Gowtham SB
www.linkedin.com/in/sbgowtham/ Instagram - @dataengineeringtamil

🔐 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."
