Static Methods in Python:

In Python, static methods are methods that are bound to a class rather than an instance of the class. They don't require access to the instance or class attributes and can be called without creating an instance of the class. You define a static method using the `@staticmethod` decorator.

Here's an example:

```
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method")

# You can call the static method directly from the class
MyClass.static_method()
```

Static methods are often used for utility functions that don't depend on instance or class state but are related to the class conceptually.

For example, in a `MathUtils` class, you might have a static method to compute the square of a number:

```
class MathUtils:
    @staticmethod
    def square(x):
        return x * x

# Calling the static method
print(MathUtils.square(5))  # Output: 25
```

Remember, static methods don't have access to instance or class variables unless explicitly passed as arguments. They're essentially just like regular functions but scoped within the class namespace.

CODE:

```
class Car:
    # Class variable
    wheels = 4

    def __init__(self, make, model):
        self.make = make
        self.model = model

    # Instance method
    def display_info(self):
        print(f"Car: {self.make} {self.model}, Wheels: {self.wheels}")

    # Static method to check if a car is electric
    @staticmethod
    def is_electric(model):
        electric_models = ['Tesla', 'Nissan Leaf', 'Chevrolet Bolt']
        return model in electric_models


# Creating instances of Car
car1 = Car('Toyota', 'Corolla')
car2 = Car('Tesla', 'Model S')

# Calling instance method
car1.display_info()  # Output: Car: Toyota Corolla, Wheels: 4

# Calling static method
print(Car.is_electric('Tesla'))  # Output: True
print(Car.is_electric('Toyota'))  # Output: False
```

Output:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\34_1.py
Car: Toyota Corolla, Wheels: 4
True
False
```

Abstraction & Encapsulation:

Abstraction and encapsulation are two important concepts in object-oriented programming, including Python.

Abstraction: Abstraction is the process of hiding the complex implementation details and showing only the necessary features of an object to the outside world. It allows you to focus on what an object does rather than how it does it. In Python, abstraction can be achieved through abstract classes and interfaces, as well as through method signatures.

Here's a simple example demonstrating abstraction using Python's abstract base classes (ABCs) from the `abc` module:

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Creating objects
circle = Circle(5)
print("Area of circle:", circle.area())  # Output: Area of circle: 78.5
```

In this example, the `Shape` class defines an abstract method `area()`, which any concrete subclass must implement. The `Circle` class is a concrete implementation of `Shape`, providing its own implementation of the `area()` method.

Encapsulation: Encapsulation is the bundling of data (attributes) and methods that operate on that data into a single unit, often referred to as a class. It hides the internal state of an object from the outside world and only exposes the necessary functionalities through methods. Encapsulation helps in achieving data abstraction and prevents direct access to the data, thereby ensuring data integrity and security.

Here's a simple example demonstrating encapsulation in Python:

```python
class Car:
    def __init__(self, make, model):
        self.__make = make  # Encapsulated attribute
        self.__model = model  # Encapsulated attribute

    def display_info(self):
        print(f"Car: {self.__make} {self.__model}")

    def get_make(self):
        return self.__make

    def set_make(self, make):
        self.__make = make

# Creating object
my_car = Car('Toyota', 'Corolla')

# Accessing encapsulated attribute using getter method
print("Make of car:", my_car.get_make())  # Output: Make of car: Toyota

# Attempting to directly access encapsulated attribute
# This will result in an AttributeError because the attribute is private
# print(my_car.__make)

# Modifying encapsulated attribute using setter method
my_car.set_make('Honda')
my_car.display_info()  # Output: Car: Honda Corolla
```

In this example, __make and __model are encapsulated attributes of the Car class, which are accessed and modified through getter and setter methods (get_make() and set_make()). This encapsulation hides the internal state of the Car object and provides controlled access to its attributes.

CODE:

```
from abc import ABC, abstractmethod

# Abstract class defining a Vehicle
class Vehicle(ABC):
    def __init__(self, make, model):
        self.make = make
        self.model = model

    @abstractmethod
    def display_info(self):
        pass

# Concrete class Car inheriting from Vehicle
class Car(Vehicle):
    def __init__(self, make, model, color):
        super().__init__(make, model)
        self.color = color

    def display_info(self):
        print(f"Car: {self.make} {self.model}, Color: {self.color}")

# Concrete class Plane inheriting from Vehicle
class Plane(Vehicle):
    def __init__(self, make, model, max_speed):
        super().__init__(make, model)
        self.max_speed = max_speed

    def display_info(self):
        print(f"Plane: {self.make} {self.model}, Max Speed: {self.max_speed}
km/h")

# Encapsulation example
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.__account_number = account_number  # Encapsulated attribute
        self.__balance = balance  # Encapsulated attribute

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")
```

```
    def get_balance(self):
        return self.__balance

# Creating objects
car = Car("Toyota", "Corolla", "Red")
plane = Plane("Boeing", "747", 1000)
print("Vehicle Information:")
car.display_info()  # Output: Car: Toyota Corolla, Color: Red
plane.display_info()  # Output: Plane: Boeing 747, Max Speed: 1000 km/h

# Encapsulation usage
account = BankAccount("123456", 1000)
print("\nBank Account Information:")
print("Balance:", account.get_balance())  # Output: Balance: 1000
account.deposit(500)
print("Balance after deposit:", account.get_balance())  # Output: Balance
after deposit: 1500
account.withdraw(2000)  # Output: Insufficient funds
print("Balance after withdrawal:", account.get_balance())  # Output: Balance
after withdrawal: 1500
```

Output:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\34_2.py
Vehicle Information:
Car: Toyota Corolla, Color: Red
Plane: Boeing 747, Max Speed: 1000 km/h

Bank Account Information:
Balance: 1000
Balance after deposit: 1500
Insufficient funds
Balance after withdrawal: 1500
```