Software to Support Operation of a Driverless Car

README

The development process for the driverless car involved applying core object-oriented programming principles and implementing critical functionalities like real-world autonomous vehicle operations. Using an object-oriented design ensured the system was modular, scalable, and easy to maintain. Encapsulating data and functionalities into distinct classes, such as Sensor, Navigator, Route, and ObstacleDetector, achieved a clear and organized structure. The modularity also made the code more readable for testing and debugging.

Each class was designed with a single responsibility in mind:

1) **Sensor:** Simulates input data from vehicle sensors.
2) **Navigator**: Manages navigation and interacts with the map and route.
3) **ObstacleDetector:** Detects obstacles using sensor data and triggers appropriate responses.
4) **UserInterfaceCLI:** Provides a command-line interface for user interaction.

This clear delineation of responsibilities ensured that each class handled its specific aspect of the system, reducing coupling.

Overview and System Requirements:

The driverless car system is a command-line application that simulates the basic functionalities of an autonomous vehicle. The system includes route selection,

navigation, obstacle detection, speed adjustment, and user interaction through a command-line interface.

To run the driverless car system, ensure you have Python Version 3.6 or later and any OS that supports Python (e.g., Windows, macOS, Linux). No external libraries beyond the standard Python library are required.

Driverless Car Code:

```python
# The Sensor class ibelow is designed to simulate the input data from vehicle sensors,
# specifically for detecting obstacles. It allows setting and retrieving information about obstacles,
# including their type and proximity.

class Sensor:
    """Simulates input data from vehicle sensors."""
    def __init__(self): # initializes an instance of the Sensor class
        self.obstacle_type = None # this attribute is initialized to None and is intended to store the type of
                                  # obstacle detected by the sensor (e.g., "Pedestrian", "Car")
        self.proximity = None     # intended to store the proximity (distance) of the obstacle from the vehicle in meters.

    def set_obstacle(self, obstacle_type, proximity):
        """This method allows setting the obstacle type and proximity based on the input."""
        self.obstacle_type = obstacle_type
        self.proximity = proximity

    def get_obstacle(self):
        """Returns the set obstacle type and proximity."""
        return self.obstacle_type, self.proximity

# The Obstacle class represents an obstacle that has been detected by the vehicle's sensors.
# This class is designed to store information about the type of obstacle and its distance from the vehicle,
# and it provides a method to display this information.

class Obstacle:
    """Represents an obstacle detected by sensors."""
    def __init__(self, type, distance): # The constructor method initializes an instance of the Obstacle class
                                        # where 'type' and 'distance' of the obstacle are stored as parameters.
        self.type = type
        self.distance = distance

```

```
    def __str__(self):
        return f"{self.type} at {self.distance} meters"

# The Map class in the provided code manages static geographical data using a dictionary to store predefined routes.
# This class allows adding new routes and retrieving existing ones with their names.

class Map:
    """Manages static geographical data using a dictionary."""
    def __init__(self):

# 'routes' attribute is a dictionary that stores routes. Each key in the dictionary is a route name, and the corresponding value
# is a list of waypoints (locations) that define the route.

        self.routes = {
            'Default Route': ['Start', 'Interim Point', 'Default Destination']
        }

    def add_route(self, route_name, waypoints):
        """Adds a new route to the map."""
        self.routes[route_name] = waypoints

    def get_route(self, route_name):
        """Retrieves a route by name."""
        return self.routes.get(route_name)


# The Route class in the provided code is responsible for handling the planning and updating of paths based on the
# routes defined in the Map class


class Route:
    """Handles the planning of paths."""
    def __init__(self, map_instance):
        self.map = map_instance
        self.current_route = self.map.get_route('Default Route') # This attribute stores the current route of the vehicle

    def update_route(self, route_name, manual_update=True):
        """Updates the route automatically (e.g. due to obstacle detection) or due to manual input."""
        if manual_update:
            self.current_route[-1] = route_name  # Update the destination manually
                                                 # this is used when user overrides a new destination to autonomous vehicle.
        else:
            self.current_route[1] = route_name  # System automatic update (due to obstacle detection)
        print(f"Route updated to: {self.current_route}")

# The Navigator class in the provided code is responsible for managing the navigation of the driverless car.
# It interacts with both the Map and Route classes to set and update the vehicle's path, as well as control the vehicle's speed.

class Navigator:
    """Manages navigation and interacts with the map and route."""
    def __init__(self, map, route):
        self.map = map
        self.route = route
        self.current_speed = 0  # Initialize speed to zero

    def set_speed(self, speed):
        """Sets the vehicle's speed to a new value specified by the user or autoamtically set by vehicle."""
        """current_speed is updated each time vehicle's speed changes"""
        self.current_speed = speed
        print(f"Vehicle speed set to {self.current_speed} km/h.")

```

```
    def stop_car(self):
        """Stops the car."""
        print("Car has been stopped.")

    def select_route(self, route_name):
        """Allows selection of a route from the map."""
        self.route.update_route(route_name)

# The ObstacleDetector class in the provided code is responsible for detecting obstacles
# using data from sensors and managing the vehicle's response to these obstacles.
# It interacts with the Navigator, Sensor, and UserInterfaceCLI classes to handle obstacle detection and subsequent actions.
# Based on the type and proximity of the obstacle, it calls update_route (to change the direction), set_speed (to adjust the speed),
# alert_user (to inform the user regarding the obstacles) and stop_car (to stop the car) methods from different classes.

class ObstacleDetector:
    """Detects obstacles using data from sensors."""
    def __init__(self, navigator, sensor, user_interface):
        self.navigator = navigator
        self.sensor = sensor
        self.user_interface = user_interface

    def detect_obstacle(self):
        obstacle_type, proximity = self.sensor.get_obstacle()
        self.user_interface.alert_user(f"Obstacle detected: {obstacle_type} at {proximity} meters")
        if proximity < 10:
            self.navigator.stop_car()
        elif 10 <= proximity <= 50:
            self.navigator.set_speed(20)
            self.navigator.route.update_route('Different Route', manual_update=False)
            self.user_interface.alert_user("Route has been adjusted due to obstacle detection.")


# The UserInterfaceCLI class in the provided code represents a command-line interface (CLI) for user interaction
# with the driverless car system. It allows the user to select routes, start navigation, simulate obstacle detection,
# stop the car, and exit the system.

class UserInterfaceCLI:
    """Interface for user interaction."""
    def __init__(self, navigator, obstacle_detector):
        self.navigator = navigator
        self.obstacle_detector = obstacle_detector

    def alert_user(self, message):
        """Alert the user with a specific message."""
        print(f"ALERT: {message}")

    def start(self):
        while True:
            print("\nDriverless Car System Menu")
            print("1. Select Route")
            print("2. Start Navigation")
            print("3. Simulate Obstacle Detection")
            print("4. Stop Car")
            print("5. Exit System")
            choice = input("Enter your choice: ")

            if choice == '1':
                route_name = input("Enter route name: ") # user can input the destination
                self.navigator.select_route(route_name)
                self.navigator.set_speed(60)  # Set default speed to 60 km/h after selecting a route
                print(f"Route {route_name} selected. Default speed set to 60 km/h.")
            elif choice == '2':
                print("Navigation Menu:")
```

```
                print("a. Display Current Route")
                print("b. Update Route")
                print("c. Set Speed")
                nav_choice = input("Choose an action: ")
                if nav_choice == 'a':
                    print(f"Current route: {self.navigator.route.current_route}")
                elif nav_choice == 'b':
                    new_route_name = input("Enter new route name: ")
                    self.navigator.select_route(new_route_name)
                elif nav_choice == 'c':
                    new_speed = int(input("Enter new speed for the car: "))
                    self.navigator.set_speed(new_speed)
            elif choice == '3':
                obstacle_type = input("Enter obstacle type (e.g., Pedestrian, Vehicle): ")
                proximity = int(input("Enter proximity of obstacle (in meters): "))
                self.obstacle_detector.sensor.set_obstacle(obstacle_type, proximity)
                self.obstacle_detector.detect_obstacle()
            elif choice == '4':
                self.navigator.stop_car()
            elif choice == '5':
                print("Exiting system.")
                break
            else:
                print("Invalid choice. Please try again.")


# Execute the script directly.

if __name__ == "__main__":
    map_instance = Map()
    route_instance = Route(map_instance)
    navigator = Navigator(map_instance, route_instance)
    sensor = Sensor()
    ui = UserInterfaceCLI(navigator, None)  # Initially setting None for obstacle_detector
    obstacle_detector = ObstacleDetector(navigator, sensor, ui)
    ui.obstacle_detector = obstacle_detector  # Setting the obstacle_detector with the instance of UI
    ui.start()
```

Updates in the implementation:

In response to feedback on diagrams, several key functionalities have been added to enhance the realism and effectiveness of the driverless car system, such as the ability to start the car, reduce speed, and change routes dynamically.

Starting the Car - The system has integrated functionality to start the car. This allows the simulation to reflect a more realistic operation where the vehicle is initially stationary and requires a command to begin movement through the user manual system (which the UserInterfaceCLI class achieves).

Reducing Speed - This is crucial for scenarios where the car needs to slow down due to obstacles or other road conditions (it was added as a method under the Navigator class).

Changing Routes - A feature to change routes dynamically based on obstacle detection or user input has been implemented, which is essential for realistic navigation systems.

Data structures:

The system's primary data structures are lists and dictionaries. Lists are used in the Map and Route classes to store sequences of route waypoints. The choice of lists is driven by the need for an ordered collection of elements that can be easily accessed, updated, and iterated.

In the Map class, lists store the waypoints of each route. The waypoints are ordered to reflect the travel sequence from the starting point to the destination. Dictionaries are used in the Map class to store routes by their names, allowing efficient key-value pair mapping and quick retrieval of routes based on their names.

```
class Map:
    """Manages static geographical data using a dictionary."""
    def __init__(self):

# 'routes' attribute is a dictionary that stores routes. Each key in the dictionary is a route name, and the corresponding value
# is a list of waypoints (locations) that define the route.

        self.routes = {
            'Default Route': ['Start', 'Interim Point', 'Default Destination']
        }

    def add_route(self, route_name, waypoints):
        """Adds a new route to the map."""
        self.routes[route_name] = waypoints

    def get_route(self, route_name):
        """Retrieves a route by name."""
        return self.routes.get(route_name)
```

In the Route class, the current_route attribute is a list that stores the current sequence of waypoints. Lists provide the flexibility needed to update specific elements within the route, such as changing the destination or inserting a new waypoint due to an obstacle.

```python
class Route:
    """Handles the planning of paths."""
    def __init__(self, map_instance):
        self.map = map_instance
        self.current_route = self.map.get_route('Default Route') # This attribute stores the current route of the vehicle

    def update_route(self, route_name, manual_update=True):
        """Updates the route automatically (e.g. due to obstacle detection) or due to manual input."""
        if manual_update:
            self.current_route[-1] = route_name  # Update the destination manually
                                                 # this is used when user overrides a new destination to autonomous vehicle.
        else:
            self.current_route[1] = route_name  # System automatic update (due to obstacle detection)
        print(f"Route updated to: {self.current_route}")
```

OOP principles applied:

The code for the driverless vehicle system demonstrates the core OOP principles of encapsulation, abstraction, and composition. These principles help organize the code in a modular, reusable, and maintainable manner, making it easier to manage the system's complexity.

Encapsulation - the concept of grouping the attributes and methods that operate on the data into a single class to limit direct access to some of the object's components. This can avoid unintended data changes.

These are some examples of class and method encapsulation from the code:

```python
class Sensor:
    """Simulates input data from vehicle sensors."""
    def __init__(self): # initializes an instance of the Sensor class
        self.obstacle_type = None # this attribute is initialized to None and is intended to store the type of
                                  # obstacle detected by the sensor (e.g., "Pedestrian", "Car")
        self.proximity = None     # intended to store the proximity (distance) of the obstacle from the vehicle in meters.
```

```python
class Navigator:
    """Manages navigation and interacts with the map and route."""
    def __init__(self, map, route):
        self.map = map
        self.route = route
        self.current_speed = 0  # Initialize speed to zero
```

Here, the Sensor and Navigator classes encapsulate their respective attributes.

```python
    def set_obstacle(self, obstacle_type, proximity):
        """This method allows setting the obstacle type and proximity based on the input."""
        self.obstacle_type = obstacle_type
        self.proximity = proximity
```

```python
def set_speed(self, speed):
    """Sets the vehicle's speed to a new value specified by the user or autoamtically set by vehicle."""
    """current_speed is updated each time vehicle's speed changes"""
    self.current_speed = speed
```

Methods in these classes operate on the encapsulated data.

Abstraction - hiding the complicated implementation and instead showing only the necessary features of the object. For example, the UserInterfaceCLI class provides methods like alert_user and starts to abstract the complex logic behind a simple interface:

```python
class UserInterfaceCLI:
    """Interface for user interaction."""
    def __init__(self, navigator, obstacle_detector):
        self.navigator = navigator
        self.obstacle_detector = obstacle_detector

    def alert_user(self, message):
        """Alert the user with a specific message."""
        print(f"ALERT: {message}")

    def start(self):
```

Composition - a design principle in which a class is composed of one or more objects of other classes rather than inheriting from them. For example, the ObstacleDetector class comprises Navigator, Sensor, and UserInterfaceCLI instances.

```python
class ObstacleDetector:
    """Detects obstacles using data from sensors."""
    def __init__(self, navigator, sensor, user_interface):
        self.navigator = navigator
        self.sensor = sensor
        self.user_interface = user_interface
```

Sample dataset to test:

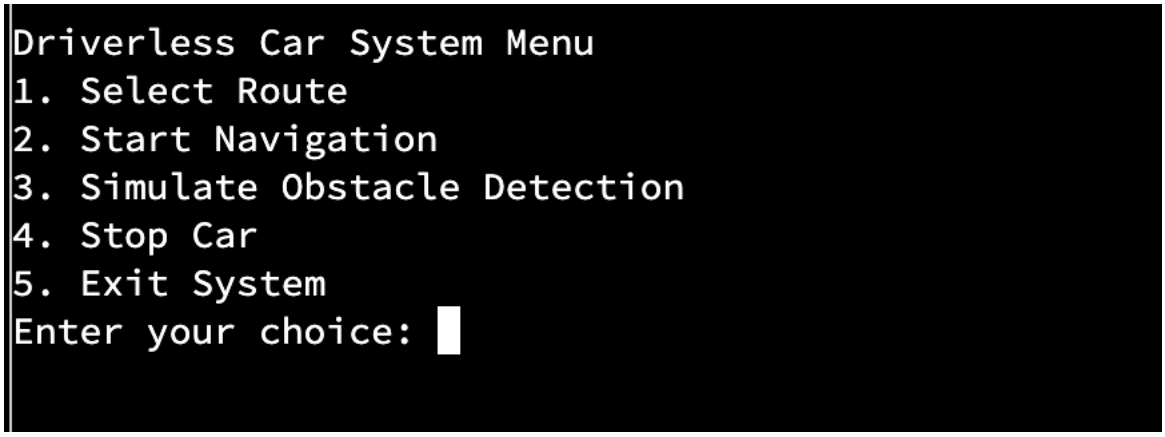
Sample datasets (route and obstacle) to test some of the functionalities (ObstacleDetector and Navigator classes) in JSON format:

```json
{
  "routes": [
    {
      "route_name": "Default Route",
      "waypoints": ["Start", "Interim Point", "Default Destination"]
    },
    {
      "route_name": "Scenic Route",
      "waypoints": ["Start", "Scenic View", "Lake"]
    },
    {
      "route_name": "City Route",
      "waypoints": ["Start", "Downtown", "Office"]
    }
  ],
  "obstacles": [
    {
      "type": "Pedestrian",
      "proximity": 5
    },
    {
      "type": "Car",
      "proximity": 30
    },
    {
      "type": "Animal",
      "proximity": 15
    }
  ]
}
```

How to use the Driverless Car application and testing approach:

Before automated testing with the Assert function, manual tests are carried out by running the code and checking if the output is correct or false at each part of the application – to observe if we see the expected output in each unit of the code.

1) When running the script, the UI needs to initiate to show five main functionalities for using a driverless car application:



Outcome: success (all five prompts are displayed when running the code).

2) By selecting ‘1’, the app should allow a user to input a route. The driverless car should then start moving towards the set destination, and the user should receive feedback about the destination, direction (e.g. “Interim Point” before destination), and the speed of the vehicle via User Interface:

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 1
Enter route name: Berlin
Route updated to: ['Start', 'Interim Point', 'Berlin']
Vehicle speed set to 60 km/h.
Route Berlin selected. Default speed set to 60 km/h.
```

Outcome: success (I was able to input the destination as “Berlin” in the “Enter route name” prompt and received feedback via UI that the car has started moving towards “Berlin” by going through “Interim Point” with the default speed of 60 km/h).

3) By selecting “2”, the app should show the navigation menu and allow the user to access any of its three functionalities: display the current route, update the route, and set speed:

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 2
Navigation Menu:
a. Display Current Route
b. Update Route
c. Set Speed
Choose an action:
```

Outcome: success (“Navigation Menu” printed on the UI, and all three functionalities can be selected)

4) By selecting “a” in the navigation menu, the user should be able to see the current route of the driverless car:

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 2
Navigation Menu:
a. Display Current Route
b. Update Route
c. Set Speed
Choose an action: a
Current route: ['Start', 'Interim Point', 'Berlin']
```

Outcome: Success

5) By selecting “2” (start navigation) and then “b,” the user should be able to update the route and change the destination. After the application updates the destination of the route, the UI should display the new route that the user inputted:

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 2
Navigation Menu:
a. Display Current Route
b. Update Route
c. Set Speed
Choose an action: b
Enter new route name: London
Route updated to: ['Start', 'Interim Point', 'London']
```

Outcome: success (the route was updated, and the destination changed from "Berlin" to "London").

6) By selecting "2" (start navigation) and then "c," the user should be able to set the speed of the driverless car (he can override the default speed of 60 km/h):

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 2
Navigation Menu:
a. Display Current Route
b. Update Route
c. Set Speed
Choose an action: c
Enter new speed for the car: 120
Vehicle speed set to 120 km/h.
```

Outcome: success (speed of the car was successfully updated to 120 km/h, and the update was displayed on UI)

7) By selecting option “3” in the system menu, the user can simulate obstacle detection (to simulate how a driverless car would behave when sensors detect an object on the road). Depending on the object and proximity (in Meters) of the object, driverless vehicles should behave differently:
    a. If the object is further than 50 Meters, the user should receive an alert warning on the UI to allow him to change the route. However, the driverless car does not make any adjustments until the object is closer than 50 meters.

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 3
Enter obstacle type (e.g., Pedestrian, Vehicle): Car
Enter proximity of obstacle (in meters): 100
ALERT: Obstacle detected: Car at 100 meters
```

Outcome: success (Alerts were displayed on the UI)

b. Suppose the distance of the obstacle/object is between 10 and 50 meters. In that case, the driverless car will slow down (speed is updated to 20 km/h), and the route changes (destination stays the same, but the vehicle will change the direction – “interim point” gets updated in the route). The feedback of these changes should be displayed to a user on the UI:

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 3
Enter obstacle type (e.g., Pedestrian, Vehicle): Car
Enter proximity of obstacle (in meters): 45
ALERT: Obstacle detected: Car at 45 meters
Vehicle speed set to 20 km/h.
Route updated to: ['Start', 'Different Route', 'London']
ALERT: Route has been adjusted due to obstacle detection.
```

Outcome: success (alert was displayed, route was adjusted, and the car slowed to 20 km/h).

c. If the distance of the obstacle/object is less than 10 meters, it's too late to reduce the speed and adjust the route (due to the physics of slowing down, the crash might not be avoided). Instead, the car entirely stops, allowing the user to start the vehicle again and choose a new route:

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 3
Enter obstacle type (e.g., Pedestrian, Vehicle): Car
Enter proximity of obstacle (in meters): 9
ALERT: Obstacle detected: Car at 9 meters
Car has been stopped.
```

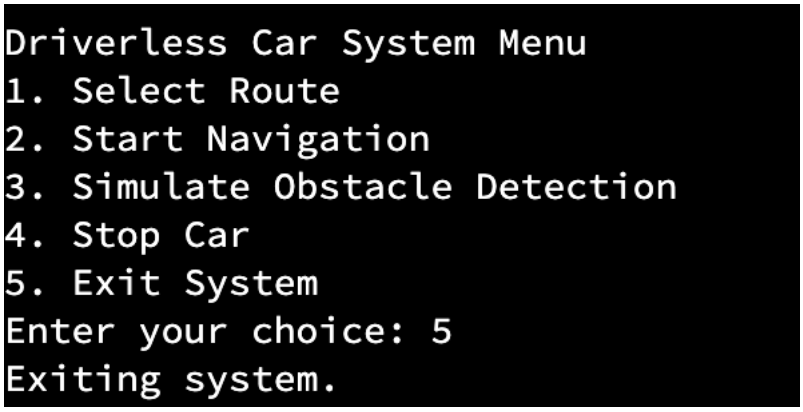
Outcome: success (alert was displayed, and the car was stopped).

8) The user should be able to override the function to stop the car by selecting option "4" in the system menu:

```
Driverless Car System Menu
1. Select Route
2. Start Navigation
3. Simulate Obstacle Detection
4. Stop Car
5. Exit System
Enter your choice: 4
Car has been stopped.
```

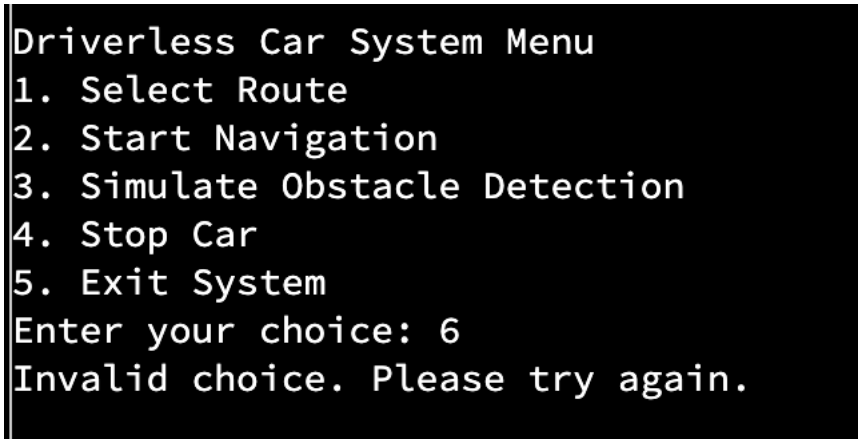
Outcome: success (the car was stopped, and the message was displayed on UI).

9) The user should be able to exit the system menu by selecting option "5" and see the message that the user is "Exiting system." of the system menu:



Outcome: success (the system menu was exited, and the message was displayed on UI).

10) On the system menu home page, if an option other than “1”, “2”, “3”, “4”, or “5” is selected, the user should be prompted that his choice was invalid and that he needs to try again and choose one of the existing functionalities.



Outcome: Success

Automated Unit testing:

The script calls its various classes and functions for automated testing with test inputs and checks if they return correct outputs. Specifically, it uses Python’s Assert

function to check if particular conditions are met. If it isn't, it raises an AssertionError exception. This can be very useful for testing and debugging code, as it allows you to ensure your assumptions about the code are correct. When a particular condition (output of a code that is being tested) is met, the Assert function equals True, and nothing happens. However, when it is false (the expected output is not met), then AssertionError is raised, and a specified error message will be returned (w3schools, 2024). Specific tests with the Assert function are shown below. In the future, these tests can be enhanced further with more elaborate tests to check other classes and methods of the code.

Unit testing refers to a part of the code created to test other units of code (DATAQUEST, 2022). Below is a unit test with the Assert() function, which individually tests various functions of the main code from different classes, such as current_speed or select_route.

The code below can be considered Unit testing due to the following:

Modularity: Each test function is focused on a specific unit of functionality.

Independence: The tests do not depend on other parts of the system to work correctly or rely on other resources to run.

Assertions - the use of assert statements verifies that the code behaves as expected. If the conditions in the assert statements are not met, the testing fails due to an AssertionError.

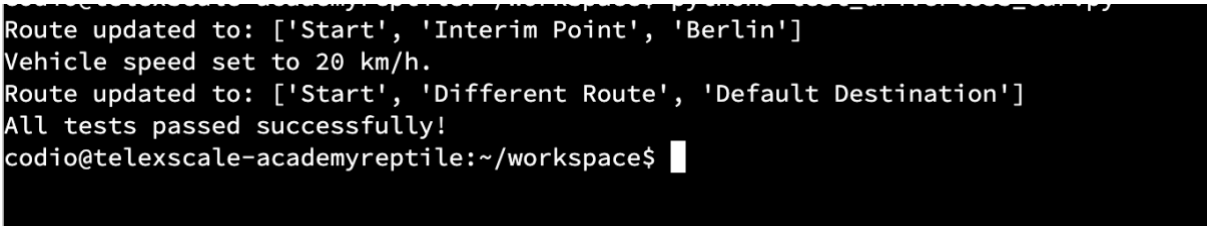
Automation - the test functions can be run automatically.

Isolation - the tests are isolated from each other, meaning the state in one test does not affect another.

```python
# test_driverless_car.py
# Import classes from the drivereless_car file to test the functions in those classes
from driverless_car import Map, Route, Navigator, Sensor, ObstacleDetector, UserInterfaceCLI

# Set up the necessary objects and their relationships to test the route selection functionality
def test_route_selection():
    map_instance = Map()
    route_instance = Route(map_instance)
    navigator = Navigator(map_instance, route_instance)

    # Test selecting a new route
    # Runnin "select_route('Berlin')" function should set "Berlin" as destination
    # therefore the Assert command is expected to return True for 'current_route' = '['Start', 'Interim Point', 'Berlin']'
    # since 'select_route' sets the destination (in this case to 'Berlin'), and 'Start' and 'Interim Point' remain unchanged in
    # the Route dataset
    navigator.select_route('Berlin')
    assert navigator.route.current_route == ['Start', 'Interim Point', 'Berlin'], "Failed to select the new Route correctly."

# Set up the necessary objects and their relationships to test the obstacle detection functionality

def test_obstacle_detection():
    map_instance = Map()
    route_instance = Route(map_instance)
    navigator = Navigator(map_instance, route_instance)
    sensor = Sensor()

# A new class MockUI is used to simulate the UserInterfaceCLI class, but instead of printing messages,
# it collects messages in a list, which are then later checked to verify that the correct
# messages were generated during the test.
# Using MockUI class allows the test to verify that the correct messages are generated when obstacles are detected
# without printing to the console. It makes it easy to check if the system behaves as expected by examining the contents of
# mock_ui.messages (python, 2024).


    class MockUI:
        def __init__(self):
            self.messages = []
        def alert_user(self, message):
            self.messages.append(message)

# Creates an instance of MockUI, which will be used in place of the actual user interface.
    mock_ui = MockUI()

# Creates an instance of the ObstacleDetector class, passing the navigator, sensor, and mock_ui instances.
# This allows the obstacle detector to use the mock user interface to retreive "alert" messages.

    obstacle_detector = ObstacleDetector(navigator, sensor, mock_ui)

    # Simulate obstacle detection within moderate proximity (10 - 50 meters) and test three functionalities:
    # 1) Setting the current_speed to 20 should update the speed of the vehicle to '20'
    # 2) After moderate proximity (10 - 50 meters) obstacle is set via 'sensor.set_obstacle', 'Routes' dataset should be updated
    # with a new destination and the 'mock_ui' variable should store the alert:
    # "Route has been adjusted due to obstacle detection.", hence the second Assert statement should be true.
    # 3) Besides the speed, the Route (direction of the vehicle) should also get updated to 'Different Route'
    sensor.set_obstacle('Pedestrian', 30)
    obstacle_detector.detect_obstacle()
    assert navigator.current_speed == 20, "Failed to set speed correctly for moderate proximity."
    assert "Route has been adjusted due to obstacle detection." in mock_ui.messages, "Route adjustment alert failed to trigger."
    assert navigator.route.current_route[1] == 'Different Route', "Route was not updated to 'Different Route'."

# Below code runs the above defined functions.
# This script will execute the tests and print "All tests passed successfully!" if all assertions pass.
# If any assertion fails, it will provide details on what went wrong.
```

```python
if __name__ == "__main__":
    test_route_selection()
    test_obstacle_detection()
    print("All tests passed successfully!")
```

Outcome: all tests are passed successfully:



Limitations and future improvements

Simulation-Based System: The current implementation is entirely simulation-based and does not interact with real-world hardware or sensors. This limits the ability to test the system under actual driving conditions and with absolute sensor data. Future system versions could integrate with real-world sensors and hardware to provide a more realistic and accurate simulation.

Essential Obstacle Detection: The obstacle detection mechanism is simplified and only considers the type and proximity of obstacles. It does not account for more complex scenarios such as moving obstacles, varying obstacle sizes, or different types of sensor data (e.g., LIDAR, radar). The future solution could incorporate data from these sensors and include the use of machine learning algorithms to classify and predict the behavior of obstacles.

Static Map Data: The map data is static and predefined within the system. It does not support dynamic updates or integration with external mapping services (e.g.,

Google Maps) to reflect real-time road conditions or traffic changes. The future system could integrate the system with external mapping services to dynamically update map data based on real-time road conditions, traffic, and other relevant factors.

Limited User Interaction: The user interaction is limited to a command-line interface, which may only be user-friendly for some users. There is no graphical user interface (GUI) or voice command integration, which is more intuitive for end-users.

Simplified Navigation and Route Planning: The route planning and navigation logic is quite basic and needs to account for real-world factors such as traffic conditions, road closures, or optimal route selection based on multiple criteria (e.g., shortest distance, fastest route).

Lack of Advanced Safety Features: The system could benefit from advanced safety features. For example, emergency braking, lane-keeping assistance, and adaptive cruise control, which are essential for modern driverless car systems.

References:

DATAQUEST (2022) A Beginner’s Guide to Unit Tests in Python. Available from: https://www.dataquest.io/blog/unit-tests-python/ [Accessed 26 May 2024].

python (2024) unittest.mock — mock object library. Available from: https://docs.python.org/3/library/unittest.mock.html [Accessed 26 May 2024].

w3schools (2024) Python User Input. Available from: w3schools.com/python/ref_keyword_assert.asp [Accessed 26 May 2024].