REPUBLIC OF THE PHILIPPINES

POLYTECHNIC UNIVERSITY OF THE PHILIPPINES

STA. MESA, MANILA

COLLEGE OF ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CMPE 102

Programming Logic and Design

Final Project Requirement

Transcript Generation System

Group 4:

BOBILES, ZAKI J.

CELLONA, DWAYNE A.

MENDOZA, MIGUEL JOSE A.

Course/Year/Section:

BSCpE 1-4

Class Schedule:

2:00 – 8:00 PM | SATURDAY

Date Submitted:

JANUARY 30, 2025

Submitted to:

ENGR. CANSINO, JULIUS S

```
# This Work Done By Group 9
# BOBILES, ZAKI J.              2024-05193-MN-0    (30%)
# CELLONA, DWAYNE A.            2024-02467-MN-0    (40%)
# MENDOZA, MIGUEL JOSE A.       2024-03099-MN-0    (30%)

# IMPORT LIBRARIES
import os
import numpy as np
import time
from datetime import datetime
import random
import sys

# Utiliy Functions
def Sleep_And_Clear():  # Pause for 4 seconds and clear the screen.
    time.sleep(2)
    os.system('cls' if os.name == 'nt' else 'clear')
    time.sleep(2)

def SaveFile(stdID, feature_name, content): # Save student data to a text file.
    file_name = f"std{stdID}{feature_name}.txt"
    with open(file_name, "w") as file:
        file.write(content)
    print(f"Data saved to {file_name}")

def DateAndTime():  # Function that will provide date and time to the system
    return                              datetime.now().strftime("%d/%m/%Y"),
datetime.now().strftime("%H:%M")

def TimeStamps(RequestType, Timestamps, stdID):
    Timestamps.append((RequestType, *DateAndTime()))  # Add RequestType, Date,
Time
    SaveFile(stdID, "PreviousRequests", PrintRequests(Timestamps, stdID))   #
Save requests to file
    Timestamps = []  # Clear the Timestamps list

def counter():  # A function to count the number of requests.
    if not hasattr(counter, 'count'):
        counter.count = 0  # Initialize the counter if it doesn't exist
    counter.count += 1
    return counter.count

def PrintRequests(Timestamps, stdID):  # Create or read the request log file
and append new requests
    file_path = f"std{stdID}PreviousRequests.txt"

    if not os.path.isfile(file_path):  # Check if the file exists and create a
header or read the existing content
        header = ('=' * 50) + f"\n{'Request':<20}{'Date':<20}{'Time':<10}\n" +
('=' * 50)
        Lines = header  # Initialize with header if file doesn't exist
    else:
        with open(file_path, "r") as file:
            Lines = file.read()  # Read file content if it exists

    for Request, Date, Time in Timestamps: # Add new request details to the log
        Lines += f"\n{Request:<20}{Date:<20}{Time:<10}"

    return Lines

def print_transcript(transcript_content):  # Split the transcript content into
individual lines and iterate over each line
    for line in transcript_content.splitlines():
        sys.stdout.write(line + "\n")  # Print the current line followed by a
newline
        sys.stdout.flush() # Flush the output buffer to ensure the line is
printed immediately
        time.sleep(0.2) # Introduce a delay of 0.2 seconds before printing the
next line

# Student Data Generation Functions (nested dictionary containing list)
def GenerateStudentDetails(): # Generates random student details and saves them
to 'studentDetails.csv'
    # Lists of first, middle, and last names
    FirstNames = ["Juan", "Maria", "Carlos", "Isabella", "Liam", "Sophia",
"Lucas", "Mia"]
    MiddleNames = ["Gabriel", "Luna", "Diego", "Ariana", "Nathan", "Ella",
"Isaiah", "Zoe"]
    LastNames = ["Santos", "Reyes", "Garcia", "Cruz", "Torres", "Mendoza"]

    # Level and degree options
    Levels = {
    "U": ["BS1", "BS2", "BS3", "BS4", "BS5"],  # Undergrad
    "G": {"M": ["M1", "M2"], "D": ["D1", "D2"]},  # Master's & Doctorate
    "B": {"U": ["BS1", "BS2", "BS3", "BS4", "BS5"], "G": ["M1", "M2", "D1",
"D2"]},  # BS + Grad
    "B0": {"M": ["M1", "M2"], "D": ["D1", "D2"]},  # Master's + Doctorate
    "BMD": {"U": ["BS1", "BS2", "BS3", "BS4"], "M": ["M1", "M2"], "D": ["D1",
"D2"]}  # BS + Master's + Doctorate
}

    Filename = "studentDetails.csv"
    if os.path.exists(Filename):
        os.remove(Filename)

    # Course IDs for each degree
    CourseIds = {f"CourseBS{i}": f"CBS{i}" for i in range(1, 46)}  # BS courses
    CourseIds.update({f"CourseM{i}": f"CM{i}" for i in range(1, 46)})  #
Master's courses
    CourseIds.update({f"CourseD{i}": f"CD{i}" for i in range(1, 46)})  #
Doctorate courses

    with open(Filename, "w") as File:
```

```
File.write("Serial,StudentID,Name,College,Department,Level,Degrees,Major,Mino
r,Terms\n")

        Serial = 1  # Initialize serial number starting at 1
        for i in range(1, 21):
            StudentId = f"20100{6000 + i}"
            FullName              =              f"{random.choice(FirstNames)}
{random.choice(MiddleNames)} {random.choice(LastNames)}"
            College = f"College{random.randint(1, 5)}"
            Department = f"C{College[-1]}Department{random.randint(1, 5)}"
            Terms = random.randint(4, 8)  # Randomly assign number of terms

            Level, Degrees = random.choice(["U", "G", "B", "B0", "BMD"]), []  #
Randomly select level and initiailize degrees

            # Writing student data based on selected level and degree(s)
            if Level == "U":
                Degree  =  random.choice(Levels["U"])   #  Select  one  Bachelor's
degree
                MajorUndergrad,      MinorUndergrad      =      (f"C{College[-
1]}D{Department[-1]}Major{Degree}",             f"C{College[-1]}D{Department[-
1]}Minor{Degree}")

File.write(f"{Serial},{StudentId},{FullName},{College},{Department},U,{Degree
},{MajorUndergrad},{MinorUndergrad},{Terms}\n")
                Serial += 1  # Increment serial number after writing row
                Degrees = [Degree]  # Store the degree as a list for course
generation

            elif Level == "G":
                Degree = random.choice(Levels["G"][random.choice(["M", "D"])])
# Choose between Master's or Doctorate and select the degree
                MajorGrad,    MinorGrad    =    f"C{College[-1]}D{Department[-
1]}Major{Degree}", f"C{College[-1]}D{Department[-1]}Minor{Degree}"

File.write(f"{Serial},{StudentId},{FullName},{College},{Department},G,{Degree
},{MajorGrad},{MinorGrad},{Terms}\n")
                Serial += 1  # Increment serial number after writing row
                Degrees = [Degree]  # Store the degree as a list for course
generation

            elif Level == "B": # Both undergraduate and graduate (BS + Graduate)
                UndergradDegree, GradDegree = random.choice(Levels["B"]["U"]),
random.choice(Levels["B"]["G"])
                Degrees = [UndergradDegree, GradDegree]  # Store both degrees

                # Write undergraduate row
                MajorUndergrad,      MinorUndergrad      =      f"C{College[-
1]}D{Department[-1]}Major{UndergradDegree}",    f"C{College[-1]}D{Department[-
1]}Minor{UndergradDegree}"
```

```
File.write(f"{Serial},{StudentId},{FullName},{College},{Department},U,{Underg
radDegree},{MajorUndergrad},{MinorUndergrad},{Terms}\n")
                Serial += 1

                # Write graduate row
                MajorGrad,    MinorGrad    =    f"C{College[-1]}D{Department[-
1]}Major{GradDegree}", f"C{College[-1]}D{Department[-1]}Minor{GradDegree}"

File.write(f"{Serial},{StudentId},{FullName},{College},{Department},G,{GradDe
gree},{MajorGrad},{MinorGrad},{Terms}\n")
                Serial += 1


            elif Level == "B0": # Master's + Doctorate (B0)
                GradTypeM,   GradTypeD   =   random.choice(Levels["B0"]["M"]),
random.choice(Levels["B0"]["D"])
                Degrees = [GradTypeM, GradTypeD]

                # Write Master's degree row
                MajorGradM,   MinorGradM   =   (f"C{College[-1]}D{Department[-
1]}Major{GradTypeM}", f"C{College[-1]}D{Department[-1]}Minor{GradTypeM}")

File.write(f"{Serial},{StudentId},{FullName},{College},{Department},G,{GradTy
peM},{MajorGradM},{MinorGradM},{Terms}\n")
                Serial += 1  # Increment serial number after writing row

                # Write Doctorate degree row
                MajorGradD,   MinorGradD   =   (f"C{College[-1]}D{Department[-
1]}Major{GradTypeD}", f"C{College[-1]}D{Department[-1]}Minor{GradTypeD}")

File.write(f"{Serial},{StudentId},{FullName},{College},{Department},G,{GradTy
peD},{MajorGradD},{MinorGradD},{Terms}\n")
                Serial += 1  # Increment serial number after writing row

            elif Level == "BMD":
                # BS + Master's + Doctorate
                UndergradDegree = random.choice(Levels["BMD"]["U"])   # Select
Bachelor's degree
                MasterDegree  =  random.choice(Levels["BMD"]["M"])    #  Select
Master's degree
                DoctorateDegree = random.choice(Levels["BMD"]["D"])   # Select
Doctorate degree
                Degrees = [UndergradDegree, MasterDegree, DoctorateDegree]   #
Store all three degrees

                # Write Undergraduate row
                MajorUndergrad,      MinorUndergrad      =      (f"C{College[-
1]}D{Department[-1]}Major{UndergradDegree}",    f"C{College[-1]}D{Department[-
1]}Minor{UndergradDegree}")
```

```
File.write(f"{Serial},{StudentId},{FullName},{College},{Department},U,{Underg
radDegree},{MajorUndergrad},{MinorUndergrad},{Terms}\n")
                Serial += 1  # Increment serial number after writing row

                # Write Master's row
                MajorMaster, MinorMaster = (f"C{College[-1]}D{Department[-
1]}Major{MasterDegree}", f"C{College[-1]}D{Department[-
1]}Minor{MasterDegree}")

File.write(f"{Serial},{StudentId},{FullName},{College},{Department},G,{Master
Degree},{MajorMaster},{MinorMaster},{Terms}\n")
                Serial += 1  # Increment serial number after writing row

                # Write Doctorate row
                MajorDoctorate, MinorDoctorate = (f"C{College[-
1]}D{Department[-1]}Major{DoctorateDegree}", f"C{College[-1]}D{Department[-
1]}Minor{DoctorateDegree}")

File.write(f"{Serial},{StudentId},{FullName},{College},{Department},G,{Doctor
ateDegree},{MajorDoctorate},{MinorDoctorate},{Terms}\n")
                Serial += 1  # Increment serial number after writing row

            # Now generate the courses CSV for each student, combining undergrad
and grad courses
            GenerateStudentCsv(StudentId, Degrees, Terms, CourseIds)

def GenerateStudentCsv(StudentId, Degrees, Terms, CourseIds):
    """Generates a single student CSV file with courses for both undergrad and
grad degrees in one file."""
    StudentFilename = f"{StudentId}.csv"  # All degrees will be in the same
file

    with open(StudentFilename, "w") as File:  # Open file in write mode
        # For each degree (Undergraduate, Graduate), generate courses
        FirstRowWritten = False  # Track if header row has been written
        for Degree in Degrees:
            LevelType = "U" if "BS" in Degree else "G"
            # Write header for each degree level, only once
            if not FirstRowWritten:

File.write("Level,Degree,Term,Course,CourseID,Type,CreditHours,Grade\n")
                FirstRowWritten = True  # Mark header row as written

            # Generate course records for the student's degree level
            for Term in range(1, Terms + 1):
                CoursesTaken, MajorCourse, MinorCourse = [], [], []
                while len(CoursesTaken) < 4:  # Ensure exactly 4 courses per
term
                    if "BS" in Degree:  # For undergraduate (BS) courses
                        MajorCourse = random.choice([c for c in CourseIds if
"CourseBS" in c and int(c.replace("CourseBS", "")) % 2 == 0])
```

```
                        MinorCourse = random.choice([c for c in CourseIds if
"CourseBS" in c and int(c.replace("CourseBS", "")) % 2 == 1])
                    elif "M" in Degree:  # For Master's (M) courses
                        MajorCourse = random.choice([c for c in CourseIds if
"CourseM" in c and int(c.replace("CourseM", "")) % 2 == 0])
                        MinorCourse = random.choice([c for c in CourseIds if
"CourseM" in c and int(c.replace("CourseM", "")) % 2 == 1])
                    elif "D" in Degree:  # For Doctorate (D) courses
                        MajorCourse = random.choice([c for c in CourseIds if
"CourseD" in c and int(c.replace("CourseD", "")) % 2 == 0])
                        MinorCourse = random.choice([c for c in CourseIds if
"CourseD" in c and int(c.replace("CourseD", "")) % 2 == 1])

                    # Check if MajorCourse (with its type) is not in
CoursesTaken before appending
                    if (MajorCourse, "Major") not in CoursesTaken:
                        CoursesTaken.append((MajorCourse, "Major"))

                    # Check if MinorCourse (with its type) is not in
CoursesTaken before appending
                    if (MinorCourse, "Minor") not in CoursesTaken:
                        CoursesTaken.append((MinorCourse, "Minor"))

                # Write the selected courses to the CSV for the student
                for CourseName, CourseType in CoursesTaken:
                    # Create the correct course ID prefix (CBS, CM, or CD)
                    if "BS" in Degree:
                        CourseId = f"CBS{CourseName.replace('CourseBS', '')}"
                    elif "M" in Degree:
                        CourseId = f"CM{CourseName.replace('CourseM', '')}"
                    elif "D" in Degree:
                        CourseId = f"CD{CourseName.replace('CourseD', '')}"

                    CreditHours = random.choice([1, 2, 3, 4])  # Random credit
hours
                    Grade = random.randint(50, 100)  # Random grade between 50
and 100

File.write(f"{LevelType},{Degree},{Term},{CourseName},{CourseId},{CourseType}
,{CreditHours},{Grade}\n")

def Get_std_Records(student_levels):
    """Retrieves student records based on the selected level(s) and student
ID."""
    try:
        os.system("cls" if os.name == 'nt' else 'clear')
        transcript_records = np.loadtxt('studentDetails.csv', dtype=str,
delimiter=',', skiprows=1)
        student_id = input("Enter student ID: ")
        Retrieved_Records = []
        has_undergraduate, has_graduate_master, has_graduate_doctorate =
False, False, False

        # Check for records that match the student ID and selected levels
        for record in transcript_records:
            degree = record[6]  # degree is in the 7th column (index 6)

            if "U" in student_levels and "BS" in degree and student_id ==
record[1]:
                Retrieved_Records.append(record)
                has_undergraduate = True
            if "M" in student_levels and "M" in degree and student_id ==
record[1]:
                Retrieved_Records.append(record)
                has_graduate_master = True
            if "D" in student_levels and "D" in degree and student_id ==
record[1]:
                Retrieved_Records.append(record)
                has_graduate_doctorate = True

        # Validate if both undergrad and graduate levels are selected
        if ['M', "D"] in student_levels:
            if not has_undergraduate:
                raise ValueError("To select 'Both', the student must have an
undergraduate record.")
            if not (has_graduate_master or has_graduate_doctorate):
                raise ValueError("To select 'Both', the student must have
either a Master's or Doctorate record.")

        # Additional validation for combinations of U, M, D
        if "U" in student_levels and "M" in student_levels:
            if not has_undergraduate or not has_graduate_master:
                raise ValueError("Both undergraduate and graduate Master's
records are required.")
        if "U" in student_levels and "D" in student_levels:
            if not has_undergraduate or not has_graduate_doctorate:
                raise ValueError("Both undergraduate and graduate Doctorate
records are required.")
        if ["U", "D", "M"] in student_levels:
            if not (has_undergraduate and has_graduate_master and
has_graduate_doctorate):
                raise ValueError("All three levels (Undergraduate, Master's,
Doctorate) must be present.")

        # If no records were found, raise an error
        if not Retrieved_Records:
            raise ValueError("No matching records found for the selected levels
and student ID.")

        # Proceed to the next menu with the retrieved records
        Sleep_And_Clear()  # Clear screen
        MenuFeature(Retrieved_Records)  # Return to the menu
```

```
    except ValueError as e:
        print(f"[Error] {e}")
        Sleep_And_Clear() # clear the screen
        startFeature() # return to start feature

def startFeature(): # Handles student level selection and retrieves student
records.
    try:
        os.system("cls" if os.name == 'nt' else 'clear')
        Selected_Levels = []

        # Ask for the student's primary level (Undergraduate, Graduate, or Both)
        First_Student_Level = input(
    "Please select your student level to view your transcript record:\n"
    "[U] Undergraduate: Select if you are an undergraduate student (e.g.,
Bachelor's).\n"
    "[G] Graduate: Select if you are a graduate student (e.g., Master's or
Doctorate).\n"
    "[B] Both: Select if you have both undergraduate and graduate records.\n"
    "Enter your student level: "
).upper()

        if First_Student_Level == "U":
            Selected_Levels.append("U")
        elif First_Student_Level == "G" or First_Student_Level == "B":
            if First_Student_Level == "B":
                Selected_Levels.append("U")  # Include undergraduate if 'Both'
is selected
            os.system("cls" if os.name == 'nt' else 'clear')
            Second_Student_Level = input(
    "Please select your graduate level to view your transcript record:\n"
    "[M] Master's: Select if you are a Master's student.\n"
    "[D] Doctorate: Select if you are a Doctorate student.\n"
    "[B0] Both: Select if you have both Master's and Doctorate records.\n"
    "Enter your graduate level: "
).upper()

            if Second_Student_Level in ["M"]:
                Selected_Levels.append("M")
            if Second_Student_Level in ["D"]:
                Selected_Levels.append("D")
            if Second_Student_Level in ["B0"]:
                Selected_Levels.append("D")
                Selected_Levels.append("M")

            if Second_Student_Level not in ["M", "D", "B0"]:
                raise ValueError("Invalid graduate level selected.")
        else:
            raise ValueError("Invalid student level. Please choose U, G, or
B.")
```

```
        # Pass the selected levels to the function that retrieves student
records
        Get_std_Records(Selected_Levels)

    except ValueError:
        print("[Error] Invalid input. Please follow the instructions
carefully.")
        Sleep_And_Clear() # clear the sceem
        startFeature() # restart the start feature

def MenuFeature(Retrieved_Records):
    """Display the main menu and handle user selection to perform various
actions based on student records."""
    Timestamps = []  # Initialize list to store timestamps for actions

    while True:
        # Display the menu options
        print(
            "Student Transcript Generation System\n"
            + "=" * 40 + "\n"
            + "1.\tStudent details\n"
            + "2.\tStatistics\n"
            + "3.\tTranscript based on major courses\n"
            + "4.\tTranscript based on minor courses\n"
            + "5.\tFull transcript\n"
            + "6.\tPrevious transcript requests\n"
            + "7.\tSelect another student\n"
            + "8.\tTerminate the system\n"
            + "=" * 40 + "\n"
        )

        # Prompt for user input
        choice = input("Enter Your Feature: ")

        # Check user choice and call respective feature
        if choice == "1":
            counter()  # Increment the counter
            Timestamps = TimeStamps('Details', Timestamps,
Retrieved_Records[0][1])
            DetailsFeature(Retrieved_Records)

        elif choice == "2":
            counter()  # Increment the counter
            Timestamps = TimeStamps('Statistics', Timestamps,
Retrieved_Records[0][1])
            StatisticsFeature(Retrieved_Records)

        elif choice == "3":
            counter()  # Increment the counter
            Timestamps = TimeStamps('Major Transcript', Timestamps,
Retrieved_Records[0][1])
            MajorTranscriptFeature(Retrieved_Records)
        elif choice == "4":
            counter()  # Increment the counter
            Timestamps = TimeStamps('Minor Transcript', Timestamps,
Retrieved_Records[0][1])
            MinorTranscriptFeature(Retrieved_Records)

        elif choice == "5":
            counter()  # Increment the counter
            Timestamps = TimeStamps('Full Transcript', Timestamps,
Retrieved_Records[0][1])
            FullTranscriptFeature(Retrieved_Records)

        elif choice == "6":
            counter()  # Increment the counter
            Timestamps = TimeStamps('Previous Requests', Timestamps,
Retrieved_Records[0][1])
            PreviousRequestFeature(Retrieved_Records)

        elif choice == "7":
            counter()  # Increment the counter
            NewStudentFeature()  # Allow user to select a new student

        elif choice == "8":
            TerminateFeature()  # Terminate the system
            break  # Exit the loop

        else:
            print("Invalid selection! Please choose a number between 1 and 8.")
# Handle invalid input

def DetailsFeature(Retrieved_Records):
    """Displays and saves student details, ensuring correct term, college, and
department mapping."""

    # Extract common fields and create details output
    Name, StdID = Retrieved_Records[0][2], Retrieved_Records[0][1]
    Levels = ", ".join(sorted(set(Record[5] for Record in Retrieved_Records),
reverse=True))
    Degrees = ", ".join(sorted(set(Record[6] for Record in Retrieved_Records)))
    Term_Numbers = ", ".join(Record[9] for Record in Retrieved_Records)
    Colleges = ", ".join(Record[3] for Record in Retrieved_Records)
    Departments = ", ".join(Record[4] for Record in Retrieved_Records)

    Details_Output = (f"Name: {Name}\nstdID: {StdID}\nLevel(s): {Levels}\n"
                  f"Degree(s): {Degrees}\nNumber Of Terms: {Term_Numbers}\n"
                  f"College(s): {Colleges}\nDepartment(s): {Departments}")

    print_transcript(Details_Output)

    # Save details and return to menu
    SaveFile(StdID, "Details", Details_Output) # save to file
    input("Press Enter to go back to the menu...")
    Sleep_And_Clear()  # Clear screen
    MenuFeature(Retrieved_Records)  # Return to the menu

def StatisticsFeature(Retrieved_Records):
    """Handles the statistics display for the student's grades and processes 
grades."""
    stdID = Retrieved_Records[0][1]
    Data = np.loadtxt(f'{stdID}.csv', dtype=str, delimiter=',', skiprows=1)

    Stat_Content = ""  # Holds the final text output to be saved

    for Record in Retrieved_Records:
        Stat_Content_Inner = ""  # Temporary container to format statistics for 
the current record
        Scores = []  # Initialize Scores as a list to hold grade data
        Max_Term = 0  # Initialize Max_Term as an integer
        Is_Repeating = False  # Initialize Is_Repeating as False
        Level = ""
        Courses = []  # Initialize Courses as an empty list
        Valid_Terms = []  # This will store the valid terms for each degree

        # Process grades based on the degree (BS, M, D)
        if "BS" in Record[6]:  # Undergraduate (BS)
            Level, Valid_Terms = "Undergraduate", [1, 2, 3, 4, 5, 6, 7, 8]  # 
Example terms for Undergraduate
            for Row in Data:
                if "BS" in Row[1]:  # Only process grades for BS students
                    Scores.append(int(Row[7]))  # Collect the grades
                    Max_Term = max(Max_Term, int(Row[2]))  # Track the highest 
term
                    if Row[4] in Courses:  # Check for repeated courses
                        Is_Repeating = True
                    Courses.append(Row[4])

        elif "D" in Record[6]:  # Graduate (Doctorate)
            Level, Valid_Terms = "Graduate (D)", [1, 2, 3, 4, 5, 6, 7, 8]
            for Row in Data:
                if "D" in Row[1]:  # Only process grades for Doctorate students
                    Scores.append(int(Row[7]))  # Collect the grades
                    Max_Term = max(Max_Term, int(Row[2]))  # Track the highest 
term
                    if Row[4] in Courses:  # Check for repeated courses
                        Is_Repeating = True
                    Courses.append(Row[4])

        elif "M" in Record[6]:  # Graduate (Master's)
            Level, Valid_Terms = "Graduate (M)", [1, 2, 3, 4, 5, 6, 7, 8]  # 
Example terms for Graduate Master's
            for Row in Data:
                if "M" in Row[1]:  # Only process grades for Master's students
                    Scores.append(int(Row[7]))  # Collect the grades
                    Max_Term = max(Max_Term, int(Row[2]))  # Track the highest
term
                    if Row[4] in Courses:  # Check for repeated courses
                        Is_Repeating = True
                    Courses.append(Row[4])

        # Format the data and append to the final text container
        Stat_Content_Inner += FormatStatistics(Record, Data, Level,
Is_Repeating, Valid_Terms)

         # Append the formatted string to the final text container
        Stat_Content += Stat_Content_Inner
        print_transcript(Stat_Content_Inner)

    # Save the statistics to a file
    SaveFile(stdID, "Statistics", Stat_Content) # Save to File
    input("Press Enter to go back to the menu...")
    Sleep_And_Clear()  # Clear screen
    MenuFeature(Retrieved_Records)  # Return to the menu

def FormatStatistics(Record, Data, Level, Is_Repeating, Valid_Terms):
    """Formats the statistics into a readable string."""

    # Initialize variables
    Max_Score, Min_Score = 0, 100
    Max_Term, Min_Terms, Courses_Taken, Repeated_Courses = [], [], set(), set()

    # Variables to calculate weighted averages
    Total_Major_Score, Total_Major_Credit_Hours = 0, 0
    Total_Minor_Score, Total_Minor_Credit_Hours = 0, 0
    Term_Averages = []

    # Loop through Data to process courses, grades, and credit hours
    for Row in Data:
        Term, Grade, Credit_Hours, Course_ID, Course_Type = int(Row[2]),
int(Row[7]), int(Row[6]), Row[4], Row[5]

        # Process grades for valid terms
        if Term in Valid_Terms and Row[1] == Record[6]:
            if Course_ID in Courses_Taken:
                Repeated_Courses.add(Course_ID)
            else:
                Courses_Taken.add(Course_ID)

            # Track max and min grades for display
            if Grade > Max_Score:
                Max_Score = Grade
                Max_Term = [Term]
            elif Grade == Max_Score:
                Max_Term.append(Term)

            if Grade < Min_Score:
                Min_Score = Grade
                Min_Terms = [Term]
            elif Grade == Min_Score:
                Min_Terms.append(Term)

            # Calculate total scores and credit hours for major and minor
courses
            if Course_Type == "Major":
                Total_Major_Score += Grade * Credit_Hours
                Total_Major_Credit_Hours += Credit_Hours
            elif Course_Type == "Minor":
                Total_Minor_Score += Grade * Credit_Hours
                Total_Minor_Credit_Hours += Credit_Hours

    # Calculate overall averages using weighted averages
    Overall_Avg = (Total_Major_Score + Total_Minor_Score) /
(Total_Minor_Credit_Hours+Total_Major_Credit_Hours)
    # Calculate term averages and display them
    Stat_Content =
f"""============================================================
***********        {Level:<10} Level        ***********
============================================================
Overall average (major and minor) for all terms: {Overall_Avg:8.2f}
Average (major and minor) of each term:
"""

    # Loop through terms to calculate term averages
    for i in Valid_Terms:
        Term_Credit_Hours ,Term_Grades = [], []

        for Row in Data:
            if int(Row[2]) == i and Row[1] == Record[6]:
                Term_Grades.append(int(Row[7]))
                Term_Credit_Hours.append(int(Row[6]))

        if Term_Grades:
            Term_Avg = np.average(Term_Grades, weights=Term_Credit_Hours)
            Term_Averages.append(Term_Avg)
            Stat_Content += f"Term {i}: {Term_Avg:8.2f}\n"


    # Output maximum and minimum grades with their terms
    Stat_Content += f"""Maximum grade(s) and in which term(s): {Max_Score} in
term(s) {Max_Term}
Minimum grade(s) and in which term(s): {Min_Score} in term(s) {Min_Terms}
Do you have any repeated course(s)? {'Yes' if Is_Repeating else 'No'}"""

    if Is_Repeating:
        Stat_Content += f" [{', '.join(Repeated_Courses)}]"
    Stat_Content += "\n"
    return Stat_Content
```

```
def GenerateTranscriptHeader(stdID, Name, College, Department, Level, NumTerms, 
Major, Minor):
    Header = ""  # Initialize the string container for the header
    # Construct the header with proper alignment
    Header = (f"\n{'Name: ' + Name:<35}{'Student ID: ' + stdID:<50}\n"
          f"{'College: ' + College:<35}{'Department: ' + Department:<50}\n"
          f"{'Major: ' + Major:<35}{'Minor: ' + Minor:<50}\n"
          f"{'Level: ' + Level:<35}{'Number of Terms: ' + 
str(NumTerms):<50}\n")
    return Header

def MajorTranscriptFeature(Retrieved_Records):
    """Generates the major transcript for a student, showing all levels and 
degrees selected."""

    # Extract student details from RetrievedRecords
    stdID, Name, College, Department = Retrieved_Records[0][1], 
Retrieved_Records[0][2], Retrieved_Records[0][3], Retrieved_Records[0][4]

    Levels = sorted(set(row[5] for row in Retrieved_Records), reverse=True)  # 
Collect all unique levels
    Degrees = sorted(set(row[6] for row in Retrieved_Records if row[1] == 
stdID))

    # Get Major and Terms for each Degree
    StudentDegreesInfo = {
        (row[5], row[6]): {"Major": row[7], "Terms": int(row[9])} for row in 
Retrieved_Records if row[1] == stdID}

    # Load student's course data from CSV
    Data = np.loadtxt(f'{stdID}.csv', dtype=str, delimiter=',', skiprows=1)

    MajorTranscripts = ""

    for Level in Levels:  # Loop through each level (undergraduate, graduate, 
etc.)
        for Degree in Degrees:  # Loop through each degree
            if (Level, Degree) not in StudentDegreesInfo:
                continue  # Skip if no matching major is found

            Major, Minor = Retrieved_Records[0][7], Retrieved_Records[0][8]
            NumTerms = StudentDegreesInfo[(Level, Degree)]["Terms"]

            # Filter courses based on Level & Degree
            CoursesData = [row for row in Data if row[0] == Level and row[1] == 
Degree]
            if not CoursesData:
                continue  # Skip this degree if no course data is found

            # Generate transcript header
            MajorTranscriptContent = GenerateTranscriptHeader(stdID, Name, 
College, Department, Level, NumTerms, Major, Minor)
```

```
            TotalMajorScore, TotalCreditHours  = 0, 0 # Initialize total major
score and total credit  score
            TermAverages = [] # Initialize term average

            # Process transcript per term
            for Term in sorted(set(row[2] for row in CoursesData)):
                MajorTranscriptContent += "=" * 60
                MajorTranscriptContent   +=   f"\n{'*'*25}      Term   {Term}
{'*'*25}\n"
                MajorTranscriptContent += "=" * 60 + "\n"
                MajorTranscriptContent                                       +=
"{:<15}{:<15}{:<15}{:<15}\n".format("Course ID", "Course Name", "Credit Hours",
"Grade")

                CourseGrades, CourseCreditHours = [], [] # Initialize grades
and credit hours

                for row in CoursesData:
                    if row[5] == "Major":
                        if row[2] == Term:
                            Course,  CourseId,  CreditHours,  Grade  =  row[3],
row[4], int(row[6]), int(row[7])
                            MajorTranscriptContent                           +=
"{:<15}{:<15}{:<15}{:<15}\n".format(CourseId, Course, CreditHours, Grade)
                            CourseGrades.append(Grade)
                            CourseCreditHours.append(CreditHours)

                # Convert to NumPy arrays for vectorized calculations
                CourseGrades = np.array(CourseGrades)
                CourseCreditHours = np.array(CourseCreditHours)

                # Calculate term averages
                TermMajorAvg            =            np.average(CourseGrades,
weights=CourseCreditHours)
                TermAvg = np.mean(CourseGrades)

                # Update totals for overall calculation
                TotalMajorScore += np.sum(CourseGrades * CourseCreditHours)
                TotalCreditHours += np.sum(CourseCreditHours)
                TermAverages.append(TermAvg)

                MajorTranscriptContent      +=      f"\nMajor      Average:
{TermMajorAvg:8.2f}"
                MajorTranscriptContent     +=     f"\t\tOverall     Average:
{np.mean(TermAverages) if TermAverages else 0:8.2f}\n"

            # Add missing closing section
            MajorTranscriptContent += "=" * 60 + "\n"
            MajorTranscriptContent += f"*******    End of Transcript for Level
({Level}-{Degree})     *******\n"
            MajorTranscriptContent += "=" * 60 + "\n"
            MajorTranscripts += MajorTranscriptContent  # Append each degree's
transcript

    print_transcript(MajorTranscripts)  # Display the transcript
    SaveFile(stdID, "MajorTranscript", MajorTranscripts)  # Save to file
    input("Press Enter to go back to the menu...")
    Sleep_And_Clear()  # Clear screen
    MenuFeature(Retrieved_Records)  # Return to the menu

def MinorTranscriptFeature(Retrieved_Records):
    """Generates the minor transcript for a student, showing all levels and
degrees selected."""
    # Extract student details
    stdID, Name, College, Department = Retrieved_Records[0][1],
Retrieved_Records[0][2], Retrieved_Records[0][3], Retrieved_Records[0][4]

    Levels = sorted(set(row[5] for row in Retrieved_Records), reverse=True)  #
Get unique levels
    Degrees = sorted(set(row[6] for row in Retrieved_Records if row[1] ==
stdID))  # Get degrees for the student

    # Get Minor and Terms for each Degree
    StudentDegreesInfo = {
        (row[5], row[6]): {"Minor": row[8], "Terms": int(row[9])} for row in
Retrieved_Records if row[1] == stdID }

    # Load course data from CSV
    Data = np.loadtxt(f'{stdID}.csv', dtype=str, delimiter=',', skiprows=1)

    MinorTranscript = ""  # Initialize transcript content

    # Initialize cumulative variables
    TotalMinorScore, TotalCreditHours = 0, 0
    TermAverages, CumulativeGrades, CumulativeCreditHours = [], [], []

    for Level in Levels:  # Loop through levels
        for Degree in Degrees:  # Loop through degrees
            if (Level, Degree) not in StudentDegreesInfo:
                continue  # Skip if no minor found

            Major, Minor = Retrieved_Records[0][7], Retrieved_Records[0][8]
            NumTerms = StudentDegreesInfo[(Level, Degree)]["Terms"]

            # Filter courses for the specific Level & Degree
            CoursesData = [row for row in Data if row[0] == Level and row[1] ==
Degree]
            if not CoursesData:
                continue  # Skip if no course data is found

            # Generate transcript header
            MinorTranscriptContent = GenerateTranscriptHeader(stdID, Name,
College, Department, Level, NumTerms, Major, Minor)

            # Process each term
            for Term in sorted(set(row[2] for row in CoursesData)):
                MinorTranscriptContent += "=" * 60
                MinorTranscriptContent += f"\n{'*'*25} Term {Term}
{'*'*25}\n"
                MinorTranscriptContent += "=" * 60 + "\n"
                MinorTranscriptContent +=
"{:<15}{:<15}{:<15}{:<15}\n".format("Course ID", "Course Name", "Credit Hours",
"Grade")

                CourseGrades, CourseCreditHours = [], []

                for row in CoursesData:
                    if row[5] == "Minor" and row[2] == Term:  # Filter minor
courses
                        Course, CourseId, CreditHours, Grade = row[3], row[4],
int(row[6]), int(row[7])
                        MinorTranscriptContent +=
"{:<15}{:<15}{:<15}{:<15}\n".format(CourseId, Course, CreditHours, Grade)
                        CourseGrades.append(Grade)
                        CourseCreditHours.append(CreditHours)

                # Calculate term averages
                CourseGrades = np.array(CourseGrades)
                CourseCreditHours = np.array(CourseCreditHours)
                TermMinorAvg = np.average(CourseGrades,
weights=CourseCreditHours)
                TermAvg = np.mean(CourseGrades)

                # Update totals
                TotalMinorScore += np.sum(CourseGrades * CourseCreditHours)
                TotalCreditHours += np.sum(CourseCreditHours)
                TermAverages.append(TermAvg)

                # Accumulate for overall average
                CumulativeGrades.extend(CourseGrades)
                CumulativeCreditHours.extend(CourseCreditHours)

                MinorTranscriptContent += f"\nMinor Average:
{TermMinorAvg:8.2f}"
                MinorTranscriptContent += f"\t\tOverall Average:
{np.average(CumulativeGrades, weights=CumulativeCreditHours):8.2f}\n"

            # Closing section for the transcript
            MinorTranscriptContent += "=" * 60 + "\n"
            MinorTranscriptContent += f"*******    End of Transcript for Level
({Level}-{Degree})     *******\n"
            MinorTranscriptContent += "=" * 60 + "\n"
            MinorTranscript += MinorTranscriptContent    # Append to full
transcript

    print_transcript(MinorTranscript)  # Display the transcript
    SaveFile(stdID, "MinorTranscript", MinorTranscript)  # Save to file
    input("Press Enter to go back to the menu...")
    Sleep_And_Clear()  # Clear screen
    MenuFeature(Retrieved_Records)  # Return to the menu

def FullTranscriptFeature(Retrieved_Records):
    # Extract student details from retrieved_records
    stdID, Name, College, Deparment = Retrieved_Records[0][1],
Retrieved_Records[0][2], Retrieved_Records[0][3], Retrieved_Records[0][4]

    # Get unique levels and degrees
    Levels = sorted(set(row[5] for row in Retrieved_Records), reverse=True)
    Degrees = sorted(set(row[6] for row in Retrieved_Records if row[1] ==
stdID))

    # Store major, minor, and terms for each level-degree pair
    Student_Degrees_Info = {
        (row[5], row[6]): {"Major": row[7], "Minor": row[8], "Terms":
int(row[9])}
        for row in Retrieved_Records if row[1] == stdID
    }

    # Load student's course data from CSV
    Data = np.loadtxt(f'{stdID}.csv', dtype=str, delimiter=',', skiprows=1)

    full_transcripts = ""

    # Process each level in order (ensuring 'U' is first)
    for Level in Levels:
        for Degree in Degrees:
            if (Level, Degree) not in Student_Degrees_Info:
                continue  # Skip if there's no matching major/minor for this
degree

            Major = Student_Degrees_Info[(Level, Degree)]["Major"]
            Minor = Student_Degrees_Info[(Level, Degree)]["Minor"]
            Num_Terms = Student_Degrees_Info[(Level, Degree)]["Terms"]

            # Filter course data for this Level & Degree
            Courses_Data = [Row for Row in Data if Row[0] == Level and Row[1]
== Degree]
            if not Courses_Data:
                continue  # Skip this degree if no course data is found

            # Generate transcript header
            Transcript_Content = GenerateTranscriptHeader(stdID, Name,
College, Deparment, Level, Num_Terms, Major, Minor)
```

```
            # Variables to calculate weighted averages
            Total_Major_Score, Total_Major_Credit_Hours = 0, 0
            Total_Minor_Score, Total_Minor_Credit_Hours = 0, 0
            Term_Averages = []

            # Process transcript per term
            for Term in sorted(set(Row[2] for Row in Courses_Data)):
                Transcript_Content += "=" * 60
                Transcript_Content += f"\n{'*' * 25}  Term {Term}  {'*' * 25}\n"
                Transcript_Content += "=" * 60 + "\n"
                Transcript_Content                                          +=
"{:<15}{:<15}{:<15}{:<15}\n".format("Course ID", "Course Name", "Credit Hours",
"Grade")

                Course_Grades_Major, Course_Credit_Hours_Major = [], []
                Course_Credit_Hours_Minor, Course_Grades_Minor = [], []

                total_term_score, total_term_credit_hours = 0 , 0

                for Row in Courses_Data:
                    if Row[2] == Term:
                        Course, Course_ID, Course_Type, Credit_Hours, Grade =
Row[3], Row[4], Row[5], int(Row[6]), int(Row[7])
                        Transcript_Content                                  +=
"{:<15}{:<15}{:<15}{:<15}\n".format(Course_ID, Course, Credit_Hours, Grade)

                        total_term_credit_hours += Credit_Hours
                        total_term_score += Grade * Credit_Hours

                        if Course_Type == "Major":
                            Course_Grades_Major.append(Grade)
                            Course_Credit_Hours_Major.append(Credit_Hours)
                            Total_Major_Score += Grade * Credit_Hours
                            Total_Major_Credit_Hours += Credit_Hours
                        elif Course_Type == "Minor":
                            Course_Grades_Minor.append(Grade)
                            Course_Credit_Hours_Minor.append(Credit_Hours)
                            Total_Minor_Score += Grade * Credit_Hours
                            Total_Minor_Credit_Hours += Credit_Hours

                # Calculate major and minor averages for the term (weighted)
                Term_Major_Avg        =        np.average(Course_Grades_Major,
weights=Course_Credit_Hours_Major)
                Term_Minor_Avg        =        np.average(Course_Grades_Minor,
weights=Course_Credit_Hours_Minor)
                Term_Avg = total_term_score / total_term_credit_hours
                Overall_Avg  =  (Total_Major_Score  +  Total_Minor_Score)   /
(Total_Major_Credit_Hours + Total_Minor_Credit_Hours)
                Term_Averages.append(Term_Avg)

                Transcript_Content        +=        f"\nMajor        Average:
{Term_Major_Avg:8.2f}\t\tMinor Average: {Term_Minor_Avg:8.2f}\n"
```

```
                Transcript_Content += f"Term Average:
{Term_Avg:8.2f}\t\tOverall Average: {Overall_Avg:8.2f}\n"

            # Add closing section
            Transcript_Content += "=" * 60 + "\n"
            Transcript_Content += f"*******      End of Transcript for Level
({Level}-{Degree})     *******\n"
            Transcript_Content += "=" * 60 + "\n"

            full_transcripts += Transcript_Content   # Append each degree's
transcript

    print_transcript(full_transcripts)  # Display the transcript
    SaveFile(stdID, "fulltranscript", full_transcripts)  # Save to file
    input("Press Enter to go back to the menu...")
    Sleep_And_Clear()  # Clear screen
    MenuFeature(Retrieved_Records)  # Return to the menu

def PreviousRequestFeature(Retrieved_Records): # Displays previous requests for
a student by reading from the request file
    stdID = Retrieved_Records[0][1]  # Extract student ID
    file_path = f"std{stdID}PreviousRequests.txt"

   # Print the content or show a message if no file exists
    print_transcript(open(file_path, "r").read() if os.path.isfile(file_path)
else "No previous requests found.")
    input("Press Enter to go back to the menu...")
    Sleep_And_Clear()  # Clear screen
    MenuFeature(Retrieved_Records)  # Return to the menu

def NewStudentFeature(): # Clears the screen and returns to the start feature
    Sleep_And_Clear() # clear the screen
    startFeature() # return to the start feature

def TerminateFeature():
    """Terminate the system and display the request counter summary."""
    total_requests = counter() - 1  # Get the total number of requests
    print(f"==================================================\nTotal number
of requests during this session:
{total_requests}\n==================================================")
    exit()  # Exit the program

# Main Funtion
def main():
    # Check if student details file exists before calling
generate_student_details"
    if not os.path.exists("studentDetails.csv"):
        GenerateStudentDetails()
    startFeature()

main()
```