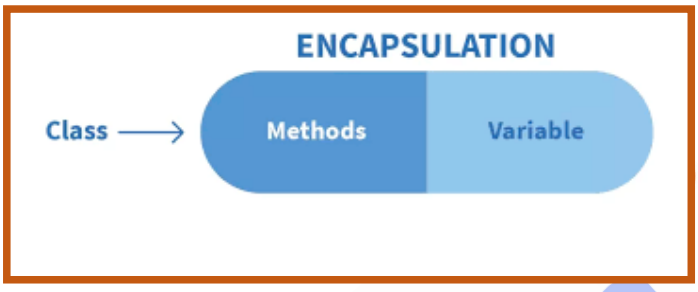
Encapsulation in Python

Encapsulation in Python

Encapsulation in Python is an Object-Oriented Programming (OOP) concept that restricts direct access to the internal data of a class and bundles data and methods together. It helps in protecting the data and controlling how it is accessed or modified.



Key Points

- Encapsulation hides internal details of a class from the outside world.
- Achieved using access specifiers:
- Public – Accessible from anywhere.
- Protected (prefix _) – Accessible within the class and its subclasses.
- Private (prefix __) – Accessible only within the class.
- Improves data security and maintainability.

Types of Access Modifiers in Python

Modifier	Example	Access Level
Public	self.name	Accessible from inside and outside the class
Protected (convention)	self._name	Accessible inside class and subclasses, but not recommended outside
Private	self.__name	Accessible only inside the class; name is mangled

Why do we use Encapsulation?

- Data Hiding – Protects internal class variables from accidental or unauthorized access.
- Controlled Access – Allows data to be accessed or modified only through getter/setter methods.
- Improved Security – Keeps sensitive information safe from outside interference.
- Better Maintainability – Changes in internal implementation do not affect external code using the class.

Encapsulation in Python

Type 1: Public Encapsulation

- Variables and methods are accessible from anywhere.
- No restriction on access.
- Syntax: Just define normally.

```
class Student:
    def __init__(self, name, age):
        self.name = name  # public
        self.age = age    # public

s = Student("Alice", 18)
print(s.name)  # Alice
s.age = 20     # Modify directly
print(s.age)   # 20
```

Brief explanation:

- name and age are **public attributes** — they can be **accessed and changed directly** from outside the class.
- s = Student("Alice", 18) → creates an object with name "Alice" and age 18.
- print(s.name) → prints "Alice".
- s.age = 20 → changes age directly to 20.
- print(s.age) → prints 20.

In short: Public attributes are fully accessible and modifiable from outside the class.

Type 2: Protected Attributes

- Indicated by a single underscore _ before the name.
- Meant to be accessed only within the class and its subclasses (not strictly enforced by Python, but by convention).

```
class Student:
    def __init__(self, name, age):
        self._name = name  # protected
        self._age = age    # protected

s = Student("Bob", 19)
print(s._name)  # Bob (accessible but discouraged)
```

Brief Explanation:

- _name and _age are **protected attributes** (indicated by a single underscore _).
- They **can be accessed outside the class**, but it's **not recommended** — meant to be used **only within the class or subclasses.**

Encapsulation in Python

• print(s._name) works and shows "Bob", but it's **against good practice** to access protected data directly.

Type 3: Private Attributes

• Indicated by a double underscore __ before the name.
• These are name-mangled by Python (internally changed to _ClassName__memberName).
• Cannot be accessed directly outside the class.

```
class Student:
    def __init__(self, name, age):
        self.__name = name  # private
        self.__age = age    # private

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

    # Setter method
    def set_name(self, name):
        self.__name = name

s = Student("Charlie", 20)
# print(s.__name)  # Error: private attribute cannot be accessed directly
print(s.get_name())  # Charlie

s.set_name("David")  # Change value using setter
print(s.get_name())  # David
```

Brief Explanation:

• __name and __age are **private attributes** — they **cannot be accessed directly** from outside the class.
• To access or modify them, we use **getter** (get_name()) and **setter** (set_name()) methods.
• print(s.get_name()) → shows "Charlie".
• s.set_name("David") → changes the name safely.
• print(s.get_name()) → now shows "David".

In short: Private attributes are hidden — we use getters and setters to access or update them securely.