What is Encapsulation?

Encapsulation is a fundamental principle of object-oriented programming (OOP) that involves bundling data (attributes) and methods (functions) that operate on that data into a single unit, typically a class, while restricting direct access to some of the object's components. This is achieved by using access modifiers like private and protected attributes to hide the internal state of an object and only expose a controlled interface (public methods) to interact with it. Encapsulation promotes data security, modularity, and maintainability by preventing unauthorized access and modification of an object's internal data.

Key aspects of encapsulation:

- **Data Hiding**: Sensitive data is hidden from external access, reducing the risk of unintended interference or misuse.

- **Controlled Access**: Public methods (getters and setters) provide controlled access to private or protected data.

- **Abstraction**: The internal implementation is hidden, and only the necessary functionality is exposed to the user.

Code Description

The provided Python code demonstrates encapsulation using a `BankAccount` class to simulate a banking system. Below is a detailed description of the code and how it implements encapsulation:

Code Breakdown

```python
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        self._account_holder = account_holder  # Protected attribute
```

```
        self.__balance = initial_balance      # Private attribute

        self.__account_number = self.__generate_account_number()  # Private attribute
```

- **Class Definition**: The `BankAccount` class encapsulates the properties and behaviors of a bank account.

- **Constructor (`__init__`)**: Initializes a bank account with:

  - `_account_holder`: A protected attribute (single underscore) to store the account holder's name. Protected attributes are meant to be accessible within the class and its subclasses but discouraged for external access.

  - `__balance`: A private attribute (double underscore) to store the account balance, ensuring it cannot be accessed directly from outside the class.

  - `__account_number`: A private attribute generated by a private method, representing a unique account number.

- **Encapsulation**: The use of `_` and `__` prefixes enforces data hiding. The double underscore (`__`) triggers Python's name mangling, making the attribute inaccessible directly (e.g., `account.__balance` raises an error).

```python
    def __generate_account_number(self):

        # Private method to generate a unique account number

        import random

        return random.randint(10000000, 99999999)
```

- **Private Method**: `__generate_account_number` is a private method (double underscore) used internally to generate a random 8-digit account number. It cannot be called from outside the class, ensuring the account number generation logic is hidden and secure.

```python
    def deposit(self, amount):
        # Public method to deposit money
        if amount > 0:
            self.__balance += amount
            return f"Deposited ${amount:.2f}. New balance: ${self.__balance:.2f}"
        return "Invalid deposit amount"
```

- **Public Method (`deposit`)**: Allows users to deposit money into the account.
- **Validation**: Checks if the deposit amount is positive. If valid, it updates the private `__balance` attribute.
- **Encapsulation**: The method provides controlled access to modify `__balance`, ensuring that only valid operations affect the balance.

```python
    def withdraw(self, amount):
        # Public method to withdraw money
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return f"Withdrew ${amount:.2f}. New balance: ${self.__balance:.2f}"
        return "Invalid withdrawal amount or insufficient funds"
```

- **Public Method (`withdraw`)**: Allows users to withdraw money from the account.
- **Validation**: Ensures the withdrawal amount is positive and does not exceed the available `__balance`.

- **Encapsulation**: Like `deposit`, this method safely modifies `__balance` while enforcing business rules (e.g., preventing overdrafts).

```python
    def get_balance(self):
        # Public method to access private balance
        return f"Current balance: ${self.__balance:.2f}"
```

- **Getter Method (`get_balance`)**: A public method that provides read-only access to the private `__balance` attribute in a formatted way. This is a common encapsulation practice to allow controlled access to private data without exposing it directly.

```python
    def get_account_holder(self):
        # Public method to access protected account holder
        return self._account_holder
```

- **Getter Method (`get_account_holder`)**: Provides access to the protected `_account_holder` attribute. Since it’s protected (not private), it’s less restrictive but still part of the encapsulation strategy to control access.

```python
# Example usage
if __name__ == "__main__":
    # Create a new bank account
```

```
    account = BankAccount("John Doe", 1000)

    # Access public methods
    print(account.get_account_holder())  # Output: John Doe
    print(account.get_balance())        # Output: Current balance: $1000.00
    print(account.deposit(500))         # Output: Deposited $500.00. New balance: $1500.00
    print(account.withdraw(200))        # Output: Withdrew $200.00. New balance: $1300.00
    print(account.get_balance())        # Output: Current balance: $1300.00

    # Attempt to access private attribute directly (will raise AttributeError)
    # print(account.__balance)  # Error: 'BankAccount' object has no attribute '__balance'

    # Attempt to access protected attribute (discouraged but possible)
    print(account._account_holder)  # Output: John Doe
```

- **Usage Example**: Demonstrates how to create a `BankAccount` object and interact with it using public methods.
- **Encapsulation in Action**:
  - Accessing `__balance` directly (`account.__balance`) raises an `AttributeError` due to name mangling, enforcing data hiding.
  - Accessing `_account_holder` directly (`account._account_holder`) is possible but discouraged, as it's a protected attribute meant for internal or subclass use.
  - Public methods (`deposit`, `withdraw`, `get_balance`, `get_account_holder`) provide a controlled interface to interact with the object's data.

How Encapsulation is Achieved

1. **Private Attributes (`__balance`, `__account_number`)**:

   - These are hidden from external access using double underscores, which triggers Python's name mangling (e.g., `__balance` becomes `_BankAccount__balance` internally).

   - This prevents accidental or malicious modification of critical data like the account balance.

2. **Protected Attribute (`_account_holder`)**:

   - The single underscore indicates that this attribute is intended for internal use or by subclasses, but it's still accessible (though discouraged) from outside the class.

3. **Private Method (`__generate_account_number`)**:

   - The method is only accessible within the class, protecting the logic for generating account numbers from external interference.

4. **Public Interface**:

   - Methods like `deposit`, `withdraw`, `get_balance`, and `get_account_holder` provide controlled ways to interact with the private and protected attributes, ensuring that all operations follow defined rules (e.g., no negative deposits, no overdrafts).

5. **Data Security**:

   - By restricting direct access to `__balance` and `__account_number`, the class ensures that the account's state can only be modified through validated operations, maintaining data integrity.

Benefits of Encapsulation in This Example

- **Security**: Sensitive data like balance and account number are protected from unauthorized access.

- **Controlled Access**: Public methods enforce validation rules (e.g., preventing negative deposits or overdrafts).

- **Maintainability**: The internal implementation (e.g., how account numbers are generated) can change without affecting external code, as long as the public interface remains consistent.

- **Modularity**: The class encapsulates all banking-related logic in one place, making it easier to manage and extend.

This code is a clear example of encapsulation, demonstrating how to hide internal details and expose only the necessary functionality to ensure secure and reliable operation.