🚗 “SmartRide – AI-Powered Car Fleet Management System”

🔹 1. Class: `Car`

Represents a smart vehicle in the system.

Attributes

- `plate_number`
- `model`
- `battery_level` (0–100)
- `mileage`
- `status` (available / on_trip / charging)

Methods

- `drive(km)`
- `charge(amount)`
- `__str__()` → Return a nice string summary
- `__eq__()` → Two cars are equal if their plate numbers match
- `__lt__()` → Compare cars by mileage (for sorting)

1. Create 3 cars with different mileage.
2. Sort them using Python's `sorted()` — test `__lt__`.
3. Compare two cars using `==`.
4. Print a car object directly → check `__str__`.

🔹 2. Class: `Driver`

Represents a person controlling a car.

Attributes

- `name`
- `experience_level` (1–10)
- `assigned_car` (can be `None`)

Methods

- `assign_car(car)`
- `__repr__()` → Developer-friendly object display
- `__bool__()` → Returns `False` if driver has no assigned car

1. Create two drivers and assign one a car.
2. Test `if driver:` logic — see how `__bool__` affects flow.
3. Use `repr(driver)` inside a list to display drivers.

🔹 3. Class: `Customer`

Represents a SmartRide app user.

Attributes

- `name`
- `credit_balance`
- `trips` → list of past trip objects

Magic Methods

- `__iadd__()` → Add credit with `+=`
- `__isub__()` → Deduct credit when paying for a trip
- `__len__()` → Return number of completed trips

1. Add 200 units of credit to a customer using `+=`.
2. Deduct trip cost using `-=`.
3. Use `len(customer)` to get trip count.

🔹 4. Class: `Trip`

Represents a single ride or delivery.

Attributes

- `trip_id`
- `driver`
- `customer`
- `distance_km`
- `price_per_km`

Methods

- `calculate_cost()`
- `__str__()` → Pretty summary (e.g. "Trip #T145: Ali → Sara (12km, $30)")
- `__call__()` → When called like `trip()`, return total cost immediately

1. Create a trip and print it.
2. Call the trip object directly to get its cost: `print(trip())`.
3. Add the trip to the customer's trip history.

🔹 5. Class: `Fleet`

Manages all cars and drivers.

Attributes

- `cars`
- `drivers`

Methods

- `__len__()` → Return total number of cars
- `__getitem__()` → Access a car by index
- `__iter__()` → Iterate over all available cars
- `add_car(car)`
- `add_driver(driver)`
- `find_available_car()`

1. Add multiple cars to the fleet and iterate over them with a loop.
2. Access the 2nd car with `fleet[1]`.
3. Get `len(fleet)` for total car count.
4. Find the first available car.

🔹 6. Class: `SystemDatabase`

Handles saving/loading fleet and customer data.

Methods

- `save_to_file(filename, data)` using `pickle`
- `load_from_file(filename)` using `pickle`
- `__enter__()` / `__exit__()` → Context manager for safe I/O

1. Save all customer objects to a `.pkl` file.
2. Reload them into a new list.
3. Use `with SystemDatabase('file.pkl') as db:` style context management.

🔹 7. AI Logic

Add a subclass `SmartCar(Car)` that overrides `drive()`:

- If `battery_level < 10`, print a warning and auto-charge.
- If `mileage > 100000`, mark as "maintenance needed".
- Override `__str__()` to include an AI status note.

🌍 Expansion

- Add a `TripHistory` class that supports slicing (`__getitem__`)
- Implement a `Billing` system using `__add__` to merge bills

🧩 New Features

1. **@log_action Decorator for Trip**
    - Logs every trip's start and completion to a file (`trip.log`).

Example:

```
 @log_action("trip_started")
def start(self): ...
```

    -
2. **@check_battery Decorator for Car**
    - Ensures the car's battery level is above 20% before driving.
    - If not, automatically charges before continuing.
3. **@require_balance(min_amount)** for Customer
    - Prevents booking a trip if the customer's credit is below the minimum required amount.
4. **@measure_time Decorator for SystemDatabase**
    - Measures how long `save_to_file()` and `load_from_file()` take and prints the duration.
5. **TripSession Context Manager**
    - Used with `with` block to log trip lifecycle (start, end, duration) automatically.

Example:

```
with TripSession(trip):
   trip.start()
```

   ○

6. **FleetMaintenance Context Manager**
   - Temporarily puts all cars into maintenance mode and restores their previous status afterward.
7. **Generator: trip_id_generator()**
   - Yields unique trip IDs endlessly (e.g., `T0001`, `T0002`, …).
8. **Generator: available_cars() in Fleet**
   - Lazily yields only available cars — useful for large fleets.
9. **Generator: trip_history(customer)**
   - Streams a customer's past trips one by one instead of returning the full list.
10. **AI SmartCar Extension**
    - Subclass of `Car` that overrides `drive()` with AI decision logic
    - Includes private attribute `__ai_version` with property getter/setter
    - Custom `__str__()` showing AI version and smart status.

🧩 Enhancements

1. **@validate_data Decorator**
   - Validates input arguments for methods like `Trip(distance_km, price_per_km)`
   - Raises a `ValueError` if distance or price are negative.

Example:

@validate_data

def __init__(self, distance_km, price_per_km): ...

-

2. **@retry_on_failure Decorator (for SystemDatabase)**
   - Retries saving/loading data up to 3 times if a file operation fails.
   - Great for simulating network/storage reliability handling.
3. **@track_usage Decorator (for Fleet)**
   - Tracks how many times methods like `find_available_car()` or `add_driver()` are called.
   - Stores stats in a class variable like `Fleet.usage_stats`.
4. **@notify_admin Decorator**
   - When a serious issue occurs (like low credit, car breakdown), logs a notification or prints a simulated alert to the "admin console".
5. **SystemLogger Context Manager**
   - Opens a log file and logs every major system event (trip, charge, save).
   - Auto-closes and timestamps at the end of the session.
6. **Transaction Context (CustomerPayment)**
   - Simulates a safe transaction:
     - Deducts money on enter
     - Refunds it automatically if an error occurs inside the block

Example:

with CustomerPayment(customer, amount):

trip.start()

    ○

7. **CarRental Context Manager**
    - Temporarily assigns a car to a driver during a trip.
    - When the context ends, the car is automatically unassigned and marked available.
8. **generate_trip_reports()**
    - Streams summary reports of completed trips (one per yield).
    - Helps simulate dashboards or analytics tools.
9. **fleet_iterator(condition)**
    - Takes a condition (like `lambda c: c.battery_level < 30`)
    - Yields only cars matching the condition — perfect for filtering large datasets.
10. **customer_rankings(customers)**

- A generator that yields customers sorted by total credit or trips completed — in chunks (lazy loading).

11. **event_stream()**

- A background generator simulating real-time events (`trip_started`, `trip_completed`, `car_charging`, etc.)
- Can be integrated with async or multithreading later.

12. **Abstract Base Class: `Vehicle`**

- Use `abc` module — both `Car` and `SmartCar` inherit from it.
- Enforce abstract methods like `drive()` and `charge()`.

13. **Mixin Class: `Loggable`**

- Provides reusable logging functionality.
- Any class (Driver, Fleet, SystemDatabase) can inherit it to gain `.log_event()` method.

14. **Interface Simulation: `Serializable`**

- Enforces `save()` and `load()` method signatures for all data-handling classes.

15. **Custom Exception Classes**

- Create:
    - `LowBatteryError`
    - `InsufficientCreditError`
    - `CarNotAvailableError`
- Raise them in respective logic — and handle gracefully with decorators or try/except.

16. **AutoBackup Mechanism**

- Every time data is saved, automatically create a `.bak` backup file using a context manager.

17. **Versioned Save Files**

- Use timestamps in filenames like `fleet_2025_10_17.pkl` for history tracking.

18. **Data Encryption (Simulation)**

- Before pickling, “encrypt” data by encoding with Base64.
- Upon loading, decode automatically.

19. **TripAnalytics Class**

- Analyzes data: average trip distance, average price, total revenue, etc.
- Uses generator inputs (e.g., `generate_trip_reports()`).

20. **ChargingStation Manager**

- Handles multiple cars charging simultaneously.
- Use a generator to simulate each car's charging progress (yield every 10% increment).

21. **Dynamic Pricing System**

- Use a decorator `@dynamic_pricing` on `Trip.calculate_cost()`
- Adjusts price based on time of day or distance (e.g., surge pricing).

22. **EventLogger (Singleton)**

- Implements Singleton pattern via `__new__` to ensure only one global logger exists.

23. **CLI Mode (Simulated)**

- Add a class `SmartRideCLI` that lets you run commands like `list_cars()`, `create_trip()`, `view_fleet()`.

24. **AutoSave Decorator**

- After every change (add driver, trip complete), automatically call `SystemDatabase.save_to_file()`.

25. **Iterator Pattern for Trips**

- Allow iterating directly over a customer's trips with `for trip in customer:` (using `__iter__` and `yield` internally).
