The Sorting Hat: A Program To Create Groups Based On Student Preferences

Executive summary: The SEG group brainstormed an idea to help create groups of students according to their working habits/preferences. This concept was formed to refine the random group allocation process currently in place so that students obtain the benefits of random allocation (i.e. meeting new people, simulating the workforce) and have a better chance of being in a compatible group with similar working styles.

Aim: To develop a standalone program that creates groups of students based on student preferences for group work and their working habits.

Approach:

Coded in python, the sorting hat program uses pop-up message boxes to step the user through the program. The user chooses a csv file, and this csv file is the Canvas quiz download containing the results of the student preferences. The program then reads this input and creates a student object for each student's preference.

To form groups, the function alphabetically sorts the students according to the meeting time preference first, and then by their preferred mode of meeting. Using this sorted list of students, the program forms groups of the desired number (or one less than the desired number).

The sorting hat writes these groups to a csv file named by the user and then the program ends.

Technical Step-Through:

Input: The csv file that contains the student identifier (i.e. SID or full name), their preference for the time of the day to meet and their preference for mode of meeting. The students in such csv file are from the same tutorial.

name	id	sis_id	section	section_id	section_sis_i	submitted	attempt	3360577: Wh	1	3360579: Pre	1	n correct	n incorrect	score
Alessia Perni	270286	510414807	Group Work A	225239		2024-07-02 0	1	A.Morning,B.	0	A.In-person	1	1	1	1
Brandon		510414808						A.Morning,B.Noon		A.In-person				
Dylan		510414809						A.Morning,D.Evening		A.In-person				
Mo		510414810						B.Noon,C.Night,D.Evening		A.In-person				
Idraki		510414811						B.Noon, C.Night, D.Evenin		A.In-person				

The input file is read and each student is created as a Student object. A list of Student objects is returned:

```python
class Student:
    '''
    This class is to represent a student. A student must have a name, SID and preferred study times and modes.
    '''
    def __init__(self, name, sid, preferred_time, preferred_mode):
        self.name = name
        self.sid = sid
        self.preferred_time = preferred_time
        self.preferred_mode = preferred_mode
```

```python
def collect_preferences(file_path):
    '''
    This function collects the relevant data from the .csv file provided by the user. This function anticipates
    that the file is of the type of a canvas quiz download, containing the student's name, SID and quiz answers
    in certain columns.
    MODIFY THIS FUNCTION IF THE CANVAS QUIZ QUESTIONS CHANGE.
    Input: file path to the .csv file.
    Output: list of students with their name, SID, preferred times and preferred modes.
    '''
    students = []
    with open(file_path, 'r') as file:
        csv_reader = csv.reader(file)
        next(csv_reader)  # Skip the header row
        for row in csv_reader:
            # Check if the row has at least 15 columns - this is the format for Canvas quiz downloads.
            if len(row) < 15:
                    return None
            # Unpack data from row
            name, _, sid, _, _, _, _, _, preferred_times_answer, _, preferred_mode, _, _, _, _ = row
            # Unpack preferred times to a list
            preferred_time = preferred_times_answer.split(",")
            students.append(Student(name, sid, preferred_time, preferred_mode))

    return students
```

The form_groups function initially figures out how many groups can be made with the user's desired group size. If the total number of students is not exactly divided by the desired group size, then the program works out how many groups need to have a group with desired size minus one:

```python
def form_groups(students, group_size):
    '''
    This function is where the sorting happens. It firstly determines how many groups can be in the desired
    size (i.e. if the total number of students divided by the desired group size contains remainders) - note
    that the program opts down, so if the user enters 5 for group size then the range of group size is 4-5.
    GThe function then sorts students according to their preferred time of the day and then sorts students based on their
    preffered mode of meeting. Finally it groups the students according to this sorting.
    Input: list of students, integer of desired group size.
    Output: list of groups, list of dominating preferences for each group.
    '''
    groups = []
    group_preferences = []  # This will store the dominant preference for each group

    # Determine the actual size of each group and the optimal number of groups
    total_students = len(students)
    num_full_groups = total_students // group_size
    remainder = total_students % group_size

    # if there exists a remainder smaller than 1-less of the desired group size, check how many groups need to
    # contain a smaller size to meet the amount of total students.
    if 0 < remainder < group_size-1:
        adj_group_size = group_size-1
        num_smaller_groups = total_students // adj_group_size
        remainder = total_students % adj_group_size
        if 0 < remainder < adj_group_size-1:
            num_full_groups = remainder
            num_smaller_groups = num_smaller_groups - remainder
    else:
        if remainder == 0:
            adj_group_size = group_size
```

The function then sorts the list of students first by preferred time and then by preferred mode of meeting. There is also a counter for the amount of groups so that the program knows when to switch the group size (if applicable):

```
# Sort students by preferred time and then by preferred mode
sorted_students = sorted(students, key=lambda s: (s.preferred_time, s.preferred_mode))

# Group students according to desired group size
current_group = []
current_group_preferences = {'times': {}, 'modes': {}}

group_size_counter = 0
```

The program then iterates through each student in the list of sorted students and appends them to the current group. I added a couple lines there to track the preferences in each group, just as an FYI for testing. If the number of students in the current group is equal to desired group size then that group is added to the list of groups. I have added some lines in there to record the dominant preference, for testing reasons. Then the current group is reset to be able to add new students to it:

```
for student in sorted_students:
    current_group.append(student)


    # Count preferences for the current group
    for time in student.preferred_time:
        current_group_preferences['times'][time] = current_group_preferences['times'].get(time, 0) + 1
    current_group_preferences['modes'][student.preferred_mode] = current_group_preferences['modes'].get(student.preferred_mode, 0) + 1

    # switch group size to the smaller alternative if required.
    if group_size_counter == num_full_groups:
        if len(current_group) == adj_group_size:
            groups.append(current_group)

            # Identify dominant preferences for the group
            dominant_time = max(current_group_preferences['times'], key=current_group_preferences['times'].get)
            dominant_mode = max(current_group_preferences['modes'], key=current_group_preferences['modes'].get)
            group_preferences.append((dominant_time, dominant_mode))
            # Reset for the next group
            current_group = []
            current_group_preferences = {'times': {}, 'modes': {}}

    # if the current group has reached the desired size, add it to the list of groups.
    if len(current_group) == group_size:
        groups.append(current_group)
        group_size_counter += 1

        # Identify dominant preferences for the group
        dominant_time = max(current_group_preferences['times'], key=current_group_preferences['times'].get)
        dominant_mode = max(current_group_preferences['modes'], key=current_group_preferences['modes'].get)
        group_preferences.append((dominant_time, dominant_mode))
        # Reset for the next group
        current_group = []
        current_group_preferences = {'times': {}, 'modes': {}}

# Print group preferences
# for i, preferences in enumerate(group_preferences):
#     print(f"Group {i+1} is for {preferences[0]} and {preferences[1]}")

return groups, group_preferences
```


Output: A csv file with the list of groups and the students in each one. At present, the student names are used, but this can easily be modified to return their SIDs:

```python
def write_groups_to_csv(groups, output_file_path):
    '''
    This function writes the groups to a .csv file. It uses the students' names but can be easily modified to
    use their SIDs.
    Input: list of groups, file path for output.
    Output: None
    '''
    with open(output_file_path, 'w', newline='') as file:
        csv_writer = csv.writer(file)
        for i, group in enumerate(groups):
            # print(f"Group {i+1}: {[student.name for student in group]}")
            group_data = [f"Group {i+1}"] + [student.name for student in group]
            csv_writer.writerow(group_data)
```

Group 1	Crystal	Kaylee	Alessia Pernice	Brandon	Colin
Group 2	Oscar	Sam	Matthew	Dylan	Lexi
Group 3	Queena	Idraki	Ryan	Mo	
Group 4	Yvette	Jeanette	Gian	Lincoln	
Group 5	Anita	Isabella	Sama	Anand	

The main function runs and calls the other functions. There are message boxes that pop up to the user and allows them to choose their input csv file, desired group size and the name for the output csv file:

```python
def main(root):
    '''
    There is no GUI window, but there are dialog boxes that pop up on the user's screen to instruct them and
    communicate what is happening.
    '''

    file_path = filedialog.askopenfilename(title="Select CSV file with student preferences")
    if not file_path:
        return

    group_size = simpledialog.askinteger("Input", "Please enter the desired size of each group:",
                                         parent=root, minvalue=2, maxvalue=15)
    if not group_size:
        return

    students = collect_preferences(file_path)
    if students is None:
        messagebox.showerror("File Error", "Failed to parse the file. Please check the file format and retry.")
        return
    groups, _ = form_groups(students, group_size)

    output_file_path = filedialog.asksaveasfilename(title="Save the output CSV file", defaultextension=".csv")
    if not output_file_path:
        return

    write_groups_to_csv(groups, output_file_path)
    messagebox.showinfo("Success", f"Groups have been written to {output_file_path}\nThank you for using the Sorting Hat!")

if __name__ == "__main__":
    root = tk.Tk()
    root.withdraw()  # we don't want a full GUI, so keep the root window from appearing
    messagebox.showinfo("Welcome", "Welcome to the Sorting Hat Program. \nPlease select the CSV file with student preferences.")
    main(root)
```

Issues:

There are 2 main issues preventing the release of the Sorting Hat.

The first issue is creating a standalone executable file of the sorting hat. The aim of the sorting hat is to be an easy-to-use application that anyone can download and open to get the program running. Currently the sorting_hat.py file works well but requires users to have downloaded python, installed relevant libraries and use terminal to run the python file. I have used Pyinstaller to create a standalone executable of the sorting_hat.py file however this only works on computers with arm64 architecture (e.g. MacBook M2 chip) and still sometimes requires installation of some python libraries.

The second issue is the approach to forming groups. The aim of the sorting hat is to create groups of students in a way that maximises their group-work habits/preferences from the Canvas quiz. Currently, the program sorts the list of students according to their preferred time of meeting and then sorts students according to their preferred mode of meeting. The issue is that the preferred time of meeting can be a combination of answers, for example: “A. Morning, B. Noon” or “A. Morning, B. Noon, C. Evening”. The second sorting of students (i.e. preferences for mode of meeting) only sorts students with the exact same combination of meeting time preferences and will therefore make groups of students with potentially different mode of meeting preferences

. This is a very simple, heuristic approach that seems to work well enough for our intention, I did try a linear regression model but that was quite unsuccessful.

Other issues to consider:

- Unsatisfied students due to shortage of students with similar preferences.
- Might need to distinguish between tutorials if there is not a setting to do that through the Canvas quiz.

Recommendation:

Fix the 2 main issues mentioned above and dedicate some time to testing the program. Test cases can be made or to keep within budget (and timeline), black-box user testing can be conducted to ensure robustness for the time being. There are 20 hours left out of the allocated 30 to achieve this. Recommended to add more resourcing if after 20 hours, the 2 main issues have not been resolved.