Table of Contents

Encapsulation is one of the core concepts of **Object-Oriented Programming (OOP)**. It simply means **hiding the internal details of an object and only showing necessary parts to the outside world.** This helps keep data safe and makes code easier to maintain.

🧠 What is Encapsulation?

> **Encapsulation** is the practice of **binding data (variables)** and **methods (functions)** that operate on the data into a single unit (a class), and **restricting direct access** to some of the object's components.

🔧 Why Use Encapsulation?

- To **protect data** from unwanted changes
- To **control access** using getters and setters
- To make your code **modular, reusable, and easy to debug**

📦 Example in Real Life:

Imagine a **bank ATM machine**.

- You insert a card and enter a PIN to withdraw money.
- You **can't see** the inner workings of the machine.
- You **interact only with buttons and screen** (public interface).

🔁 This is encapsulation: hiding the complex internal logic and only exposing the necessary parts.

Example 1: Basic Encapsulation

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # private attribute

    # Getter method
    def get_balance(self):
        return self.__balance
```

```
    # Setter method with validation
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient balance or invalid amount")

# Using the class
account = BankAccount("Alice", 1000)

# Try to access private attribute directly (won't work as intended)
# print(account.__balance)  # ❌ AttributeError

# Can't access __balance directly
print(account.get_balance())  # ✅ 1000


account.deposit(500)
print(account.get_balance())  # ✅ 1500

account.withdraw(2000)  # ❌ Insufficient balance
```

🧱 How It Works:

- `__balance` is a **private variable** (name mangling in Python makes it hard to access directly).
- You access it **only through methods** like `get_balance()`, `deposit()`, and `withdraw()`.

See also:

- Understanding Name Mangling in Python - Syntax and Purpose

🔐 Access Levels in Python:

Python doesn't have strict private/public keywords, but we use **naming conventions**:

Type	Syntax	Access
Public	`self.name`	Accessible everywhere
Protected	`_name`	Suggests limited access
Private	`__name`	Not directly accessible

See also:

- What Does Protected Access Level Mean in Python?

Example 2: Real-World Usage - Temperature Converter

```
class TemperatureConverter:
    def __init__(self):
        self.__celsius = 0  # private attribute

    def set_celsius(self, temp):
        if temp < -273.15:  # Absolute zero check
            print("Temperature below absolute zero is not possible")
        else:
            self.__celsius = temp

    def get_celsius(self):
        return self.__celsius

    def get_fahrenheit(self):
        return (self.__celsius * 9/5) + 32

    def get_kelvin(self):
        return self.__celsius + 273.15

# Usage
thermo = TemperatureConverter()
thermo.set_celsius(25)

print(f"Celsius: {thermo.get_celsius()}")      # 25
print(f"Fahrenheit: {thermo.get_fahrenheit()}")  # 77.0
print(f"Kelvin: {thermo.get_kelvin()}")       # 298.15

# Invalid temperature is prevented
thermo.set_celsius(-300)  # Shows warning message
```

🎯 Real-World Usage Benefits

1. **Banking Systems**: Protect account balances while allowing deposits/withdrawals
2. **User Authentication**: Hide password storage details while providing login methods
3. **Game Development**: Control character attributes like health points
4. **E-commerce**: Manage product inventory with validation rules

---

✅ Tips

- Remember, in Python, encapsulation is more about convention than **enforcement**. The single and double underscore prefixes are signals to other programmers about how attributes should be used, not strict access controls.

✅ Summary

- Encapsulation is about **hiding internal data** and exposing only what's needed.
- Use **methods** to control access to internal data.

- It helps keep your code **clean, secure, and maintainable**.

📘 Related Topics

- **Classes and Objects** – Learn how to create and use classes in Python with practical examples. 👉 Learn more
- **Inheritance in Python** – Learn Python OOP inheritance with Beginner's examples! Understand parent & child classes, method overriding, `super()`, and multilevel inheritance. 👉 Learn more
- **Polymorphism** – Learn polymorphism in Python OOP with practical examples. Understand method overriding, duck typing, and operator overloading for flexible and reusable code. 👉 Learn more
- **Abstraction and Interfaces** – Learn about abstraction in python 👉 Learn more
- **Protected Access Level** – Learn about Protected Access Level in python 👉 Learn more