4 Pillars of OOP

Encapsulation

That encapsulate

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number  # Private attribute
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount ⩽ self.__balance:
            self.__balance -= amount

    def get_balance(self):
        return self.__balance

# Usage
account = BankAccount('123456', 1000)
account.deposit(500)
account.withdraw(200)
print(account.get_balance())  # Output: 1300
```

Inheritance

a new class (child class) is derived from an existing class (parent class)

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start_engine(self):
        print("Engine started")

class SportsCar(Car):
    def __init__(self, make, model, turbo):
        super().__init__(make, model)
        self.turbo = turbo

    def activate_turbo(self):
        print("Turbo activated")

# Usage
ferrari = SportsCar('Ferrari', '488', True)
ferrari.start_engine()  # Output: Engine started
ferrari.activate_turbo()  # Output: Turbo activated
```

Abstraction

Abstraction means hiding the complex implementation details and showing only the necessary features of an object

```
from abc import ABC, abstractmethod

class Notification(ABC):
    @abstractmethod
    def format_message(self, message):
        """Format the message for the specific notification type."""
        pass

    @abstractmethod
    def send(self, message):
        """Send the formatted message."""
        pass
```

```
class EmailNotification(Notification):
    def format_message(self, message):
        return f"Subject: Notification\n\n{message}"

    def send(self, message):
        formatted_message = self.format_message(message)
        print(f"Sending Email: {formatted_message}")
```

```
class SMSNotification(Notification):
    def format_message(self, message):
        return f"SMS: {message}"

    def send(self, message):
        formatted_message = self.format_message(message)
        print(f"Sending SMS: {formatted_message}")
```

```
class PushNotification(Notification):
    def format_message(self, message):
        return f"Push Notification: {message}"

    def send(self, message):
        formatted_message = self.format_message(message)
        print(f"Sending Push Notification: {formatted_message}")
```

```
# Usage example
def notify_user(notification: Notification, message):
    notification.send(message)

# Create instances of different notification types
email_notification = EmailNotification()
sms_notification = SMSNotification()
push_notification = PushNotification()

# Send notifications using the common interface
notify_user(email_notification, "You have a new email!")
notify_user(sms_notification, "You have a new SMS!")
notify_user(push_notification, "You have a new push notification!")
```

Polymorphism

Same same , But Different !

```
class Employee:
    def calculate_salary(self):
        pass

class FullTimeEmployee(Employee):
    def calculate_salary(self):
        return 4000

class PartTimeEmployee(Employee):
    def calculate_salary(self):
        return 2000

def print_salary(employee: Employee):
    print(f"Salary: {employee.calculate_salary()}")

# Usage
full_time = FullTimeEmployee()
part_time = PartTimeEmployee()

print(full_time.calculate_salary())
print(part_time.calculate_salary())
```