Python Exercise Report

Isla Kim: shining04@csu.fullerton.edu

1. Program documentation

1) `ran_gen.py` : This file generates an array filled with 10 elements, each being a floating-point number within the range of -10 to 10.

rand_gen.py > ...

```python
# Isla Kim

import random

array1 = [1.0] * 10

for i in range(len(array1)):
    array1[i] = random.uniform(-10, 10)

print("\nRandomly generated array with 10 floating numbers in range -10 to 10: \n", array1, "\n")

```

1. It creates and prints an array filled with 10 random numbers.

Execution Result:

```
Randomly generated array with 10 floating numbers in range -10 to 10:
 [-7.7163461923830585, 4.307144075148662, 3.4066179920408466, -8.453167680026706, -2.020
2996237873165, 3.77592446675518, -7.707980069551644, -8.271712633669052, -1.275719181725
9645, 8.408571193294474]
```

2) `Data_generator.py`: This file generates arrays filled with random numbers and variables created according to specific formulas, saving them under the names L1.csv, L1.json, Q1.csv, and Q2.csv.

```python
# Isla Kim

import random
import json
import csv

# Initialize the two-dimensional array with floating-point numbers
linear_data_array = []

# Generate random numbers as x values, calculate y values, and save them in the array
for i in range(1000000):
    x = round(random.uniform(-1000, 1000), 3)
    y = (x * 0.5) + 2
    linear_data_array.append([y, x])

# Display the first, middle, and last pairs
print("The first pair:", linear_data_array[0])
print("The middle pair:", linear_data_array[len(linear_data_array)//2])
print("The last pair:", linear_data_array[-1])
```

1. It generates a random number x, then creates y using a specific formula involving x, and constructs an array consisting of 100,000,000 [y, x] pairs. The first, middle, and last [y, x] pairs are printed.

```python
# Saves the data in the array into L1.json
with open('L1.json', 'w') as jsonfile:
    json.dump({"linear": linear_data_array}, jsonfile)  # Store data with "linear" key
print("L1.json well generated!")

# Saves the data in the array into L1.csv
with open('L1.csv', 'w', newline='') as csvfile:
    write = csv.writer(csvfile)
    write.writerows(linear_data_array)  # Write the rows of data to the CSV
print("L1.csv well generated!")
```

2. A key-value pair format is used for L1.json, where the key is "linear," and a comma-separated L1.csv file is created.

```python
# Initialize the two-dimensional array with floating-point numbers
polynomial_data_array1 = []
polynomial_data_array2 = []

# Generate random numbers as x values, calculate x1, x2, y values, and save them in the array
for i in range(1000000):
    x = round(random.uniform(-1000, 1000), 3)
    x1 = 0.5*x
    x2 = -3*x**2
    y = (-3 * x**2) + (x * 0.5) + 2
    polynomial_data_array1.append([y, x])
    polynomial_data_array2.append([y, x1, x2])

# Saves the data in the array into Q1.csv with two columns y and x
with open('Q1.csv', 'w', newline='') as csvfile2:
    write = csv.writer(csvfile2)
    write.writerows(polynomial_data_array1)
print("Q1.csv well generated!")

# Saves the data in the array into Q2.csv with two columns (y, x1, x2)
with open('Q2.csv', 'w', newline='') as csvfile3:
    write = csv.writer(csvfile3)
    write.writerows(polynomial_data_array2)
print("Q2.csv well generated!")
```

3. It generates a random number x, then creates y, x1, and x2 using specific formulas involving x. The resulting data is stored in Q1.csv as [y, x] pairs and in Q2.csv as [y, x, x1, x2] pairs.

Execution Result:

```
The first pair: [443.2325, 882.465]
The middle pair: [352.557, 701.114]
The last pair: [442.9145, 881.829]
L1.json well generated!
L1.csv well generated!
Q1.csv well generated!
Q2.csv well generated!
```

3) `data_loader.py`: This file defines functions for loading data from CSV and JSON files.

```python
# Isla Kim

import json
import csv

# Define a function that loads csv file and save the data into Python array data structures.
def csv_file_loader(file_name):
    with open(file_name, 'r', newline='') as csvfile:
        csvreader = csv.reader(csvfile)
        data = list(csvreader)
    return data

# Define a function that loads json file and save the data into Python array data structures.
def json_file_loader(file_name):
    with open(file_name, 'r') as jsonfile:
        json_data = json.load(jsonfile)
        json_data_array = json_data["linear"]
    return json_data_array
```

1. It defines functions that load .csv and .json files and store them as Python arrays.

```python
# Read and load the data stored in each of the file “L1.csv”, “Q1.csv”, “Q2.csv”, “L1.json”
if __name__ == "__main__": # Process only when this script runs directly
    print('\n"L1.csv": ', end='')
    L1_data = csv_file_loader("L1.csv")
    if L1_data:
        print("Well loaded!")

    print('\n"Q1.csv": ', end='')
    Q1_data = csv_file_loader("Q1.csv")
    if L1_data:
        print("Well loaded!")

    print('\n"Q2.csv": ', end='')
    Q2_data = csv_file_loader("Q2.csv")
    if L1_data:
        print("Well loaded!")

    print('\n"L1.json": ', end='')
    L1_json_data = json_file_loader("L1.json")
    if L1_data:
        print("Well loaded!\n")
```

2. The data is read from "L1.csv," "Q1.csv," "Q2.csv," and "L1.json" files. If loading is successful, "Well loaded" is printed.

Execution Result:

```
"L1.csv": Well loaded!

"Q1.csv": Well loaded!

"Q2.csv": Well loaded!

"L1.json": Well loaded!
```

4) `data_processor1.py`: This file loads, normalizes, and standardizes the data, converting it into Pandas and NumPy arrays.

```python
# Isla Kim

import numpy as np
import pandas as pd
import data_loader

# Load the data from Q1.csv and Q2.csv using the functions from data_loader.py
Q1_data = np.array(data_loader.csv_file_loader("Q1.csv"), dtype=float)
Q2_data = np.array(data_loader.csv_file_loader("Q2.csv"), dtype=float)

# Compute Mean and Standard Deviation for each column in Q1.csv and Q2.csv
def calculate_mean_std(data): # Define a function that calculates mean and Standard Deviation
    mean = np.mean(data, axis=0)
    std_dev = np.std(data, axis=0)
    return mean, std_dev

Q1_mean, Q1_std = calculate_mean_std(Q1_data)
Q2_mean, Q2_std = calculate_mean_std(Q2_data)

print("\n-----------------------------------------------")
print(f'Mean of "Q1.csv" for each column: {Q1_mean}')
print(f'Standard Deviation of "Q1.csv for each column": {Q1_std}')
print(f'\nMean of "Q2.csv for each column": {Q2_mean}')
print(f'Standard Deviation of "Q2.csv for each column": {Q2_std}')
print("-----------------------------------------------\n")
```

1. It loads Q1.csv and Q2.csv as NumPy arrays, calculates, and prints the mean and standard deviation for each column.

```python
# Identify Outliers whose values are beyond 2 Standard Deviations
def identify_outliers(data, mean, std): # Define a function that identifies outliers
    outliers_2std = (data < mean - 2 * std) | (data > mean + 2 * std)
    outliers_1std = (data < mean - std) | (data > mean + std)

    if np.any(outliers_2std): # if there are outliers
        print("There are Outliers (beyond 2 std dev):", data[outliers_2std])
    else: # if there are no outliers
        print("No values beyond 2 std dev, showing values beyond 1 std dev:", data[outliers_1std])

print("\n----------------------------------------------")
print("In Q1.csv:")
identify_outliers(Q1_data, Q1_mean, Q1_std)

print("\nIn Q2.csv:")
identify_outliers(Q2_data, Q2_mean, Q2_std)
print("----------------------------------------------\n")
```

2. Outliers are identified and printed from Q1.csv and Q2.csv.

```python
# Remove Outliers and Normalize Data
def normalize_data(data, mean, std):
    filtered_data = data[(data >= mean - 2 * std) & (data <= mean + 2 * std)]
    normalized = (filtered_data - np.min(filtered_data)) / (np.max(filtered_data) - np.min(filtered_data)) # Convert data to value between 0 and 1
    normalized = normalized - np.mean(normalized)  # Adjust mean to 0
    return normalized

Q1_normalized = normalize_data(Q1_data, Q1_mean, Q1_std)
Q2_normalized = normalize_data(Q2_data, Q2_mean, Q2_std)

# Randomly sample 10 values to verify normalization
print("\n----------------------------------------------")
Q1_normalized_random = np.random.choice(Q1_normalized, 10)
print("10 Random Samples from Normalized Q1 Data:\n", Q1_normalized_random)
if ((Q1_normalized_random > -1) & (Q1_normalized_random < 1)).all():
    print("-> Well converted!")
else:
    print("-> Something wrong happened. Try again.")

Q2_normalized_random = np.random.choice(Q2_normalized, 10)
print("\n10 Random Samples from Normalized Q2 Data:\n", Q2_normalized_random)
if ((Q2_normalized_random > -1) & (Q2_normalized_random < 1)).all():
    print("-> Well converted!")
else:
    print("-> Something wrong happened. Try again.")
print("----------------------------------------------\n")
```

3. Outliers are filtered out, and the data is normalized (mean-adjusted to 0). To verify normalization, 10 sample points are examined, and the results are printed.

```python
# Convert to Pandas DataFrame and NumPy Array
Q1_df = pd.DataFrame(Q1_normalized, columns=['Normalized Q1'])
Q2_df = pd.DataFrame(Q2_normalized, columns=['Normalized Q2'])

print("\n-----------------------------------------------")
print("10 Random Rows from Pandas Q1 DataFrame:\n", Q1_df.sample(10))
print("\n10 Random Rows from Pandas Q2 DataFrame:\n", Q2_df.sample(10))
print("-----------------------------------------------\n")

# Convert to NumPy arrays
Q1_array = Q1_df.to_numpy()
Q2_array = Q2_df.to_numpy()

# Split into X (input) and Y (output) data
X_data = Q1_array[:, 0]
Y_data = Q2_array[:, 0]

print("\n-----------------------------------------------")
print("X_data (First 10 values):\n", X_data[:10])
print("\nY_data (First 10 values):\n", Y_data[:10])
print("-----------------------------------------------\n")
```

4. The normalized Q1 and Q2 data are converted into Pandas DataFrames, and 10 samples are printed. The data is then converted into NumPy arrays and split into X_data and Y_data, which are also printed.

Execution Result:

```
-------------------------------------------
Mean of "Q1.csv" for each column: [-1.00025244e+06  8.14976561e-01]
Standard Deviation of "Q1.csv for each column": [8.94854116e+05 5.77423258e+02]

Mean of "Q2.csv for each column": [-1.00025244e+06  4.07488280e-01 -1.00025485e+06]
Standard Deviation of "Q2.csv for each column": [8.94854116e+05 2.88711629e+02 8.94854124e+05]
-------------------------------------------



-------------------------------------------
In Q1.csv:
There are Outliers (beyond 2 std dev): [-2806081.186528 -2904254.3217   -2906391.774592 ... -2849061.772288
 -2897663.877607 -2845764.953152]

In Q2.csv:
There are Outliers (beyond 2 std dev): [-2806081.186528 -2806566.798528 -2904254.3217   ... -2897174.521107
 -2845764.953152 -2845280.017152]
-------------------------------------------



-------------------------------------------
10 Random Samples from Normalized Q1 Data:
 [ 0.16351725  0.16372897 -0.5300075   0.15066496  0.00924872  0.1636684
  0.16355951  0.16367543 -0.45303162  0.16383072]
-> Well converted!

10 Random Samples from Normalized Q2 Data:
 [ 0.21943107 -0.55585298 -0.6544563   0.21934774  0.21942867  0.21931486
  0.21632296 -0.68860641 -0.75987827  0.18249025]
-> Well converted!
-------------------------------------------
```

```
------------------------------------------------
10 Random Rows from Pandas Q1 DataFrame:
         Normalized Q1
1126890      -0.502454
1175284       0.163647
345833        0.163625
421532        0.146082
1160408       0.163844
1248581       0.157415
1268522      -0.449691
1111782       0.163352
383183       -0.155315
137132        0.163703

10 Random Rows from Pandas Q2 DataFrame:
         Normalized Q2
2909165      -0.113880
1912608      -0.305928
2405480       0.219330
2755851       0.219582
1672119       0.026685
2910102      -0.670599
2555483      -0.188577
1438555       0.219535
1746823       0.202624
2208966       0.219335
------------------------------------------------


------------------------------------------------
X_data (First 10 values):
 [-0.11896584  0.16372602 -0.47070301  0.1632671  -0.22861861  0.16375875
 -0.61942838  0.16384813 -0.28142199  0.1633118 ]

Y_data (First 10 values):
 [-0.06312597  0.21952408 -0.06321856 -0.41492542  0.21929458 -0.41478851
 -0.17279816  0.21954044 -0.17290712 -0.56367713]
------------------------------------------------
```

5) `dot_product.py`: This file performs vector and matrix multiplication and analyzes performance differences.

```python
# Isla Kim

import data_loader
import numpy as np
import time

# Defines a function that calculate dot product
def dot_product_manual(a, b):
    result = 0
    for i in range(len(a)):
        result += a[i] * b[i]
    return result

# 1. Load the data from L1.csv and Q1.csv and save them in python array
L1_python_array = data_loader.csv_file_loader("L1.csv")
Q1_python_array = data_loader.csv_file_loader("Q1.csv")

# Loads the x values from "L1.csv" and from "Q1.csv"
L1_python_x = [float(row[0]) for row in L1_python_array]
Q1_python_x = [float(row[0]) for row in Q1_python_array]

# Calculate how long did it take while processing element-wise multiplication in python array
start_time = time.time()
dot_product_result = dot_product_manual(L1_python_x, Q1_python_x)
end_time = time.time()

# Print the result of time and element-wise multiplication
print("\nExecution Time while processing element-wise multiplicaition of python arrays in L1.csv and Q1.csv: {:.4f} seconds".format(end_time - start_time))
print("----------------------------------------")
```

1. It loads data from L1.csv and Q1.csv into Python arrays and extracts x values. The time taken for element-wise multiplication is measured and printed.

```python
# 2. Load the data from L1.csv and Q1.csv and save them in numpy array
L1_numpy_array = np.array(data_loader.csv_file_loader("L1.csv"), dtype=float)
Q1_numpy_array = np.array(data_loader.csv_file_loader("Q1.csv"), dtype=float)

# Loads the x values from "L1.csv" and from "Q1.csv"
L1_numpy_x = L1_numpy_array[:, 0]
Q1_numpy_x = Q1_numpy_array[:, 0]

# Calculate how long did it take while processing element-wise multiplication in numpy arrays
start_time = time.time()
dot_product_result = np.dot(L1_numpy_x, Q1_numpy_x)
end_time = time.time()

# Print the result of time and element-wise multiplication
print("Execution Time while processing element-wise multiplicaition of numpy arrays in L1.csv and Q1.csv: {:.4f} seconds".format(end_time - start_time))
print("----------------------------------------")
```

2. It loads data from L1.csv and Q1.csv into NumPy arrays and extracts x values. The time taken for element-wise multiplication is measured and printed.

```python
# 3. Load the data from Q2.csv and save them in python array and numpy array
Q2_python_array = data_loader.csv_file_loader("Q2.csv")
Q2_numpy_array = np.array(data_loader.csv_file_loader("Q2.csv"), dtype=float)

# Calculate how long did it take while processing matrix multiplication in python arrays
python_start_time = time.time()
Q2_python_array = [[float(value) for value in row] for row in Q2_python_array] # All values in Q2_python_array are transformed to float type
matrix_multi_result = [
    sum(Q2_python_array[i][j] * Q2_python_array[i][j] for j in range(len(Q2_python_array[0])))
    for i in range(len(Q2_python_array))
]
python_end_time = time.time()

# Print the result of time and matrix multiplication in python arrays
print("Execution Time while processing matrix multiplicaition of python arrays in Q2.csv: {:.4f} seconds".format(python_end_time - python_start_time))
print("----------------------------------------")
```

3. It loads data from Q2.csv into Python and NumPy arrays, then performs matrix multiplication on the Python array and measures the computation time.

```
# 4. Calculate how long did it take while processing matrix multiplication in numpy arrays
numpy_start_time = time.time()
try:
    matrix_multi_result = np.einsum("ij,ij->i", Q2_numpy_array, Q2_numpy_array)
except:
    print(f"Unable to allocate memory for the matrix multiplication: numpy.core._exceptions._ArrayMemoryError")
    print("It takes up too much space like 7.28 TiB when you calculate matrix multiplication in large arrays.\n")
else:
    numpy_end_time = time.time()

    # Print the result of time and matrix multiplication in python arrays
    print("Execution Time while processing matrix multiplicaition of numpy arrays in Q2.csv: {:.4f} seconds.\n".format(numpy_end_time - numpy_start_time))
```

4. It loads data from Q2.csv into Python and NumPy arrays, then performs matrix multiplication on the NumPy array and measures the computation time. When attempting to use the `np.dot()` function, a memory shortage issue occurred due to loading and computing all data at once. Since full matrix multiplication was unnecessary and only row-wise computations were required, the `np.einsum()` function was used to reduce memory consumption.

Execution Result:

```
Execution Time while processing element-wise multiplicaition of python arrays in L1.csv and Q1.csv: 0.0480 seconds
----------------------------------------
Execution Time while processing element-wise multiplicaition of numpy arrays in L1.csv and Q1.csv: 0.0010 seconds
----------------------------------------
Execution Time while processing matrix multiplicaition of python arrays in Q2.csv: 2.6201 seconds
----------------------------------------
Execution Time while processing matrix multiplicaition of numpy arrays in Q2.csv: 0.0126 seconds.
```

2. Result Analysis:

From the computation results in **dot_product.py**,

```
Execution Time while processing element-wise multiplicaition of python arrays in L1.csv and Q1.csv: 0.0480 seconds
----------------------------------------
Execution Time while processing element-wise multiplicaition of numpy arrays in L1.csv and Q1.csv: 0.0010 seconds
----------------------------------------
Execution Time while processing matrix multiplicaition of python arrays in Q2.csv: 2.6201 seconds
----------------------------------------
Execution Time while processing matrix multiplicaition of numpy arrays in Q2.csv: 0.0126 seconds.
```

When storing data from the L1.csv and Q1.csv files in a Python array and measuring the execution time for element-wise multiplication, it took 0.0480 seconds, whereas storing the data in a NumPy array and performing the same operation took only 0.0010 seconds. Additionally, for matrix multiplication, storing the data in a Python array took 2.6201 seconds, while storing it in a NumPy array took only 0.0126 seconds. This demonstrates that storing data in a NumPy array is significantly faster than storing it in a Python array for the same computations.

The first reason for this difference is the structural differences between the data storage methods. A Python list is a general object array where each element is stored in a separate memory location and accessed via pointers. Therefore, additional processing is required to interpret references to each element during computation, which slows down performance.
In contrast, a NumPy array is a fixed-type data array stored in a contiguous memory block. Since all elements have the same data type and can be accessed directly in memory, operations are much faster.

The second reason is the difference in computation methods. Python lists rely on loop-based operations (for-loops), meaning the interpreter must process each element individually. However, NumPy employs C-based optimizations, such as vectorized operations, and utilizes SIMD (Single Instruction, Multiple Data) instructions to significantly accelerate computation.
Since NumPy operations are executed as compiled C code rather than interpreted Python loops, they are much faster.

The third reason is memory management differences. Python lists store dynamically typed objects, meaning each element is stored as a pointer to a memory location. This leads to inefficient memory access since multiple locations must be referenced during computation, resulting in cache inefficiencies. On the other hand, NumPy arrays store data in a contiguous C memory block, where only values of the same type are stored, making memory access much more efficient.

Therefore, using NumPy provides significantly better performance, especially for large-scale computations.