**CMPE 30052**

**Data Structures and Algorithms**

**Final Project Requirement**

**Transcript Generation System**

**Group 2:**

LIM, SHIN I.

ILAO, KENJI C.

ESGUERRA, EDGAR JR. P.

MINA, SIEGFRED LORELLE C.

**Course/Year/Section:**

BSCOE 2-1

**Class Schedule:**

3:00 – 5:00 PM | WEDNESDAY

**Date Submitted:**

FEBRUARY 28, 2023

**Submitted to:**

CANSINO, JULIUS S

"""

This work is done by Group 2

Edgar Esguerra Jr. - 2021-05780-MN-0 - 25%

Kenji Ilao - 2021-05784-MN-0 - 25%

Shin Lim - 2021-05789-MN-0 - 25%

Siegfred Lorelle Mina - 2021-05794-MN-0 - 25%

"""

import csv

import sys

import os

from datetime import date, datetime

def main():

    """ Initialize the App upon starting the program """

    App()

class App:

    def \_\_init\_\_(self):

        """ Initialize all variables, then assign them in start feature """

        self.student\_level = set()

        self.student\_type = set()

        self.student\_id = ""

        self.student = {

            "Name": "",

            "Levels": set(),

            "Colleges": set(),

            "Departments": set(),

            "Num of terms": 0,

        }

        self.student\_list = []

        self.student\_grades = []

        self.history = []

        self.num\_of\_request = 0

        # Read student details CSV file, save each row as a dictionary then append that dictionary to student list

        self.setStudentList()

        # Call the startFeature method to assign data about student level, type, and id, and other information about the student

        self.startFeature()

    def startFeature(self):

        """ Prompt user for their student level, type. load student data from csv files. """

        # Resets the variables to give space for new student

        self.resetVariables()

        # Prompt user to select their student level and type/degree

        self.setStudentLevel()

        # Prompt user to enter their student ID

        self.setStudentID()

        # Ensure that the given student level, degree and id exists in the student details

        if not self.isStudentRegistered():

            print("\nA student with the given information doesn't exists. Try Again.")

            buffer()

            return self.startFeature()

        # Read the he student details csv of the which contains all students, save only the information relevant to the user

        self.setStudentInfos()

        # Read the csv file of the student's grade/record, then save it to student grades

        self.setStudentGrade()

        # Read the txt file of the student's previous requests, then save it to  history

        self.setHistory()

        # Pause the program then clear the screen then redirect to menu screen

        buffer()

        self.menuFeature()

    def resetVariables(self):

        """ Resets the variables to give space for new student """

        self.student\_level.clear()

        self.student\_type.clear()

        self.student["Name"] = ""

        self.student["Levels"].clear()

        self.student["Colleges"].clear()

        self.student["Departments"].clear()

        self.student["Num of terms"] = 0

        self.student\_grades.clear()

        self.history.clear()

    def setStudentList(self):

        """ Read student details CSV file, save each row as a dictionary then append that dictionary to student l """

        with open('studentDetails.csv') as file:

            reader = csv.DictReader(file)

            for row in reader:

                self.student\_list.append(row)

    def setStudentLevel(self):

        """ Prompt user to select student level and then ask for type/degree if they have one """

        LEVELS = {

            "U": "Undergraduate",

            "G": "Graduate",

        }

        # Asks for their student level

        print("\nSelect Student Level:\nUndergraduate (U)\nGraduate (G)\nBoth (B)")

        while True:

            level = input("\nChoice: ").upper()

            if level in LEVELS:

                self.student\_level.add(level)

                break

            elif level == "B":

                self.student\_level.update(["U", "G"])

                break

            print("\nPlease use: U/G/B")

        if "U" in self.student\_level:

            self.student\_type.add("BS")

        if "G" in self.student\_level:

            self.setDegreeLevel()

    def setDegreeLevel(self):

        """ Prompt user to enter degree level """

        DEGREES = {

            "M": "Master",

            "D": "Doctorate",

        }

        # Ask for their degree/type

        print("\nSelect your level type:\nMaster (M)\nDoctorate (D)\nBoth (B0)")

        while True:

            degree = input("\nChoice: ").upper()

            if degree in DEGREES:

                self.student\_type.add(degree)

                break

            elif degree == "B0":

                self.student\_type.update({"M", "D"})

                break

            print("\nPlease use: M/D/B0")

    def setStudentID(self):

        """ Prompt user to enter student ID and check if it exists in the data list """

        # Prompt for the new student id

        self.student\_id = input("\nEnter your Student ID: ")

        # Check if the given student id is registered

        for student in self.student\_list:

            if student['stdID'] == self.student\_id:

                return

        return self.setStudentID()

    def setStudentInfos(self):

        """ Save student information relevant to the user """

        # Loop through the student list to find information relevant to the user

        for student in self.student\_list:

            if student["stdID"] != self.student\_id:

                continue

            if student["Level"] not in self.student\_level:

                continue

            # Degrees in the student details csv has digits, so strip them to match degree inputted by the user

            degree\_without\_digits = ''.join(i for i in student["Degree"] if not i.isdigit()).strip()

            if degree\_without\_digits not in self.student\_type:

                continue

            # If student matches the information given by user then save the information in student dictionary

            self.student["Name"] = student["Name"]

            self.student["Levels"].add(student["Level"])

            self.student["Colleges"].add(student["College"])

            self.student["Departments"].add(student["Department"])

            self.student["Num of terms"] += int(student["Terms"])

    def isStudentRegistered(self):

        """ Checks if the student information given is registered in the student details csv """

        for student in self.student\_list:

            if student["stdID"] != self.student\_id:

                continue

            if student["Level"] not in self.student\_level:

                continue

            degrees\_without\_digits = ''.join(i for i in student["Degree"] if not i.isdigit()).strip()

            if degrees\_without\_digits not in self.student\_type:

                continue

            return True

        return False

    def setStudentGrade(self):

        """ Read the csv file of the student, then save it to student grades """

        # Read the record/grade of the student, save each row as dictionary, then append it to student grades

        with open(f'{self.student\_id}.csv', "r") as file:

            reader = csv.DictReader(file)

            for row in reader:

                degree\_without\_digits = ''.join(i for i in row["Degree"] if not i.isdigit()).strip()

                if row["Level"] in self.student\_level and degree\_without\_digits in self.student\_type:

                    self.student\_grades.append(row)

        # Convert term and grade into int

        for data in self.student\_grades:

            data["Term"] = int(data["Term"])

            data["Grade"] = int(data["Grade"])

        # [print(i) for i in self.student\_grades]

    def setHistory(self):

        """ Read the csv file of the student's previous request and save it in history """

        filename = f"std{self.student\_id}PreviousRequest.txt"

        # If the previous request txt file of the student doesn't exists then do nothing

        if not os.path.exists(filename):

            return

        # Open the previous request txt file of the student

        with open(filename, 'r') as file:

            # Skip the first two line (header)

            reader = file.readlines()[2:]

            # Loop through each line

            for line in reader:

                # Separate data in a line by space

                line = line.split()

                # Via request dictionary, save each data to their appropriate header value

                request = {

                    "req": line[0],

                    "date": line[1],

                    "time": line[2]

                }

                # Append the request in history

                self.history.append(request)

    def saveHistory(self):

        """ Save previous requests of the student via txt file """

        # Create a txt file with a the student's name and previous request in its filename, if it exists, then overwrite it

        with open(f"std{self.student\_id}PreviousRequest.txt", 'w') as file:

            # Write the header

            file.write("{:^12} {:^15} {:^7}\n".format('Request','Date','Time'))

            file.write("=========================================\n")

            # Loop through history and write each request

            for request in self.history:

                file.write("{:^12} {:^15} {:^7}\n".format(request['req'], request['date'], request['time']))

    def recordRequest(self, request):

        """ Record this request with the current date and time, save it in history list via append """

        today = date.today()

        now = datetime.now()

        new\_h = {

            "req": request,

            "date": str(today.strftime("%d/%m/%Y")),

            "time": f"{now.strftime('%H:%M')}"

        }

        self.history.append(new\_h)

        self.saveHistory()

        # Increment the number of requests

        self.num\_of\_request += 1

    def menuFeature(self):

        """ Show all available options in menu. Asks for a choice, then redirect to that feature """

        print('Student Transcript Generation System\n====================================================\n1. Student Details\n2. Statistics\n3. Transcript based on major courses\n4. Transcript based on minor courses\n5. Full transcript\n6. Previous transcript requests\n7. Select another student\n8. Terminate system\n====================================================')

        # Asks the user what to do, then redirects it to that feature

        self.menuManager()

    def detailsFeature(self, student):

        """ Display Details and save it in a text file """

        text = (

            f"Name: {student['Name']}"

            f"\nstdID: {self.student\_id}"

            f"\nLevel(s): {', '.join(student['Levels'])}"

            f"\nNumber of Terms: {student['Num of terms']}"

            f"\nCollege(s): {', '.join(student['Colleges'])}"

            f"\nDepartment(s): {', '.join(student['Departments'])}"

        )

        print(text)

        with open(f'std{self.student\_id}Details.txt', 'w') as file:

            file.write(text)

        # Record this request

        self.recordRequest("Details")

    def statisticsFeature(self, student\_level, student\_id):

        """ Show some statistics about the student's grade/record. Examples are average grade per term, minimum and maximum grades. """

        text= ""

        for level in sorted(self.student["Levels"], reverse=True):

            # Initialize variables

            grades\_info = [grade for grade in self.student\_grades if grade["Level"] == level]

            grades = [grade["Grade"] for grade in grades\_info]

            average = int(sum(grades) / len(grades))

            max\_grade = max(grades)

            min\_grade = min(grades)

            terms\_with\_max\_grade = {str(grade["Term"]) for grade in grades\_info if grade["Grade"] == max\_grade}

            terms\_with\_min\_grade = {str(grade["Term"]) for grade in grades\_info if grade["Grade"] == min\_grade}

            last\_term = max([grade["Term"] for grade in grades\_info])

            # Change the text header depending on what level (undergraduate for U, graduate for G)

            if level == "U":

                # Save and print the header

                text += (

                    "===================================================================\n"

                    f"Undergraduate Level\n"

                    "===================================================================\n"

                )

            else:

                text += (

                    "===================================================================\n"

                    f"Graduate Level\n"

                    "===================================================================\n"

                )

            # After the header, concatenate the text for the averages

            text += (

                f"Overall average (major and minor) for all terms: {average}\n"

                f"Average (major and minor) of each term:\n"

            )

            # Loop through each terms

            for term in range(1, last\_term + 1):

                grades\_per\_term = []

                # Loop through the student's grade/record, get the average grade per term

                for grade in grades\_info:

                    if term == grade["Term"]:

                        grades\_per\_term.append(grade["Grade"])

                average\_per\_term = int(sum(grades\_per\_term) / len(grades\_per\_term))

                # Concatenate the the average grade per term to text string

                text += f"\tTerm {term}: {average\_per\_term}\n"

            # Concatenate the minimum and maximum grades to text string

            text += (

                f"\nMaximum grade(s) and in which term(s): {max\_grade} in term {', '.join(terms\_with\_max\_grade)}"

                f"\nMinimum grade(s) and in which term(s): {min\_grade} in term {', '.join(terms\_with\_min\_grade)}"

                "\nDo you have any repeated course(s)?: Y\n\n"

            )

        # Print and write the text

        print(text, end="")

        with open(f"std{student\_id}Statistics.txt", "w") as file:

            file.write(text)

        # Record this request

        self.recordRequest("Statistics")

    def majorTranscriptFeature(self, student\_level, student\_id, student):

        """ Shows the transcript of the student's major courses """

        # Initialize variables

        major\_courses = [grade for grade in self.student\_grades if grade["courseType"] == "Major"]

        minor\_courses = [grade for grade in self.student\_grades if grade["courseType"] == "Minor"]

        name = student['Name']

        student\_id = student\_id

        num\_of\_major = str(len(major\_courses))

        num\_of\_minor = str(len(minor\_courses))

        colleges = ', '.join(sorted(self.student['Colleges'], reverse=True))

        departments = ', '.join(sorted(self.student['Departments'], reverse=True))

        levels = ', '.join(sorted(self.student['Levels'], reverse=True))

        last\_term = max([grade["Term"] for grade in self.student\_grades])

        text = (

            f"{'Name: ' + name: <50} student ID: {student\_id}\n"

            f"{'College: ' + colleges:<50} Department: {departments}\n"

            f"{'Major: ' + num\_of\_major:<50} Minor: {num\_of\_minor}\n"

            f"{'Level: ' + levels:<50} Number of terms: {last\_term}\n\n"

        )

        # Compute overall average in all major courses

        major\_grades = [grade["Grade"] for grade in major\_courses]

        overall\_major\_average = str(int(sum(major\_grades) / len(major\_grades)))

        for term in range(1, last\_term + 1):

            # Concatenate the header for each term

            text += (

                f"=================================================================================\n"

                f"Term {term}"

                f"\n=================================================================================\n"

            )

            grades\_per\_term = []

            course\_text = ""

            for grade in major\_courses:

                if term == grade["Term"]:

                    grades\_per\_term.append(grade["Grade"])

                    course\_text += f"{grade['courseID']:^12} {grade['courseName']:^43} {grade['creditHours']:^12} {grade['Grade']:^8}\n"

            # Solve the average

            # grades\_per\_term = [grade["Grade"] for grade in major\_courses if term == grade["Term"]]

            if len(grades\_per\_term) <= 0:

                text += "\nNo registered major course this term.\n\n"

                continue

            text += f"{'Course ID':^12} {'Course Name':^43} {'Credit Hours':^12} {'Grade':^10}\n"

            text += course\_text

            average\_grade\_per\_term = str(int(sum(grades\_per\_term) / len(grades\_per\_term)))

            # # Concatenate the the average grade per term to text string

            text += f"\n{'Overall Major Average: ' + overall\_major\_average:<52} Term Major Average: {average\_grade\_per\_term}\n\n"

        print(text)

        with open(f"std{self.student\_id}MajorTranscript.txt", "w") as file:

            file.write(text)

        # Record this request

        self.recordRequest("Major")

    def minorTranscriptFeature(self, student\_level, student\_id, student):

        """ Shows the transcript of the student's minor courses """

        # Initialize variables

        major\_courses = [grade for grade in self.student\_grades if grade["courseType"] == "Major"]

        minor\_courses = [grade for grade in self.student\_grades if grade["courseType"] == "Minor"]

        name = student['Name']

        student\_id = student\_id

        num\_of\_major = str(len(major\_courses))

        num\_of\_minor = str(len(minor\_courses))

        colleges = ', '.join(sorted(self.student['Colleges'], reverse=True))

        departments = ', '.join(sorted(self.student['Departments'], reverse=True))

        levels = ', '.join(sorted(self.student['Levels'], reverse=True))

        last\_term = max([grade["Term"] for grade in self.student\_grades])

        text = (

            f"{'Name: ' + name: <50} student ID: {student\_id}\n"

            f"{'College: ' + colleges:<50} Department: {departments}\n"

            f"{'Major: ' + num\_of\_major:<50} Minor: {num\_of\_minor}\n"

            f"{'Level: ' + levels:<50} Number of terms: {last\_term}\n\n"

        )

        # Compute overall average in all major courses

        minor\_grades = [grade["Grade"] for grade in minor\_courses]

        overall\_minor\_average = str(int(sum(minor\_grades) / len(minor\_grades)))

        for term in range(1, last\_term + 1):

            # Concatenate the header for each term

            text += (

                f"=================================================================================\n"

                f"Term {term}"

                f"\n=================================================================================\n"

            )

            grades\_per\_term = []

            course\_text = ""

            for grade in minor\_courses:

                if term == grade["Term"]:

                    grades\_per\_term.append(grade["Grade"])

                    course\_text += f"{grade['courseID']:^12} {grade['courseName']:^43} {grade['creditHours']:^12} {grade['Grade']:^8}\n"

            # Solve the average

            if len(grades\_per\_term) <= 0:

                text += "\nNo registered minor course this term.\n\n"

                continue

            text += f"{'Course ID':^12} {'Course Name':^43} {'Credit Hours':^12} {'Grade':^10}\n"

            text += course\_text

            average\_grade\_per\_term = str(int(sum(grades\_per\_term) / len(grades\_per\_term)))

            # # Concatenate the the average grade per term to text string

            text += f"\n{'Overall Minor Average: ' + overall\_minor\_average:<52} Term Minor Average: {average\_grade\_per\_term}\n\n"

        print(text)

        with open(f"std{self.student\_id}MinorTranscript.txt", "w") as file:

            file.write(text)

        # Record this request

        self.recordRequest("Minor")

    def fullTranscriptFeature(self, student\_level, student\_id, student):

        """ Shows the transcript of the student's courses (both major and minor courses) """

        # Initialize variables

        major\_courses = [grade for grade in self.student\_grades if grade["courseType"] == "Major"]

        minor\_courses = [grade for grade in self.student\_grades if grade["courseType"] == "Minor"]

        name = student['Name']

        student\_id = student\_id

        num\_of\_major = str(len(major\_courses))

        num\_of\_minor = str(len(minor\_courses))

        colleges = ', '.join(sorted(self.student['Colleges'], reverse=True))

        departments = ', '.join(sorted(self.student['Departments'], reverse=True))

        levels = ', '.join(sorted(self.student['Levels'], reverse=True))

        last\_term = max([grade["Term"] for grade in self.student\_grades])

        text = (

            f"{'Name: ' + name: <50} student ID: {student\_id}\n"

            f"{'College: ' + colleges:<50} Department: {departments}\n"

            f"{'Major: ' + num\_of\_major:<50} Minor: {num\_of\_minor}\n"

            f"{'Level: ' + levels:<50} Number of terms: {last\_term}\n\n"

        )

        # Compute overall average in all major courses

        full\_grades = [grade["Grade"] for grade in self.student\_grades]

        overall\_average = str(int(sum(full\_grades) / len(full\_grades)))

        for term in range(1, last\_term + 1):

            # Concatenate the header for each term

            text += (

                f"=================================================================================\n"

                f"Term {term}"

                f"\n=================================================================================\n"

            )

            grades\_per\_term = []

            course\_text = ""

            for grade in self.student\_grades:

                if term == grade["Term"]:

                    grades\_per\_term.append(grade["Grade"])

                    course\_text += f"{grade['courseID']:^12} {grade['courseName']:^43} {grade['creditHours']:^12} {grade['Grade']:^8}\n"

            # Solve the average

            if len(grades\_per\_term) <= 0:

                text += "\nNo registered course this term.\n\n"

                continue

            text += f"{'Course ID':^12} {'Course Name':^43} {'Credit Hours':^12} {'Grade':^10}\n"

            text += course\_text

            average\_grade\_per\_term = str(int(sum(grades\_per\_term) / len(grades\_per\_term)))

            # # Concatenate the the average grade per term to text string

            text += f"\n{'Overall Average: ' + overall\_average:<58} Term Average: {average\_grade\_per\_term}\n\n"

        print(text)

        with open(f"std{self.student\_id}FullTranscript.txt", "w") as file:

            file.write(text)

        # Record this request

        self.recordRequest("Full")

    def previousRequestsFeature(self, history, student\_id):

        """ Shows the previous requests of the student """

        # Print the header

        print("{:^12} {:^15} {:^7}".format('Request','Date','Time'))

        print("=========================================")

        # Loop through history and print and write each request

        for request in history:

            print ("{:^12} {:^15} {:^7}".format(request['req'], request['date'], request['time']))

        # Save the requests in history in a text file

        self.saveHistory()

        # Increment the number of requests

        self.num\_of\_request += 1

    def newStudentFeature(self):

        """ Asks for the new student's info (similar to start feature) """

        # Resets the variables to give space for new student

        self.resetVariables()

        # Prompt user to select the new student's level and type/degree

        self.setStudentLevel()

        # Prompt user to enter the new student ID

        self.setStudentID()

        # Ensure that the given student level, degree and id exists in the student details

        if not self.isStudentRegistered():

            print("\nA student with the given information doesn't exists. Try Again.")

            buffer()

            return self.newStudentFeature()

        # Read the he student details csv of the which contains all students, save only the information relevant to the user

        self.setStudentInfos()

        # Read the csv file of the student's grade/record, then save it to student grades

        self.setStudentGrade()

        # Read the txt file of the student's previous requests, then save it to  history

        self.setHistory()

    def terminateFeature(self):

        """ Terminates/exit/close the program """

        print(f"There was a total of {self.num\_of\_request} requests this session.")

        print("The program is closing ...\nGoodbye!\n")

        sys.exit()

    def menuManager(self):

        """ Asks user which feature to use, then redirects to that feature """

        # Asks for a feature

        choice = input("Enter your Feature: ")

        clearScreen()

        # Redirect to specific feature based on user choice

        match choice:

            case "1":

                self.detailsFeature(self.student)

            case "2":

                self.statisticsFeature(self.student\_level, self.student\_id)

            case "3":

                self.majorTranscriptFeature(self.student\_level, self.student\_id, self.student)

            case "4":

                self.minorTranscriptFeature(self.student\_level, self.student\_id, self.student)

            case "5":

                self.fullTranscriptFeature(self.student\_level, self.student\_id, self.student)

            case "6":

                self.previousRequestsFeature(self.history, self.student\_id)

            case "7":

                self.newStudentFeature()

            case "8":

                self.terminateFeature()

            case \_:

                print("Invalid Input!")

        # After every feature, add a buffer to let user read, then clear screen and go back to menu screen

        buffer()

        self.menuFeature()

def buffer():

    """ Acts as buffer to give enough time for user to read the texts """

    input("\nPress enter to proceed ...")

    clearScreen()

def clearScreen():

    """ Clears the screen regardless of the OS """

    # Posix is OS name for Linux or Mac, 'clear' cmd clears the screen for Linux and Mac

    if os.name == "posix":

        os.system("clear")

    # for Windows (os name is 'nt'), 'cls' command clears the screen

    else:

        os.system("cls")

if \_\_name\_\_ == "\_\_main\_\_":

    main()