Grand Prix Ticketing Experience

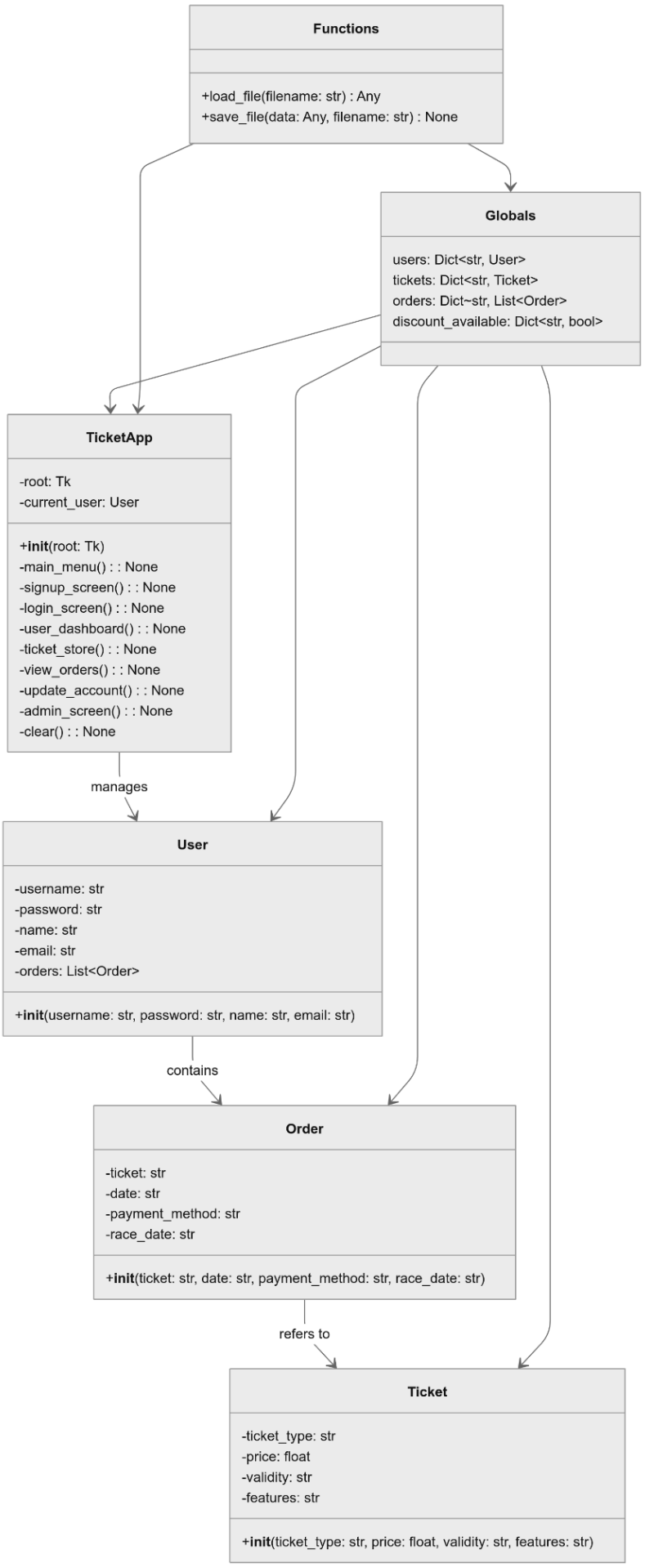
Mohammad Ismail 202105609

Nasser Lootah 202221929

ICS220 > 21383 Program. Fund.

May 13, 2025

UML Class Diagram:


Functions
+load_file(filename: str) : Any
+save_file(data: Any, filename: str) : None
Globals
users: Dict<str, User>
tickets: Dict<str, Ticket>
orders: Dict~str, List<Order>
discount_available: Dict<str, bool>
TicketApp
-root: Tk
-current_user: User
+init(root: Tk)
-main_menu() : : None
-signup_screen() : : None
-login_screen() : : None
-user_dashboard() : : None
-ticket_store() : : None
-view_orders() : : None
-update_account() : : None
-admin_screen() : : None
-clear() : : None
manages
User
-username: str
-password: str
-name: str
-email: str
-orders: List<Order>
+init(username: str, password: str, name: str, email: str)
contains
Order
-ticket: str
-date: str
-payment_method: str
-race_date: str
+init(ticket: str, date: str, payment_method: str, race_date: str)
refers to
Ticket
-ticket_type: str
-price: float
-validity: str
-features: str
+init(ticket_type: str, price: float, validity: str, features: str)

User

- Attributes: username, password, name, email, and a list of orders
- Each User can create multiple Order objects. These are stored in the orders list.
- The class models a customer account and handles individual purchase history.

So the attributes allow each user to log in securely, manage their identity in the system, and it will keep track of all of the ticket purchases they've made. The list of orders ensures that the system maintains a purchase history for each user. This relationship allows users to make several purchases over time, so with each purchase saved separately. It ensures traceability and helps in features like viewing past orders or filtering based on ticket type or race date. The class serves as the central representation of a system user, encapsulating both login credentials and the user's activity too (such as the orders) in a single place for better organization control.

Order

- Attributes: ticket, date, payment_method, and race_date
- Each Order object stores details of a single ticket purchase made by a user.
- It references ticket type as a string, not a direct object, to keep storage simple.

They capture the essential details of a transaction. So the "ticket" identifies the type of pass, and the "date" stores when the order was made, "payment_method" records how it was paid, and " race_date" indicates when the ticket will be used. It ensures every transaction is isolated, and makes it easier to manage cancellations, updates, and receipt generation. It also helps with analytics like tracking peak purchase days or popular races. For the references, the design choices reduces the memory overhead and it simplifies file serialization. However, it assumes ticket types are constant and doesn't allow for dynamic changes during runtime.

Ticket

- Attributes: ticket_type, price, validity, and features
- Represents the available ticket options in the system.
- Tickets are predefined in the code and not editable by the admin.

These define what each ticket offers. "Ticket_type" names the kind of ticket, "price" defines its costs, "validity" shows the duration or time restrictions, and "features" outline special perks like VIP access or food vouchers. For representing the available ticket options in the system, the class acts as a catalog or product list that users choose from during purchase. It also ensures that all of the tickets are standardized and structured for consistent display and logic handling. Finally for the tickets are predefined in the code and not editable by the admin, this restriction is likely for the system stability,

which ensures that the admins can't accidentally disrupt business rules. It also makes the ticket logic easier to manage since its static and tested in advance.

TicketApp

- Attribute: current_user for session tracking
- Methods handle all GUI views and user actions: login, signup, ticket purchase, admin panel
- Manages system logic and controls access to users, orders, and tickets

This attribute keeps track of who is currently logged in, which allows personalized views and actions like viewing their orders or updating their profile. Secondly, for the methods handling all GUI views and user actions: login, signup, ticket purchase, admin panel, they can connect the graphical user interface with the backend logic which will let users interact with the system smoothly , and they all ensures that the key actions from registering to purchasing are managed within one centralized app class. Lastly, for the managing system logic and controls access to users, orders, and tickets, It acts like a core controller enforcing rules , invoking necessary data updates, and providing structured navigation between different features of the application.

Relationships

- User has a one-to-many relationship with Order (composition)
- Order is associated with Ticket (by ticket type string, not object reference)
- TicketApp acts as the controller, handling GUI events, data saving, and class interaction

This means that one user can have many orders, but the thing is that each order belongs to only one user, so deleting the user will also delete their orders, reflecting real world behavior. Secondly for the Order associated with the ticket, Its indirect association simplifies data loading and reduces dependencies, but it requires consistency in ticket type naming to avoid mismatches. Lastly, for the Ticket acts as the controller, handling GUI events, data saving, and class interaction, it functions like the brain of the application, ensuring each part (like the user, order, and tickets) works together correctly. It also updates the interface based the current state or action

Assumptions

- Tickets are hardcoded for simplicity and to meet the requirement of disabling admin ticket creation
- Data is stored using pickle in separate binary files: users.pkl, orders.pkl, and discount.pkl

- No need for inheritance as all classes serve distinct roles without overlapping behavior
- Errors like invalid input or missing files are handled gracefully using basic exception handling

This design simplifies the system and reduces complexity, though it sacrifices flexibility in case the business wants to add or update ticket types later. Secondly, for the Data is stored using pickle in separate binary files: users.pkl, orders.pkl, and discount.pkl, Using pickle allows for a quick data saving and loading with the minimal setup , ideal for small scale applications or prototypes. Each file keeps related data grouped and manageable. Thirdly, the system uses composition over inheritance, which suits this scenario because the classes have clearly defined, non-overlapping purposes. This also keeps the design simple and focused. Finally, For errors like invalid input or missing files are handled gracefully using basic exception handling, this improves user experience and system stability. Even if a file is missing or a user inputs invalid data, the application will provide feedback or fallback behavior rather than crashing.

Full Code:

```
import tkinter as tk
from tkinter import messagebox
import pickle
import os
from datetime import datetime

# Models
class User:
    def __init__(self, username, password, name, email):
        self.username = username
        self.password = password
        self.name = name
        self.email = email
        self.orders = []

class Ticket:
    def __init__(self, ticket_type, price, validity, features):
        self.ticket_type = ticket_type
        self.price = price
        self.validity = validity
        self.features = features

class Order:
    def __init__(self, ticket, date, payment_method, race_date):
        self.ticket = ticket
        self.date = date
        self.payment_method = payment_method
```

```
        self.race_date = race_date

# File Helpers
def load_file(filename):
    if os.path.exists(filename):
        with open(filename, 'rb') as f:
            return pickle.load(f)
    return {}

def save_file(data, filename):
    with open(filename, 'wb') as f:
        pickle.dump(data, f)

# Predefined Tickets
tickets = {
    'Single Race': Ticket('Single Race', 100.0, '1 Day', 'Access to 1 race
event'),
    'Weekend Pass': Ticket('Weekend Pass', 250.0, '3 Days', 'Full weekend
access'),
    'Season Membership': Ticket('Season Membership', 1000.0, 'Full Season',
'All races access'),
    'Group Discount': Ticket('Group Discount', 80.0, '1 Day', 'Minimum 5
tickets')
}

users = load_file('users.pkl')
orders = load_file('orders.pkl')
discount_available = load_file('discount.pkl') or {'status': False}

class TicketApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Grand Prix Ticket Booking")
        self.root.geometry("800x600")
        self.current_user = None
        self.main_menu()

    def main_menu(self):
        self.clear()
        tk.Label(self.root, text="Grand Prix Ticket Booking System",
font=("Arial", 20)).pack(pady=20)
        tk.Button(self.root, text="Login", width=30,
command=self.login_screen).pack(pady=10)
        tk.Button(self.root, text="Sign Up", width=30,
command=self.signup_screen).pack(pady=10)
        tk.Button(self.root, text="Admin Dashboard", width=30,
command=self.admin_screen).pack(pady=10)

    def signup_screen(self):
        self.clear()
        tk.Label(self.root, text="Create New Account", font=("Arial",
16)).pack(pady=10)
        entries = {}
```

```
        for field in ["Username", "Password", "Name", "Email"]:
            tk.Label(self.root, text=field).pack()
            entry = tk.Entry(self.root, show="*" if field == "Password" else
"")
            entry.pack()
            entries[field.lower()] = entry

        def create_user():
            u = entries['username'].get()
            if u in users:
                messagebox.showerror("Error", "Username already exists")
                return
            users[u] = User(u, entries['password'].get(),
entries['name'].get(), entries['email'].get())
            save_file(users, 'users.pkl')
            self.main_menu()

        tk.Button(self.root, text="Create Account",
command=create_user).pack(pady=10)
        tk.Button(self.root, text="Back", command=self.main_menu).pack()

    def login_screen(self):
        self.clear()
        tk.Label(self.root, text="Login", font=("Arial", 16)).pack(pady=10)
        tk.Label(self.root, text="Username").pack()
        username_entry = tk.Entry(self.root)
        username_entry.pack()
        tk.Label(self.root, text="Password").pack()
        password_entry = tk.Entry(self.root, show="*")
        password_entry.pack()

        def login():
            u = username_entry.get()
            p = password_entry.get()
            if u in users and users[u].password == p:
                self.current_user = users[u]
                self.user_dashboard()
            else:
                messagebox.showerror("Error", "Invalid credentials")

        tk.Button(self.root, text="Login", command=login).pack(pady=10)
        tk.Button(self.root, text="Back", command=self.main_menu).pack()

    def user_dashboard(self):
        self.clear()
        tk.Label(self.root, text=f"Welcome {self.current_user.name}",
font=("Arial", 16)).pack(pady=10)
        tk.Button(self.root, text="Buy Ticket", width=30,
command=self.ticket_store).pack(pady=5)
        tk.Button(self.root, text="My Orders", width=30,
command=self.view_orders).pack(pady=5)
        tk.Button(self.root, text="Update Account", width=30,
command=self.update_account).pack(pady=5)
```

```
        tk.Button(self.root, text="Logout", width=30,
command=self.main_menu).pack(pady=5)

    def ticket_store(self):
        self.clear()
        tk.Label(self.root, text="Available Tickets", font=("Arial",
16)).pack(pady=10)
        selected_date = tk.Entry(self.root)
        tk.Label(self.root, text="Enter Race Date (YYYY-MM-DD)").pack()
        selected_date.pack()
        payment_var = tk.StringVar(value="Credit Card")
        tk.Label(self.root, text="Select Payment Method").pack()
        for method in ["Credit Card", "PayPal", "Digital Wallet"]:
            tk.Radiobutton(self.root, text=method, variable=payment_var,
value=method).pack(anchor='w')

        def purchase(ticket_key):
            race_date = selected_date.get()
            if not race_date:
                messagebox.showerror("Error", "Enter race date")
                return
            ticket = tickets[ticket_key]
            order = Order(ticket.ticket_type,
datetime.now().strftime('%Y-%m-%d'), payment_var.get(), race_date)
            self.current_user.orders.append(order)
            orders.setdefault(self.current_user.username, []).append(order)
            save_file(users, 'users.pkl')
            save_file(orders, 'orders.pkl')
            messagebox.showinfo("Success", "Ticket purchased")
            self.user_dashboard()

        for key, ticket in tickets.items():
            text = f"{ticket.ticket_type}: ${ticket.price} | {ticket.validity}
| {ticket.features}"
            tk.Button(self.root, text=text, command=lambda k=key:
purchase(k)).pack(pady=2)

        tk.Button(self.root, text="Back",
command=self.user_dashboard).pack(pady=10)

    def view_orders(self):
        self.clear()
        tk.Label(self.root, text="Your Orders", font=("Arial",
16)).pack(pady=10)
        for idx, order in enumerate(self.current_user.orders):
            race_date = getattr(order, 'race_date', 'N/A')
            payment_method = getattr(order, 'payment_method', 'N/A')
            msg = f"{idx + 1}. {order.ticket} on {race_date} via
{payment_method} (Booked: {order.date})"
            tk.Label(self.root, text=msg).pack()

        tk.Label(self.root, text="Select Order Number to
Modify/Delete").pack()
```

```
        idx_entry = tk.Entry(self.root)
        idx_entry.pack()

        def modify_order():
            try:
                i = int(idx_entry.get()) - 1
                order = self.current_user.orders[i]
            except:
                messagebox.showerror("Error", "Invalid index")
                return

            self.clear()
            tk.Label(self.root, text="Modify Order", font=("Arial",
16)).pack(pady=10)
            race_entry = tk.Entry(self.root)
            race_entry.insert(0, getattr(order, 'race_date', ''))
            tk.Label(self.root, text="New Race Date (YYYY-MM-DD)").pack()
            race_entry.pack()

            pay_var = tk.StringVar(value=getattr(order, 'payment_method',
'Credit Card'))
            tk.Label(self.root, text="New Payment Method").pack()
            for method in ["Credit Card", "PayPal", "Digital Wallet"]:
                tk.Radiobutton(self.root, text=method, variable=pay_var,
value=method).pack(anchor='w')

            def save_changes():
                order.race_date = race_entry.get()
                order.payment_method = pay_var.get()
                save_file(users, 'users.pkl')
                save_file(orders, 'orders.pkl')
                messagebox.showinfo("Success", "Order updated")
                self.user_dashboard()

            tk.Button(self.root, text="Save",
command=save_changes).pack(pady=5)
            tk.Button(self.root, text="Back",
command=self.view_orders).pack(pady=5)

        def delete_order():
            try:
                i = int(idx_entry.get()) - 1
                del self.current_user.orders[i]
                orders[self.current_user.username] = self.current_user.orders
                save_file(users, 'users.pkl')
                save_file(orders, 'orders.pkl')
                messagebox.showinfo("Deleted", "Order removed")
                self.view_orders()
            except:
                messagebox.showerror("Error", "Invalid index")

        tk.Button(self.root, text="Modify Order",
command=modify_order).pack(pady=5)
```

```
        tk.Button(self.root, text="Delete Order",
command=delete_order).pack(pady=5)
        tk.Button(self.root, text="Back",
command=self.user_dashboard).pack(pady=10)

    def update_account(self):
        self.clear()
        tk.Label(self.root, text="Update Account Details", font=("Arial",
16)).pack(pady=10)
        email_entry = tk.Entry(self.root)
        email_entry.insert(0, self.current_user.email)
        tk.Label(self.root, text="Email").pack()
        email_entry.pack()

        def save_updates():
            self.current_user.email = email_entry.get()
            save_file(users, 'users.pkl')
            messagebox.showinfo("Updated", "Details updated")
            self.user_dashboard()

        tk.Button(self.root, text="Save Changes",
command=save_updates).pack(pady=10)
        tk.Button(self.root, text="Back", command=self.user_dashboard).pack()

    def admin_screen(self):
        self.clear()
        tk.Label(self.root, text="Admin Dashboard", font=("Arial",
16)).pack(pady=10)
        tk.Label(self.root, text=f"Discount is {'ON' if
discount_available['status'] else 'OFF'}").pack()

        def toggle_discount():
            discount_available['status'] = not discount_available['status']
            save_file(discount_available, 'discount.pkl')
            self.admin_screen()

        tk.Button(self.root, text="Toggle Discount",
command=toggle_discount).pack(pady=5)

        tk.Label(self.root, text="Ticket Sales by Day").pack()
        sales = {}
        for user_orders in orders.values():
            for o in user_orders:
                sales[o.date] = sales.get(o.date, 0) + 1
        for day, count in sales.items():
            tk.Label(self.root, text=f"{day}: {count} tickets").pack()

        tk.Label(self.root, text="User List").pack(pady=10)
        for u in users.values():
            tk.Label(self.root, text=f"{u.username} | {u.name} |
{u.email}").pack()
```

```
        tk.Button(self.root, text="Back",
command=self.main_menu).pack(pady=10)

    def clear(self):
        for widget in self.root.winfo_children():
            widget.destroy()

root = tk.Tk()
app = TicketApp(root)
root.mainloop()
```

File Structure Explanation:

The system uses the pickle library to store data persistently in binary format. The following files are created and managed during program execution:

1. users.pkl

- Stores all registered user accounts.
- Each user object includes username, password, name, email, and a list of their orders.

So the file captures every user that signs up, along with their personal data and list of orders, and it ensures that the application can recognize returning users and restore their activity history. For each user object includes username, password, name, email, and a list of their orders, these attributes are essentials for login validation, displaying user specific data, and it maintains a consistent purchase record per user.

2. orders.pkl

- Stores all ticket orders placed by users.
- Each entry links to a user via their username and includes ticket type, purchase date, payment method, and selected race date.

It acts as a historical record of every purchase made in the system, useful for reporting, order modifications, or admin review. This linking makes it simple to retrieve and display orders for each user, while still allowing admin-level aggregation of all sales data.

3. discount.pkl

- Stores the current discount toggle status for the admin dashboard.
- Used to simulate discount availability without modifying the ticket prices directly.

Instead of recalculating ticket prices or modifying the ticket structure, the discount status provides a flexible way for the admin to control promotional availability. Lastly, Used to simulate discount

availability without modifying the ticket prices directly, it keeps the ticket definitions clean and allows temporary changes to be reversed easily without data corruption or inconsistency.

File Handling Logic

- Each file is read using load_file() when the application starts.
- All changes (e.g., new user, ticket purchase, order update) are saved using save_file().
- The system uses separate files to keep user, order, and admin-related data modular and easier to debug or reset.

First, “Each file is read using load_file() when the application starts” ensures that all existing data is immediately loaded into memory, allowing the app to function with the latest user, order, and discount information from previous sessions. Secondly, “All changes (e.g., new user, ticket purchase, order update) are saved using save_file().” The system automatically persists every critical update so there will be no manual save action that is needed by the user or admin after performing an operation. Finally, “The system uses separate files to keep user, order, and admin-related data modular and easier to debug or reset” the separation provides better flexibility during development and troubleshooting, as each component can be managed independently without affecting the others as well.

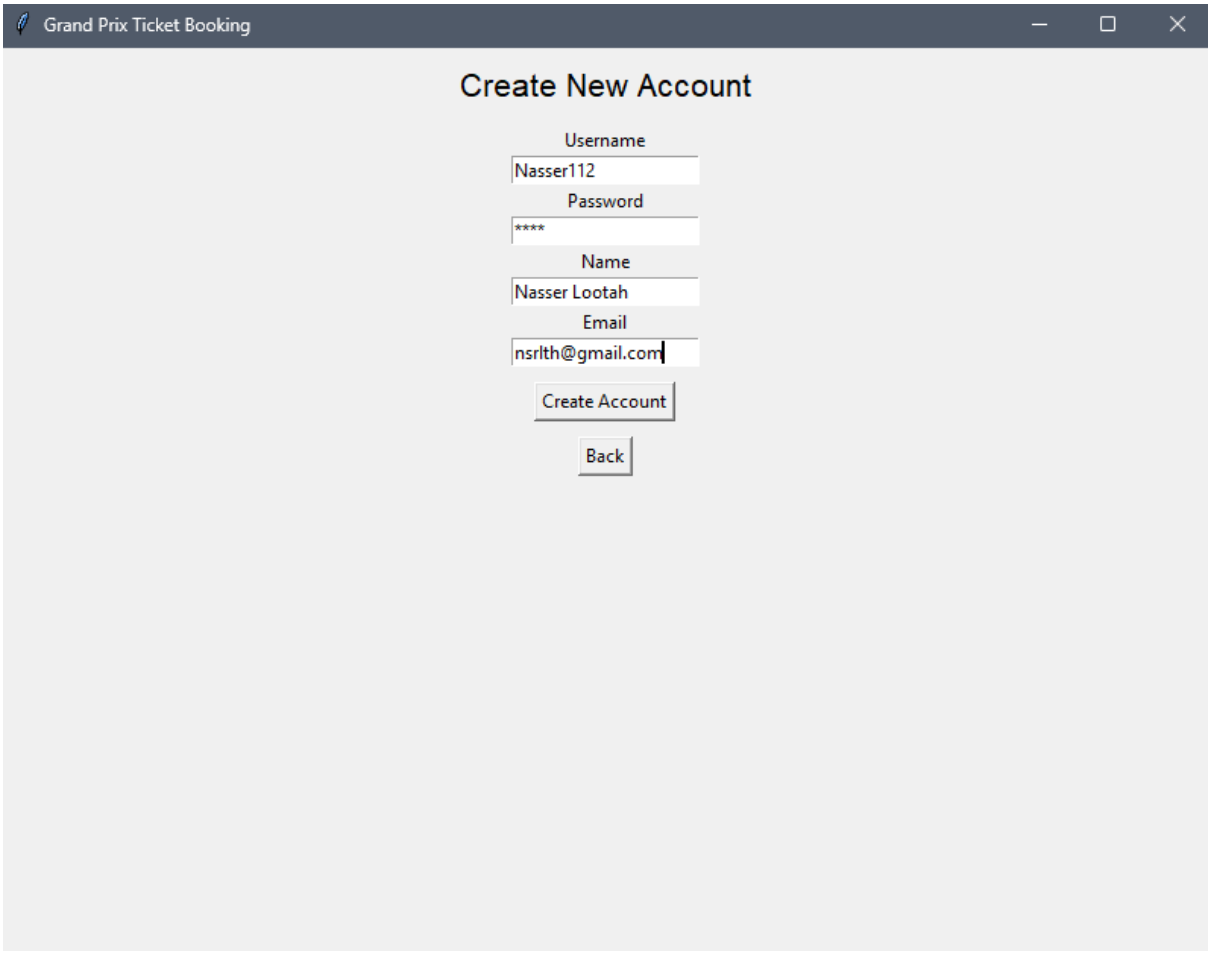
Assurance

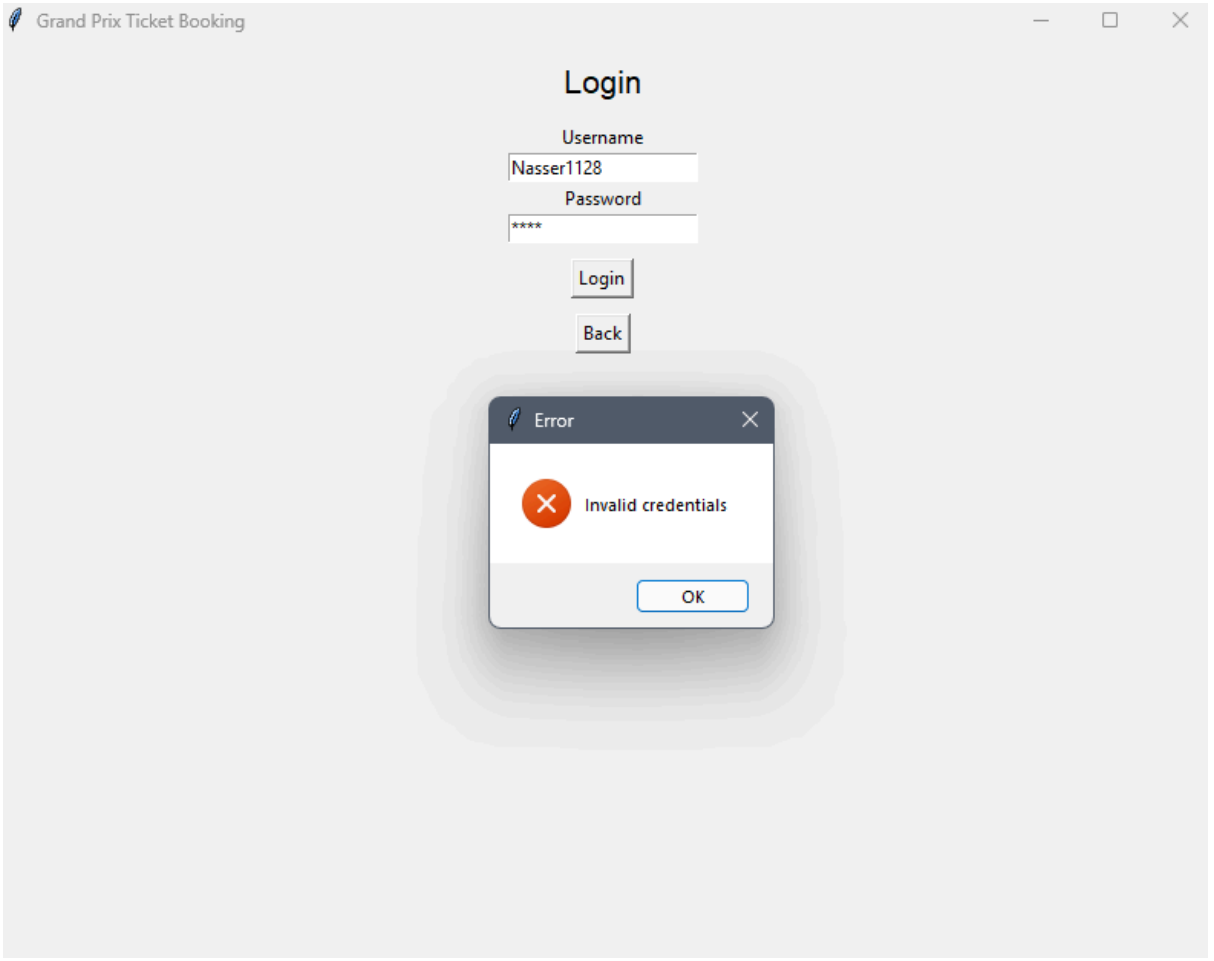
- Before accessing any file, the program checks if it exists using os.path.exists() to avoid file errors.
- If a file does not exist, it initializes an empty dictionary or default value.

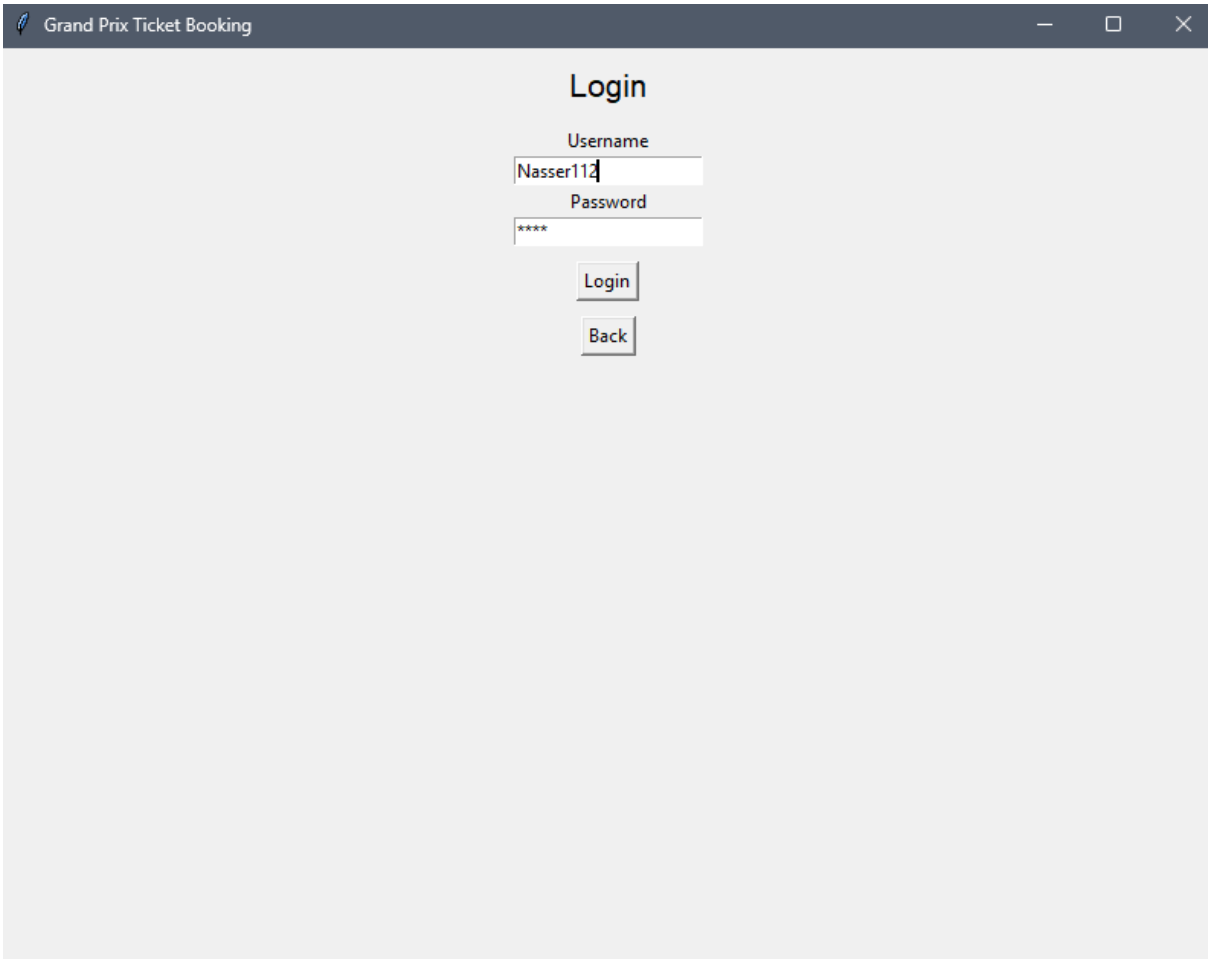
So this prevents the application from crashing when files are missing, especially during the first launch or after file deletions. And for the “If a file does not exist, it initializes an empty dictionary or default value” This fallback design makes the application more robust, allowing it to self-heal and generate new files when necessary without needing manual setup.

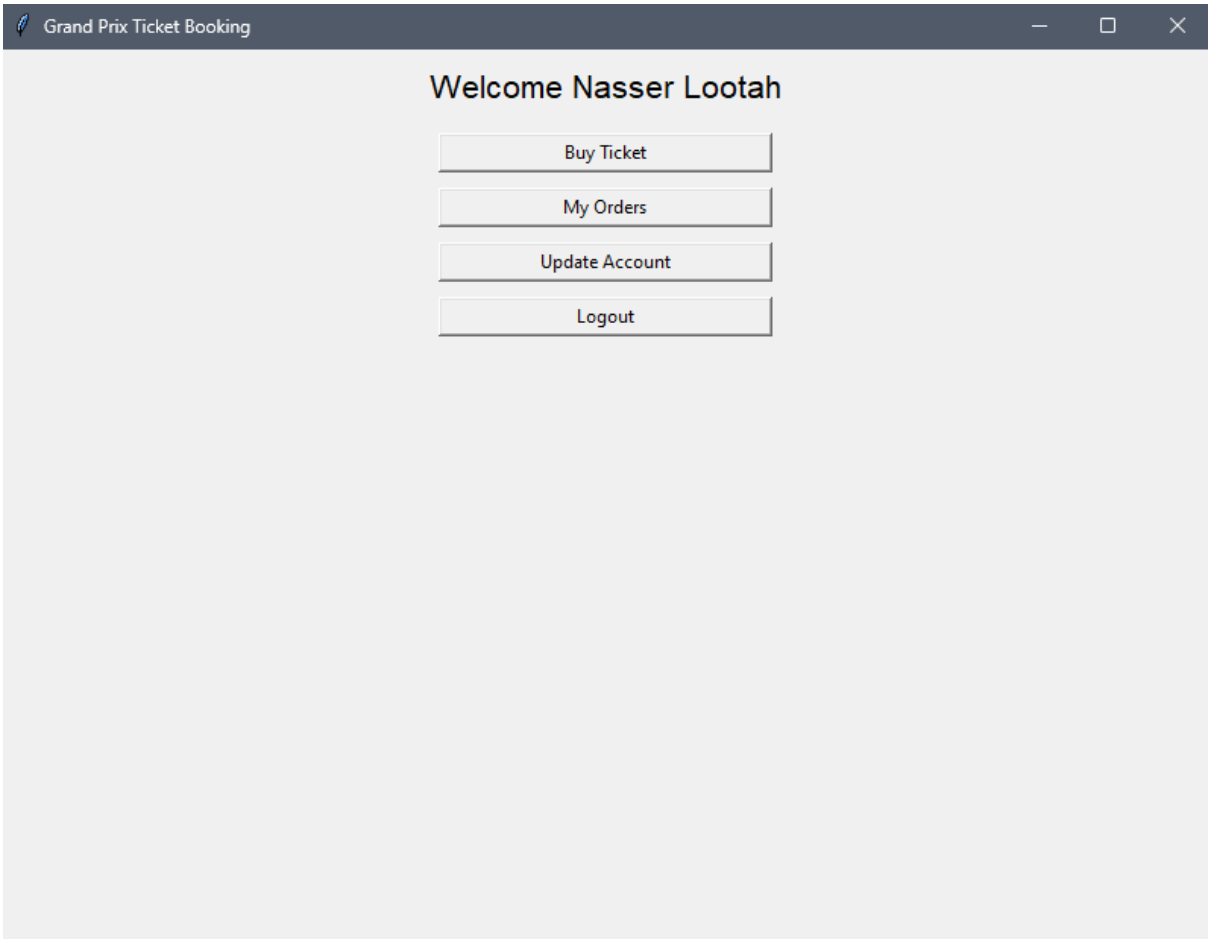
Test cases:

Test Case 1: User Registration and Login

- 
Grand Prix Ticket Booking
Create New Account
Username
Nasser112
Password
****
Name
Nasser Lootah
Email
nsrlth@gmail.com
Create Account
Back

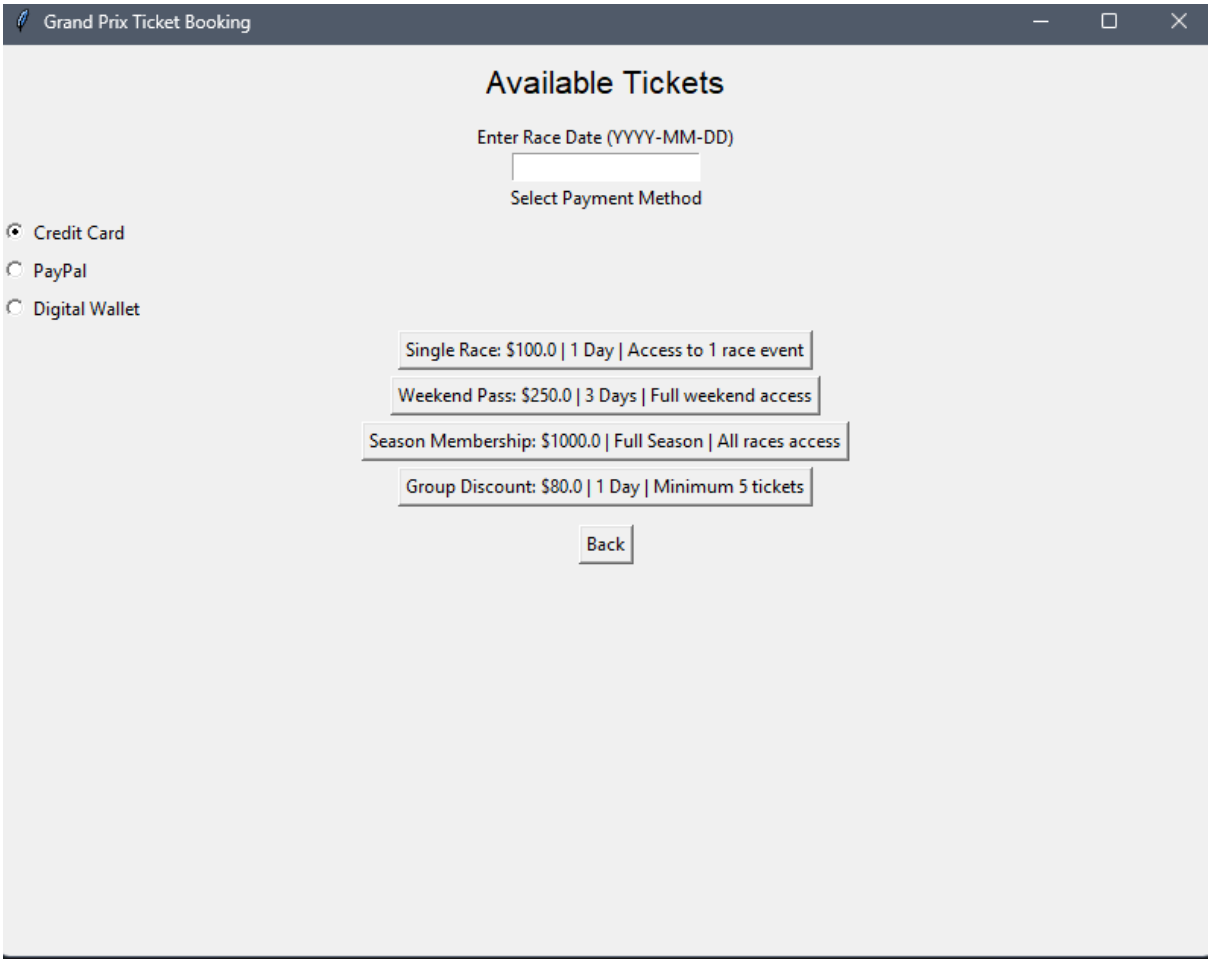
- 
Grand Prix Ticket Booking
Login
Username
Nasser1128
Password
****
Login
Back
Error
Invalid credentials
OK

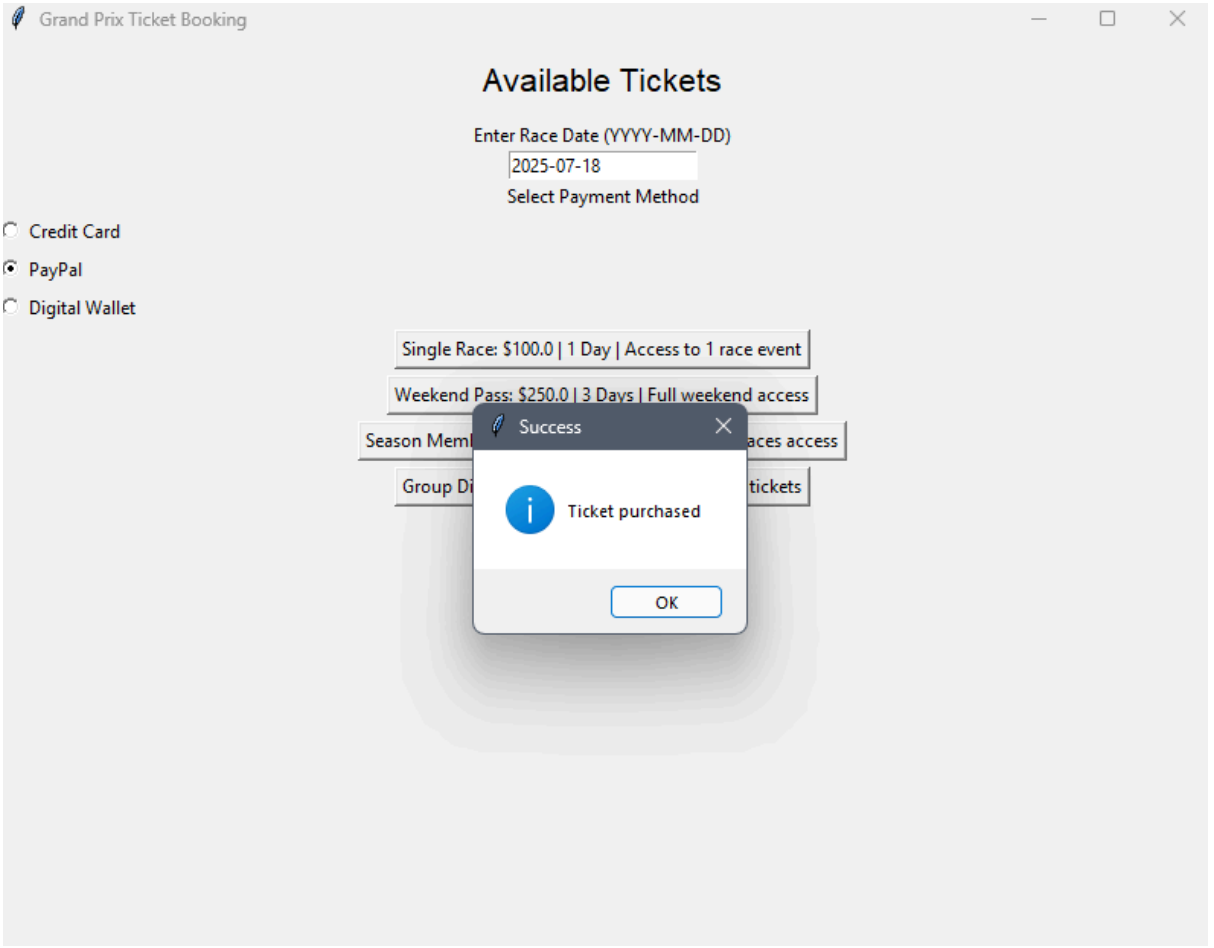
- 
Grand Prix Ticket Booking
Login
Username
Nasser112
Password
****
Login
Back

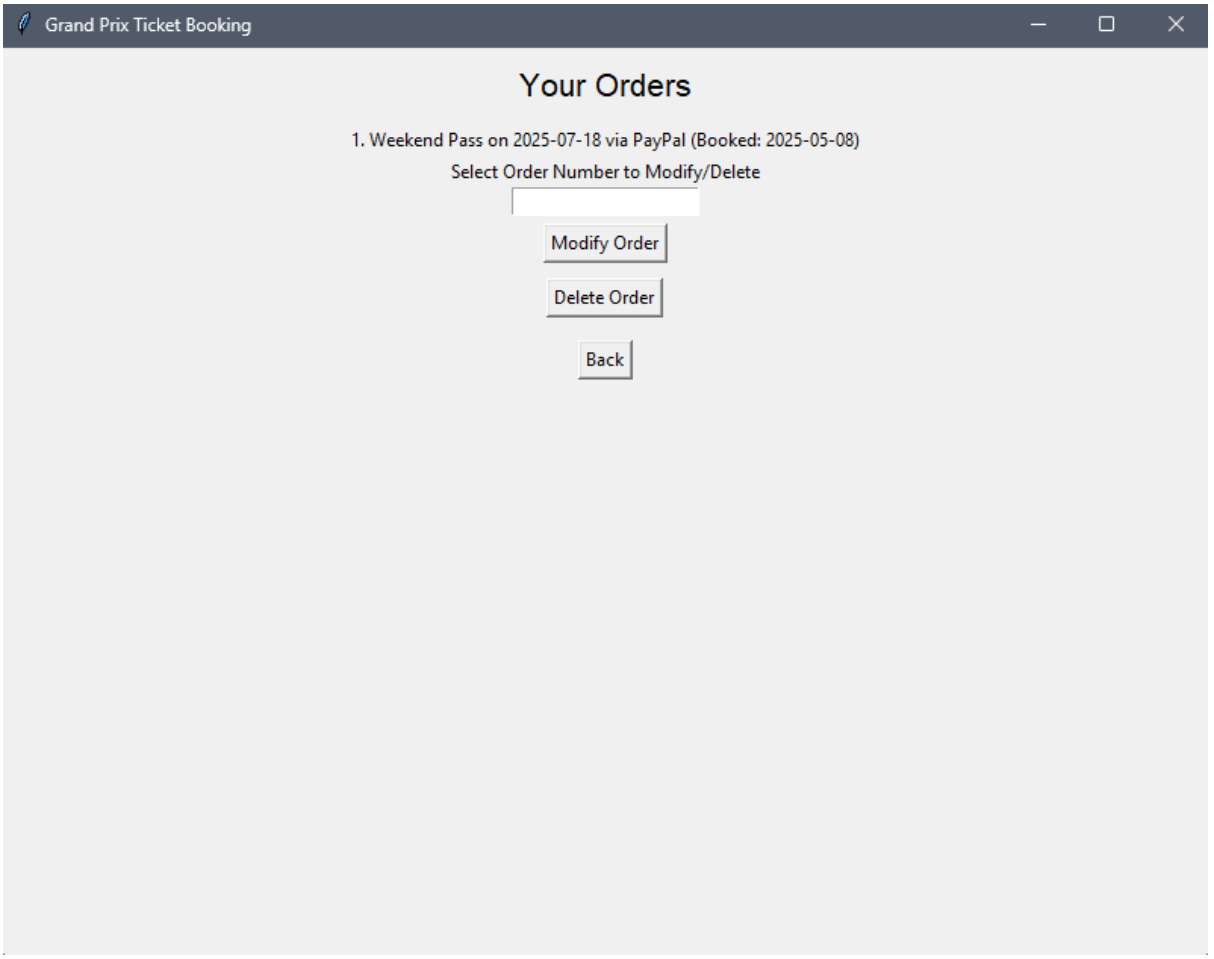


-
-

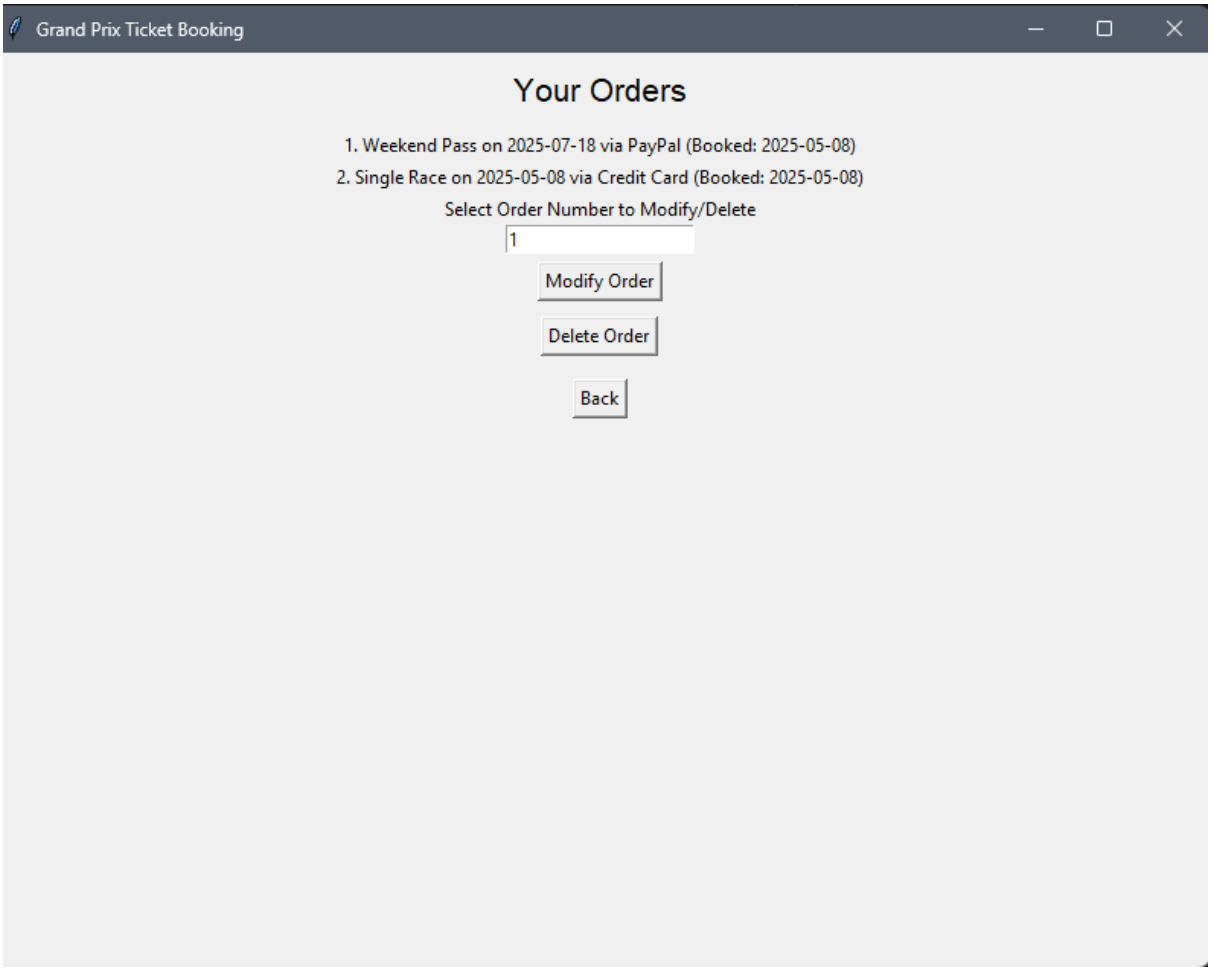
Test Case 2: Ticket Purchase

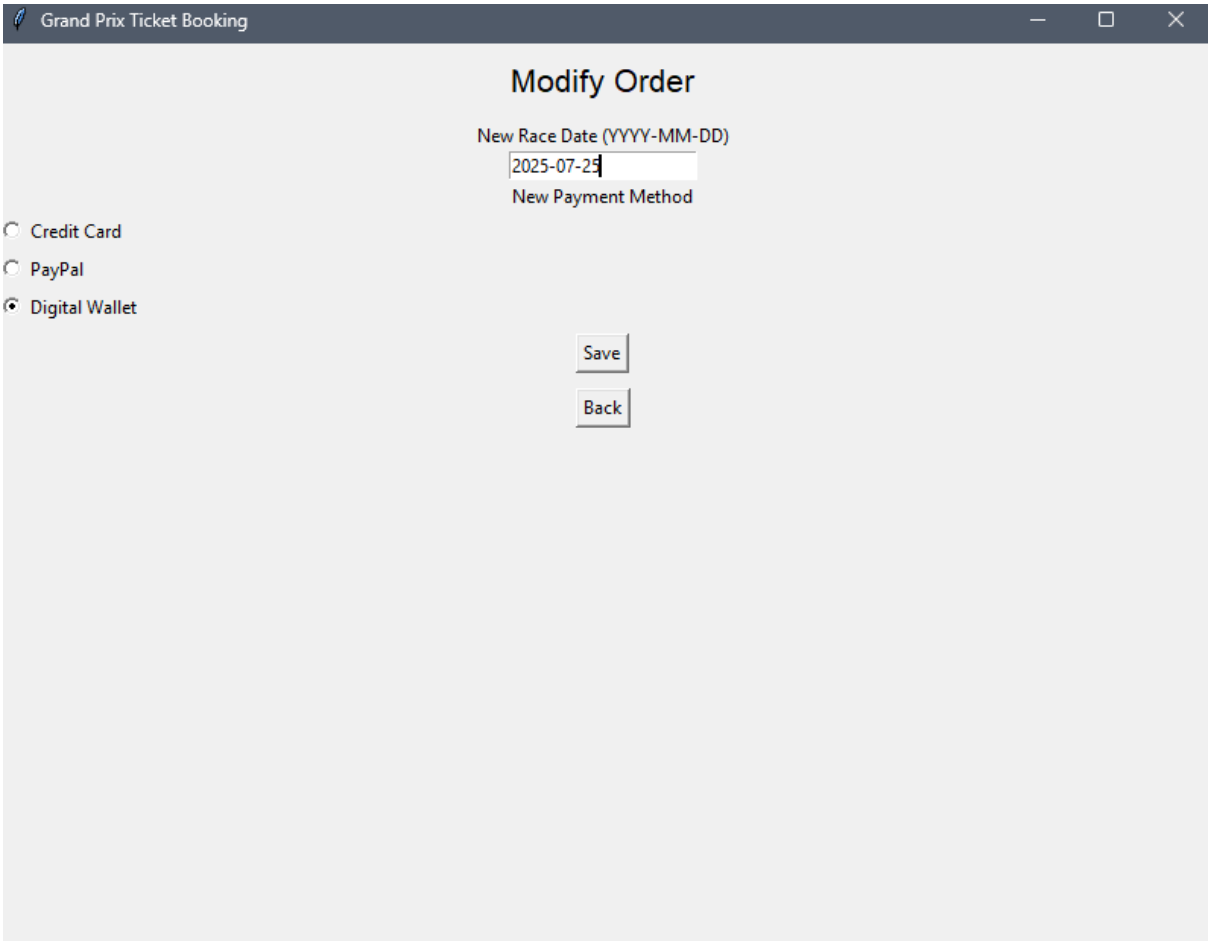

Grand Prix Ticket Booking
Available Tickets
Enter Race Date (YYYY-MM-DD)
Select Payment Method
Credit Card
PayPal
Digital Wallet
Single Race: $100.0 | 1 Day | Access to 1 race event
Weekend Pass: $250.0 | 3 Days | Full weekend access
Season Membership: $1000.0 | Full Season | All races access
Group Discount: $80.0 | 1 Day | Minimum 5 tickets
Back

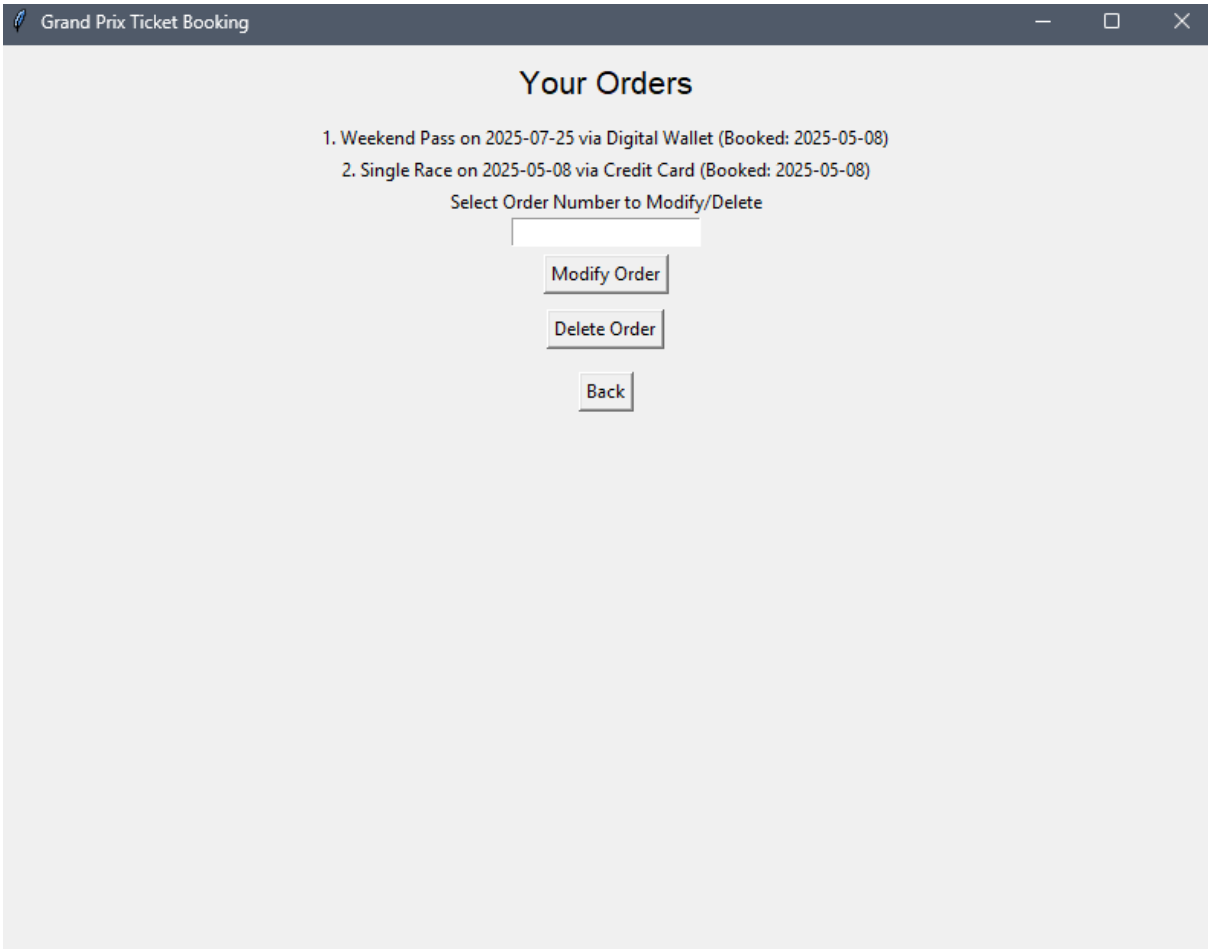
- 
Grand Prix Ticket Booking
Available Tickets
Enter Race Date (YYYY-MM-DD)
2025-07-18
Select Payment Method
Credit Card
PayPal
Digital Wallet
Single Race: $100.0 | 1 Day | Access to 1 race event
Weekend Pass: $250.0 | 3 Days | Full weekend access
Season Mem
aces access
Group Di
tickets
Success
Ticket purchased
OK

- 

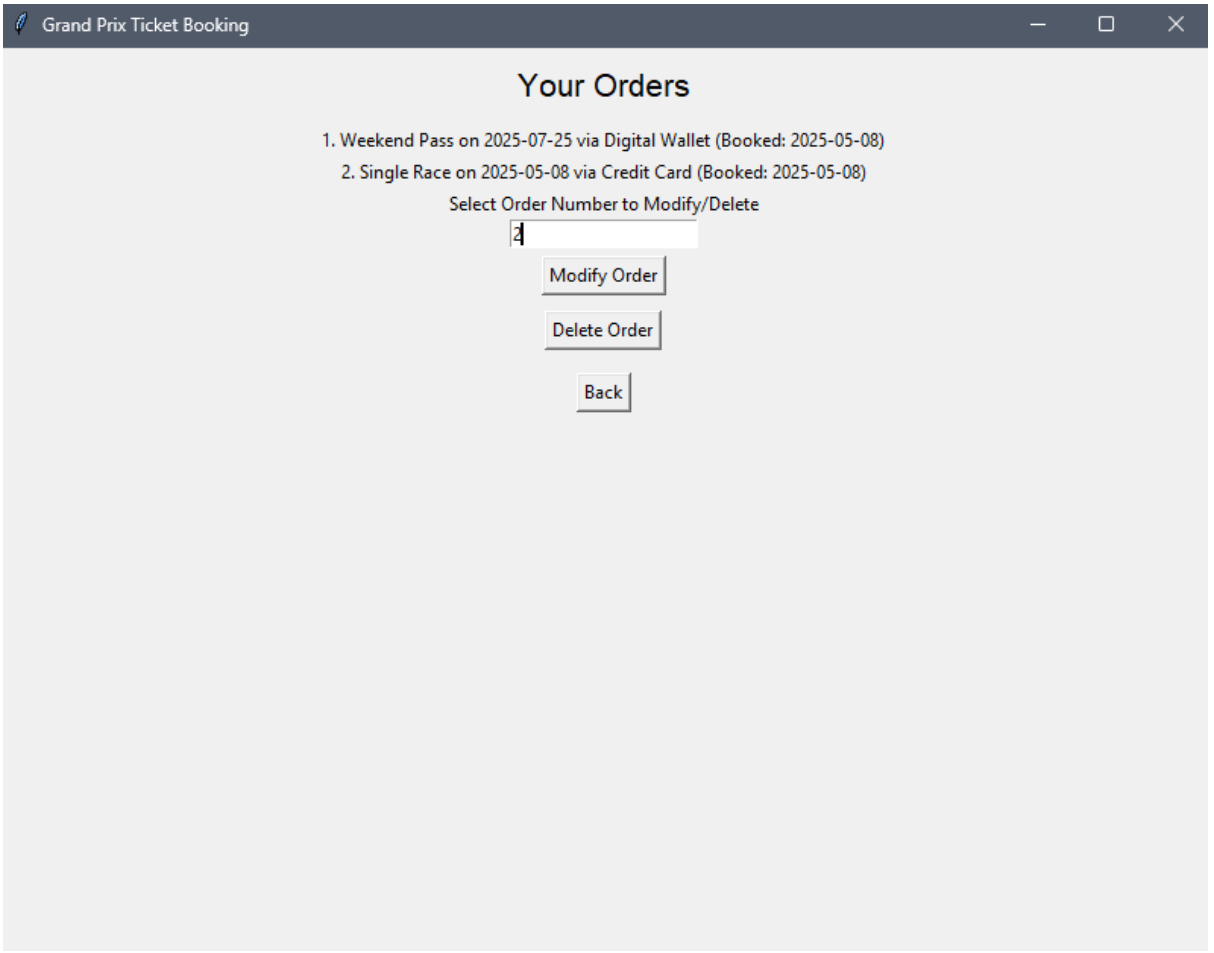
Test Case 3: Modify Order

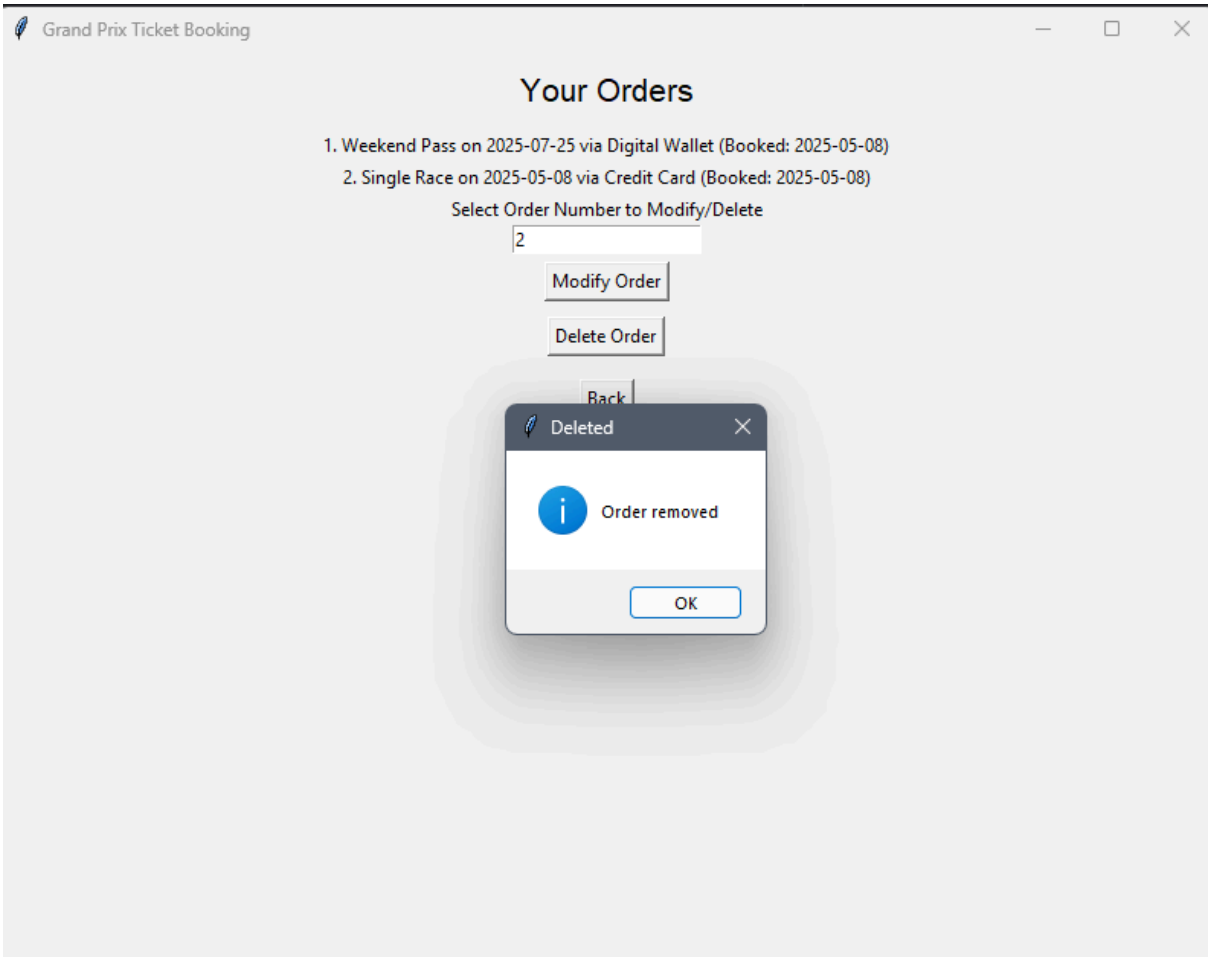
- 
Grand Prix Ticket Booking
Your Orders
1. Weekend Pass on 2025-07-18 via PayPal (Booked: 2025-05-08)
2. Single Race on 2025-05-08 via Credit Card (Booked: 2025-05-08)
Select Order Number to Modify/Delete
1
Modify Order
Delete Order
Back

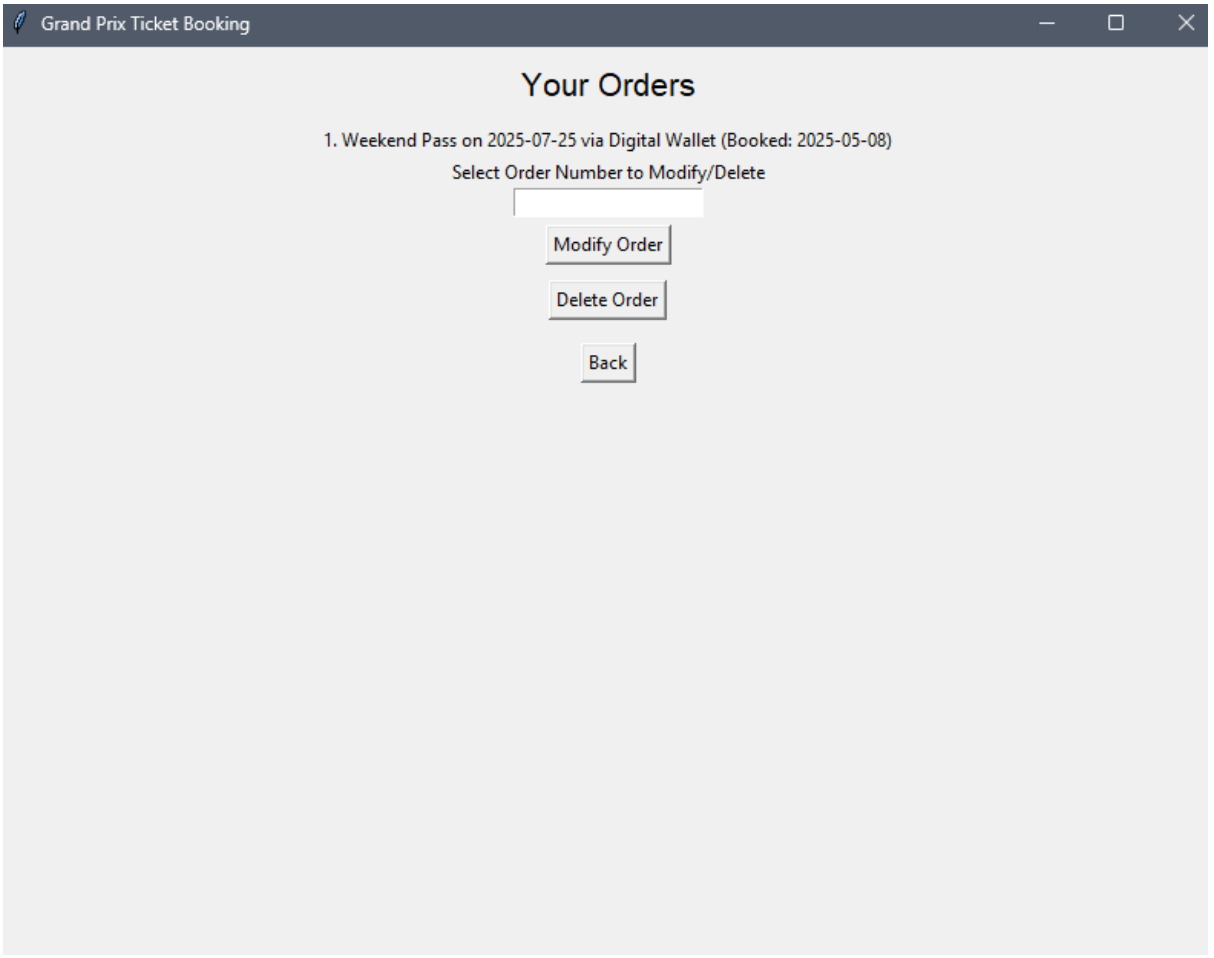
- 
Grand Prix Ticket Booking
Modify Order
New Race Date (YYYY-MM-DD)
2025-07-25
New Payment Method
Credit Card
PayPal
Digital Wallet
Save
Back

- 

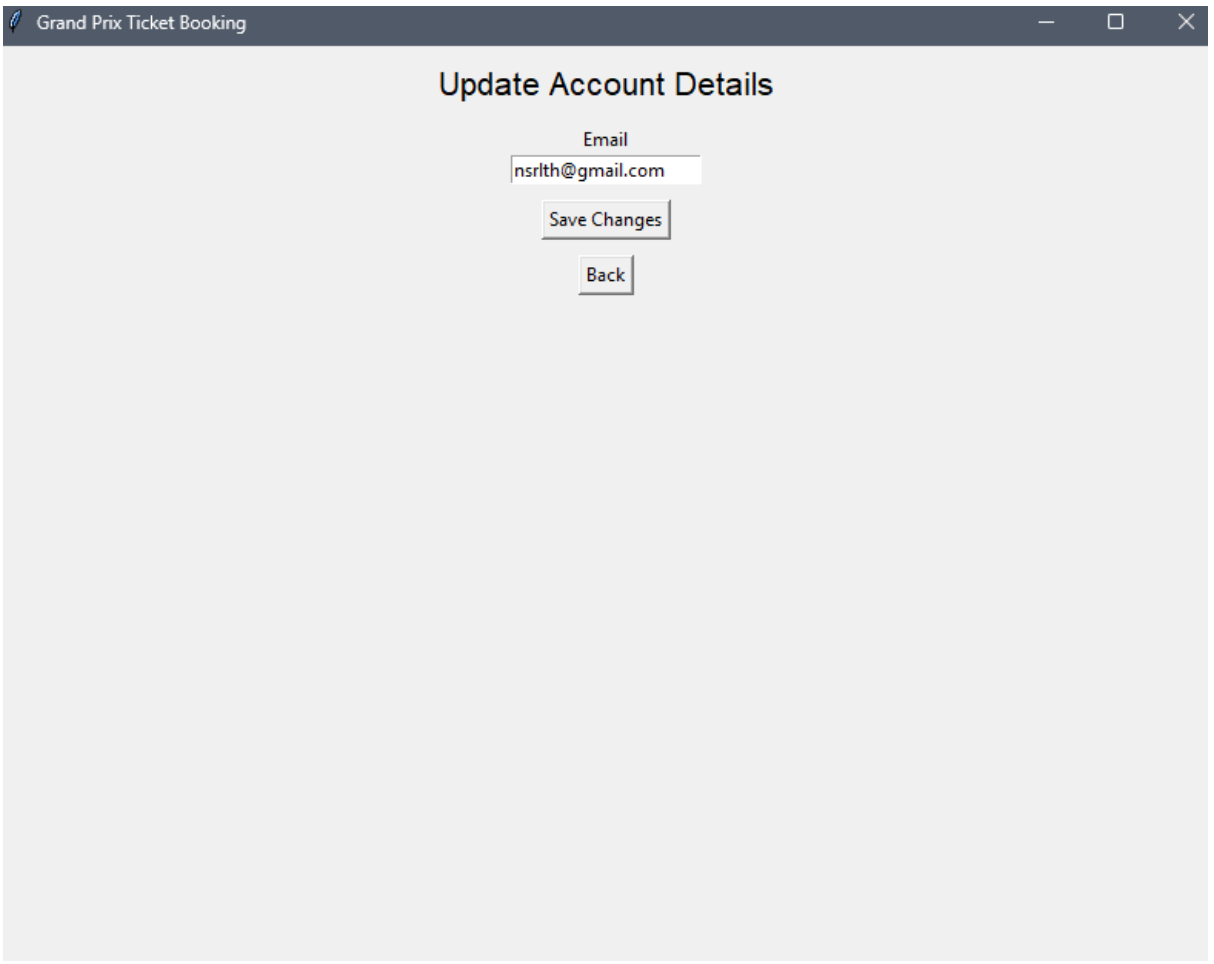
Test Case 4: Delete Order

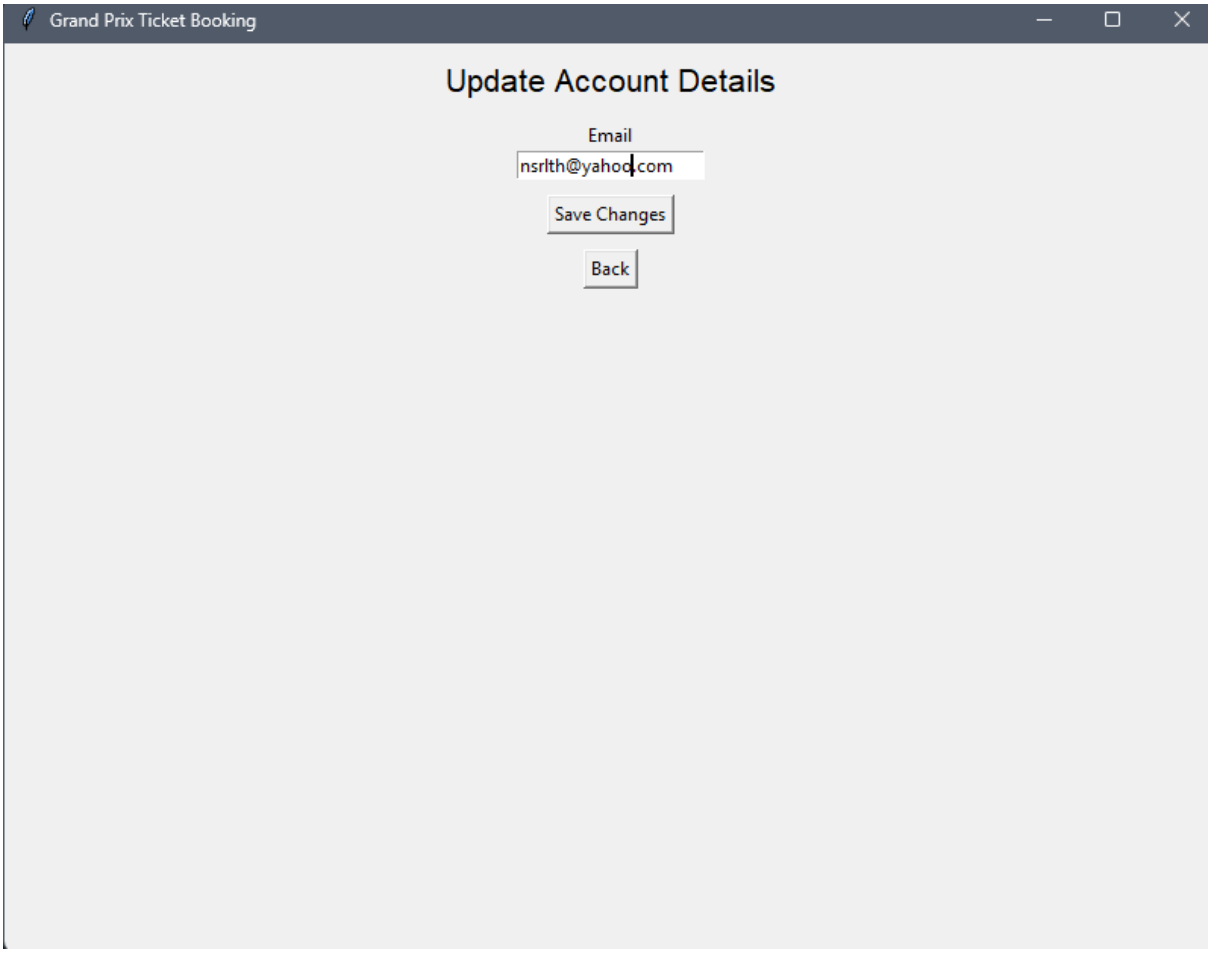
- 
Grand Prix Ticket Booking
Your Orders
1. Weekend Pass on 2025-07-25 via Digital Wallet (Booked: 2025-05-08)
2. Single Race on 2025-05-08 via Credit Card (Booked: 2025-05-08)
Select Order Number to Modify/Delete
2
Modify Order
Delete Order
Back

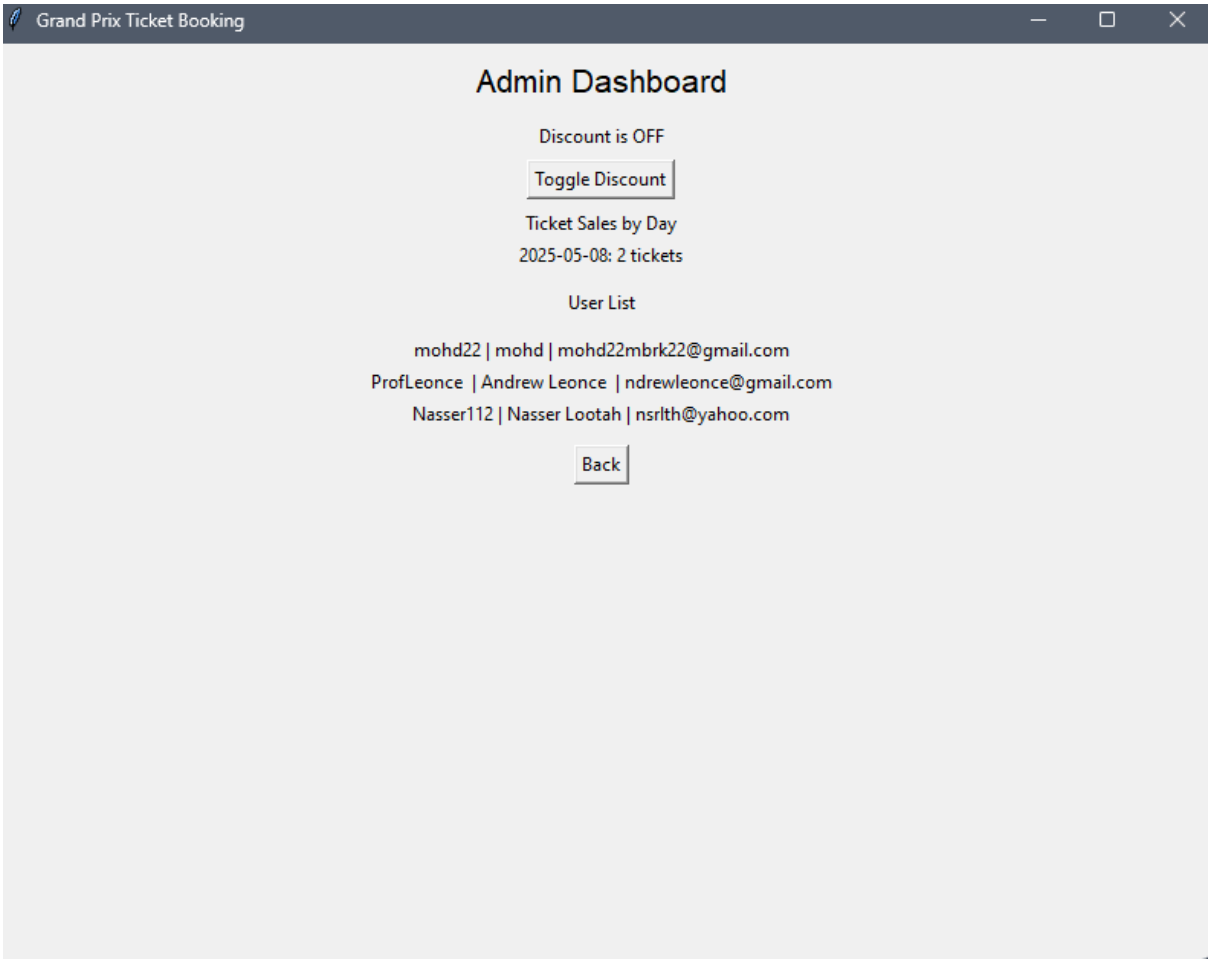
- 
Grand Prix Ticket Booking
Your Orders
1. Weekend Pass on 2025-07-25 via Digital Wallet (Booked: 2025-05-08)
2. Single Race on 2025-05-08 via Credit Card (Booked: 2025-05-08)
Select Order Number to Modify/Delete
2
Modify Order
Delete Order
Back
Deleted
Order removed
OK

- 

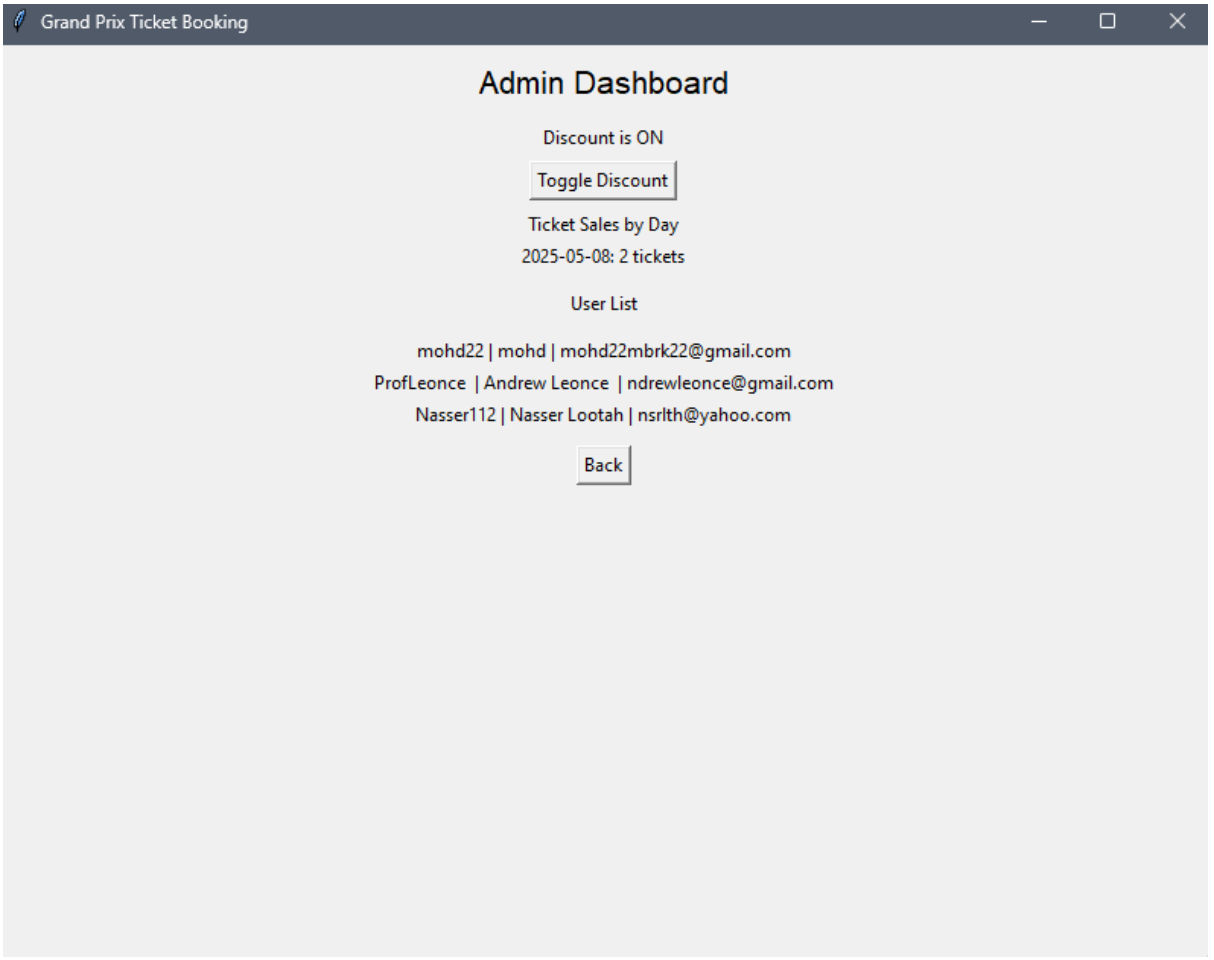
Test Case 5: Update Account Info

- 
Grand Prix Ticket Booking
Update Account Details
Email
nsrlth@gmail.com
Save Changes
Back

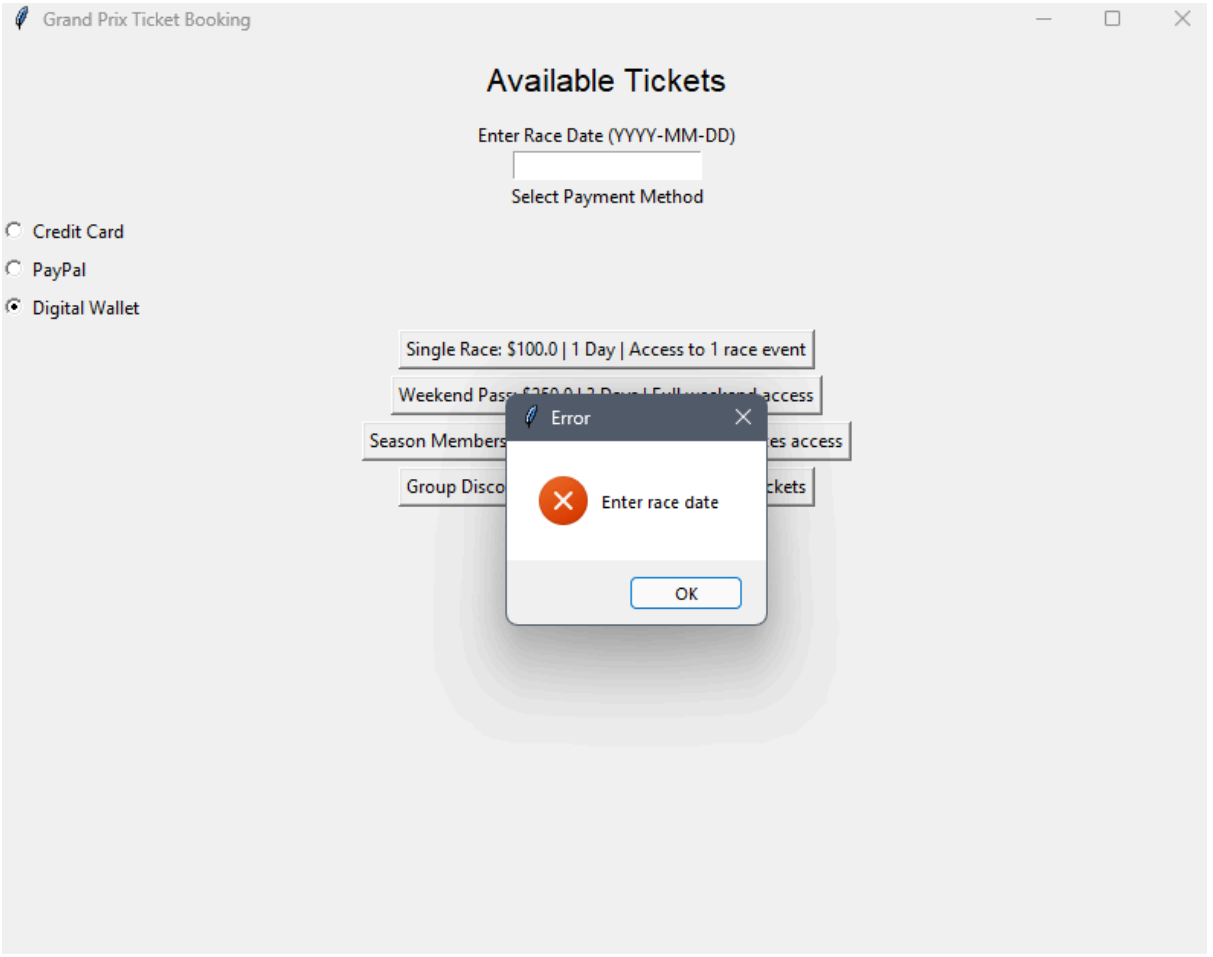
- 
Grand Prix Ticket Booking
Update Account Details
Email
nsrlth@yahoo.com
Save Changes
Back

- 

Test Case 6: Admin Dashboard Access

- 

Test Case 7: Invalid Input Handling

- 

Github repository link: https://github.com/NasserLootah/Nasser-and-Mohammed

Summary of Learning:

Through this project, we developed a complete understanding of object-oriented design, GUI development, and file-based data management in Python. By building the Grand Prix ticket booking system from scratch, we applied core programming concepts such as class relationships, composition, and modular coding.

We also practiced using the tkinter library to create a user-friendly interface and handled real-world scenarios like login, data persistence, and error validation. Implementing binary file storage with pickle helped us learn how to maintain user and order data effectively without relying on external databases.

This project improved our skills in structuring large programs, debugging multi-component systems, and following clear software design standards.