Saman Aijaz Siddiqui
Email: siddiqui.1@iitj.ac.in

Smart Cab Allocation System for Efficient Trip Planning

Introduction

This document describes a simplified system for cab allocation and employee cab searching. The system includes classes for `Cab`, `Trip`, `CabAllocator`, `EmployeeCabSearch`, and `RealTimeLocationData`. Example usage demonstrates the allocation of cabs for trips, employee cab searches, and integration with real-time location data.

Classes Overview

1. `Cab` Class

Represents a cab with an `id` and `location`.

```
class Cab:
    def __init__(self, id, location):
        self.id = id
        self.location = location
```

2. `Trip` Class

Represents a trip with an `id` and `start_location`.

```
class Trip:
    def __init__(self, id, start_location):
        self.id = id
        self.start_location = start_location
```

3. `CabAllocator` Class

Allocates the best-suited cab for a trip based on distance.

```
class CabAllocator:
    def __init__(self, cabs):
        self.cabs = cabs

    def suggest_best_cab(self, trip):
        best_cab = min(self.cabs, key=lambda cab: self.calculate_distance(cab.location, trip.start_location))
        return best_cab

    @staticmethod
    def calculate_distance(location1, location2):
        # Simplified distance calculation using Euclidean distance
        return math.sqrt((location1[0] - location2[0])**2 + (location1[1] - location2[1])**2)
```

The `import math` statement at the beginning of the code is used to include the Python math module in the script. In this specific code, the `math` module is utilized for performing mathematical operations, particularly for calculating distances using the `sqrt` function (square root). The Euclidean distance formula, which involves square roots, is employed in the distance calculations within the `CabAllocator` and `EmployeeCabSearch` classes.

We were able to use `math.sqrt` because we have imported math.

4. `EmployeeCabSearch` Class

Facilitates searching for nearby cabs based on an employee's location.

```
class EmployeeCabSearch:
    def __init__(self, cabs):
        self.cabs = cabs

    def suggest_nearby_cabs(self, employee_location):
        nearby_cabs = [cab for cab in self.cabs if self.is_nearby(cab.location, employee_location)]
        return nearby_cabs

    @staticmethod
    def calculate_distance(location1, location2):
        return math.sqrt((location1[0] - location2[0])**2 + (location1[1] - location2[1])**2)

    def is_nearby(self, location1, location2):
        # Simplified distance check
        return self.calculate_distance(location1, location2) < 5  # Assuming a 5 km radius for simplicity
```

5. `RealTimeLocationData` Class

Stores and updates real-time location data for cabs.

```python
class RealTimeLocationData:
    def __init__(self):
        self.cab_locations = {}

    def update_cab_location(self, cab_id, new_location):
        self.cab_locations[cab_id] = new_location
```

Example Usage

```python
# Example Usage:
cabs = [Cab(1, (10, 10)), Cab(2, (15, 15)), Cab(3, (20, 20))]
trips = [Trip(101, (12, 12))]

cab_allocator = CabAllocator(cabs)
employee_cab_search = EmployeeCabSearch(cabs)
real_time_location_data = RealTimeLocationData()

# Simulate real-time data update
real_time_location_data.update_cab_location(1, (11, 11))
```

1. Data Initialization:
    a. `cabs`: A list of `Cab` instances representing the available cabs with their IDs and initial locations.
    b. `trips`: A list of `Trip` instances representing planned trips with IDs and start locations.
2. Object Instantiation:
    a. `cab_allocator`: An instance of the `CabAllocator` class initialized with the list of available cabs.
    b. `employee_cab_search`: An instance of the `EmployeeCabSearch` class initialized with the list of available cabs.
    c. `real_time_location_data`: An instance of the `RealTimeLocationData` class to manage real-time location data for cabs.
3. Real-Time Data Simulation:
    a. `real_time_location_data.update_cab_location(1, (11, 11))`: Simulates updating the location of Cab with ID 1 to (11, 11).

1. Admin's Cab Allocation Optimization:

```
# 1. Admin's Cab Allocation Optimization
for trip in trips:
    best_cab = cab_allocator.suggest_best_cab(trip)
    print(f"Trip {trip.id}: Allocate Cab {best_cab.id}")
```

Explanation:

- The code simulates an admin optimizing cab allocation for each trip in the `trips` list.
- For each trip, the `cab_allocator.suggest_best_cab(trip)` method is called to find the best-suited cab based on the minimum distance to the trip's start location.
- The result is printed, indicating the trip ID and the allocated cab ID.

2. Employee's Cab Search Optimization:

```
# 2. Employee's Cab Search Optimization
employee_location = (14, 14)
nearby_cabs = employee_cab_search.suggest_nearby_cabs(employee_location)
print(f"Employee at {employee_location}: Nearby Cabs {[cab.id for cab in nearby_cabs]}")
```

Explanation:

- The code simulates an employee optimizing cab search based on their current location `(14, 14)`.
- The `employee_cab_search.suggest_nearby_cabs(employee_location)` method is called to find cabs within a 5 km radius of the employee's location.
- The result is printed, showing the employee's location and the IDs of nearby cabs.

3. Real-Time Location Data Integration:

```
# 3. Real-Time Location Data Integration
real_time_location_data.update_cab_location(2, (16, 16))

# Refresh suggestions with updated location data
nearby_cabs = employee_cab_search.suggest_nearby_cabs(employee_location)
print(f"After update - Employee at {employee_location}: Nearby Cabs {[cab.id for cab in nearby_cabs]}")
```

Explanation:

- The code simulates the integration of real-time location data.
- The location of Cab with ID 2 is updated to `(16, 16)`.
- After the update, the employee re-runs the cab search to get the latest suggestions based on the updated location data.
- The result is printed, showing the employee's location and the IDs of nearby cabs after the update.

Conclusion:

The example usage demonstrates the functionality of the cab allocation and employee cab search systems, including the optimization of allocation, real-time data updates, and efficient employee cab searching. The integration of classes showcases a modular and organized approach to managing cab-related operations in a transportation system.

MODIFICATIONS IN THE CODE FOR ADDITONAL POINTS:

1. Authentication:

```
class UserAuthentication:
    def authenticate_user(self, username, password):
        # Simulated authentication logic for demonstration purposes
        return username == "admin" and password == "admin123"
```

```
# 1. Authentication
username = input("Enter your username: ")
password = input("Enter your password: ")

authenticated = authenticator.authenticate_user(username, password)
```

OUTPUT:

```
Enter your username: admin
Enter your password: admin123
Authentication successful!
```

```
Enter your username: admin
Enter your password: admin123
Authentication successful!
```

The `authenticate_user` method returns `True` if the provided username and password match the database. In this case, the considered username is "admin" and the password is "admin123", indicating successful authentication. Otherwise, it returns `False`.

Authentication Check:

- The snippet begins with a conditional check (`if authenticated`) to verify the success of user authentication.
- If authentication is successful, the subsequent operations are executed; otherwise, a failure message is displayed.

2. Cost Estimation - Time and Space:

- The time and space complexity of the system are analyzed using methods from the `system_analyzer` object.

```
# 2. Cost Estimation - Time and Space
system_analyzer.analyze_time_complexity()
system_analyzer.analyze_space_complexity()
```

3. Handling System Failure Cases:

- The system's fault tolerance mechanism is invoked to handle potential failure scenarios.

```
# 3. Handling System Failure Cases
fault_tolerance_handler.handle_system_failure()
```

4. Object-Oriented Programming Language (OOPS):
   Entire code is written in OOPS

5. Trade-offs in the System:

- Documentation on trade-offs in the system is generated using the `trade_off_documentation` object.

```
# 5. Trade-offs in the System
trade_off_documentation.document_trade_offs()
```

6. System Monitoring:
   - An event indicating the start of the system is logged using the `system_monitor` object.

```
# 6. System Monitoring
system_monitor.log_event("System started")
```

7. Caching:
   - Data is added to the system's cache using the `caching_mechanism` object.

```
# 7. Caching
caching_mechanism.add_to_cache("key", "value")
```

8. Error and Exception Handling:
   - A try-except block demonstrates error and exception handling, with the `error_handler` object managing any raised exceptions.

```
# 8. Error and Exception Handling
try:
    # Code that may raise an error
    pass
except Exception as e:
    error_handler.handle_error(e)
```

Overall Generated Outputs:

```
Trip 101: Allocate Cab 1
Employee at (14, 14): Nearby Cabs [2]
```

```
After update - Employee at (14, 14): Nearby Cabs [2]
```

For successful authentication:

```
Enter your username: admin
Enter your password: admin123
Authentication successful!
```

```
Employee at (14, 14): Nearby Cabs [2]
After update - Employee at (14, 14): Nearby Cabs [2]
System Logs:
System started
System started
Employee at (14, 14) searched for nearby cabs
System failure handled
Real-time location data updated
```

```
After update - Employee at (14, 14): Nearby Cabs [2]
```

For unsuccessful authentication:

```
Enter your username: saman
Enter your password: siddiqui
Authentication failed. Exiting.
```

Conclusion

This sequence of operations represents a comprehensive approach to system functionality, covering aspects such as authentication, system analysis, fault tolerance, documentation, monitoring, caching, and error handling. The modular structure allows for easy integration of additional functionalities or modifications to existing operations.