2025

CAR RENTAL SYSTEM

PROJECT:
DEBASISH PARIDA (25BAI10081)

Car Rental System: Project Report

2. Introduction

This report documents the design, implementation, and testing of the **Car Rental System**, a console-based application built in Python. The system is designed to manage a fleet of vehicles, track their availability, handle customer rental and return transactions, and calculate rental costs. It provides a simple, interactive user interface suitable for a small-scale, internal car rental operation management.

3. Problem Statement

The core problem addressed is the inefficient manual tracking of car inventory and rental transactions within a small-scale rental business. Without a centralized system, it is difficult to accurately determine which cars are available, who currently possesses which vehicle, and to calculate rental costs quickly and reliably. The solution requires an automated system to manage car status and active customer rentals efficiently.

4. Functional Requirements

ID	Requirement	Description
FR1.0	Car Inventory Management	The system must maintain a dynamic list of cars with attributes (Make, Model, ID, Availability).
FR1.1	Add Car	Users (Admin) must be able to add a new car to the inventory.
FR2.0	View Available Cars	The system must display a list of all cars currently marked as available for rent.
FR3.0	Rent Car Transaction	The system must allow a customer to rent an available car for a specified number of days, marking the car as unavailable and creating an active rental record.
FR3.1	Cost Calculation	The system must calculate and display the estimated rental cost based on the daily rate and rental duration.

FR4.0	Return Car Transaction	The system must allow a customer to return a rented car, removing the active rental record and marking the car as available.
FR5.0	View Active Rentals	The system must display a list of all ongoing rentals, including the customer name, car ID, and rental duration.
FR6.0	User Interface	The system must provide a simple, menu-driven command-line interface for interaction.

5. Non-functional Requirements

ID	Requirement	Description
NFR1.0	Usability	The command-line interface must be intuitive, with clear menus and prompts.
NFR2.0	Error Handling	The system must gracefully handle invalid inputs (e.g., non-numeric IDs, empty fields) during transactions.
NFR3.0	Maintainability	The code must be well-organized into logical classes (CarManager, RentalManager, UserInterface) following Object-Oriented Programming (OOP) principles.
NFR4.0	Reliability	Car availability status must be correctly synchronized between the inventory and active rentals.

6. System Architecture

The Car Rental System utilizes a **Tiered Monolithic Architecture** consisting of three main logical components (Python classes):

1. **Data/Service Layer (CarManager):** Responsible for managing the core inventory data (car information, availability status) and providing data access methods (add, check, update).

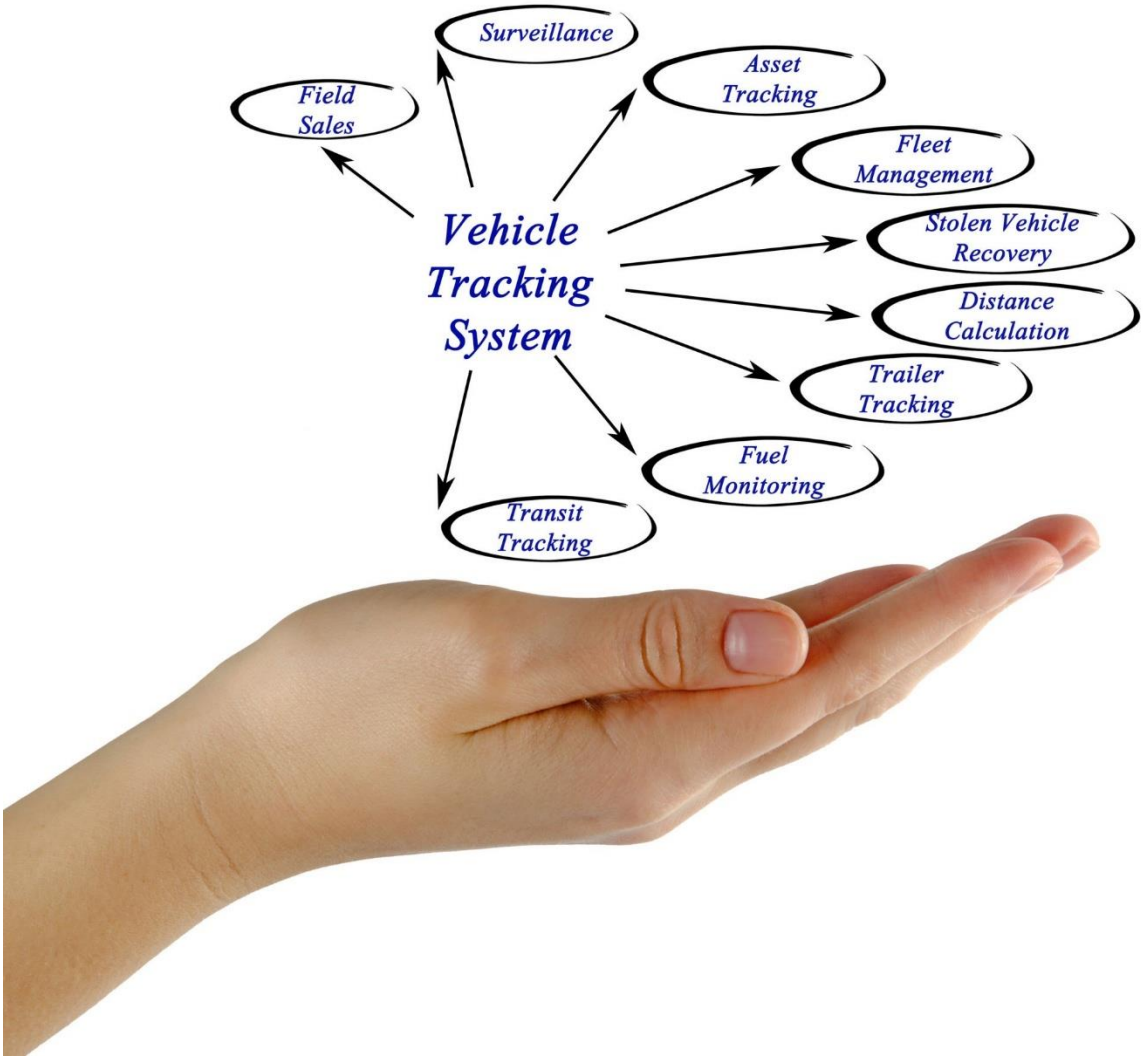
2. **Business Logic Layer (RentalManager):** Encapsulates the transactional logic (renting, returning, cost calculation) and relies on the CarManager for inventory updates.
3. **Presentation Layer (UserInterface):** Provides the command-line interface, handles user input, displays menus, and orchestrates calls to the RentalManager and CarManager.

This design separates concerns, making the logic easy to test and maintain.

7. Design Diagrams

Use Case Diagram

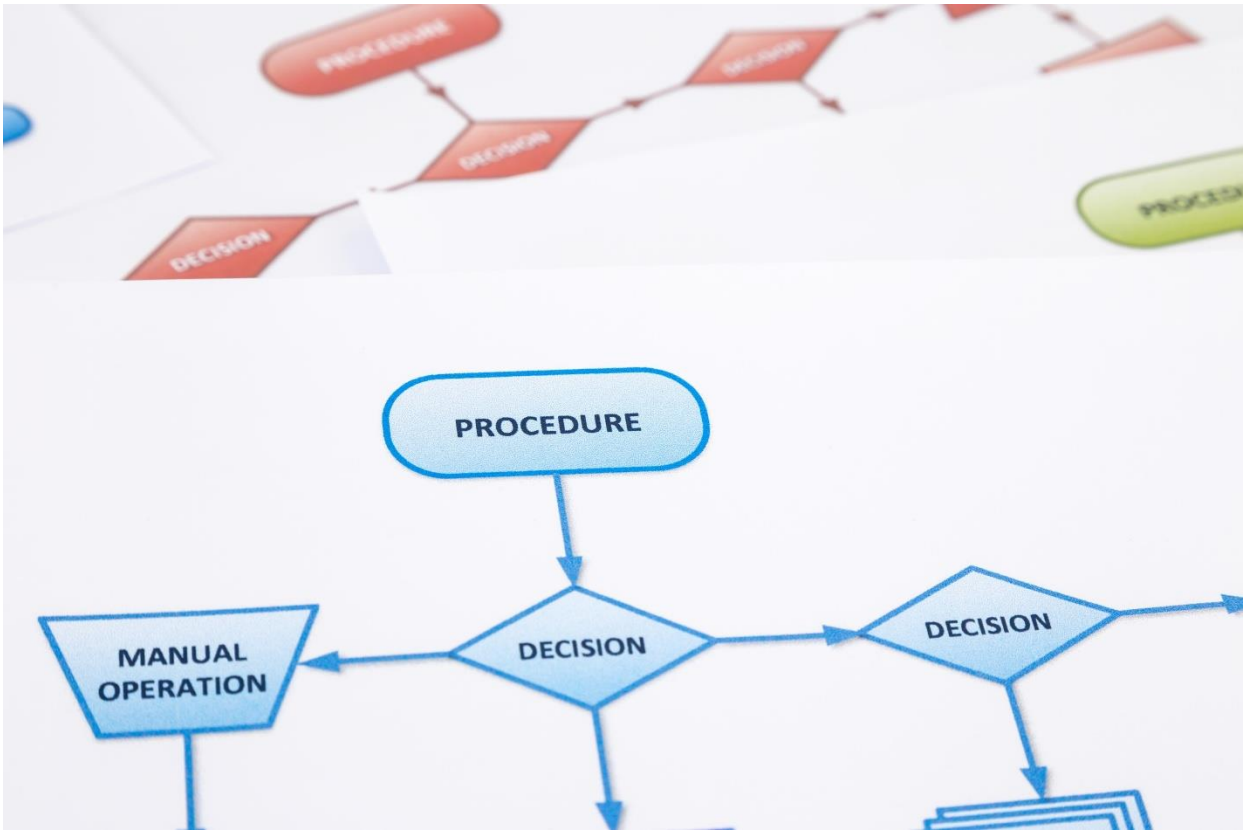
The Use Case Diagram illustrates the system's functions from the perspective of the primary actors: the Customer and the Admin (implicitly the user accessing admin functions).



Getty Images

Workflow Diagram

The Workflow Diagram visualizes the steps involved in the primary business process: renting a car.

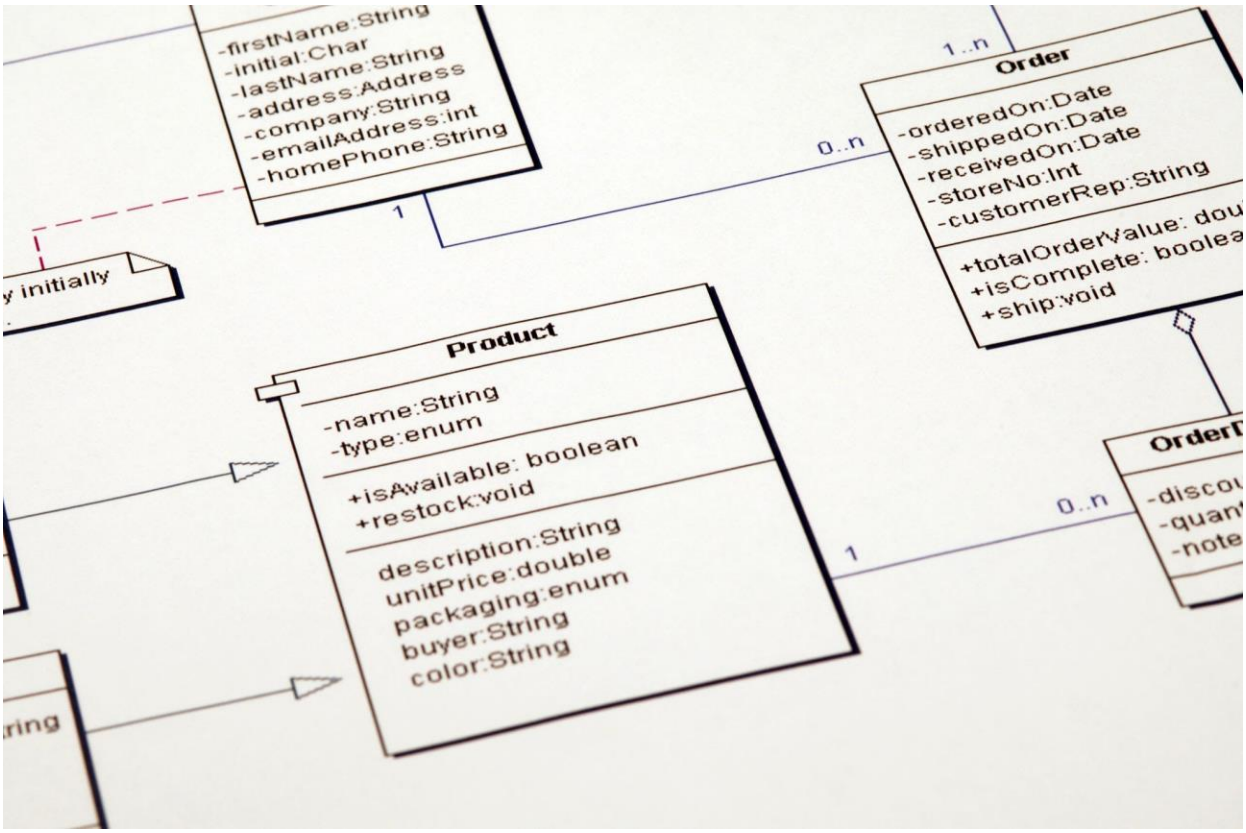


Shutterstock

Explore

Sequence Diagram

The Sequence Diagram demonstrates the interaction and order of messages between the UserInterface, RentalManager, and CarManager during the "Rent a Car" transaction.



Getty Images

Explore

Class/Component Diagram

This diagram shows the main components (classes) and their relationships, illustrating the OOP structure of the application.

- **CarManager**: Manages the inventory dictionary.
- **RentalManager**: Depends on (aggregates) CarManager to perform rental transactions, manages the active_rentals dictionary.
- **UserInterface**: Depends on both managers to run the application logic and interact with the user.

ER Diagram (Conceptual Data Model)

Although the storage uses in-memory Python dictionaries, the conceptual data model can be mapped as follows:

1. **CAR** Entity:

- Attributes: Car ID (PK), Make, Model, Available (Boolean)

2. **RENTAL** Entity:
    - Attributes: Customer Name (PK/Identifier), Car ID (FK to CAR), Rental Days

The relationship between **CAR** and **RENTAL** is **1-to-Many**, where one car can be part of many past rental transactions, but at any given time, a car is either available or tied to **at most one** active rental. (In the current code, the active_rentals dictionary enforces the 1:1 active relationship by using customer_name as the key, implicitly linking one customer to one currently rented car).

8. Design Decisions & Rationale

Decision	Rationale
OOP Structure	Separating inventory management (CarManager) from transaction logic (RentalManager) ensures a clean Separation of Concerns. The RentalManager only needs to know *that* a car can be checked/updated, not *how* the inventory is stored.
In-Memory Dictionary Storage	Using Python dictionaries (self.inventory, self.active_rentals) allows for fast prototyping and simplifies the implementation for a command-line application, as persistence is not a requirement of the initial design.
Integer Car IDs	Using sequential integers (next_car_id) as primary keys for the cars simplifies user input and dictionary lookups.
Input Validation & Error Handling	Implemented try-except blocks (e.g., in check_availability, rent_car) and input checks (e.g., ensuring fields are not empty) to make the application robust against user errors.

9. Implementation Details

The system is implemented entirely in Python, using standard libraries.

- **CarManager**:
    - The inventory dictionary uses the car ID as the key for quick access.

    - The add_car method uses self.next_car_id to ensure unique, sequential IDs.

- **RentalManager**:
    - The active_rentals dictionary uses the customer_name as the key, enforcing that one customer can only have one active rental at a time. The value is a tuple of (car_id, rental_days).
    - The rent_car method orchestrates the availability check, updates the car status via CarManager, and records the rental.
- **UserInterface**:
    - Contains the main run() loop and helper methods (_handle_rent_car, etc.) to manage user interaction flow.

11. Testing Approach

Testing was conducted manually using the console interface (black-box testing) and through unit-like checks (white-box testing) by directly calling class methods.

Key Test Cases:

Test Case	**Steps**	**Expected Result**
T1: Successful Rental	1. View Available Cars (e.g., ID 1). 2. Rent Car ID 1.	Car ID 1 is marked unavailable; Rental record created; Cost displayed.
T2: Rent Unavailable Car	1. Attempt to rent car ID 3 (initially unavailable).	Error message displayed; Car ID 3 remains unavailable.
T3: Successful Return	1. Rent Car ID 2. 2. Return Car for that customer.	Rental record removed; Car ID 2 is marked available.
T4: Invalid Input	Attempt to rent a car using non-numeric ID or days.	Error message displayed; No changes to inventory or rentals.

12. Challenges Faced

The primary challenge was ensuring transactional integrity, specifically:

1. **Synchronization:** Ensuring that when a car is rented, its status in the CarManager's inventory is *atomically* updated to False before or immediately after recording the rental in RentalManager. This was solved by making the RentalManager directly call CarManager.update_availability().
2. **Unique Rental:** The decision to key active_rentals by customer_name was made to prevent a single customer from renting multiple cars, which simplified the return_car process since only the customer name is required for lookup.

13. Learnings & Key Takeaways

- **OOP Design:** The use of distinct, collaborative classes (Manager objects) demonstrated how to effectively model real-world business domains in code, making the system modular and easier to read.
- **Encapsulation:** The RentalManager is successfully encapsulated from the internal data structure of CarManager, relying only on its public methods (check_availability, update_availability).
- **Robust Input:** The necessity of strong input validation and try-except blocks in a console application became clear to handle unexpected user entries (e.g., strings where integers are expected).

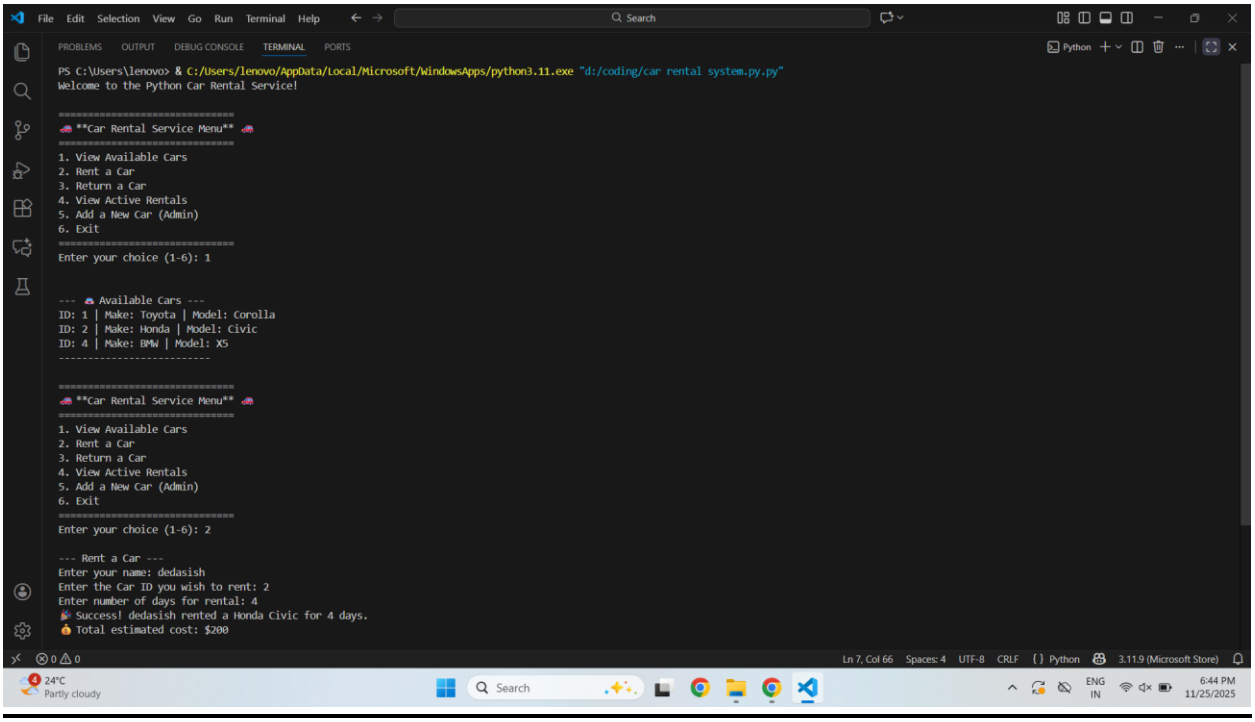
14. Future Enhancements

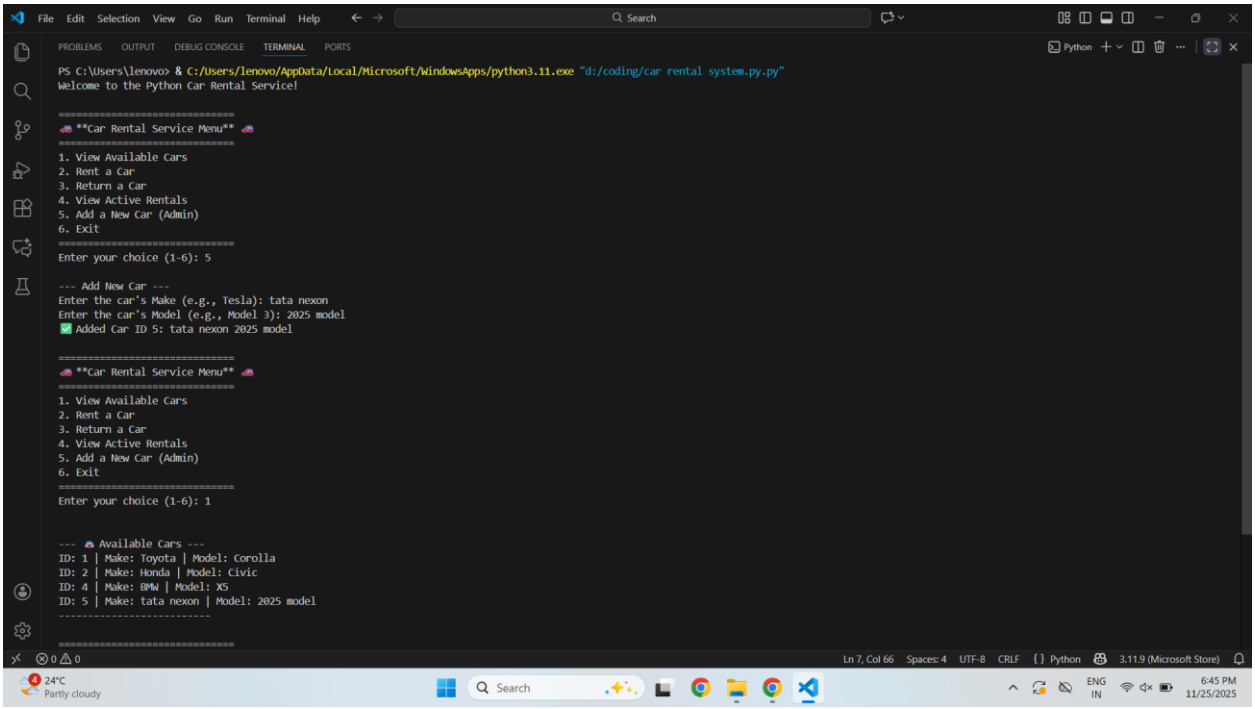
1. **Persistence:** Implement a database (e.g., SQLite, PostgreSQL, or even JSON/CSV file storage) to save inventory and rental data across sessions.
2. **Customer Management:** Introduce a separate CustomerManager class to handle customer registration, verification, and history.
3. **Pricing Models:** Introduce variable pricing based on car make/model/type instead of a single static DAILY_RATE.
4. **Admin Functions:** Add functionality to remove cars, modify car details, and generate reports (e.g., total revenue).
5. **GUI:** Develop a graphical user interface (GUI) using libraries like Tkinter or PyQt to improve usability beyond the command line.

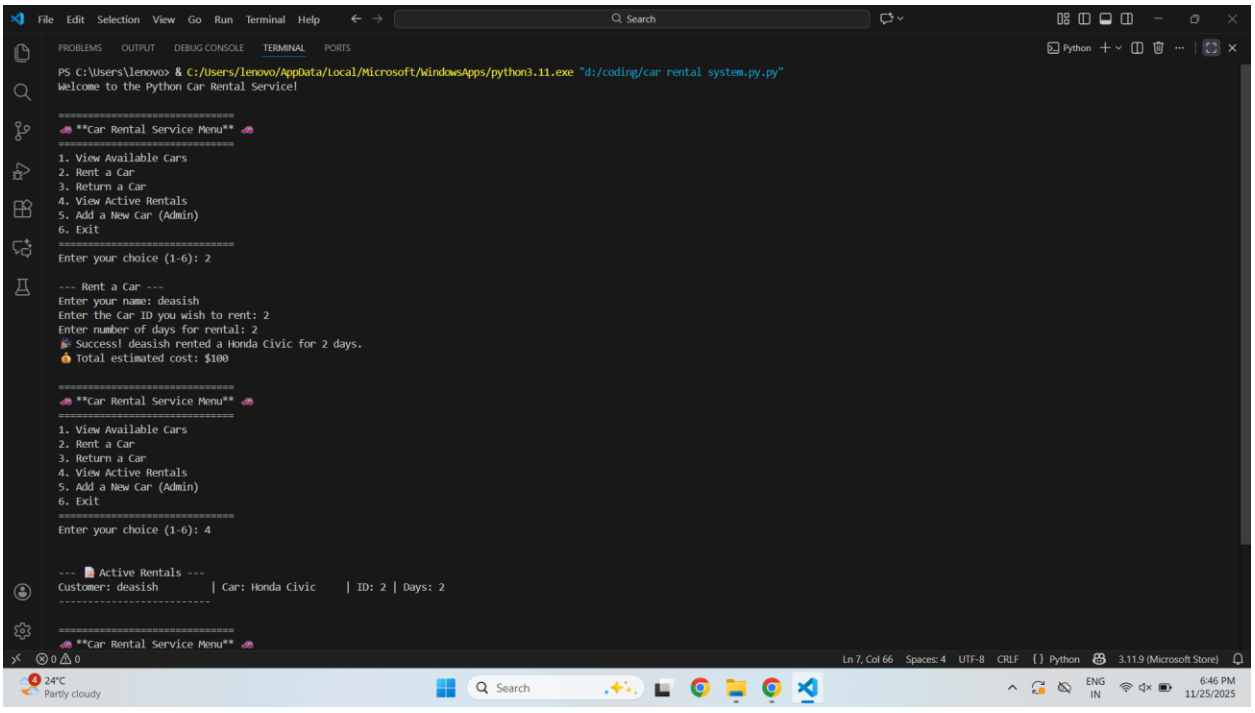
15. References

1. Python Standard Library Documentation (for basic data structures and IO).
2. Object-Oriented Programming (OOP) principles for class design.

Screenshots:






File Edit Selection View Go Run Terminal Help
Search
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python
PS C:\Users\lenovo> & C:/Users/lenovo/AppData/Local/Microsoft/WindowsApps/python3.11.exe "d:/coding/car rental system.py.py"
Welcome to the Python Car Rental Service!
==========
🚗 **Car Rental Service Menu** 🚗
==========
1. View Available Cars
2. Rent a Car
3. Return a Car
4. View Active Rentals
5. Add a New Car (Admin)
6. Exit
==========
Enter your choice (1-6): 2
--- Rent a Car ---
Enter your name: deasish
Enter the Car ID you wish to rent: 2
Enter number of days for rental: 2
🎉 Success! deasish rented a Honda Civic for 2 days.
💰 Total estimated cost: $100
==========
🚗 **Car Rental Service Menu** 🚗
==========
1. View Available Cars
2. Rent a Car
3. Return a Car
4. View Active Rentals
5. Add a New Car (Admin)
6. Exit
==========
Enter your choice (1-6): 4
--- 📄 Active Rentals ---
Customer: deasish | Car: Honda Civic | ID: 2 | Days: 2
==========
🚗 **Car Rental Service Menu** 🚗
Ln 7, Col 66 Spaces: 4 UTF-8 CRLF { } Python 3.11.9 (Microsoft Store)
24°C
Partly cloudy
Search
ENG
IN
6:46 PM
11/25/2025