**Homework 7**

과목명 : 컴퓨팅 사고와 파이썬 프로그래밍

담당 교수 : 김영탁

수강 번호 : 1541

제출 일자 : 2020년 10월 14일

제출자 : 21611817 박 준 형

**7.1**

최소 10개 국가의 기본 정보 (국가 이름, 수도 이름, 인구수, 면적) 데이터를 텍스트 파일 (demography.txt)에 준비하고, 이 파일의 국가 기본 정보를 읽어 들인 후 순차적으로 화면에 출력하는 파이썬 프로그램을 작성하라. 한 줄에 한 국가씩 출력할 것. 데이터 파일로 부터 읽어 들인 국가들의 기본 정보에서 인구 수를 기준으로 내림차순 정렬을 하고, 그 순서대로 국가 기본 정보를 출력하는 파이썬 프로그램을 작성하라.

**- Source Code**

#7.1

Country\_info = []

f1 = open("demography.txt", 'r')

# 파일로 부터 정보를 읽어 리스트에 저장 #

for line in f1:

    name, capital, population, area = line.split()

    tmp = [name, capital, int(population), int(area)]

    Country\_info.append(tmp)

f1.close()

# 리스트에 담긴 국가 기본 정보 출력 #

count = 0

print("List of countries")

for info in Country\_info:

    print("Country[%2d] = " %(count), end = "")

    print(info)

    count += 1

# 인구수를 기준으로 내림차순 정렬 #

Country\_info = sorted(Country\_info, key=lambda x : x[2], reverse=True)

# 정렬 후 국가 기본 정보 출력 #

count = 0

print("\n\nList of countries sorted by demography(number of people")

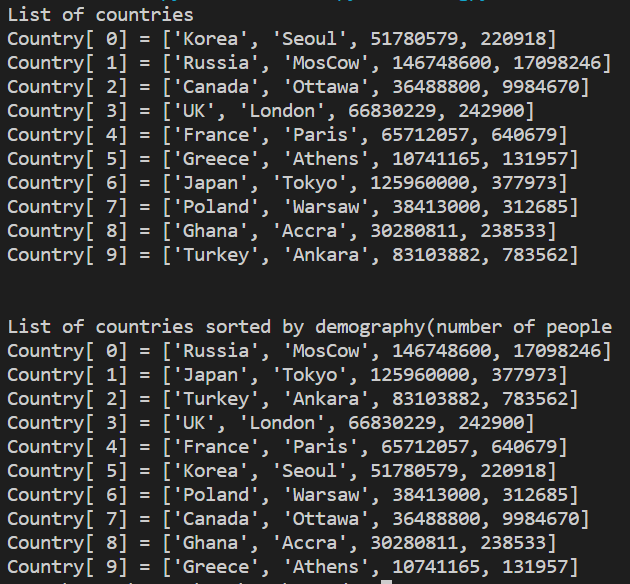
for info in Country\_info:

    print("Country[%2d] = " %(count), end = "")

    print(info)

    count += 1

**- 결과창**



**7.2**

최소 10명의 학생 정보인 (학생 이름, 학번, 국어점수, 영어점수, 수학점수, 과학점수) 데이터를 텍스트 파일 student\_records.txt에 준비하고, 이 파일을 읽어 들인 후, 각 학생의 평균점수를 계산하여 학생 정보에 추가하고, 각 과목별로 학생들의 성적을 종합하여 평균 점수를 계산한 후, 결과를 output.txt 텍스트 파일에 출력하는 파이썬 프로그램을 작성하라.

**- Source Code**

#7.2

def fread\_data(file\_name): # 파일에 있는 정보를 읽어서 이중리스트로 반환해주는 함수

    data\_list = []

    f = open(file\_name, 'r') # 파일을 읽기 모드로 열고 새로운 파일 객체 f생성.

    for line in f: # f2.readlines()과 동일한 역할 -> 한줄씩 뽑아서 각 리스트원소로 만들어줌

        name, ID, kor, eng, math, science = line.split()

        tmp = [name, int(ID), int(kor), int(eng), int(math), int(science)]

        data\_list.append(tmp)

    f.close()

    return data\_list

def fwrite\_data(file\_name, data\_list): # fime\_name이라는 파일에 data\_list라는 리스트 쓰는 함수

    f = open(file\_name, 'w', -1, "utf-8") # 파일을 쓰기 모드로 열고 새로운 파일 객체 f생성

    f.write(" name,       ID,   kor,  eng,  math,  sci,  sum,  avg\n")

    f.write("‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐\n")

    for data in data\_list: # data\_list의 정보를 하나씩 쓰기

        s = ""

        s = "{0:<5s},".format(data[0])

        s += "{0:>10},".format(data[1])

        s += "{0:>5},".format(data[2])

        s += "{0:>5},".format(data[3])

        s += "{0:>5},".format(data[4])

        s += "{0:>5},".format(data[5])

        s += "{0:6d}".format(data[6])

        s += "{0:8.2f}".format(data[7])

        s += '\n'

        f.write(s)

    f.close()

def calculate\_score(data\_list): # 각 학생의 성적 합과 평균을 계산하여 리스트의 원소로 추가해주는 함수

    i = 0

    for name, ID, kor, eng, math, science in data\_list: # 원소차례대로 data\_list[i][0] = name, data\_list[i][1] = kor, data\_list[i][2] = eng, data\_list[i][3] = math

        sumScore = kor + eng + math + science

        data\_list[i].append(sumScore)

        data\_list[i].append(sumScore/4.0)

        i = i + 1

#########################################

# Application

f\_st\_name = "student\_records.txt"

students = fread\_data(f\_st\_name) # 파일에 있는 정보를 읽어와서 students[]에 저장

print("\nStudent records read from {} :".format(f\_st\_name))

for st in students: # student리스트에 담긴 정보를 한줄씩 출력

    print(st)

calculate\_score(students) # 성적의 합과 평균을 각 리스트에 추가

print("\nAfter calculate\_score(students)")

print(" name,       ID,   kor,  eng,  math,  sci,  sum,  avg")

print("‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐")

for st in students: # 출력 포맷에 맞게 학생들의 정보 출력

    s = ""

    s = "{0:<5s},".format(st[0])

    s += "{0:>10},".format(st[1])

    s += "{0:>5},".format(st[2])

    s += "{0:>5},".format(st[3])

    s += "{0:>5},".format(st[4])

    s += "{0:>5},".format(st[5])

    s += "{0:6d}".format(st[6])

    s += "{0:8.2f}".format(st[7])

    print(s)

fwrite\_data("output.txt", students) # students리스트를 result\_student.txt파일에 쓰는 함수

# 파일에서 읽어 오는 부분 #

print("\nAfter calculate\_scores({}):".format("Students"))

f = open("output.txt", 'r', -1, "utf-8")

for line in f:

    print(line)

f.close()

# 각 과목별 학생들의 성적 평균 점수 출력 #

sum = [0] \* 4

for i in range(len(students)):

    sum[0] += students[i][2]

    sum[1] += students[i][3]

    sum[2] += students[i][4]

    sum[3] += students[i][5]

print("Average score of each class")

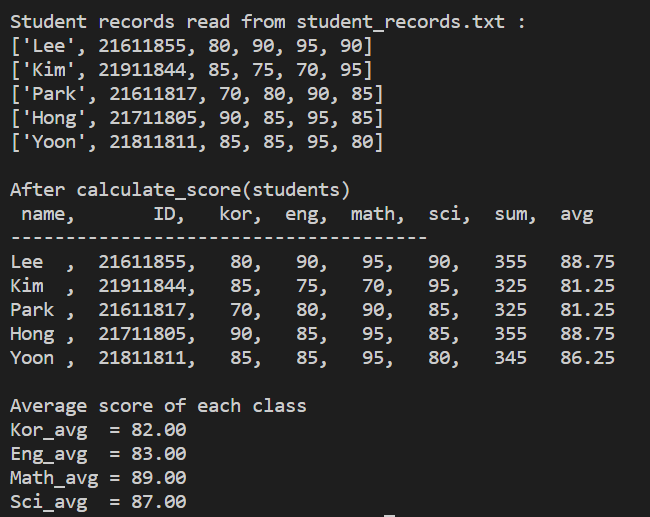
print("Kor\_avg  = %.2f" %(sum[0]/len(students)))

print("Eng\_avg  = %.2f" %(sum[1]/len(students)))

print("Math\_avg = %.2f" %(sum[2]/len(students)))

print("Sci\_avg  = %.2f" %(sum[3]/len(students)))

**- 결과창**



**7.3**

행렬 (matrix)의 초기화 및 덧셈, 뺄셈, 곱셈 및 출력 기능을 메소드로 가지는 class MyMtrx를 파이썬 프로그램으로 작성하고, 이를 myClassMtrx.py에 저장하라. class MyMtrx의 초기화를 담당하는 멤버함수 \_\_init\_\_(self, name, num\_rows, num\_cols, list\_data)에는 행렬의 이름 (name), 행의 개수 (num\_rows), 열의 개수 (num\_cols), 원소 데이터를 포함하는 리스트

(list\_data)가 전달된다. 행렬을 출력하는 print(self)는 행렬의 원소들을 콘솔로 행 단위로 한줄씩 출력한다. 행렬의 덧셈, 뺄셈, 곱셈 연산은 ‘+’, ‘-’, ‘\*’ 연산자를 사용할 수 있도록 연산

자 오버로딩 (operator overloading) 함수로 구현하며, \_\_add\_\_(self, other), \_\_sub\_\_(self, other), \_\_mul\_\_(self, other)의 함수 원형을 가지며, 연산 결과를 class MyMtrx의 객체로 반환하도록 구현하여야 한다.

**- Source Code**

**1) myClassMtrx.py**

class Mtrx:

    def \_\_init\_\_(self, name, n\_row, n\_col, L\_data): #직접 구현

        self.name = name

        self.n\_row = n\_row # 행

        self.n\_col = n\_col # 열

        self.L\_data = L\_data

    def \_\_str\_\_(self): #직접 구현

        s = "\n"

        for i in range(self.n\_row):

            for j in range(self.n\_col):

                s += "{:5.1f}".format(self.L\_data[i][j], end=" ")

            s += "\n"

        return s

    def \_\_add\_\_(self, other): # operator overloading of '+'

        result = [[0 for i in range(self.n\_col)] for i in range(self.n\_row)]

        for i in range(self.n\_row):

            for j in range(self.n\_col):

                result[i][j] = self.L\_data[i][j] + other.L\_data[i][j]

        return Mtrx("mtrxAdd", self.n\_row, self.n\_col, result)

    def \_\_sub\_\_(self, other): # operator overloading of '-'

        result = [[0 for i in range(self.n\_col)] for i in range(self.n\_row)]

        for i in range(self.n\_row):

            for j in range(self.n\_col):

                result[i][j] = self.L\_data[i][j] - other.L\_data[i][j]

        return Mtrx("mtrxSub", self.n\_row, self.n\_col, result)

    def \_\_mul\_\_(self, other): # operator overloading of '\*'

        result = [[0 for i in range(other.n\_col)] for i in range(self.n\_row)]

        for i in range(self.n\_row):

            for j in range(other.n\_col):

                for k in range(self.n\_col):

                    result[i][j] += self.L\_data[i][k] \* other.L\_data[k][j]

        return Mtrx("mtrxMul", self.n\_row, other.n\_col, result)

**7.4**

행렬의 크기 num\_rows과 num\_columns을 정수 데이터로 지정하며, num\_rows × num\_columns 크기의 행렬 데이터들을 실수 자료형으로 포함하는 텍스트 파일 (matrix\_data.txt)을 준비하라. 텍스트 파일에선 2개의 행렬 데이터를 가지며, 각 행렬마다 행의 개수 (num\_rows)과 열의 개수(num\_columns) 및 num\_rows×num\_columns개의 실수형 (float) 행렬 원소

를 가진다. 텍스트 파일로부터 2개를 행렬 크기 및 행렬 데이터를 읽어class MyMtrx의

객체 mA와 mB에 각각 저장하고, mC = mA + mB, mD = mA – mB, mE = mA × mB를 각각 계산하여 화면으로 출력하는 파이썬 응용 프로그램(test\_myClassMtrx.py)을 작성하라.

이 파이썬 응용 프로그램은 7.3에서 작성한 myClassMtrx.py 모듈을 import하여 사용한다.

파이썬 응용 프로그램 (test\_myClassMtrx.py)에서 class MyMtrx의 객체 mA를 json.dump() 함수를 사용하여 텍스트 파일 mA\_json.txt에 저장하라. 이때, 사용자 정의 클래스 객체를 JSON serialization을 수행하기 위하여 다음과 같은 Class CustomEncoder를 사용하라.

**- Source Code**

**1) main.py**

#7.4

# Application Program of myClassMtrx

import json, pickle

import os, os.path

import myClassMtrx

import CustomJsonEncoder

def main():

    # prepare mA, mB #

    data\_1 = []

    data\_2 = []

    f = open("matrix\_data.txt", 'r')

    count = 0

    for line in f:

        if count == 0:

            num\_rows\_A, num\_columns\_A = map(int, line.split())

        elif count == 6:

            num\_rows\_B, num\_columns\_B = map(int, line.split())

        elif 0 < count < 6:

            e0, e1, e2, e3, e4 = map(float, line.split())

            tmp = [e0, e1, e2, e3, e4]

            data\_1.append(tmp)

        else:

            e0, e1, e2, e3, e4 = map(float, line.split())

            tmp = [e0, e1, e2, e3, e4]

            data\_2.append(tmp)

        count += 1

    f.close()

    # operations for mC, mD, mE #

    mA = myClassMtrx.Mtrx("MA", num\_rows\_A, num\_columns\_A, data\_1)

    print("n\_row = %d, n\_col = %d" %(num\_rows\_A, num\_columns\_A))

    print("mA = ", mA)

    print("n\_row = %d, n\_col = %d" %(num\_rows\_B, num\_columns\_B))

    mB = myClassMtrx.Mtrx("M2", num\_rows\_B, num\_columns\_B, data\_2)

    print("mB = ", mB)

    mC = mA + mB

    print("mC = mA + mB =", mC)

    mD = mA - mB

    print("mD = mA ‐ mB =", mD)

    mE = mA \* mB

    print("mE = mA \* mB =", mE)

    # json을 이용해 텍스트 파일에 저장 #

    f\_json = open("mA\_json.txt", "w")

    json.dump(mA, f\_json, indent=4, cls=CustomJsonEncoder.CustomEncoder)

    f\_json.close()

    size\_f\_json = os.path.getsize("mA\_json.txt")

    print("size of mA\_json.txt = ", size\_f\_json)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**2)CustomJsonEncoder.py**

# CustomJsonEncoder for JSON serialization

import json

from datetime import datetime

class CustomEncoder(json.JSONEncoder):

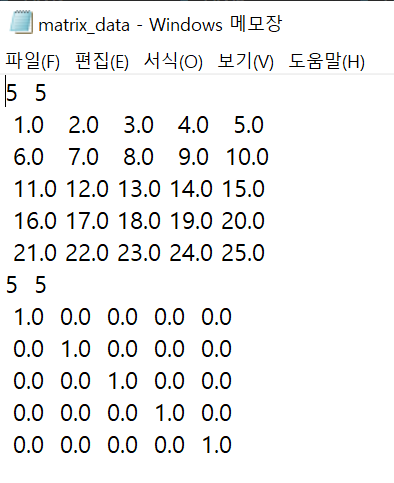
    def default(self, o):

        if isinstance(o, datetime):

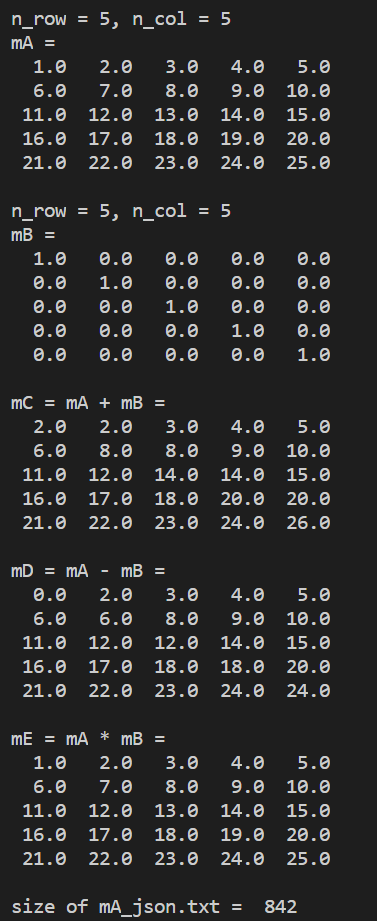
            return {'\_\_datetime\_\_': o.replace(microsecond=0).isoformat()}

        return {'\_\_{}\_\_'.format(o.\_\_class\_\_.\_\_name\_\_): o.\_\_dict\_\_}

**3) matrix\_data.txt**

****

**- 결과창**



**7.5**

위 homework 7.4에서 생성된 mA를 pickle 파일 (mA\_pickle.bin)에 저장하고, 파일의 크기를 7.4에서 생성된 텍스트 파일 mA\_json.txt와 비교하라. 그리고 왜 크기에 차이가 나는지에 대하여 설명하라.

**- Source Code**

#7.5

import json, pickle

import os, os.path

import myClassMtrx

import CustomJsonEncoder

def main():

    # prepare mA, mB #

    data\_1 = []

    data\_2 = []

    f = open("matrix\_data.txt", 'r')

    count = 0

    for line in f:

        if count == 0:

            num\_rows\_A, num\_columns\_A = map(int, line.split())

        elif count == 6:

            num\_rows\_B, num\_columns\_B = map(int, line.split())

        elif 0 < count < 6:

            e0, e1, e2, e3, e4 = map(float, line.split())

            tmp = [e0, e1, e2, e3, e4]

            data\_1.append(tmp)

        else:

            e0, e1, e2, e3, e4 = map(float, line.split())

            tmp = [e0, e1, e2, e3, e4]

            data\_2.append(tmp)

        count += 1

    f.close()

    # operations for mC, mD, mE #

    mA = myClassMtrx.Mtrx("MA", num\_rows\_A, num\_columns\_A, data\_1)

    print("n\_row = %d, n\_col = %d" %(num\_rows\_A, num\_columns\_A))

    print("mA = ", mA)

    print("n\_row = %d, n\_col = %d" %(num\_rows\_B, num\_columns\_B))

    mB = myClassMtrx.Mtrx("M2", num\_rows\_B, num\_columns\_B, data\_2)

    print("mB = ", mB)

    mC = mA + mB

    print("mC = mA + mB =", mC)

    mD = mA - mB

    print("mD = mA ‐ mB =", mD)

    mE = mA \* mB

    print("mE = mA \* mB =", mE)

    # mA를 json을 이용해 텍스트 파일에 저장 #

    f\_json = open("mA\_json.txt", "w")

    json.dump(mA, f\_json, indent=4, cls=CustomJsonEncoder.CustomEncoder)

    f\_json.close()

    size\_f\_json = os.path.getsize("mA\_json.txt")

    print("size of mA\_json.txt = ", size\_f\_json)

    # mA를 pickle파일에 저장 #

    f\_pickle = open("mA\_pickle.bin", "wb")

    pickle.dump(mA, f\_pickle)

    f\_pickle.close()

    size\_f\_pickle = os.path.getsize("mA\_pickle.bin")

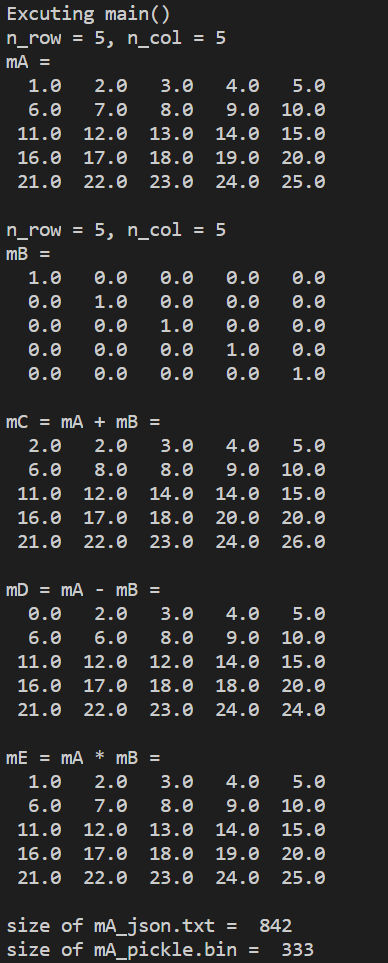
    print("size of mA\_pickle.bin = ", size\_f\_pickle)

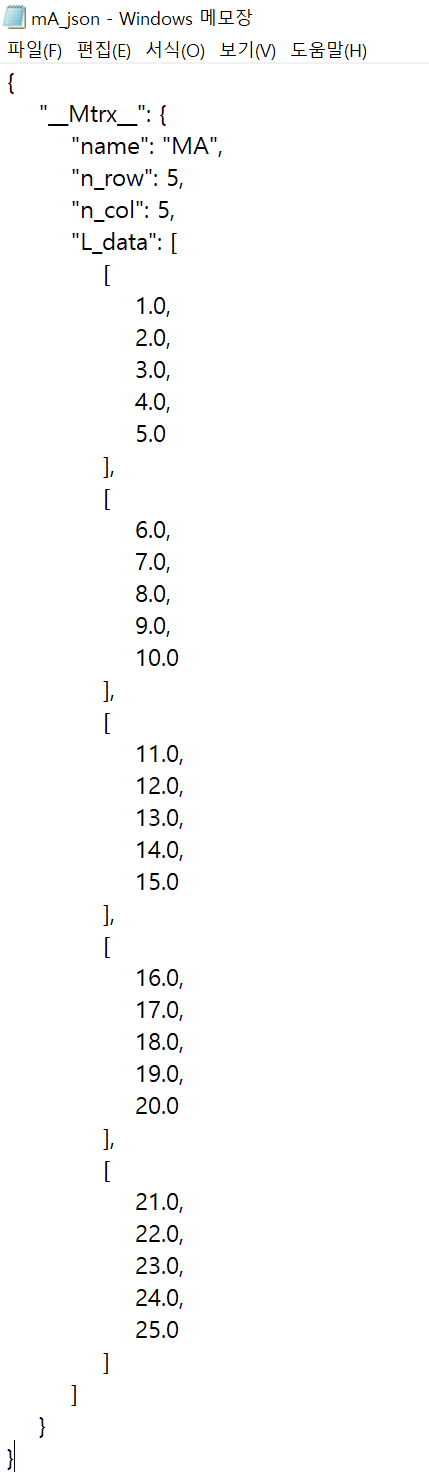
if \_\_name\_\_ == "\_\_main\_\_":

    print("Excuting main()")

    main()

**- 결과창**





Q) json파일과 pickle파일 간의 크기가 차이가 나는 이유

⇒ json파일은 텍스트 직렬화 방식으로 사람이 읽을 수 있지만 pickle은 바이너리 직렬화 방식으로 읽을 수 없다. 또한 json은 파이썬 이외에 다른 환경에서도 널리 사용되는 등의 외부 표준에 의해 부과된 제약이 없는 반면, pickle은 오직 파이썬에서만 사용가능하여 비 파이썬 프로그램은 피클 된 파이썬 객체를 재구성할 수 없다. 그렇기에 pickle데이터 포맷은 상대적으로 간결한 바이너리 표현을 사용하여 효율적으로 데이터를 압출할 수 있고, 그에 비해 Json은 pickle보다 부가적인 정보를 더 많이 담고 있기에 pickle보다 상대적으로 큰 크기를 갖는다.