Encapsulation

Encapsulation means binding data (variables) and methods (functions) together into a single unit (class) and controlling access to that data.
It used to provide

1. Data hiding
2. Security
3. Controlled modification

Python provides three levels of data access control using **access modifiers:**

1. **Public members** → Accessible from anywhere.
2. **Protected members** (_var) → Accessible within the class and its subclasses (not enforced, but a convention).
3. **Private members** (__var) → Accessible only within the class.

```python
class Student:
    def __init__(self, name, age, roll):
        self.name = name           # Public
        self._age = age            # Protected
        self.__roll = roll         # Private

    def show(self):
        print("Name:", self.name)
        print("Age:", self._age)
        print("Roll:", self.__roll)


s = Student("Neeraj", 21, 101)

print(s.name)         # Public → Works
print(s._age)         # Protected → Works but not recommended
# print(s.__roll)     # Private → Error

s.show()              # Accessing private via method

# Name mangling trick
print(s._Student__roll)    # Works, but not recommended
```

Name Mangling in Python
Name Mangling is a mechanism in Python that changes the name of a class’s private variables or

methods so they cannot be accessed directly from outside the class.

If you define a variable with two leading underscores (__var) and no trailing underscores, Python automatically changes its name internally.

```python
# _ClassName__variableName

class Student:
    def __init__(self, name, roll):
        self.name = name          # Public
        self.__roll = roll        # Private

s = Student("Neeraj", 101)

print(s.name)
# print(s.__roll)      # Error

# Internal name after name mangling
print(s._Student__roll)
```

Getter and Setter in Python
Since private variables cannot be accessed directly, getter and setter methods are used to read and update them in a safe way insted of name manging.

Example 1: Normal Getter & Setter Methods

```python
class Student:
    def __init__(self, name):
        self.__name = name   # Private variable

    def get_name(self):       # Getter
        return self.__name

    def set_name(self, new_name):   # Setter
        if len(new_name) > 0:
            self.__name = new_name
        else:
            print("Name cannot be empty!")

s = Student("Neeraj")
print(s.get_name())     # Using getter
s.set_name("Amit")      # Using setter
print(s.get_name())
s.set_name("")          # Invalid input
```

Example 2: Using @property Decorator (Pythonic Way)

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    @property
    def balance(self):        # Getter
        return self.__balance

    @balance.setter
    def balance(self, amount):  # Setter
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid balance!")

acc = BankAccount(5000)
print(acc.balance)      # Looks like attribute → Getter
acc.balance = 10000     # Looks like assignment → Setter
print(acc.balance)
acc.balance = -2000     # Invalid value
```

Real-Life Example: Bank Account with Encapsulation

```python
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number   # Private
        self.__balance = balance                 # Private

    @property
    def balance(self):       # Getter
        return self.__balance

    @balance.setter
    def balance(self, amount):  # Setter
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid balance!")

    def display(self):
        print(f"Account: {self.__account_number}, Balance: {self.__balance}")
```

```python
acc = BankAccount("12345", 5000)

acc.display()
print(acc.balance)       # Accessing balance via getter

acc.balance = 10000      # Updating balance via setter
acc.display()

acc.balance = -2000      # Invalid update
```