Python answers:

Ques 1: def reverse\_in\_groups(lst, n):

result = []

i = 0

while i < len(lst):

# Create an empty temporary list to store reversed elements

temp = []

# Loop over the next 'n' elements or less if fewer than 'n' are left

for j in range(i, min(i + n, len(lst))):

temp.insert(0, lst[j]) # Insert elements at the start to reverse

result.extend(temp) # Add the reversed group to the result

i += n

return result

def group\_strings\_by\_length(lst):

# Create an empty dictionary to store grouped strings by their length

result = {}

# Iterate over each string in the list

for string in lst:

length = len(string) # Calculate the length of the current string

# If the length is not already a key in the dictionary, create a new list for it

if length not in result:

result[length] = []

# Append the string to the list corresponding to its length

result[length].append(string)

# Sort the dictionary by keys (string lengths) and return it

return dict(sorted(result.items()))

Explanation:

The function iterates through the list in steps of size n.

For each group of n elements, it reverses the elements by inserting them at the start of a temporary list.

The reversed group is then appended to the result list.

This process continues until the entire list is processed.

Explanation:

We loop through each string in the input list.

For each string, we calculate its length and check if the length is already a key in the dictionary:

If not, we create a new list for that length.

If it exists, we append the string to the existing list.

Finally, we sort the dictionary by keys (string lengths) to ensure the result is in ascending order of lengths.

Ques 3

def flatten\_dict(d, parent\_key='', sep='.'):

items = [] # List to store the flattened key-value pairs

for key, value in d.items():

# Create a new key by combining the parent key with the current key

new\_key = f"{parent\_key}{sep}{key}" if parent\_key else key

# If the value is a dictionary, recursively flatten it

if isinstance(value, dict):

items.extend(flatten\_dict(value, new\_key, sep=sep).items())

# If the value is a list, flatten each element and use the index in the key

elif isinstance(value, list):

for i, elem in enumerate(value):

items.extend(flatten\_dict(elem, f"{new\_key}[{i}]", sep=sep).items())

else:

# For other types (int, string, etc.), just add the key-value pair

items.append((new\_key, value))

return dict(items)

def unique\_permutations(nums):

def backtrack(path, used):

if len(path) == len(nums):

result.append(path[:])

return

for i in range(len(nums)):

# Skip if the number is already used or if it's a duplicate and hasn't been used in the previous recursion

if used[i] or (i > 0 and nums[i] == nums[i - 1] and not used[i - 1]):

continue

used[i] = True

path.append(nums[i])

backtrack(path, used)

path.pop() # Remove the last element to backtrack

used[i] = False # Mark the element as unused for the next iteration

nums.sort() # Sort the list to handle duplicates

result = []

used = [False] \* len(nums) # Track which elements are used in the current path

backtrack([], used)

return result

import re

def find\_all\_dates(text):

# Regular expression to match dd-mm-yyyy, mm/dd/yyyy, yyyy.mm.dd

date\_pattern = r"\b\d{2}-\d{2}-\d{4}\b|\b\d{2}/\d{2}/\d{4}\b|\b\d{4}\.\d{2}\.\d{2}\b"

# Find all dates that match the pattern

dates = re.findall(date\_pattern, text)

return dates

Ques 4

import pandas as pd

def check\_time\_completeness(df):

# Convert the start and end times to datetime objects for easier comparison

df['startTimestamp'] = pd.to\_datetime(df['startDay'] + ' ' + df['startTime'])

df['endTimestamp'] = pd.to\_datetime(df['endDay'] + ' ' + df['endTime'])

# Create a MultiIndex for (id, id\_2)

multi\_index = pd.MultiIndex.from\_tuples(df[['id', 'id\_2']].drop\_duplicates().values.tolist(), names=['id', 'id\_2'])

# Initialize the result as True (assuming timestamps are correct)

result = pd.Series(True, index=multi\_index)

# Define the complete set of 7 days of the week

full\_week = set(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'])

# Function to check completeness for each (id, id\_2) group

def check\_group(group):

# Extract the unique days in the group

days = set(group['startDay'])

# Check if the days cover all 7 days of the week

if days != full\_week:

return False

# Check the time coverage for each day

for day in full\_week:

day\_group = group[group['startDay'] == day]

# Check if the timestamps for this day cover a full 24-hour period

min\_time = day\_group['startTimestamp'].min().time()

max\_time = day\_group['endTimestamp'].max().time()

if min\_time != pd.to\_datetime('00:00:00').time() or max\_time != pd.to\_datetime('23:59:59').time():

return False

# If all checks pass, return True

return True

# Apply the check to each group of (id, id\_2)

for name, group in df.groupby(['id', 'id\_2']):

if not check\_group(group):

result.loc[name] = False # Mark as False if the group has incomplete timestamps

return result

def rotate\_matrix\_90\_clockwise(matrix):

# Step 1: Rotate the matrix by 90 degrees clockwise (transpose + reverse rows)

n = len(matrix)

# Transpose the matrix (swap rows with columns)

for i in range(n):

for j in range(i, n):

matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

# Reverse each row to complete the 90-degree rotation

for i in range(n):

matrix[i].reverse()

return matrix

def transform\_matrix(matrix):

# Step 2: Replace each element with the sum of the row and column, excluding itself

n = len(matrix)

transformed = [[0] \* n for \_ in range(n)] # Create an empty matrix of the same size

# Precompute the sum of all elements in each row and column

row\_sums = [sum(row) for row in matrix]

col\_sums = [sum(matrix[i][j] for i in range(n)) for j in range(n)]

# Calculate the transformed matrix

for i in range(n):

for j in range(n):

# Sum of elements in the same row and column, excluding the current element

transformed[i][j] = row\_sums[i] + col\_sums[j] - 2 \* matrix[i][j]

return transformed

def rotate\_and\_transform(matrix):

# First rotate the matrix by 90 degrees

rotated\_matrix = rotate\_matrix\_90\_clockwise(matrix)

# Then transform the rotated matrix

final\_matrix = transform\_matrix(rotated\_matrix)

return final\_matrix

import pandas as pd

def unroll\_distance\_matrix(distance\_matrix):

# Reset the index to include the IDs in the DataFrame

distance\_matrix\_reset = distance\_matrix.reset\_index()

# Melt the DataFrame to long format

long\_format = pd.melt(distance\_matrix\_reset, id\_vars=['index'], var\_name='id\_end', value\_name='distance')

# Rename the index column to id\_start

long\_format = long\_format.rename(columns={'index': 'id\_start'})

# Filter out rows where id\_start is the same as id\_end and distances are not infinite

long\_format = long\_format[(long\_format['id\_start'] != long\_format['id\_end']) &

(long\_format['distance'] != float('inf'))]

return long\_format

# Example usage

# Assuming result\_df is the DataFrame returned from calculate\_distance\_matrix

# unrolled\_df = unroll\_distance\_matrix(result\_df)

# print(unrolled\_df)

import pandas as pd

import numpy as np

def calculate\_distance\_matrix(file\_path):

# Read the dataset

df = pd.read\_csv(file\_path)

# Create a set of unique IDs

unique\_ids = pd.unique(df[['id\_start', 'id\_end']].values.ravel('K'))

# Create an empty DataFrame with unique IDs as both rows and columns

distance\_matrix = pd.DataFrame(np.inf, index=unique\_ids, columns=unique\_ids)

# Fill in the direct distances from the dataframe

for index, row in df.iterrows():

distance\_matrix.at[row['id\_start'], row['id\_end']] = row['distance']

distance\_matrix.at[row['id\_end'], row['id\_start']] = row['distance'] # Ensure symmetry

# Set diagonal to 0 (distance from a node to itself)

np.fill\_diagonal(distance\_matrix.values, 0)

# Calculate cumulative distances using Floyd-Warshall algorithm

for k in unique\_ids:

for i in unique\_ids:

for j in unique\_ids:

if distance\_matrix.at[i, j] > distance\_matrix.at[i, k] + distance\_matrix.at[k, j]:

distance\_matrix.at[i, j] = distance\_matrix.at[i, k] + distance\_matrix.at[k, j]

return distance\_matrix

# Example usage

# result\_df = calculate\_distance\_matrix('dataset-2.csv')

# print(result\_df)

import polyline

import pandas as pd

import math

# Function to calculate the Haversine distance

def haversine(lat1, lon1, lat2, lon2):

# Radius of the Earth in meters

R = 6371000

# Convert latitude and longitude from degrees to radians

phi1 = math.radians(lat1)

phi2 = math.radians(lat2)

delta\_phi = math.radians(lat2 - lat1)

delta\_lambda = math.radians(lon2 - lon1)

# Haversine formula

a = math.sin(delta\_phi / 2) \* 2 + math.cos(phi1) \* math.cos(phi2) \* math.sin(delta\_lambda / 2) \* 2

c = 2 \* math.atan2(math.sqrt(a), math.sqrt(1 - a))

# Distance in meters

distance = R \* c

return distance

# Function to decode polyline and calculate distances

def decode\_polyline\_to\_dataframe(polyline\_str):

# Decode the polyline string into a list of (latitude, longitude) tuples

coordinates = polyline.decode(polyline\_str)

# Create a DataFrame with the latitude and longitude columns

df = pd.DataFrame(coordinates, columns=['latitude', 'longitude'])

# Initialize the distance column with zeros

df['distance'] = 0.0

# Calculate the distance for each row (from the previous row)

for i in range(1, len(df)):

lat1, lon1 = df.loc[i - 1, 'latitude'], df.loc[i - 1, 'longitude']

lat2, lon2 = df.loc[i, 'latitude'], df.loc[i, 'longitude']

df.loc[i, 'distance'] = haversine(lat1, lon1, lat2, lon2)

return df

import pandas as pd

import numpy as np

def find\_ids\_within\_ten\_percentage\_threshold(distance\_df, reference\_id):

# Calculate the average distance for the specified reference\_id

average\_distance = distance\_df.loc[distance\_df['id\_start'] == reference\_id, 'distance'].mean()

# Calculate 10% of the average distance

ten\_percent\_threshold = average\_distance \* 0.10

# Calculate the lower and upper bounds

lower\_bound = average\_distance - ten\_percent\_threshold

upper\_bound = average\_distance + ten\_percent\_threshold

# Filter the DataFrame for ids within the threshold

filtered\_ids = distance\_df[

(distance\_df['distance'] >= lower\_bound) &

(distance\_df['distance'] <= upper\_bound)

]

# Get the unique id\_start values and sort them

result\_ids = sorted(filtered\_ids['id\_start'].unique())

return result\_ids

# Example usage

# Assuming unrolled\_df is the DataFrame returned from unroll\_distance\_matrix

# reference\_value = 1001400

# ids\_within\_threshold = find\_ids\_within\_ten\_percentage\_threshold(unrolled\_df, reference\_value)

# print(ids\_within\_threshold)