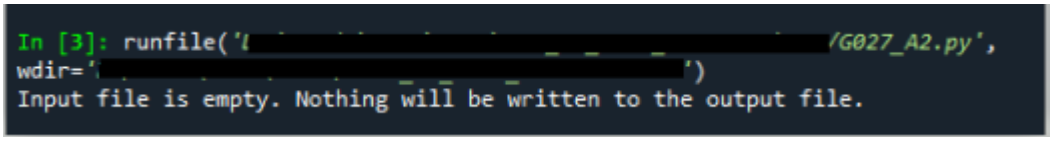
Solution Details:

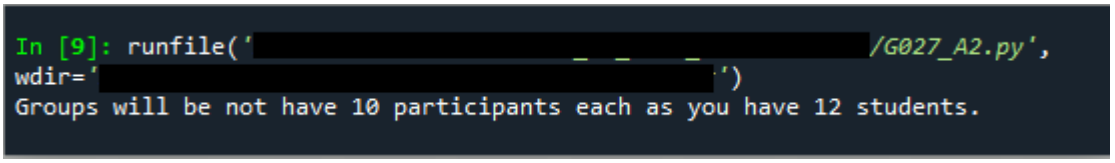
- Reading input file as per the input specification in the sample input file.

- As we read the input file, we create a list (student_data). While creating the list, we skip the first line (header) and filter out the lines that are empty or do not contain exactly two elements. Note - We are assuming all the student IDs to be of length 4.

- After reading the input data, we check if the student_data list is empty. If it's empty, it prints a message indicating that the input file is empty, and nothing will be written to the output file.



- If the student_data has more or less than 20 students' details, we abort the process as each group must have 10 students i.e., one per location.

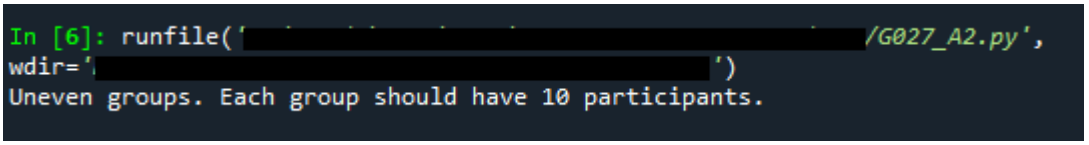


- Now, to sort the students based on their ids, we are using the Quick sort algorithm (quick_sort) which uses divide and conquer strategy. The function quick_sort takes a list of student IDs and the type of sort (ascending and descending) as input and then sorts the data based on student IDs. The pivot is chosen as the middle element and then the list is divided into left and right sublists based on whether the student ID is less than or greater than the pivot. And again, quick sort is applied the same on the sub lists recursively.

```
def quick_sort(student_data, ascending=True):
    if len(student_data) <= 1:
        return student_data

    pivot = student_data[len(student_data) // 2][0]
    left = []
    right = []
    equal = []

    for student in student_data:
        if student[0] == pivot:
            equal.append(student)
        elif (student[0] < pivot and ascending) or (student[0] > pivot and not
ascending):
            left.append(student)
        else:
            right.append(student)

    equal = [student for student in student_data if student[0] == pivot]
```

return quick_sort(left, ascending) + equal + quick_sort(right, ascending)

- Once the list is sorted, we divide it into two groups, G1 and G2. G1 will have students whose IDs start from 0 to 4 and G2 will have students whose IDs start from 5 to 9. Then we check if both groups have 10 participants, if not we do not proceed as each group must have 10 participants.



- If we have 10 participants in each group, we write the details of Group G1 to the output file, including student IDs and names, in ascending order. And to get details for Group G2 in descending we use the quick sort again to sort Group G2 student details in descending order. List group2_descending holds the sorted data.

- Then we are writing details of students at the 3rd and 7th locations for both Group G1 and Group G2. This includes student IDs, names, and the clue locations.

- We are also dealing with error using error handling try and except blocks to catch potential exceptions. It handles the cases like input file not being found and other general exceptions by printing appropriate error messages.

- The time complexity of quick sort is O(nlogn).

- We have used list data structure because lists make it easy to read and process data from a file line by line. Also, Lists in python can store heterogeneous data types, which means you can have a mix of strings, integers, and other data types in the same list. This flexibility is useful when dealing with diverse data, such as student IDs and names. Lists support indexing, which allows easy access to individual elements. This is important when you need to sort.

- One alternate recursive algorithm which uses divide and conquer strategy that can be used to solve this problem is Merge Sort.Though, Merge sort is a stable sorting algorithm with consistent O(nlogn) time complexity, making it a good choice when worst-case performance is concern and it is easier to understand and implement, when it comes to small input sizes and partially sorted data quick sort can be faster than merge sort.In our case where we have small input dataset, both sorting algorithms should perform well and almost similar.