Practical 5

1.1 Polymorphism, Encapsulation, and Method & Attributes

Aim:

Use private attributes and provide public getter and setter methods to access them.
Demonstrate polymorphism by defining a common interface and implementing it in multiple classes.
Override methods in a subclass to provide specific implementations.
Define multiple methods with the same name but different parameters to handle different types of inputs.
Define class methods using the @classmethod decorator to operate on class-level data.
Define static methods using the @staticmethod decorator for utility functions that do not depend on instance or class data.
Use the @property decorator to create getter and setter methods for attributes.

Code:

```
class Vehicle:
    # Class variable
    total_vehicles = 0

    def __init__(self, make, model, year):
        # Private attributes with encapsulation
        self._make = make
        self._model = model
        self._year = year
        self._is_running = False
        Vehicle.total_vehicles += 1

    # Getter methods
    @property
    def make(self):
        return self._make

    @property
    def model(self):
        return self._model

    @property
    def year(self):
        return self._year

    @property
    def is_running(self):
        return self._is_running

    # Setter methods
    @make.setter
    def make(self, value):
```

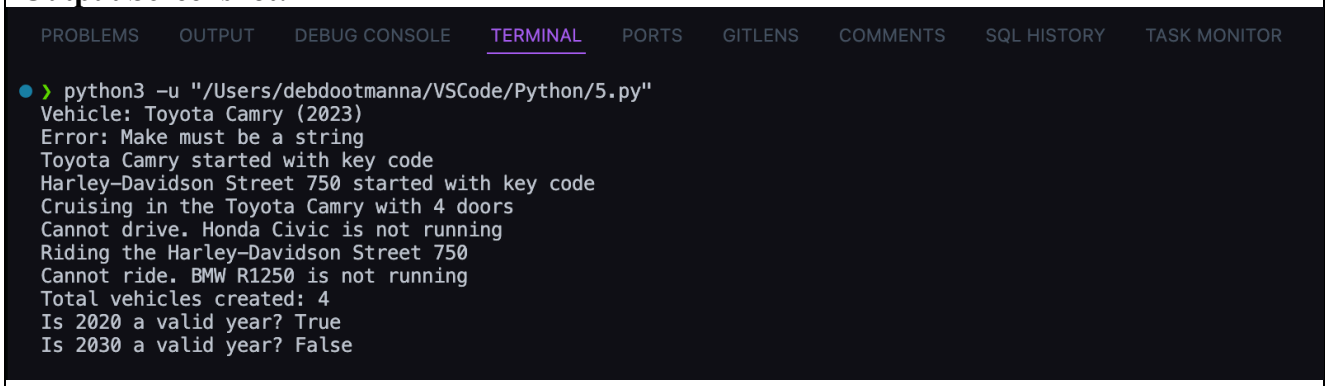
```
        if isinstance(value, str):
            self._make = value
        else:
            raise ValueError("Make must be a string")

    @model.setter
    def model(self, value):
        if isinstance(value, str):
            self._model = value
        else:
            raise ValueError("Model must be a string")

    # Method overloading (different parameters)
    def start(self):
        self._is_running = True
        return f"{self.make} {self.model} started"

    def start(self, key_code):
        if key_code == 1234:  # Simple validation
            self._is_running = True
            return f"{self.make} {self.model} started with key code"
        return f"Invalid key code for {self.make} {self.model}"

    def stop(self):
        self._is_running = False
        return f"{self.make} {self.model} stopped"

    # Method to be overridden by subclasses (polymorphism)
    def drive(self):
        if self._is_running:
            return f"Driving the {self.make} {self.model}"
        return f"Cannot drive. {self.make} {self.model} is not running"

    # Class method
    @classmethod
    def get_total_vehicles(cls):
        return f"Total vehicles created: {cls.total_vehicles}"

    # Static method
    @staticmethod
    def validate_year(year):
        current_year = 2025
        return 1900 <= year <= current_year
```

```
class Car(Vehicle):
    def __init__(self, make, model, year, doors=4):
        super().__init__(make, model, year)
        self._doors = doors

    # Property for the additional attribute
    @property
    def doors(self):
        return self._doors

    @doors.setter
    def doors(self, value):
        if isinstance(value, int) and value > 0:
            self._doors = value
        else:
            raise ValueError("Doors must be a positive integer")

    # Override the drive method (polymorphism)
    def drive(self):
        if self._is_running:
            return f"Cruising in the {self.make} {self.model} with
{self._doors} doors"
        return f"Cannot drive. {self.make} {self.model} is not running"


class Motorcycle(Vehicle):
    def __init__(self, make, model, year, has_sidecar=False):
        super().__init__(make, model, year)
        self._has_sidecar = has_sidecar

    @property
    def has_sidecar(self):
        return self._has_sidecar

    @has_sidecar.setter
    def has_sidecar(self, value):
        if isinstance(value, bool):
            self._has_sidecar = value
        else:
            raise ValueError("has_sidecar must be a boolean")

    # Override the drive method (polymorphism)
```

```
    def drive(self):
        if self._is_running:
            base = f"Riding the {self.make} {self.model}"
            if self._has_sidecar:
                return f"{base} with a sidecar"
            return base
        return f"Cannot ride. {self.make} {self.model} is not running"


# Usage example
if __name__ == "__main__":
    # Create different vehicle types (demonstrating polymorphism)
    sedan = Car("Toyota", "Camry", 2023)
    coupe = Car("Honda", "Civic", 2022, doors=2)
    bike = Motorcycle("Harley-Davidson", "Street 750", 2024)
    bike_with_sidecar = Motorcycle("BMW", "R1250", 2023,
has_sidecar=True)

    # Using encapsulated attributes through properties
    print(f"Vehicle: {sedan.make} {sedan.model} ({sedan.year})")

    # Testing invalid input with setters
    try:
        sedan.make = 123  # Should raise an error
    except ValueError as e:
        print(f"Error: {e}")

    # Starting vehicles
    print(sedan.start(1234))
    print(bike.start(1234))

    # Demonstrating polymorphism with drive method
    vehicles = [sedan, coupe, bike, bike_with_sidecar]
    for vehicle in vehicles:
        print(vehicle.drive())

    # Using class method
    print(Vehicle.get_total_vehicles())

    # Using static method
    valid_year = 2020
    invalid_year = 2030
    print(f"Is {valid_year} a valid year?
{Vehicle.validate_year(valid_year)}")
```

```
    print(f"Is {invalid_year} a valid year?
{Vehicle.validate_year(invalid_year)}")
```

Output Screenshot:



Conclusion/Summary:

This practical demonstrates key object-oriented programming concepts in Python, showcasing:

Encapsulation through private attributes (with leading underscore) and controlled access via getter/setter methods, protecting data integrity and providing a stable interface.

Polymorphism implemented in two ways:

Method overriding: Different vehicles provide specialized implementations of common methods
Interface consistency: All vehicle objects can be treated uniformly despite being different types
Method varieties:

Instance methods operating on object state
Class methods using the @classmethod decorator to access class-level data
Static methods using the @staticmethod decorator for utility functions
Property getters/setters using the @property decorator for elegant attribute access
Inheritance hierarchy with a base Vehicle class and specialized subclasses (Car and Motorcycle), promoting code reuse and extensibility.

Student Signature & Date	**Marks:**	**Evaluator Signature & Date**