[DelaField]
1	47	eweaben1@salon.com	Female	[Elenore]	[Gravett]
2	27	akillerby2@gravatar.com	Male	[Abbe]	[Stockdale]
3	46	gsainz3@zdnet.com	Male	[Guido]	[Comfort]
4	72	bdanilewicz4@4shared.com	Male	[Brody]	[Pinckard]
...	...	...	...	...	...
995	3	pstroulgerrn@time.com	Female	[Penelopa]	[Roman]
996	49	kbasnettro@seattletimes.com	Female	[Anthiathia, Kandy]	[Baldwin, Cossam]
997	75	pmortlockrp@liveinternet.ru	Male	[Paco]	[Weatherburn]
998	81	sphetterq@toplist.cz	Male	[Sammy]	[Dymick]
999	70	jtyresrr@slashdot.org	Male	[Josiah]	[Ayshford]

1000 rows × 5 columns

Part 3: Working with CSV Files

Continuing with our exploration of pre-collected data formats, we delve into CSV files, which are renowned for their simplicity and widespread use in representing tabular data. This stage involves leveraging libraries like pandas in Python, which simplify the process of reading, manipulating, and analyzing CSV data.

Overview

For this assignment, you are provided with two data files that contain information on a sample of people. The two files and their columns are:

- `age_steps.csv` : Contains one row for each person.
  - `id` : Unique identifier for the person.
  - `age` : Age of the person.
  - `steps` : Number of steps the person took on average in January 2018.
- `incomes.json` : Contains one record for each person.
  - `id` : Unique identifier for the person. Two records with the same ID between `age_steps.csv` and `incomes.json` correspond to the same person.
  - `last_name` : Last name of the person.
  - `first_name` : First name of the person.
  - `income` : Income of the person in 2018.

You can find the given data under the folder `'./data/data_wrangling'` . To finish the assignment, we recommend looking at the official 10 minutes to pandas guide: http://pandas.pydata.org/pandas-docs/stable/10min.html

Question 3a: Load the `age_steps.csv` file into a `pandas` DataFrame named `df_steps` . It should have 11257 rows and 3 columns.

In [20]:
```
df_steps = pd.read_csv('data/data_wrangling/age_steps.csv')
df_steps
```

Out[20]:

	id	age	steps
0	18875	31	9159
1	36859	48	6764
2	99794	39	4308
3	33364	36	6410
4	73874	35	7870
...	...	...	...
11252	42474	28	7307
11253	61626	44	7752
11254	52336	41	-1
11255	54972	44	7548
11256	17411	43	8765

11257 rows × 3 columns

In [21]:
```
assert isinstance(df_steps, pd.DataFrame)
assert df_steps.shape == (11257, 3)
```

Question 3b: Load the `incomes.json` file into a `pandas` DataFrame called `df_income`. The DataFrame should have 13332 rows and 4 columns.

Hint: Find a pandas function similar to `read_csv` for JSON files.

In [22]:
```
df_income = pd.read_json('data/data_wrangling/incomes.json')
df_income
```

Out[22]:

	id	last_name	first_name	income
0	84764	Wolfe	Brian	99807.16
1	49337	Keith	George	0.00
2	54204	Wilcox	Zachary	5242.96
3	41693	Glass	Catherine	0.00
4	98170	Perez	Bob	18077.78
...	...	...	...	...
13327	43140	Gonzalez	John	81081.56
13328	21142	Green	James	3826.20
13329	68473	Meyer	Ian	7617.27
13330	60486	Russell	Carl	34479.99
13331	13915	Johnson	Curtis	12133.79

13332 rows × 4 columns

In [23]:
```
assert isinstance(df_income, pd.DataFrame)
assert df_income.shape == (13332, 4)
```

Question 3c: Drop the `first_name` and `last_name` columns from the `df_income` DataFrame. The resulting DataFrame should only have two columns.

In [24]:
```
df_income = df_income.drop(['first_name', 'last_name'], axis=1)
df_income
```

Out[24]:

	id	income
0	84764	99807.16
1	49337	0.00
2	54204	5242.96
3	41693	0.00
4	98170	18077.78
...	...	...
13327	43140	81081.56
13328	21142	3826.20
13329	68473	7617.27
13330	60486	34479.99
13331	13915	12133.79

13332 rows × 2 columns

In [25]:

```
assert 'first_name' not in df_income.columns
assert 'last_name' not in df_income.columns
```

Question 3d: Merge the `df_steps` and `df_income` DataFrames into a single combined DataFrame called `df`. Use the `id` column to match rows together.

The final DataFrame should have 10,135 rows and 4 columns: `id`, `income`, `age`, and `steps`.

Call an appropriate `pandas` method to perform this operation; don't write a `for` loop. (In general, writing a `for` loop for a DataFrame will produce poor results.)

In [26]:

```
df = df_steps.merge(df_income, on='id')
df
```

Out[26]:

	id	age	steps	income
0	36859	48	6764	10056.43
1	99794	39	4308	13869.47
2	33364	36	6410	79634.92
3	73874	35	7870	12369.03
4	66956	56	7670	41150.18
...	...	...	...	...
10130	42474	28	7307	49128.60
10131	61626	44	7752	20096.38
10132	52336	41	-1	0.00
10133	54972	44	7548	18350.20
10134	17411	43	8765	88965.55

10135 rows × 4 columns

In [27]:

```
assert isinstance(df, pd.DataFrame)
assert set(df.columns) == set(['id', 'income', 'age', 'steps'])
assert df.shape == (10135, 4)
```

Question 3e: Reorder the columns of `df` so that they appear in the order: `id`, `age`, `steps`, then `income`.

(Note: If your DataFrame is already in this order, just put `df` in this cell.)

In [28]:
```
df = df[['id', 'age', 'steps', 'income']]
df
```

Out[28]:

	id	age	steps	income
0	36859	48	6764	10056.43
1	99794	39	4308	13869.47
2	33364	36	6410	79634.92
3	73874	35	7870	12369.03
4	66956	56	7670	41150.18
...	...	...	...	...
10130	42474	28	7307	49128.60
10131	61626	44	7752	20096.38
10132	52336	41	-1	0.00
10133	54972	44	7548	18350.20
10134	17411	43	8765	88965.55

10135 rows × 4 columns

In [29]:
```
assert list(df.columns) == ['id', 'age', 'steps', 'income']
```

Question 3f: You may have noticed something strange: the merged `df` DataFrame has fewer rows than either of `df_steps` and `df_income`. Why did this happen? (If you're unsure, check out the documentation for the `pandas` method you used to merge these two datasets. Take note of the default values set for this method's parameters.)

Please select the **one** correct explanation below and save your answer in the variable `q1f_answer`. For example, if you believe choice number 4 explains why `df` has fewer rows, set `q1f_answer = 4`.

1. Some steps were recorded inaccurately in `df_steps`.
2. Some incomes were recorded inaccurately in `df_income`.
3. There are fewer rows in `df_steps` than in `df_income`.
4. There are fewer columns in `df_steps` than in `df_income`.
5. Some `id` values in either `df_steps` and `df_income` were missing in the other DataFrame.
6. Some `id` values were repeated in `df_steps` and in `df_income`.

You may use the cell below to run whatever code you want to check the statements above. Just make sure to set `q1f_answer` once you've selected a choice.

In [30]:
```
q1f_answer = 5
```

In [31]:
```
assert isinstance(q1f_answer, int)
```

Part 4 - Data Cleaning

Post data collection, a pivotal step ensues—Data Cleaning. This phase is crucial for ensuring the reliability and accuracy of our analysis. It involves scrutinizing the data for inaccuracies, inconsistencies, and incompleteness. Techniques such as removing duplicates, handling missing values, and correcting errors are employed to refine the dataset. A common phenomenon is that the collected data may contain missing values. Here are two common ones:

- **Nonresponse.** For example, people might have left a field blank when responding to a survey, or left the entire survey blank.
- **Lost in entry.** Data might have been lost after initial recording. For example, a disk cleanup might accidentally wipe older entries of a database.

In general, it is **not** appropriate to simply drop missing values from the dataset or pretend that if filled in they would not change your results. In 2016, many polls mistakenly predicted that Hillary Clinton would easily win the Presidential election by committing this error. In this particular dataset, however, the **missing values occur completely at random**. This criteria allows us to drop missing values without significantly affecting our conclusions.

In this section, we continue use the data mentioned in Part 3.

Question 4a: How many values are missing in the `income` column of `df`? Save this number into a variable called `n_nan`.

In [32]:
```
n_nan = df['income'].isna().sum()
n_nan
```

Out[32]: 451

In [33]:
```
assert (n_nan)
```

Question 4b: Remove all rows from `df` that have missing values.

In [34]:
```
df.dropna(inplace=True)
```

In [35]:
```
assert sum(np.isnan(df['income'])) == 0
assert df.shape == (9684, 4)
```

Question 4c: Note that we can now compute the average income. If your `df` variable contains the right values, `df['income'].mean()` should produce the value `25508.84`.

Suppose that we didn't drop the missing incomes. What will running `df['income'].mean()` output? Use the variable `q2c_answer` to record which of the below statements you think is true. As usual, you can use the cell below to run any code you'd like in order to help you answer this question as long as you set `q2c_answer` once you've finished.

1. No change; `df['income'].mean()` will ignore the missing values and output `25508.84`.
2. `df['income'].mean()` will produce an error.
3. `df['income'].mean()` will output `0`.
4. `df['income'].mean()` will output `nan` (not a number).
5. `df['income'].mean()` will fill in the missing values with the average income, then compute the average.
6. `df['income'].mean()` will fill in the missing values with `0`, then compute the average.

In [36]:
```
q2c_answer = 1
```

In [37]:
```
assert isinstance(q2c_answer, int)
```

Question 4d: Suppose that missing incomes did not occur at random, and that individuals with incomes below $10000 a year are less likely to report their incomes. If so, which of the following statements below is true? Record your choice in the variable `q2d_answer`.

1. `df['income'].mean()` will likely output a value that is the same as the population's average income
2. `df['income'].mean()` will likely output a value that is smaller than the population's average income.
3. `df['income'].mean()` will likely output a value that is larger than the population's average income.
4. `df['income'].mean()` will raise an error.

In [38]:
```
q2d_answer = 3
```

In [39]:
```
assert isinstance(q2d_answer, int)
```

Complete!

Congrats, you're done!