Part A: Warmup Tasks

Folder Structure

```
MiniYou
├── fun.py
├── main.py
├── README.md
└── datasets
    ├── health_activity_data.csv
    └── Sleep_health_and_lifestyle_dataset.csv
```

Datasets

The two datasets are discussed below:

health_activity_data.csv:

The Sleep Health and Lifestyle dataset is structured with 13 columns that describe various personal health and lifestyle attributes. The columns include Person ID, Gender, Age, Occupation, Sleep Duration, Quality of Sleep, Physical Activity Level, Stress Level, BMI Category, Blood Pressure, Heart Rate, Daily Steps, and Sleep Disorder.

Sleep_health_and_lifestyle_dataset.csv:

The Health Activity Data file contains 16 columns providing a detailed structure of an individual's health and daily activities. These columns are ID, Age, Gender, Height_cm, Weight_kg, BMI, Daily_Steps, Calories_Intake, Hours_of_Sleep, Heart_Rate, Blood_Pressure, Exercise_Hours_per_Week, Smoker, Alcohol_Consumption_per_Week, Diabetic, and Heart_Disease status.

Functions

mini_load_csv_dict()

The function takes the csv dataset as a dictionary and loads it with the `csv.DictReader()`. This function uses the return method to return all the rows in a single list.

```python
def mini_load_csv_dict(input_dict):
    filename = input_dict.get('data')
    if not filename:
        raise ValueError("No 'data' key provided in the input dictionary.")
    data_rows = []
    try:
        with open(filename, "r", newline='', encoding="utf-8") as csvfile:
            reader = csv.DictReader(csvfile)
            for row in reader:
                data_rows.append(row)
```

```
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return []
    return data_rows
```

Time complexity: `O(n)`: looping over every row once, which is O(1), so O(n). Space complexity: `O(n)`: creating the list makes it take the same space as the rows in the dataset.

mini_load_csv_yield()

This function does the same with the `yield` method.

```
def mini_load_csv_yield(input_dict):
    filename = input_dict.get("data")

    if not filename:
        raise ValueError("No 'data' key provided in the input dictionary.")

    try:
        with open(filename, newline='', encoding='utf-8') as csvfile:
            reader = csv.DictReader(csvfile)
            for row in reader:
                yield row
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return
```

Time complexity: `O(n)`: same as the return method Space complexity: `O(1)`: we process one at a time and not using any list. so does not need extra spaces.

Part B

Task 01

`mini_len()` function that takes a column name as input and returns: 1. Whether the column exists in the dataset (Exists: True or False) 2. If it exists: 1. name of the column 1. number of records 1. number of missing values

Usage:

```
input_data = {"data": "datasets/Sleep_health_and_lifestyle_dataset.csv"}
print(fun.mini_len(input_data, 'Gender'))
```

Output: If exists: `{"Exists": True, "Column": "Gender", "Number of records": 374, "Number of missing records": 0}` if not exists; `{"Exists": False}`

Time complexity:`O(n)`, as each row is processed on by one Space complexity: `O(1)`, one row at a time from the generator, no need of extra spaces.

Task 02

`mini_search()` function takes the dataset and a column name and the value as input, searches for it and returns if it exists or not > Here `return` is used as we want to exit immediately upon finding the first match

Usage:

```
print(fun.mini_search(input_data, "Occupation", 'Lawyer'))
```

Output: If exists: `{"Exists": True, "Column": "Occupation", "Value": "Lawyer"}` If not exists: `{"Exists": False, "Column": "Occupation", "Value": "Lawyer"}`

Time: `O(n)` Space: `O(n)`

Task 03

`mini_proportion_count()` takes the column name and value as input and finds the proportion of the value in the entire column. The proportion is calculated as follows: if 3 out of 10 rows have the value “Engineer”, the proportion is 3/10 = 0.3.

> `return` is used intead of `yield` as we need to traverse through everything and give a final aggregated value, rather than processing one by one

Usage:

```
print(fun.mini_proportion_count(input_data, "Heart Rate", '70'))
```

Output: If exists: `{'Exists': True, 'Column': 'Heart Rate', 'Value': '70', 'Proportion': 0.20320855614973263}` If not exists: `{'Exists': False}`

Time complexity: `O(N)` Space complexity: `O(N)`

task 04

`mini_count_match()` takes two column names and two values and find how many instances are common between them

> Used `return` method for the same purpose as the previous

Usage:

```
print(fun.mini_count_match(input_data, "Occupation", "Doctor", "BMI Category", "Normal"))
```

Output: If exists: `{'Conditions': {'Occupation': 'Doctor', 'BMI Category': 'Normal'}, 'Count': 65}` If not exist: `columns exist but no match between them` If none of the columns exist: `{"Exists": False}`

Time Complexity: `O(N)` Space Complexity: `O(N)`

Task 05

`mini_average()` function takes a numeric column and finds the average of the values, ignoring missing or non-numeric values. And return the average in float.

> `return` method is used instead of `yield` as we need to traverse all the instances and return a single value as return.

Usage:

```
print(fun.mini_average(input_data, 'Daily Steps'))
```

Output: If exists: `{'Exists': True, 'Column': 'Daily Steps', 'Average': 6816.84}`
If not exists: `{"Exists": False, "Column":`Daily Steps`}`

Time Complexity: `O(N)` Space Complexity: `O(N)`

Task 06

`mini_extract_metrics()` takes a dictionary of column names and extracts them on the basis of 'Occupation'='teacher'

> `yield` is used to process each item one by one, as we don't need a final aggregate value. That makes the code memory-efficient.

Usage:

```
print(list(fun.mini_extract_metrics(input_data, {"Col1":"Person ID",
"Col2":"Quality of Sleep", "Col3":"Physical Activity Level"})))
```

Example Output:

```
[
    {'Person ID': '7', 'Quality of Sleep': '6', 'Physical Activity Level':
'40'},
    {'Person ID': '83', 'Quality of Sleep': '7', 'Physical Activity Level':
'40'},
    {'Person ID': '84', 'Quality of Sleep': '7', 'Physical Activity Level':
'40'},.....
]
```

Time complexity: `O(N)` as goes over every instance Space complexity: `O(1)`, processed one at a time

Task 07

`mini_stats()` takes a column name and a function name(min or max), then finds the min or max of that column. Ignoring missing or non-integer values.

> `Return` method used, as it needs to travese all the points and find a final answer

Usage:

```
print(fun.mini_states({"Data": input_data, "function": "max", "column": "Age"}))
```

Output: If exists: `{'Function': 'max', 'Column': 'Age', 'Result': 59.0}` If not exist: `{"Function": 'max', "Column": 'Age', "Result": None}`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 08

`mini_bubble_sort()` takes a numeric column and bubble sorts it in ascending order

> `return` is used as we need to go through all the data points and gives a sorted list

Usage:

```
print(fun.mini_bubble_sort({"Data": input_data, "column": "Age"}))
```

Output: If exists: `{'column': 'Age', 'sorted data': [27.0, 28.0, 28.0, 28.0, 28.0, 28.0, 29.0, 29.0,.....]}` If not exists: `return {"Exists": "False"}`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 09

`mini_value_list_exists()` finds a specific value in the sorted list

> `return` is used as in the previous bubble sort function

Usage:

```
print(fun.mini_value_list_exists({"Data": input_data, "column": "Age", "value": 29.0}))
```

Output: If exists: `{'Value': 29.0, 'Exists': True}` If not exists: `{'Value': 7.0, 'Exists': False}`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 10

`mini_frequency_table()` counts the values in a specific column

> `return` is used instead of `yield` as we need to go output one final, fully sorted list of values

Usage:

```
print(fun.mini_frequency_table({"Data": input_data, "column": "Gender"}))
```

Output: If exists: `{'Gender': {'male': 188, 'female': 185}}` If not exists: `{"Column": col, "Exists": False}`

Time complexity: `O(N)` Space complexity: `O(N)`

Part C

Task 01

`mini_check_missing_values()` takes one column and counts how many missing values are there

`return` method is used

Usage:

```
print(fun.mini_check_missing_values({"Data": input_data, "column": 'Daily Steps'}))
```

Output: If column exists: `{'Column': 'Daily Steps', 'Missing Values': 0}` If column not exist: `{'Exists': False}`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 02

`mini_report_daily_steps()` finds the individuals report Daily_Steps of 6457 hours.

Usage:

```
def mini_report_daily_steps(input_data):
    total = mini_len(input_data, "Daily Steps").get('Number of records')
    proportion = mini_proportion_count(input_data, "Daily Steps", "7000").get('Proportion')
    return {"Column": "Daily Steps", "Individuals Report": proportion*total}


print(fun.mini_report_daily_steps(input_data))
```

Output: `{'Column': 'Daily Steps', 'Individuals Report': 66.0}`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 03

`mini_calculate_average_BMI()` takes the BMI column from the health activity data and finds the average BMI

’return` method is used as in the mini_average() function

Usage:

```
def mini_calculate_average_BMI(input_data):
    return mini_average(input_data, "BMI")


print(fun.mini_calculate_average_BMI(input_data_2))
```

Output: if exists: `{'Exists': True, 'Column': 'BMI', 'Average': 26.73}` If not exists: `{Exists': False}`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 04

`mini_count_specific_sleep_based_on_gender()` finds how many female users sleep for 7.4 hours

`return` as in the function used

Usage:

```
def mini_count_specific_sleep_based_on_gender(input_data):
    return mini_count_match(input_data.get("Data"),
                            input_data.get('col_one'),
                            input_data.get('val_one'),
                            input_data.get('col_two'),
                            input_data.get('val_two'))
                            .get("Count")

print(fun.mini_count_specific_sleep_based_on_gender({"Data": input_data,
"col_one": "Gender", "val_one": "Female", "col_two": "Sleep Duration",
"val_two": "7.4"}))
```

Output: `1`

Time complexity: `O(N)` Space complexity: `O(N)`

Part D

Task 01

`mini_weather_data()` function takes the list of cities locations and extracts the data. The data is the hourly temperature of a given city. A 1-second delay is included between requests to avoid being rate-limited or blocked.

```
def mini_weather_data(locations):
    url = "https://api.open-meteo.com/v1/forecast"
    results = []
    for loc in locations:
        params = {
            "latitude": loc["lat"],
```

```
            "longitude": loc["lon"],
            "hourly": "temperature_2m"
        }
        response = requests.get(url, params=params)
        data = response.json()
        hourly = data.get("hourly", {})
        temperatures = hourly.get("temperature_2m", [])
        results.append({
            "city": loc["city"],
            "temperatures": temperatures
        })
        time.sleep(1)
    return results
```

Time complexity: `O(N)` Space complexity: `O(N)`

Task 02

`mini_hottest_city()` function takes the locations list and retrieves the data with the help of `mini_weather_data()` function to calculate the hottest temperature

```
def mini_hottest_city(locations):
    data = mini_weather_data(locations[:10])
    hottest_city = None
    hottest_temperature = None
    for item in data:
        city = item.get('city')
        temperatures = item.get("temperatures")
        city_max = temperatures[0]
        for temp in temperatures[1:]:
            if temp > city_max:
                city_max = temp
        if (hottest_temperature is None) or (city_max > hottest_temperature):
            hottest_temperature = city_max
            hottest_city = city
    return {"city":hottest_city, "Hottest Temperature": hottest_temperature}
```

Usage:

```
print(fun.mini_hottest_city(locations))
```

Output: `{'city': 'Paris', 'Hottest Temperature': 39.4}`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 03

`mini_coldest_city()` function takes the locations list and retrieves the data with the help of `mini_weather_data()` function to calculate the coldest temperature

```
def mini_coldest_city(locations):
    data = mini_weather_data(locations[:10])
    coldest_city = None
    coldest_temperature = None
    for item in data:
        city = item.get('city')
        temperatures = item.get("temperatures")
        city_min = temperatures[0]
        for temp in temperatures[1:]:
            if temp < city_min:
                city_min = temp
        if (coldest_temperature is None) or (city_min > coldest_temperature):
            coldest_temperature = city_min
            coldest_city = city
    return {"city":coldest_city, "coldest Temperature": coldest_temperature}
```

Usage:

```
print(fun.mini_coldest_city(locations))
```

Output: `{'city': 'Mumbai', 'coldest Temperature': 27.6}`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 04

`mini_cities_between_range()` function takes the list of cities as input and finds the cities whose temperatures are between 20 degree and 30 degree celcius.

```
def mini_cities_between_range(locations):
    data = mini_weather_data(locations[:10])
    cities_within_range = []
    for item in data:
        city = item.get('city')
        temperatures = item.get("temperatures")
        for temp in temperatures:
            if city not in cities_within_range:
                cities_within_range.append(city)
    return cities_within_range
```

Usage: `print(fun.mini_cities_between_range(locations))`

Output: `['New York', 'London', 'Paris', 'Tokyo',.....]`

Time complexity: `O(N)` Space complexity: `O(N)`

Task 05

`mini_highest_temp_diff()` function takes the list of locations and gets the cities that have the highest temperature differences (max-min)

```
def mini_highest_temp_diff(locations):
    data = mini_weather_data(locations[:10])
    difference = {}
    for item in data:
        city = item.get('city')
        temperatures = item.get("temperatures")
        low = temperatures[0]
        high = temperatures[0]
        for temp in temperatures:
            if temp < low:
                low = temp
            elif temp > high:
                high = temp
        diff = high - low
        difference[city] = diff

    items = []
    for city in difference:
        items.append((city, difference[city]))
    n = len(items)
    for i in range(n):
        for j in range(0, n - i - 1):
            if items[j][1] < items[j+1][1]:
                items[j], items[j+1] = items[j+1], items[j]
    result = {}
    for city, difference in items:
        result[city] = difference
    return result
```

Output: `{'Paris': 19.5, 'London': 16.9, 'Rio de Janeiro': 15.7, 'New York': 13.3, 'Sydney': 12.7}`

Time complexity: `O(N)` Space complexity: `O(N)`