**CMPT 2021 Ticketing System Project**

This assignment was an interesting project that allowed us to test our knowledge on data structures, queues, and advanced programming logic. I had a lot of fun writing this program! In this report I will be walking through my program function by function. I will first do a high-level overview of the program.

**High-Level Overview**

The task was to create a dynamic ticketing system to performs tasks such as booking and cancelling a ticket, adjusting ticket details, and expanding the venue all while managing the waitlist of the ticketed event. I chose to utilize a dynamic array as my data structure for the venue (ie. seating) and the queue. This choice was due to its ability to allow a O(1) time complexity to access a seat by a row and column value to see if it’s already booked and whether a specific ticket ID was in the list/queue. This was also chosen because it made geometric sense to view the grid as a row by column list (ie. 2d list structure), as well as the ability to expand the venue without needing to shift all elements.

I have utilized a dictionary to store the values of the ticket details such as naming etc. This is due to the dictionaries ability to store items as key value pairs to allow ease of access in constant time complexity for lookups. Since the tickets are associated to their ID’s we have unique values to distinguish tickets from each other.

In terms of the structure of the code, I have created two classes; one to store the ticket object with its details (ie. name, ticket ID, etc) and the other to manage the venue. This object-oriented encapsulation helps keep the code clean while ensuring objects that are similar stay together (ie. the ticketing info vs the venue info). If we are to call the \_\_repr\_\_() function of the Ticket class we can easily view all the ticket details. If we need to adjust certain parts of the ticket we can adjust the Ticket class and not have to touch the Seating class.

For the queue I have utilized a dynamic array and the heapq library to implement a priority queue. After looking at the examples from class I decided this was most practical in my usage case as it was the most straightforward option.

A computer screen shot of a program

AI-generated content may be incorrect.

A computer screen with text and images

AI-generated content may be incorrect.Initialize the Ticket class. This class has attributes such as ID, and customer info, with seat being seat to None initially. The \_\_repr\_\_() function allows the ticket details to be printed out. We are importing 2 libraries: heapq and random.

Initialize the Seating class. We require a 20x25 grid (ie. 500 seats). The seats will be a 2D grid using list comprehension. Purchased tickets will be the dictionary storing the details of the tickets for customers with seats. Job count will be the counter for the queue (ie. used to keep track of customers with same priority and their ranking)

**Functions**

Display Seating Grid

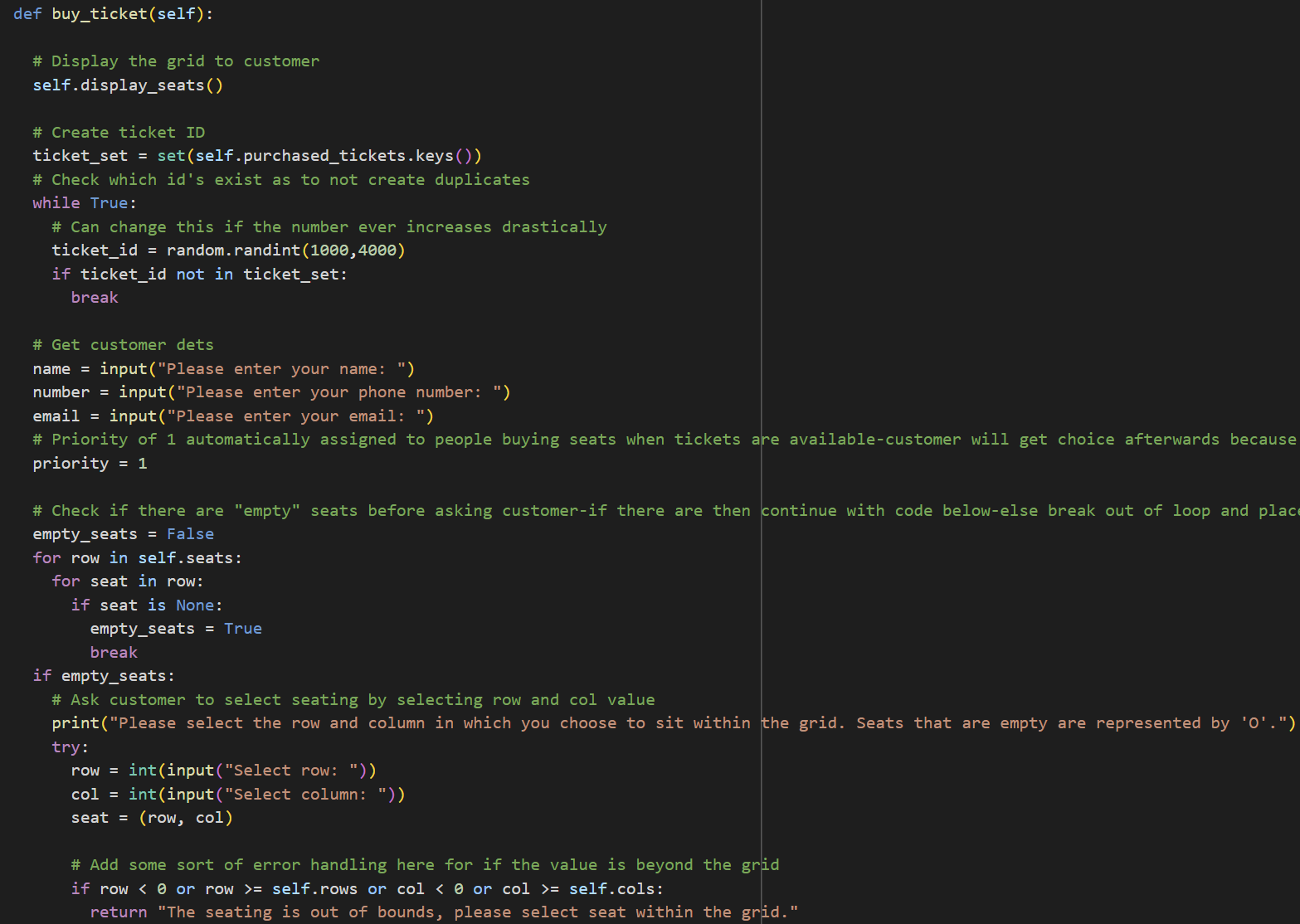
A screen shot of a computer code

AI-generated content may be incorrect.

A screenshot of a number table

AI-generated content may be incorrect.

This function took some time to get ensure the grid looked visually appealing. I wanted to visual the O’s and X’s we see in real life ticketing systems and allow the user to view which row and column they could choose from. The function first prints the columns at the top using “f”{i:2d}” which iterated through the length of the columns and spaces the values by 2 digits. We then are printing “---” for each column. This was the part of the code that I needed to use guess and check to get fully correct. Either the numbers weren’t lining up or there was some other visual issue. Next, I print the row values on the left side. Finally, we are iterating through the seats and assigning “X” or “O” depending on if the seat is “empty” (ie. if no value there).

Purchase a Ticket

A screen shot of a computer program

AI-generated content may be incorrect.

A black background with white text

AI-generated content may be incorrect.

This function will allow the user to purchase a ticket or place the user in queue. We first call the function created previously to display the grid. Next, we create a set of the ticket ID’s. This will allow us to create unique ID’s of 4 digits using the random library. The program loops the ID generation until it finds a value that is not in the set. The user will then be prompted to input their details and will automatically be assigned a priority of 1. This will change depending on if we place the customer in queue or not. We must first check if there are any empty seats in the venue before moving onto our questions to the user. Iterate through the columns and rows to check this in the loop. If seats are empty, we can continue to assign customer seating, or if they are not we must move to our queue. For the first option (ie. empty seats available), we continue to prompt the user assigning a row and column value to their seat selection. We then create a ticket object to store the users ticket info, appending their ticket to the dictionary of sold tickets with seats (ie. purchased\_tickets) and filling in our seat with that customer. If there are no seats the program will continue to the else loop. It will ask the user its priority with 1 being the highest paying (ie. top choice) and 3 being the lowest. We will then create a ticket object assigning the customers details (ie. name, priority, ID, etc.) and pushing it to the queue using heapq library. Note: I added to the print message at the return of the function stating that users in the waitlist will be assigned a seat once one opens up.

Cancelling Ticket

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

In this function we handle ticket cancellation. We place everything into a try-except block to handle if the ticket ID is not found in either the queue or the purchased tickets dictionary. We first search for the ticket in the purchased tickets dictionary. If found, we assign it to a variable and obtain the tickets seat value (ie. row & column value). Then we assign a value of none to the actual seat in the grid and delete the ID from our dictionary. Once this is done, we handle the queue. If there are values in the queue we obtain the priority, order (found through the job\_count variable assigned during ticket purchase), and ticket id of the next item in queue. The heapq library ensures the queue is kept in order through using the job\_count variable and priority values to rank the tickets in terms of order placed and priority. Next, we assign the values of the emptied seat to the next person in queue and update the values in the seating data structure (ie. the array) and the dictionary of purchased tickets with a seat. If the ticket ID is not found in the dictionary, we look at the queue. To do this, we first create a new queue and iterate through the queue until we can find a match for the ticket ID. This will be the accessing the ticket ID as the queue is a tuple (priority, job\_count, ticket\_ID) and then from that we will be accessing the ticket\_id attribute of that object. If the ID’s don’t match the ticket is appended to the new list and the queue is then “heapified” (ie. sorted) outside the loop.

Modifications to Tickets

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

This function handles modifications to the ticket. It is once again nested inside a try-except block to handle the case where the id is found in neither the dictionary nor the queue. The values for a new row and column, and priority will be inputted as parameters into this function. They will all be assigned None to handle cases where only 1 value needs to be adjusted. We first do some error handling where we look at the boundaries of the new seat values if they are given. Within the main function we will be displaying the grid to allow the user to choose an empty seat. Next, we will look in the purchased tickets with seats to see if our ID is found. If it is, we can loop through the items we want to change (ie. either priority or seat). We will not be adjusting the queue here as there will be no new empty seats-only switching from one seat to another. If there were values in queue they would have been placed before the customer was allowed to switch seats within another function. If the ticket ID was not found in the dictionary of tickets with a seat, we look in the queue. If the ticket is found here, we can adjust the priority and re-heapify the new list just as we did in the previous function.

Lookup your Ticket Details

A screenshot of a computer

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.

We can use this function to view our ticket details by inputting a ticket ID. We check in both the queue and dictionary. For the queue we use the indexing of the queue as explained above for previous functions.

A screen shot of a computer

AI-generated content may be incorrect.Expand the Venue Seating

This function handles the case of venue expansion in case the venue team wishes to expand the seating. We first define the capacity and then determine how many new rows we must add. We keep the columns the same as this value is usually fixed in venues when adjusting seating. To update the grid, we iterate through the length of the new rows and append a spot for each position in the row. Finally, we update the value of the rows. To handle the queue, we start by iterating through the seats to see if any are empty as the same time as checking if the queue is empty. We then pop the value of the next item in queue as usual and assign it the empty seat and update the dictionary. We use the variable popped to keep track of how many customers from the queue gained a seat with the seating expansion. At the end of the function, we show an updated view of the grid and state how many people were assigned seats.

A screen shot of a computer

AI-generated content may be incorrect.Main Function to Call All Others

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

This is our main function where we will be interacting with the user to determine which functions to call. We will ask the user to choose between booking, cancelling, modifying, etc a ticket and then looping through based off the choices.

I have added some test cases to the .ipynb file that can be viewed. I have also adjusted the grid to initialize a 1x1 grid to test the seating expansion and how it handles the queue. The time complexities of each case will vary depending on what we are performing. #######ADD INFO

**AI Generated Portion**

Through prompting I was able to work with Chat GPT to create an optimized code. The ticketing system still utilizes two classes for the ticket and venue management. There are some changes, and new features added such as:

* Optimization for time and space complexity through:
  + Maintaining available seats set to turn searching for an empty seat into O(1) complexity
  + The image shows the optimized seat check feature. This is excellent in that it creates a dictionary of all the available seats and removes as we assign seats. Rather than searching through the grid of rows and columns to find seating each time we can look through this list of tuples instead, reducing the complexity from O(n^2) to O(1)

A computer screen shot of a program

AI-generated content may be incorrect.

* + Indexing tickets by attributes to improve lookups and searchability
    - Can locate all tickets with a given customer name without scanning through every record.
    - The implementation of this feature is very interesting. Chat GPT has created helper functions to store each of the attributes. If a customer has a specific field, the program will combine the fields to provide the ticket info

A screen shot of a computer program

AI-generated content may be incorrect.

* + Efficient updates on seats when cancelling or modifying
* Creating a function for ticket ID generation to reduce redundancy
  + This is good for further development as we will be required to generate ID’s for each instance, we are to create a ticket object
* Enhanced feature: Group bookings
  + Create a new function that accepts a list of dictionaries containing each of the group’s details
  + A screenshot of a computer program

    AI-generated content may be incorrect.First checks if there are empty seats in the dictionary of tuples available\_seats. Then it searches for a contiguous block of seats, if none then it checks for any scattered seats, if not enough seats for the entire group some members are placed in queue. This is an excellent feature in that it handles group booking effectively with sound logic.
* Enhanced feature: Seat recommendations with dynamic pricing models
  + Calculates price of a seat based on its row with seats closer to the front costing more money as compared to seats in the back
  + Suggest best available seats for groups
  + Incorporates priorities with priority 1 being the top tier tickets
  + This is a unique creative feature for the AI to come up with when it comes to a ticket booking features which mostly consist of standard processes such as the ones we already have. The program splits up the rows into three tier ranges aligning with the 3 priorities. Then it iterates through available tickets to determine which tier the ticket falls into and uses a helper function to determine the price points based on the tiers.

A screenshot of a computer program

AI-generated content may be incorrect.

The optimization and enhancements provided through using GenAI really take the program further in terms of flexibility and usage. It has provided a faster search program that gives more options when it comes to seat selection. Overall, the program is a great piece that would perform well if implemented into a company’s systems. This is shown through the successful test cases in both the original and GenAI program. In the original program I have added a case where we initialize the grid with only one seat to properly display the success of the queuing and seat expansion functions. The test cases for GenAI are thorough and document the edge cases and error handling of the program.