HOMEWORK 1 (110 points + 1) : CAUTION! CONTENTS ARE HOT 🌋

DUE: *SEPTEMBER 18, 2025 @ 11:59 PM*

24-HR LATE DUE DATE WITH A 15% PENALTY: *SEPTEMBER 19, 2025 @ 11:59 PM*

The NCEI/WDS Global Significant Volcanic Eruptions Database is a very comprehensive collection of +600 volcanic eruptions dating from 4360 BC to the present. Due to the nature of this assignment, we will be dealing with relatively newer volcanoes (in which some are still obviously still older than anyone on Earth currently). Each eruption in the database is classified as significant if it meets one or more criteria, such as causing fatalities, incurring **damage on property** (**+$1 million**), reaching a **Volcanic Explosivity Index (VEI)** of **6 or higher**, generating a tsunami, or being linked to a significant earthquake. The VEI is a scale that measures the explosiveness of volcanic eruptions, providing insight into the magnitude and potential consequences of the eruptions. The database includes detailed information on the location, type of volcano, last known eruption, VEI, casualties, property damage, and much more.

We are going to dive straight into these volcanoes (well... their dataset), to swim our way into Pandas proficiency!

You will find the Pandas Documentation helpful. There are also some helpful links to guide you along the way! Don't get burned 🔥

For this assignment, we are including a special bonus question to help everyone get familiar with the proper assignment submission routine. We strongly encourage you to complete Bonus Question 1, learn the process, and follow this practice for all future assignments.

For this assignment only, you will not lose points if you skip the bonus question. However, starting from Homework 2, submitting both the Jupyter Notebook file and the PDF will be required in order to receive full credit.

DO NOT REMOVE ANY PART OF ANY OF THE QUESTIONS OR YOU LOSE CREDIT

No Hardcoding either 😋 💗

REMEMBER TO SHOW OUTPUT

Part 1: Maintenance 😈 (25 POINTS TOTAL)

First, we're going to familiarize ourselves with the process. As in most languages, Python looks best when its modules are imported first before any other code is written ✨

In [ ]:
```
# Make sure these code blocks run properly and that you have properly installed the appropriate modules required.
import pandas as pd
import requests
import numpy
```

```
import math
# import other libraries here

# Don't Remove this
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

As you may have noticed, there's another library aside from Pandas called "requests." **The requests library allows you to send HTTP requests to a server, retrieve the content, and process it at ease.** It's very beginner friendly for those attempting to get into webscraping (super important for collecting and creating datasets). We also recommend looking into BeautifulSoup (yeah, soup LOL), another wonderful library that can be paired with the requests library for webscraping.

As shown below, sometimes specific websites require specific headers in order to process a request to access the data.

To check if a request was processed successfully, use the status_code function to see if the process returned 200.

In [ ]:
```
# API URL and headers in case request gets denied.
api_url = "https://www.ngdc.noaa.gov/hazel/hazard-service/api/v1/volcanoes"

headers = {
    'accept': '*/*'
}

data = requests.get(api_url)
data.json()
```

TASK 1.0: Cute Webscraping (5 points)

To make our cute webscraper we need to **create a GET request** using the relevant information given above.

This particular dataset NOAA returns data from the API as ***json*** when a user makes a request. The json data has a particular format, so we will extract our needed information only from the field called **items** to make a dataframe.

After properly scraping the data, name this dataframe *df*

Save this dataframe into a CSV file named 'volcanoes.csv'

You won't need to run this more than once

In [ ]:
```
df = pd.DataFrame(data.json()["items"])
df.to_csv("volcanoes.csv")
```

TASK 1.1: 1 Liner Thingz (3 points)

We need to get an idea of what this dataset is going to look. In order to do that, let's take a look at some of the most basic things our dataframe has.

Read the directions carefully and code your answer with only one line of code.

CAN'T USE LOOPS. DO NOT DISPLAY THE DATAFRAME, JUST YOUR CODE OUTPUT HERE.

1.1.1: In one line of code and **using only one single method call**, output the **summary statistics** (count, mean, std, min, max, quartiles) of all numerical features in the dataframe.

Hint: Look for a method that gives a descriptive overview of numerical columns.

In [ ]:
```
df.describe()
```

1.1.2: In one line of code and **using only one single method call**, print the **summary information** of the dataframe, including index dtype, column names, non-null counts, and memory usage.

In [ ]:
```
df.info()
```

We won't be using some of the data because there is a lot of missing data.

1.1.3: *In one line of code, create a **new dataframe** called **new_df** that **discards** all the features of the **old** dataframe **except for the following**:*

`id, year, month, day, tsunamiEventId, earthquakeEventId, volcanoLocationId, volcanoLocationNewNum, name, country, elevation, morphology, deathsTotal, vei, deaths`

Hint: Don't use any drop function

In [ ]:
```
new_df = df[["id", "year", "month", "day",      "tsunamiEventId", "earthquakeEventId", "volcanoLocationId", "volcanoLocationNe
```

TASK 1.2: 1 Liner Shenaniganz (7 points)

We're going to tidy up the **new dataframe** a little more with some more advanced 1 liner code.

Read the directions carefully and code your answer with only one line of code.

For this section, keep the method of display that is already in the box. Write your code as indicated.

YOU CAN'T USE ONE LINE LOOPS OR ANY KIND OF LOOP.

1.2.1: *In one line of code, **drop any row** that contains **NaN** in **any one** of the columns indicating a measure of **time**.*

In [ ]:
```
new_df.dropna(subset=["year", "day", "month"], inplace=True)
```

1.2.2: *In one line of code, **change** the **index column** of the dataframe so that it has **1-based indexing**.*

In [ ]:
```
new_df.index = numpy.arange(1, len(new_df) + 1)
```

The **deathsTotal** and **deaths** columns have approximations of the same data with alternating NaNs in each.

1.2.3: *In one line of code, make a **new column** called **'totalDeaths'** that takes the **max** of the values given between those **two** *columns.

- If there is **NaN** in ***one column*** and a **numerical** value in the **other**, it will ***take the numerical value***.
- **Only** if there are **NaNs** in ***both columns***, the **new column will have NaN.**

In [ ]:
```
new_df["totalDeaths"] = new_df[["deaths", "deathsTotal"]].max(axis=1)
```

TASK 1.3: Tailoring Time (10 Points)

It's pretty obvious that the year, month, and day look pretty weird in the dataset. We're going to have to do some hardcore cleaning on the time.

We need to have only ONE column called "*date*" that contains the full date (YYYY-MM-DD), not separated into three columns.

Make sure there are no floating point values in the date and sort the data from most recent to least.

Remove the old columns and place the new column next to the 'id' column.

YOU CAN USE MULTIPLE LINES OF CODE BUT CAN'T USE LOOPS.

Note: It is alright to have only a **maximum of 12 NaTs** for some dates that often go further back than the 1600s because the datetime module in Pandas has a limit (unless otherwise guided).

In [ ]:
```
new_df["year"] = new_df["year"].apply(lambda x: None if x < 1677 else x)
new_df.insert(loc=1, column="date", value=pd.to_datetime(arg=new_df[["year", "month", "day"]]))
new_df.drop(["year", "month", "day"], axis=1, inplace=True)
```

Part 2: Volcanic Matryoshkas 🪆 (30 POINTS TOTAL)

Now, that most of the data has been tidied up. We will organize the data into more sizable pieces of information in order to extract useful information.

2.1.1: *(10 points here)*

Use the **groupby function in Pandas** to create separate dataframes for each unique country.

- **Each table must only have the columns**: 'date' 'country', 'name', and 'vei' (Having the index is ok. Up to you.)
- **Sort** the dataframe of **each country** by **highest to lowest 'vei'**
- Use the **display** function to show **each sorted table**

You MUST use the groupby function here and display your results.

In [ ]:
```
from IPython.display import display

separated_dfs = {}

for group_name, group_df in new_df[["date", "country", "name", "vei"]].groupby(["country"]):
```

```
    group_df.sort_values(by="vei", inplace=True, ascending=False)
    separated_dfs[group_name] = group_df
    display(group_df)
```

2.1.2: *(5 points here)*

Using **groupby again**, print out the **maximum 'vei'** for **each unique country.**

You MUST use the groupby function here and print your results.

- **Print** out your results in a format like the following: "Country: {country_name}, Highest VEI: {vei}"

In [ ]:
```
for country, fmax in new_df.groupby("country")["vei"].max().items():
    print(f"Country: {country}, Highest VEI: {fmax}")
```

Finally, we have ALMOST REACHED THE END!! Since there is still quite a bit of missing data, we want to make use of what is still available.

A very powerful tool in Python's magnificent collection of libraries is its beautiful graphing tools.

Check out libraries such as Seaborn or Matplotlib to create meaningful visualizations! **Your final task in this section requires the use of these libraries**

2.1.3: *(15 points here)*

- Based on the **unique names of volcanoes**, **filter names that have more than 4 datapoints under their name.**
- Each datapoint in the dataframe refers to a recorded instance of a volcanic eruption.
- Make **separate line graphs for each volcano** and **plot their VEIs over time.**

Make sure to properly label all parts of the graph appropriately to receive credit 👀 (like title, axes, legend, etc...)

In [ ]:
```
import seaborn
for volcano, vol_df in new_df.groupby("name").filter(lambda x: len(x) > 4).groupby("name"):
    # display(vol_df)
    seaborn.lineplot(data=vol_df, x="date", y="vei")
```

Part 3: Fiery Jobs 🚒 (15 POINTS TOTAL)

Proficiency in SQL is also super important. SQL databases are essentially relational databases in which there are vast amounts of tabular data. which can often be used to connect with related tablular data. This is a pretty good intro into learning more about SQL.

Check out this tutorial for some clarifications on SQL.

Now! We'll be using `sqlite` to access a database.

- **!!!IMPORTANT!!! PLEASE Start by downloading the sql lite file and putting it in the same directory as this KAGGLE DATASET URL (hit the 'download' button in the upper right). You need an account in order to download the dataset.**
- Check out the description of the data so you know the table / column names.

The following code will use `sqlite3` to create a database connection. `sqlite3` is the library in Python that assists in navigating through SQL databases.

Note: If you are working on this assignment via Google Colab, sometimes the runtime resets and it will throw errors.

Instead of running through the entire notebook, run the notebook from the following code block and onwards:

- Click anywhere on the next code block.
- Go up to where it says **'Runtime'** in the toolbar (right under the title of the notebook and **in between 'Insert' and 'Tools'**)
- Hover over it and **click on the option** that says **'Run cell and below'**

In [ ]:
```
import sqlite3
import pandas as pd # Pandas was already imported from the previous sections

conn = sqlite3.connect("database.sqlite")
crsr = conn.cursor()
```


!!!IMPORTANT!!! PLEASE Start by downloading the sql lite file and putting it in the same directory as this KAGGLE DATASET URL (hit the 'download' button in the upper right). You need an account in order to download the dataset.

If you cannot see anything when executing this code, that means you did not download the Kaggle dataset correctly.

In [ ]:
```
# This code will let you check out the different tables within the database.
query = "SELECT name FROM sqlite_master WHERE type='table';"
tables = crsr.execute(query).fetchall()
print(tables)
```

Remember that each problem should be solved with a single SQL query.

Note: All outputs must be shown

- Only include whatever fields are mentioned throughout each question, nothing more and nothing less.
- Follow each instruction clearly

3.1.1: 2 Points

From the **Salaries** table, get the **average base pay** for firefighters (all job titles consisting of the word "firefighter" **(not case-sensitive)**) between the **years 2010 to 2015**.

Remember that firefighters that also occupy other professions are still considered firefighters.

Hint: Look into this

In [ ]:
```
query = "SELECT AVG(BasePay) FROM Salaries WHERE JobTitle LIKE '%firefighter%' AND Year BETWEEN 2010 AND 2015"


# KEEP THIS. It will display the whole dataframe.
df = pd.read_sql(query, conn)
df
```

3.1.2: 2 Points

From the **Salaries** table, get all the firefighters (all job titles consisting of the word "firefighter" **(not case-sensitive)**) in the **year 2011** making under **$40,000 as a base pay. Sort** them in **descending** order by their pay.

Remember that firefighters that also occupy other professions are still considered firefighters.

In [ ]:
```
query = "SELECT JobTitle, BasePay FROM Salaries WHERE JobTitle LIKE '%firefighter%' AND BasePay < 40000 AND Year = 2011 ORDER
# KEEP THIS. It will display the whole dataframe.
df = pd.read_sql(query, conn)
df
```

3.1.3: 4 Points

From the **Salaries** table, first get the **averages** of **base pay**, **benefits**, and **overtime pay** for firefighters (all job titles consisting of the word "firefighter" **(not case-sensitive)**).

- Then, make a **column with the sum** of these **three averages**
- Finally, **exclude** job titles containing "FIREFIGHTER" **(case-sensitive)**

Remember that firefighters that also occupy other professions are still considered firefighters.

In [ ]:
```
query = '''
SELECT AVG(BasePay), AVG(Benefits), AVG(OvertimePay), (AVG(BasePay) + AVG(Benefits) + AVG(OvertimePay))
FROM Salaries
WHERE JobTitle LIKE '%firefighter%' AND NOT JobTitle LIKE 'FIREFIGHTER'

'''


# KEEP THIS. It will display the whole dataframe.
df = pd.read_sql(query, conn)
df
```

3.1.4: 7 Points

Finally, we'll make our own table in our database.

- Separate the **Salaries table** by **years (2010-2015)**, and add it back to the database.
- Using a loop might be helpful.
- You may use basic python to complete the task. However, using querying on SQL is **mandatory**.
- Feel free to **use multiple lines of code for this problem only.**

Hint: Check this out

In [ ]:
```
# REMOVE THIS CONTENT AND ANSWER IN YOUR OWN WAY
query = '''
SELECT * FROM Salaries
WHERE Year AND Year >= 2010 AND Year <= 2015
'''

df = pd.read_sql(query, conn)

for year, dataf in df.groupby(by=["Year"]):
    # print(year[0])
    dataf.to_sql(name=str(year[0]), con=conn)
```

In [ ]:
```
# Run this code to check if you successfully added your table.
cursor = conn.cursor()
cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
print(cursor.fetchall())
```

Part 4: Pandas 'Group By'(40 points)

This flowchart is taken from our lecture class presentation and illustrates the process of transforming data using the Pandas GroupBy operation. First, the data is input, followed by applying the GroupBy function to one or more columns of the DataFrame. Once the data is grouped, an aggregation function (such as sum(), mean(), or count()) is applied to compute summary statistics for each group

 bonus

Your task is to translate the workflow shown in the flowchart into Pandas queries that perform these operations step by step.

Notes: Your task is to translate the workflow shown in the flowchart into Pandas queries. Ensure that the **exact input and exact output** from the flowchart are replicated using Pandas queries, step by step.

4.1 (Point 10) Create a sample dataset that includes columns for account, order, and ext price.

In [ ]:
```
# Create the Sample dataset from above flowchart
input_data = {
    'account': ['383080', '383080', '383080', '412290', '412290','412290','412290','412290','218895','218895','218895','218895
    'order': ['10001', '10001', '10001', '10005', '10005','10005','10005','10005','10006','10006','10006','10006'],
    'ext_price': [235.83, 232.32, 107.97, 2679.36, 286.02, 832.95, 3472.04, 915.12, 3061.12, 518.65, 216.9, -72.18]
}

# Create DataFrame
df2 = pd.DataFrame(input_data)

# Display Dataframe (DONT REMOVE THE CODE)
df2
```

4.2 (Point 10+10 =20) Group by order and show the intermediate results

In [ ]:
```
for ind, dataf in df2.groupby(["order"]):
    display(dataf)
```

4.3 (Point 10) Apply the Sum Aggregation for Each Group

Now we'll apply the sum aggregation to get the total ext price for each order:

In [ ]:
```
# Repeat group by 'order' again and then apply aggregation (sum of 'ext_price' for each 'order')

df3 = df2.groupby(["order"])["ext_price"].sum()

# Show the aggregated result after re-grouping  (DONT REMOVE THE CODE)
df3
```

4.4 (Point 10) Combine the Results into One Final Table Finally, we will **reset the index** and create a combined table that shows order and the sum of the ext price for each group:

Notes: In pandas, 'reset_index()' is a method used to reset the index of a DataFrame to its default integer-based index. By default, when you perform certain operations like groupby(), the resulting DataFrame may have a new index (e.g., the grouped column). The reset_index() method allows you to convert the current index back to a default sequential integer index and optionally, move the current index values into a regular column.

In [ ]:
```
# Reset index to combine result into a single DataFrame
df3.reset_index()
# Rename the columns for clarity

# Show the final result  (DONT REMOVE THE CODE)
```

Bonus(1 point): Assignment Submission with Jupyter Notebook file(ipynb) and PDF

In addition to the instruction of the special bonus question included exclusively in Assignment 1, we would like to highlight the importance of developing good practices for assignment submission in data science projects.

Think of yourself as a reviewer of your peers' work: what would you pay attention to if you had to grade their assignments? Clearly, the raw dataset itself is not the main point—why would someone need to see every line of data you downloaded? Data science is about analysis and insight, not just data collection. What truly matters is how you analyze the data and how you present your results.

For a clean and effective submission, please avoid printing entire datasets unless they are directly relevant to the presentation of your output or if you are asked to do so. Instead, focus on:

- Showing your algorithm with clear, concise code
- Displaying the key results you want us to evaluate (or the outputs you relied on while working)
- Adding a short explanation when asked to interpret your findings

For this bonus question, please submit both your .ipynb file and the PDF version. To create a PDF: go to File (top left corner in Jupyter Notebook) → Print → Destination → Save as PDF.

Are you noticing an overwhelming amount of unnecessary printed results in your PDF? This is exactly why we emphasize the importance of cleaning up your work before submission. Removing trivial data printing makes your notebook look much more professional and easier to review—just like proofreading an essay for good grammar.

Your Homework 1 submission should be lightweight and focused, since this is a starter project designed to help you ease into the semester. :)

This practice helps us grade more efficiently and ensures fairness, since we can always reference your original notebook if regrading is needed.