1. Encapsulation:

Encapsulation is defined as bundling instance attributes (data) and methods into a single unit and control access on the attributes and methods.

For example:

When you are creating class, you are implementing encapsulation – because a class bundle data (attributes) and methods (functions) into a single unit.

If you also use access modifiers (i.e. private attributes with getter and setter methods), you achieve data hiding, which is a deeper form of encapsulation.

Encapsulation is used to hide an objects internal data from outside – termed as information hiding.

Also encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.

Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice

How can we Encapsulate attributes and methods?

Encapsulation can be achieved by declaring the data members and methods of a class either as private, public, protected – termed as access modifiers

In Python, we can achieved this by using single **underscore** and **double underscores**.

1. Public Member: Accessible anywhere from outside the class.
2. Private Member: Accessible within the class.
3. Protected Member: Accessible within the class and its sub-classes.

```
class Student:
    # contructor
    def __init__(self,name,grade, roll_no):
        # Instance Attributes
        self.name = name        # Public Member
        self._grade = grade     # Protected Member
        self.__roll_no = roll_no  # Private Member
```

- **Public Member:**

Public members are accessible within and outside the class. All members of the class are public by default.

```
class Student:
    # constructor
    def __init__(self, name, course):
        # Instance Attributes
        self.name = name          # Public Member
        self.course = course      # Public Member

    # Instance Method (Public)
    def show_info(self):
        #print(f"Name: {self.name}")
        #print(f"Course: {self.course}")
        return f"Name: {self.name}\nCourse: {self.course}"

# Creating Instance (Object) of a class
student = Student("Ali", "AI and Data Science")

# access Public attributes of class
print(f"Name: {student.name}\nCourse: {student.course}")

# calling Public Method of class
print(student.show_info())
```

- **Private Member:**

Public members are accessible within the class using instance methods. To define a private member we use two underscores as prefix at the start of a variable name

```
class Student:
    # constructor
    def __init__(self, name, course, roll_no):
        # Instance Attributes
        self.name = name        # Public Member
        self.course = course    # Public Member

        self.__roll_no = roll_no # Private Member

    # Instance Method (Public)
    def show_info(self):
        return f"Name: {self.name}\nCourse: {self.course}\nRoll No: {self.__roll

# Creating Instance (Object) of a class
student = Student("Ali", "AI and Data Science", 40554)

# calling Public Method of Class
print(student.show_info())

# access Public attributes of class
print("Name: ", student.name)
print("Course: ", student.course)

# Acceesing Private Member   (Attribute Error)
print("Roll No: ", student.__roll_no)
```

We can access private members from outside a class using two approaches

- Create public Instance method inside the class to access private attributes
- By Name Mangling

```
class Student:
    # constructor
    def __init__(self, name, course, roll_no):
        # Instance Attributes
        self.name = name         # Public Member
        self.course = course     # Public Member

        self.__roll_no = roll_no # Private Member

    # Instance Method (Public)
    def show_info(self):
        return f"Name: {self.name}\nCourse: {self.course}\nRoll No: {self.__roll

# Creating Instance (Object) of a class
student = Student("Ali", "AI and Data Science", 40554)


# calling Public Method of class
print(student.show_info())

# direct access to private member using Name Mangling
print("Roll no: ",student._Student__roll_no)
```

- **Protected Members:**

Protected members are accessible within the class and also from a sub-class. We use a single underscore _ before the name of class. (Inheritance)

```
class Student:
    # constructor
    def __init__(self, name, course, roll_no, grade):
        # Instance Attributes
        self.name = name        # Public Member
        self.course = course    # Public Member

        self.__roll_no = roll_no # Private Member

        self._grade = grade      # Protected Member

    # Instance Method (Public)
    def show_info(self):
        return f"Name: {self.name}\nCourse: {self.course}\nRoll No: {self.__rol

# Creating Instance (Object) of a class
student = Student("Ali", "AI and Data Science", 40554, "A")

# calling Public Method of Class
print(student.show_info())

# access Public attributes of class
print("Name: ", student.name)
print("Course: ", student.course)

# Acceesing Private Member   (Attribute Error)
#print("Roll No: ", student.__roll_no)

# direct access to private member using Name Mangling
print("Roll no: ",student._Student__roll_no)

# access Protected Member of class
# (Not recommended, as it is used in inheritance)
print("Grade: ", student._grade)
```

- **Getters and Setters Methods:**

To implement encapsulation, we use getters and setters methods. Since Private data is not accessible from outside the class, they can only be accessible inside the class by Instance Methods. Getter method is used to access data, while Setter method is used to modify the data members.