1. Outer Function: `startFeature`

The `startFeature` function wraps the entire code logic to make it modular and callable from other parts of a program.

---

2. `validate_student_id(student_id, valid_ids)`

This function ensures the provided student ID is valid:

- **Input:** A `student_id` string and a set of `valid_ids`.
- **Logic:** If the entered ID is not in the `valid_ids` set, it repeatedly prompts the user until they provide a valid one.
- **Output:** A validated `student_id`.

---

3. `get_student_level_selection()`

This function collects and validates the student's level and degree selection:

- **Input:** Prompts the user to choose:
  - Student level: Undergraduate (U), Graduate (G), or Both (B).
  - If Graduate or Both, a further degree selection: Master (M), Doctorate (D), or Both (B).
- **Logic:**
  - Ensures valid input using `while` loops.
  - Constructs a list of levels:
    - Undergraduate: `['U']`.
    - Graduate:
      - Master's: `['GM']`.
      - Doctorate: `['GD']`.
      - Both Master's and Doctorate: `['GM', 'GD']`.
    - If Both levels are selected, includes both Undergraduate (`U`) and Graduate levels (`GM`, `GD`) based on the degree selection.
- **Output:** A list of selected levels, e.g., `['U', 'GM']`.

---

4. `load_student_details(file_path)`

This function loads student details from a CSV file:

- **Input:** A `file_path` string pointing to the CSV file.
- **Logic:**
  - Checks if the file exists using `os.path.exists`.
  - Reads the file using `csv.DictReader` to convert each row into a dictionary and appends it to a list.
- **Output:** A list of dictionaries containing student details.

---

5. `main()`

The core logic of the program resides in this function:

1. **Load student details:**
   - Reads `studentDetails.csv` to get all students' data.
   - Extracts valid student IDs into a set for validation.
2. **User Inputs:**
   - Calls `get_student_level_selection` to get the level and degree selections.
   - Validates the entered student ID using `validate_student_id`.
3. **Load Specific Student Data:**
   - Loads the student's individual file (e.g., `<student_id>.csv`) to fetch enrollment data.
4. **Enrollment Validation:**
   - **Undergraduate Level:** Checks if any row in the student data has `Level` set to `U`.
   - **Graduate Level:**
     - Filters rows with `Level` set to `G`.
     - Extracts the `Degree` field from the rows.
     - Validates against selected levels (`GM` for Master's, `GD` for Doctorate).
   - **Multiple Levels:**
     - Checks for combinations of Undergraduate, Master's, and Doctorate enrollments.
   - **Invalid Cases:** If enrollment doesn't match the selected levels or degrees, it clears the screen (`os.system('cls')`) and restarts `startFeature`.
5. **Post-validation Actions:**
   - If enrollment validation is successful, the program proceeds to another function `menuFeature`, passing the validated level and student ID.
   - If validation fails, displays an error message.

---

6. Auxiliary Features

- **Error Handling:** Checks if files exist before attempting to read them.
- **Debugging:** Prints selected levels for debugging purposes.
- **Recursive Restarts:** Calls `startFeature` recursively for invalid cases.
- **File Dependencies:** Relies on the structure and availability of CSV files like `studentDetails.csv` and individual `<student_id>.csv`.

Function Definition: `menuFeature(degree, student_id)`

The function implements a menu system for managing student transcript features. It takes two parameters:

- **`degree`**: The level(s) of the student (e.g., Undergraduate (`U`), Graduate (`GM`, `GD`)).
- **`student_id`**: The unique ID of the student.

---

1. Variable Initialization

```
stdID = int(student_id)
level = degree
```

- **`stdID`**: Converts `student_id` into an integer to ensure it's in numeric form.
- **`level`**: Stores the selected degree/level for easy reference throughout the function.

Request Tracking Variables:

```
request_history = []
dates = []
times = []
```

- **`request_history`**: Tracks the names of features accessed (e.g., "Statistics", "Major").
- **`dates`**: Logs the dates when features are accessed.
- **`times`**: Logs the times of access in a human-readable format.

Timestamp Setup:

```
now = datetime.datetime.now()
formatted_time = now.strftime("%I:%M %p")
```

- **`now`**: Captures the current date and time.
- **`formatted_time`**: Formats the time in a 12-hour format with AM/PM for better readability.

---

2. Menu Display and Loop

```
confirm = True
while confirm:
```

- **`confirm`**: A flag controlling the loop. The menu continues to display as long as `confirm` is `True`.

Display Menu:

```
print("Student Transcript Generation System")
print("====================================")
print("1. Student Details")
print("2. Statistics")
print("3. Transcript based on Major Courses")
print("4. Transcript based on Minor Courses")
print("5. Full Transcript")
print("6. Previous Transcript Requests")
print("7. Select Another Student")
print("8. Terminate the System")
print("====================================")
```

- The menu lists eight options, each corresponding to a feature.

Input Handling:

```
try:
    option = int(input("Enter Your Feature: "))
except ValueError:
    print("Invalid input. Please enter a number between 1 and 8.")
    continue
```

- Prompts the user for input and tries to convert it to an integer.
- If the user enters invalid data (e.g., letters or symbols), a `ValueError` is caught, and the user is prompted again.

---

3. Option Processing

Validating the Option:

```
if 1 <= option <= 8:
```

- Ensures the user selects a valid option (1 to 8).

Handling Each Option:

Option 1: Student Details

```
if option == 1:
    detailsFeature(level, stdID)
```

  - Calls the `detailsFeature` function to display or process the student's details.

Option 2: Statistics

```
elif option == 2:
    statisticsFeature(level, stdID)
    request_history.append("Statistics")
    dates.append(str(now.date()))
    times.append(formatted_time)
```

  - Calls the `statisticsFeature` function.
  - Updates tracking lists:
    - **`request_history`**: Appends "Statistics".
    - **`dates`**: Adds the current date.
    - **`times`**: Adds the formatted current time.

Option 3: Transcript Based on Major Courses

```
elif option == 3:
    MajorTranscript(level, stdID)
    request_history.append("Major")
    dates.append(str(now.date()))
    times.append(formatted_time)
```

  - Calls the `MajorTranscript` function.
  - Updates the tracking lists similarly to option 2.

Option 4: Transcript Based on Minor Courses

```
elif option == 4:
    MinorTranscript(level, stdID)
    request_history.append("Minor")
    dates.append(str(now.date()))
    times.append(formatted_time)
```

- Calls the `MinorTranscript` function and logs the request.

Option 5: Full Transcript

```
elif option == 5:
    FullTranscript(level, stdID)
    request_history.append("Full")
    dates.append(str(now.date()))
    times.append(formatted_time)
```

- Calls the `FullTranscript` function and logs the request.

Option 6: Previous Transcript Requests

```
elif option == 6:
    previousRequestsFeature(request_history, dates, times, stdID)
```

- Displays a history of previous requests using `previousRequestsFeature`.

Option 7: Select Another Student

```
elif option == 7:
    newStudentFeature()
```

- Calls the `newStudentFeature` function to restart the process with a new student.

Option 8: Terminate the System

```
elif option == 8:
    terminateFeature(request_history)
    confirm = False
```

- Calls `terminateFeature` to handle termination tasks.
- Sets `confirm = False` to exit the loop.

---

4. Invalid Option Handling

```
else:
    print("Invalid option. Please try again.")
    continue
```

- If the user enters a number outside the range 1–8, they are prompted to try again.

---

5. Summary of Function Flow

1. **Menu Display:**
    - Continuously displays a menu until the user exits.
2. **Feature Selection:**
    - Depending on the input, calls the appropriate feature function.
    - Logs user actions for tracking purposes.
3. **Exit Mechanism:**
    - Option 8 allows the user to terminate the system.
    - Option 7 lets the user select another student and restart the process.

Function Definition: `detailsFeature(level, student_id)`

This function retrieves and displays detailed information about a student based on their ID and selected levels (Undergraduate, Graduate - Master's, Graduate - Doctorate). It also saves the details to a text file.

Parameters:

1. **`level`**: A list of levels (e.g., `['U']`, `['GM', 'GD']`) indicating the student's enrollment levels.
2. **`student_id`**: The unique ID of the student whose details are being queried.

---

1. Open and Read the CSV File

```
with open('studentDetails.csv', mode='r') as file:
    reader = csv.DictReader(file)
```

- Opens the `studentDetails.csv` file in read mode.
- **`csv.DictReader`**: Parses each row of the CSV into a dictionary, where column headers serve as keys.

---

2. Initialize Variables

```
details_found = False
```

- **`details_found`**: A flag to track if any student details are found.

```
level_descriptions = {
    'U': "Undergraduate",
    'GM': "Graduate - Masters",
    'GD': "Graduate - Doctorate"
}
```

- **`level_descriptions`**: Maps level codes to human-readable descriptions.

```
terms_display = {}
colleges_display = set()
departments_display = set()
```

- **`terms_display`**: Stores the number of terms completed for each level.
- **`colleges_display`**: A set of unique colleges the student has attended.
- **`departments_display`**: A set of unique departments associated with the student.

---

3. Filter Rows Matching the Student ID

```
matching_rows = []
for row in reader:
    if row['stdID'] == str(student_id):
        matching_rows.append(row)
```

- **`matching_rows`**: A list of all rows in the CSV file that match the student ID.
- Filters rows where the **`stdID`** matches the input `student_id`.

---

4. Process Matching Rows

```
for row in matching_rows:
    details_found = True
```

- If any rows match, set **`details_found`** to `True`.

Loop Through the Selected Levels:

```
for l in level:
    if l == 'U' and row['Level'] == 'U':
        terms_display['Undergraduate'] = f"Undergraduate:
{row['Terms']} term(s)"
        colleges_display.add(row['College'])
        departments_display.add(row['Department'])
```

- **Checks each level (`U`, `GM`, `GD`)**:
    - If the level matches the student's enrollment in the CSV row:
        - Updates **`terms_display`** with the number of terms completed for that level.
        - Adds the college and department to their respective sets.

Example for Graduate Levels:

```
elif l == 'GM' and row['Degree'] == 'M1':
    terms_display['Graduate - Masters'] = f"Graduate - Masters:
{row['Terms']} term(s)"
    colleges_display.add(row['College'])
    departments_display.add(row['Department'])
```

- Checks for Master's (`M1`) and Doctorate (`D1`) degrees and processes them similarly.

---

5. Output or Error Handling

If Details Are Found:

```
if details_found:
    level_display = ", ".join([level_descriptions[l] for l in level if
l in level_descriptions])
```

- **`level_display`**: Constructs a human-readable string of the student's levels (e.g., "Undergraduate, Graduate - Masters").

```
student_details = (
    f"Name: {matching_rows[0]['Name']}, "
    f"\nStudent ID: {matching_rows[0]['stdID']}, "
    f"\nLevels: {level_display}, "
    f"\nTerms: {', '.join(terms_display.values())}, "
    f"\nCollege(s): {', '.join(colleges_display)}, "
    f"\nDepartment(s): {', '.join(departments_display)}"
)
```

- Formats the student's details, including:
    - **Name**
    - **Student ID**
    - **Levels**
    - **Terms**
    - **Colleges**
    - **Departments**

Save the Details to a Text File:

```
file_name = f"{student_id}_StudentDetails.txt"
with open(file_name, "w") as txt_file:
    txt_file.write(student_details)
```

- Saves the formatted details into a file named `<student_id>_StudentDetails.txt`.

Print and Redirect:

```
print(student_details)
print(f"\nStudent details have been saved to {file_name}.")
time.sleep(10)
os.system('cls' if os.name == 'nt' else 'clear')
print("Redirecting to the menu...")
time.sleep(2)
```

- Displays the details in the console and pauses for 10 seconds.
- Clears the console screen and redirects to the main menu.

If No Details Are Found:

```
else:
    print("Student ID not found or no matching level details.")
    time.sleep(2)
```

- Informs the user that no matching details were found for the student.

---

6. Summary of Function Flow

1. **Input Validation:**
    - Reads the CSV file and filters rows based on the `student_id`.
2. **Processing Levels:**
    - Matches the student's enrollment levels and collects relevant details.
3. **Output Generation:**
    - Formats the details for console display and saves them to a text file.
4. **Error Handling:**
    - Handles cases where no matching student or levels are found.
5. **User Experience:**
    - Clears the screen and redirects to the main menu after displaying the details.

Function Definition: `statisticsFeature(level, student_id)`

The `statisticsFeature` function generates a detailed statistical summary for a student's academic performance, including averages, term-specific statistics, repeated courses, and maximum/minimum grades. It also saves the output to a text file and provides a summary to the user.

1. Function Signature

```
def statisticsFeature(levels, student_ID):
```

- **`levels`**: A string or list representing the student's academic levels (`U` for Undergraduate, `GM` for Graduate - Masters, `GD` for Graduate - Doctoral).
- **`student_ID`**: The student's unique identifier.

---

2. Helper Functions

1. **`load_csv(filepath)`**
    - **Purpose**: Loads data from a CSV file and returns it as a list of dictionaries.
    - **Process**:
        - Reads the file line by line, splitting headers and values.
        - Handles file errors gracefully.

Key Code Snippets:

```
try:
    with open(filepath, 'r') as file:
        headers = file.readline().strip().split(',')
        for line in file:
            values = line.strip().split(',')
            data.append(dict(zip(headers, values)))
except FileNotFoundError:
    print(f"Error: File {filepath} not found.")
```

---

2. **`calculate_averages(courses)`**

- **Purpose**: Calculates the total credits and the weighted average grade for a given list of courses.
- **Process**:
  - Iterates through the list of courses.
  - Calculates total credits and a weighted sum based on grades and credit hours.
  - Divides the weighted sum by the total credits for the average.

Key Code Snippets:

```
total_credits = sum(int(course['creditHours']) for course in courses)
weighted_sum = sum(int(course['Grade']) * int(course['creditHours'])
for course in courses)
average = weighted_sum / total_credits if total_credits > 0 else 0
```

---

3. **`generate_statistics(stdID, student_details_path, student_courses_path, levels)`**
   1. **Purpose**: Main function to generate statistics for the student.
   2. **Inputs**:
      - Student details file and course data file paths.
      - Academic levels to process.
   3. **Process**:
      - Filters and processes data for each academic level.
      - Collects term-specific statistics, averages, and grade extremes.
4. **Steps**:
   1. **Load Student Data**:
      - Searches for the student in `studentDetails.csv`.
        Code Example:
        ```
        student_info = next((student for student in
        student_details if student['stdID'] == stdID), None)
        ```
   2. **Filter Courses by Level**:
      - Filters courses based on the specified level (`U`, `GM`, `GD`).
        Code Example

```
if level == 'GM':

    level_courses = [course for course in student_courses if
course['Level'] == 'G' and course['Degree'] == 'M1']
```

   3. **Compute Statistics**:

Term-Based Averages: Calculates the average grades for major and minor courses in each term.

```
_, term_avg_major = calculate_averages(major_courses)
_, term_avg_minor = calculate_averages(minor_courses)
```

Highest and Lowest Grades: Identifies the maximum and minimum grades for each term.

```
max_grade = max(term_courses, key=lambda x: int(x['Grade']))
min_grade = min(term_courses, key=lambda x: int(x['Grade']))
```

Repeated Courses: Detects courses that appear multiple times.

```
repeated_courses.update([cid for cid in course_ids if
course_ids.count(cid) > 1])
```

4. **Generate Output**:
    - Aggregates the results into a formatted text summary.

---

3. File Generation

The statistics are written to a text file named `<student_ID>_Statistics.txt`.

Key Code Snippet:

```
output_filename = f"{stdID}_Statistics.txt"
with open(output_filename, "w") as file:
    file.write("\n".join(statistics_lines))
```

---

4. Redirect to Menu

After displaying the statistics, the program waits for a few seconds and then redirects to the main menu.

Key Code Snippet:

```
time.sleep(10)
os.system('cls' if os.name == 'nt' else 'clear')
print("Redirecting to main menu...")
time.sleep(2)
```

Detailed Features

1. **Error Handling**:
    - Ensures the CSV files exist.
    - Handles cases where the student or course data is missing.
2. **Modular Design**:
    - Divides tasks into helper functions like `load_csv` and `calculate_averages` for better readability and reusability.
3. **Flexible Levels**:
    - Allows analysis of one or multiple academic levels.
4. **Comprehensive Statistics**:
    - Calculates averages, identifies repeated courses, and highlights grade extremes.
5. **Output**:
    - Saves results in a user-friendly text format and prints them to the console.

Function Definition: `MajorTranscript(level, student_id)`

The `MajorTranscript` function generates a detailed transcript for a student's major subjects, organized by specified academic levels (`U`, `GM`, or `GD`). The results include term-wise averages, course details, and overall averages, which are saved to a text file.

1. Function Signature

```
def MajorTranscript(levels, student_ID):
```

- **`levels`**: A string or list of strings indicating academic levels:
  - `'U'`: Undergraduate
  - `'GM'`: Graduate - Masters
  - `'GD'`: Graduate - Doctoral
- **`student_ID`**: The unique identifier of the student.

---

2. Helper Functions

1. **`load_csv(filepath)`**
   - **Purpose**: Reads data from a CSV file and returns a list of dictionaries (rows).
     Key Code Snippets:

```
with open(filepath, 'r') as file:
    headers = file.readline().strip().split(',')
    for line in file:
        values = line.strip().split(',')
        data.append(dict(zip(headers, values)))
```

   - **Error Handling**: Catches `FileNotFoundError` and other exceptions, printing appropriate messages.

---

2. **`get_number_of_terms(student_details, level)`**
   - **Purpose**: Determines the number of terms a student has completed for a given academic level.

- **Process**:
  - Filters `student_details` based on the specified level.
  - Extracts the `Terms` field, if present.

Key Code Snippets:

```
student_info = next((row for row in student_details if row['Level'] == 'U'), None)
return int(student_info['Terms']) if student_info and 'Terms' in student_info else 0
```

---

3. **`generate_transcript(stdID, student_details_path, student_courses_path, levels, course_type)`**
   - **Purpose**: Core function that creates the transcript and writes it to a text file.
   - **Inputs**:
     - `stdID`: Student ID.
     - Paths to the student details and course data files.
     - List of academic levels.
     - Type of courses to include (`"Major"` in this case).

---

3. Core Functionality

1. **Load Data**:
   - Loads student details (`studentDetails.csv`) and course data (`<student_ID>.csv`) using `load_csv`.
2. **Filter Courses**:
   - Filters courses by academic level (`U`, `GM`, `GD`) and course type (`Major`).

Example for Graduate - Masters (GM):

```
level_courses = [course for course in student_courses if course['courseType'] == course_type and course['Level'] == 'G' and course.get('Degree') == 'M1']
```

3. **Organize Data**:
    - Groups courses by term and calculates the following:

Term Average: Weighted average for each term.

```
term_avg = sum(int(course['Grade']) * int(course['creditHours']) for
course in term_courses) / total_credits
```

Overall Average: Across all terms.

```
overall_avg = sum(int(course['Grade']) * int(course['creditHours'])
for course in level_courses) / total_credit_hours
```

        - **Number of Terms**: Retrieved using `get_number_of_terms`.
4. **Format the Transcript**:
    - Adds student details, term-wise data, and averages to the transcript.

Example Transcript Lines:

```
transcript_lines.append(f"Name: {student_info['Name']}\tstdID:
{student_info['stdID']}")
transcript_lines.append(f"\n{'*' * 10} Term {term} {'*' * 10}")
transcript_lines.append(f"{'course ID':<10} {'course name':<15}
{'credit hours':<15} {'grade':<10}")
```

5. **Save to File**:
    - Writes the transcript to `<student_ID>_MajorTranscript.txt`.

---

4. File Generation

The generated transcript includes:

- **Student Information**: Name, ID, College, Department, Major, Minor, Number of Terms.
- **Term-wise Course Details**: Course ID, name, credit hours, grades, and term average.
- **Overall Average**: Across all terms.

---

5. Redirect to Menu

After generating the transcript:

- Pauses for 10 seconds.
- Clears the screen and redirects to the main menu.

Key Code Snippets:

```
time.sleep(10)
os.system('cls')
print("Redirecting to main menu...")
time.sleep(2)
```

Detailed Features

1. **Flexible Level Selection**:
    - Handles single or multiple levels (e.g., `'U'` or `['U', 'GM']`).
2. **Error Handling**:
    - Checks for missing files, student details, or incomplete data.
3. **Customizable Course Type**:
    - The `course_type` parameter allows generating transcripts for different course types (`"Major"`, `"Minor"`, etc.).
4. **Comprehensive Statistics**:
    - Includes term-wise and overall averages.
5. **Output**:
    - Generates a clean and readable text transcript.

Function Definition: `MinorTranscript(level, student_id)`

The `MinorTranscript` function generates a transcript for a student's minor subjects, segmented by academic levels (`Undergraduate`, `Graduate - Masters`, or `Graduate - Doctoral`). The function processes student and course data from CSV files, calculates term-wise and overall averages, and outputs the transcript to a text file.

---

1. Function Signature

```
def MinorTranscript(levels, student_ID):
```

- **`levels`**: Indicates the academic levels to include:
  - `'U'`: Undergraduate
  - `'GM'`: Graduate - Masters
  - `'GD'`: Graduate - Doctoral
- **`student_ID`**: A unique identifier for the student (converted to a string).

---

2. Key Components

A. `load_csv(filepath)`

Purpose: Reads a CSV file and converts its content into a list of dictionaries.

How It Works:

1. Opens the file and reads the first row as column headers.
2. Reads subsequent rows, creating a dictionary for each row by pairing column headers with values.

Error Handling:

- **FileNotFoundError**: Displays an error if the file is not found.
- **General Exceptions**: Catches other errors during file reading.

B. `get_number_of_terms(student_details, level)`

Purpose: Determines the number of terms completed by the student for a given academic level.

Logic:

1. Filters `student_details` based on the `level`:
    - `'U'`: Undergraduate (checks `Level == 'U'`).
    - `'GM'`: Graduate - Masters (checks `Degree == 'M1'`).
    - `'GD'`: Graduate - Doctoral (checks `Degree == 'D1'`).
2. Returns the `Terms` value for the matched record (or `0` if no match is found).

C. `generate_transcript(...)`

This is the core function responsible for creating the transcript. Below is a detailed breakdown of its steps.

---

3. Steps in `generate_transcript`

Step 1: Load Data

```
student_details = load_csv(student_details_path)
student_courses = load_csv(student_courses_path)
```

- Loads `studentDetails.csv` and the student-specific course file (`<student_ID>.csv`).

Step 2: Find Student Information

```
student_info = next((student for student in student_details if
student['stdID'] == stdID), None)
if not student_info:
    print(f"No details found for student ID: {stdID}")
    return
```

- Searches for the student's information in `student_details` using the `stdID`.
- If no match is found, the function exits with a message.

Step 3: Handle Multiple Levels

```
if isinstance(levels, str):
    levels = [levels]
```

- Ensures `levels` is always treated as a list (even if a single level is provided).

Step 4: Filter Courses by Level

For each academic level:

Filters courses based on `courseType` (`Minor`) and `Level`:

```
if level == 'GM':
    level_courses = [course for course in student_courses if
course['courseType'] == course_type and course['Level'] == 'G' and
course.get('Degree') == 'M1']
```

Groups courses by term:

```
all_terms = sorted(set(course['Term'] for course in level_courses))
```

Calculates **Term Average** and **Overall Average**:

```
total_credits = sum(int(course['creditHours']) for course in
term_courses)
term_avg = sum(int(course['Grade']) * int(course['creditHours']) for
course in term_courses) / total_credits
```

Uses `get_number_of_terms` to retrieve the number of terms for the level.

Step 5: Structure Data for Transcript

- Organizes the data into a dictionary:

```
transcript_data[level] = {
    "terms": level_data,
    "overall_avg": overall_avg,
    "student_info": {
        "Name": student_info['Name'],
        "stdID": student_info['stdID'],
        "College": student_info['College'],
        "Department": student_info['Department'],
        "Major": student_info['Major'],
        "Minor": student_info.get('Minor', 'N/A'),
        "Number of terms": number_of_terms
    }
}
```

Step 6: Generate the Transcript

- Formats the transcript into a readable text layout.
- For each level, includes:
    1. **Student Details**: Name, ID, College, Department, Major, Minor, and Number of Terms.
    2. **Term-Wise Courses**: Lists courses with their IDs, names, credit hours, and grades.
    3. **Averages**: Displays term-wise and overall averages.

Step 7: Save to File

- Writes the formatted transcript to `<student_ID>_MinorTranscript.txt`.

Step 8: Redirect to Menu

After creating the transcript:

1. Pauses for 10 seconds.
2. Clears the screen twice, displaying "Redirecting to main menu...".

Key Code:

```
time.sleep(10)
os.system('cls')
print("Redirecting to main menu...")
time.sleep(2)
os.system('cls')
```

---

4. Key Features

1. **Customizable Levels**:
    - Can generate transcripts for one or multiple levels (`'U'`, `'GM'`, `'GD'`).
2. **Comprehensive Averages**:
    - Calculates term-wise and overall averages, weighted by credit hours.
3. **Dynamic Output**:
    - Generates transcripts based on the student's actual data.
4. **Error Handling**:
    - Handles missing files, missing student details, and empty course lists gracefully.

Function Definition: `FullTranscript(level, student_id)`

The `FullTranscript` function generates a full transcript for a student, including details about both major and minor courses. It calculates term averages and overall averages for different academic levels (Undergraduate, Graduate - Masters, and Graduate - Doctoral) and outputs the data into a text file.

1. Function Definition and Parameters

```
def FullTranscript(levels, student_ID):
```

- `levels`: Specifies the academic levels (`'U'` for Undergraduate, `'GM'` for Graduate - Masters, `'GD'` for Graduate - Doctoral).
- `student_ID`: The unique identifier for the student.

---

2. Nested Helper Functions

`load_csv(filepath)`

Loads a CSV file and parses its contents into a list of dictionaries:

- **Input**: File path (`filepath`).
- **Output**: List of dictionaries with headers as keys and row values as values.
- Handles errors such as file not found or other exceptions.

`calculate_averages(courses)`

Calculates the total credit hours and the weighted average grade for a list of courses:

- **Input**: List of courses (each course is a dictionary with `Grade` and `creditHours`).
- **Output**: Tuple containing:
    1. `total_credits`: Sum of all credit hours.
    2. `average`: Weighted average grade.

---

3. `generate_full_transcript`

The main logic for generating the transcript:

```
def generate_full_transcript(stdID, student_details_path,
student_courses_path, levels):
```

- **Parameters**:
  - `stdID`: Student ID (converted to a string).
  - `student_details_path`: Path to the `studentDetails.csv` file.
  - `student_courses_path`: Path to the student-specific courses file (`<student_ID>.csv`).
  - `levels`: Academic levels.

Detailed Steps

Step 1: Load CSV Data

```
student_details = load_csv(student_details_path)
student_courses = load_csv(student_courses_path)
```

- Load the student details and course data from the respective CSV files.

Step 2: Filter and Process Student Information

```
student_info = next((student for student in student_details if
student['stdID'] == stdID), None)
```

- Retrieve the student record matching the `stdID`.
- If no record is found, print an error and exit.

Step 3: Process Levels

```
for level in levels:
```

Iterates over the specified levels (`'U'`, `'GM'`, `'GD'`):

1. Filters courses for the given level (e.g., `'G'` for graduate courses, `'U'` for undergraduate).
2. Group courses by term.
3. Separates courses into major and minor categories.

Step 4: Calculate Term and Overall Averages

For Each Term:

```
major_courses = [course for course in term_courses if
course['courseType'] == 'Major']
minor_courses = [course for course in term_courses if
course['courseType'] == 'Minor']
total_credits_major, term_avg_major =
calculate_averages(major_courses)
total_credits_minor, term_avg_minor =
calculate_averages(minor_courses)
```

- **Major Courses**: Courses categorized as `'Major'`.
- **Minor Courses**: Courses categorized as `'Minor'`.
- Calculates:
    - `term_avg_major`: Average grade for major courses.
    - `term_avg_minor`: Average grade for minor courses.
    - `term_avg`: Combined average for both major and minor courses.

Overall Level Averages:

```
total_credit_hours, overall_avg = calculate_averages(level_courses)
```

- Calculates the overall average grade for all courses in the level.

Step 5: Generate Transcript Data

```
transcript_data[level] = {
    "terms": level_data,
    "overall_avg": overall_avg,
    "student_info": {
        "Name": student_info['Name'],
```

```
        "stdID": student_info['stdID'],
        "College": student_info['College'],
        "Department": student_info['Department'],
        "Major": student_info['Major'],
        "Minor": student_info.get('Minor', 'N/A'),
        "Number of terms": num_terms
    }
}
```

- **`transcript_data`**: Stores the processed transcript information:
    - Per-term averages and courses.
    - Overall averages.
    - Student details (e.g., name, major, minor, etc.).

Step 6: Write to Transcript File

```
output_filename = f"{stdID}_FullTranscript.txt"
with open(output_filename, "w") as file:
    file.write("\n".join(transcript_lines))
```

- Generates a text file with the transcript details, formatted for readability.

---

7. Formatting Transcript for Output

The transcript includes:

- **Header**: Student details (name, ID, college, department, major, minor).
- **Level-Specific Data**:
    - Term-by-term breakdown:
        - Courses (ID, name, credit hours, grade).
        - Term averages (major, minor, and combined).
    - Overall averages.

---

8. Post-Processing

```
time.sleep(10)
os.system('cls')
print("Redirecting to main menu...")
```

```
time.sleep(2)
os.system('cls')
```

- Pauses for 10 seconds, clears the console, and prints a message before redirecting back to the main menu.

Function Definition:

```
previousRequestsFeature(request_history, dates, times, stdID)
```

The `previousRequestsFeature` function manages and displays a student's previous transcript requests. It also gives the option to save the history to a file for future reference.

1. Function Parameters

- **`request_history`**: A list containing previous transcript requests (e.g., types of transcripts or levels requested).
- **`dates`**: A list of dates corresponding to each request in `request_history`.
- **`times`**: A list of times corresponding to each request in `request_history`.
- **`stdID`**: The student's unique identifier, used for naming the file where request history might be saved.

---

2. Validating Input

```
if not request_history or not dates or not times:
    print("No request history available.")
    time.sleep(2)
    return
```

- **Purpose**: Ensures that all three input lists are non-empty.
    - If any of the lists are empty, it prints a message, waits for 2 seconds, and exits the function.
- This validation ensures the program doesn't fail when trying to access elements in empty lists.

---

3. Displaying Request History

```
print("\nPrevious Transcript Requests")
print("Request\tDate\tTime")
```

```
print("=====================================")
for i in range(len(request_history)):
    print(f"{request_history[i]}\t{dates[i]}\t{times[i]}")
```

- **Purpose**: Displays the previous transcript requests in a tabular format.

---

4. Option to Save History to File

```
save_to_file = input("Would you like to save this request history to a file? (Y/N): ").strip().lower()
```

- **Purpose**: Asks the user if they want to save the history to a file.
- **Input Validation**: Converts the input to lowercase (`'y'` or `'n'`) to ensure case-insensitivity.

4.1. File Name and Existence Check

```
file_name = f"{stdID}_PreviousRequests.txt"
file_exists = os.path.exists(file_name)
```

- **File Name**: The file name is dynamically created using the student's ID (`stdID`), ensuring uniqueness for each student.
- **File Existence Check**: Determines if the file already exists in the directory.

4.2. Preparing Content for the File

```
content = ""
if not file_exists:
    content += "Request\tDate\tTime\n"
    content += "=====================================\n"
```

```
for i in range(len(request_history)):
    content += f"{request_history[i]}\t{dates[i]}\t{times[i]}\n"
```

- **New File**: If the file doesn't exist, a header (`Request\tDate\tTime`) is added to the content.
- **Appending Requests**: Loops through the `request_history`, `dates`, and `times` to format each request into a tabular structure.

4.3. Writing to the File

```
with open(file_name, "a") as file:
    file.write(content)
```

- Opens the file in append mode (`"a"`):
    - If the file doesn't exist, it will be created.
    - If it exists, the content is appended to the end.

4.4. Confirmation Message

```
print(f"Request history has been {'updated' if file_exists else
'saved'} to {file_name}.")
```

- **Message**: Indicates whether the file was newly created or updated.

---

5. Handling "No Save" Choice

```
else:
    print("Request history was not saved to a file.")
```

- If the user chooses not to save (`'n'`), a message is printed.

---

6. Final Steps

```
input("Press Enter to return to the menu.")
time.sleep(3)
os.system('cls' if os.name == 'nt' else 'clear')
```

- **Pause**: Waits for the user to press Enter before proceeding.
- **Screen Clearing**: Clears the screen to maintain a clean interface:
    - Uses `cls` for Windows (`os.name == 'nt`).
    - Uses `clear` for other operating systems.

Flow Summary

1. **Input Validation**: Ensures there is a history to display.
2. **Display History**: Shows the request history in a readable table format.
3. **Save Option**: Asks the user if they want to save the history.
    - **If Yes**: Appends or writes the data to a file.
    - **If No**: Skips saving.
4. **Final Steps**: Pauses and clears the screen, redirecting to the main menu.

Function Definition:
`terminateFeature(request_history)`

The `terminateFeature` function is a simple utility that displays the total number of requests made during the current session.

1. Function Parameters

- **`request_history`**: A list that contains all the requests made by the user in the session. Each element in the list represents a single request (e.g., a transcript request).

---

2. Function Logic

2.1. Print Separator Line

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

- **Purpose**: Adds a visual separator line to make the output more readable and distinguishable in the console.

2.2. Display Number of Requests

```
print(f"Number of requests: {len(request_history)}")
```

- **`len(request_history)`**:
  - The `len()` function calculates the total number of elements in the `request_history` list.
  - This gives the total number of requests made.
- **Output Message**:
  - The message dynamically displays the calculated number of requests.

Purpose of the Function

- This function provides a concise summary of the total requests made during the session.
- It serves as part of a termination process, likely called before the program exits, to give a final report.

Example Interaction

Input:

```
request_history = ["MajorTranscript", "FullTranscript", "Statistics"]
terminateFeature(request_history)
```

Output:

```
======================================
Number of requests: 3
```

Flow Summary

1. **Separator Line**: Prints a line for clear formatting.
2. **Request Count**: Calculates and displays the total number of requests using the length of `request_history`.

Function Definition: `newStudentFeature()`

The function is designed to clear the terminal screen, notify the user about transitioning to a new student context, and redirect the user to the main menu by calling another function `startFeature`.

1. Clearing the Terminal Screen:

```
if os.name == 'nt': #for windows
    os.system('cls')
else:
    os.system('clear') #for linux and mac
```

- **Purpose**: Clears the terminal to provide a clean slate for further interactions.
- **How It Works**:
  - The `os` module is used to interact with the operating system.
  - `os.name` determines the operating system:
    - `'nt'`: Refers to Windows systems.
    - Other systems (like Linux and macOS) use POSIX-compliant commands (`os.name` is `'posix'`).
  - Depending on the operating system:
    - Windows uses the `cls` command.
    - Linux and macOS use the `clear` command.
  - The `os.system()` function executes the appropriate terminal command.

---

2. Notifying the User:

```
print("Preparing for a new student...")
time.sleep(2)
```

- **Purpose**: Notifies the user that the system is transitioning to handle a new student.
- **How It Works**:
  - The `print()` function displays the message `"Preparing for a new student..."`.
  - `time.sleep(2)` pauses the execution for 2 seconds, allowing the user to read the message before proceeding.

---

3. Redirecting to the Main Menu:

```
print("Redirecting to the main menu...")
time.sleep(1)
startFeature()
```

- **Purpose**: Informs the user about the redirection and then redirects to the main menu.
- **How It Works**:
    - The `print()` function displays `"Redirecting to the main menu..."`.
    - `time.sleep(1)` pauses for 1 second before proceeding.
    - `startFeature()`: A function call (assumed to be defined elsewhere in the code) that initializes or navigates to the main menu.

Behavior:

When `newStudentFeature` is called:

1. The terminal is cleared.
2. The user sees a message indicating preparation for a new student.
3. After a 2-second delay, the user is informed that the system is redirecting to the main menu.
4. After a 1-second delay, the `startFeature()` function is executed.