# **🔐 Python OOP - Encapsulation Guide**

## **✅ What is Encapsulation?**

**Encapsulation** is the concept of **bundling data (variables)** and **methods (functions)** into a **single unit (class)** and **restricting direct access** to internal details.

Encapsulation allows:

* Hiding internal state and logic
* Protecting data from unauthorized access
* Exposing only what is necessary using public methods

## **🔑 Real-Life Analogy**

Think of a **Zomato app order**:

* Customer sees total bill, item list
* But cannot see internal calculations or discounts unless they’re authorized (admin)
* Internal logic is hidden and only exposed as needed

That’s **encapsulation**.

## **🧱 Access Modifiers in Python**

| **Modifier** | **Syntax** | **Meaning** |
| --- | --- | --- |
| Public | self.name | Accessible everywhere |
| Protected | self.\_name | Meant for subclass/internal access |
| Private | self.\_\_name | Name mangled; internal use only |

## **🧪 Code Example: Multi-Class Encapsulation (Zomato Style)**

class Order:

def \_\_init\_\_(self, customer\_name, items, total\_amount, discount):

self.customer\_name = customer\_name # public

self.items = items # public

self.\_\_total\_amount = total\_amount # private

self.\_\_discount = discount # private

def \_\_calculate\_final(self): # private helper

return self.\_\_total\_amount - self.\_\_discount

def \_get\_admin\_view(self): # protected method

return {

"Customer": self.customer\_name,

"Items": self.items,

"Total Amount": f"₹{self.\_\_total\_amount}",

"Discount Applied": f"₹{self.\_\_discount}",

"Final Bill": f"₹{self.\_\_calculate\_final()}"

}

def get\_customer\_view(self): # public method

return {

"Customer": self.customer\_name,

"Items": self.items,

"Final Bill": f"₹{self.\_\_calculate\_final()}"

}

class AdminPortal:

def show\_order(self, order):

return order.\_get\_admin\_view() # accessing protected method

class CustomerApp:

def show\_order(self, order):

return order.get\_customer\_view()

## **🧪 Usage Demo**

order = Order("Gowtham", ["Pizza", "Pepsi"], 600, 150)

admin = AdminPortal()

customer = CustomerApp()

print(admin.show\_order(order)) # Shows full breakdown

print(customer.show\_order(order)) # Shows final bill only

### **🚫 Cannot Access Private Method/Variable Directly**

print(order.\_\_calculate\_final()) # ❌ AttributeError

print(order.\_\_discount) # ❌ AttributeError

## **🔐 Name Mangling in Python**

### **🔧 What Is It?**

Python internally **renames** private variables and methods to \_ClassName\_\_var format. This avoids accidental access or override in subclasses.

### **🧪 Example:**

class Demo:

def \_\_init\_\_(self):

self.\_\_secret = "hidden"

def \_\_private\_method(self):

return "You can't see me"

obj = Demo()

# Direct access fails

# print(obj.\_\_secret) ❌

# print(obj.\_\_private\_method()) ❌

# Access via name mangling (not recommended)

print(obj.\_Demo\_\_secret) # ✅ "hidden"

print(obj.\_Demo\_\_private\_method()) # ✅ "You can't see me"

⚠️ This is not real security — it’s just a convention + internal name change.

## **🎯 Summary Points**

* Use \_\_var to hide internal data
* Provide safe access using getters or public methods
* Prevent misuse of internal logic by exposing only required interfaces
* Use protected methods (\_method) for subclasses or controlled access

## **❓ Interview Questions**

1. What is encapsulation?
2. How do you implement encapsulation in Python?
3. What is name mangling?
4. Can you access private variables from outside?
5. What’s the difference between \_protected and \_\_private?
6. Real-world use cases of encapsulation?

## **🧳 Resume Tip**

**Project Line Example:**

"Designed encapsulated class models for order processing with private financial data and controlled access views using role-based method exposure for admin vs customer."