Day 37

Inheritance in Python:

What is Inheritance?

Inheritance allows one class (child / subclass) to reuse and extend another class (parent / base class).

Think of it like:

- Parent class: common features
- Child class: uses those features + adds its own

Why use inheritance?

- Code reusability
- Avoid duplication
- Easy maintenance
- Logical relationship between classes

Basic inheritance syntax

```
class Parent:
    def show(self):
        print("This is the parent class")
class Child(Parent):
    pass
```

Example: a basic example without any inheritance.

```
1   class Employee:
2       def __init__(self,name,id):
3           self.name = name
4           self.id = id
5       #method
6       def showDetails(self):
7           print(f"Name is {self.name} whose id is {self.id} ")
8   #object
9   e = Employee("Aditya", 111)
10  e.showDetails()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS E:\Python\Day31-40\Day37> python main.py
Name is Aditya whose id is 111
```

Example: introducing the inheritance, where the programmer class inherits the properties of the employee class.

```python
class Employee:
    def __init__(self,name,id):
        self.name = name
        self.id = id
    #method
    def showDetails(self):
        print(f"Name is {self.name} whose id is {self.id} ")
class Programmer(Employee):
    def showLang(self):
        print("Python is the lang")
#object for employee
e = Employee("Aditya", 111)
e.showDetails()
#object for programmer
f = Programmer("Utsav", 112)
f.showDetails()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day31-40\Day37> python main.py
Name is Aditya whose id is 111
Name is Utsav whose id is 112
```

Example:

```python
class Employee:
    def __init__(self,name,id):
        self.name = name
        self.id = id
    #method
    def showDetails(self):
        print(f"Name is {self.name} whose id is {self.id} ")
class Programmer(Employee):
    def showLang(self):
        print("Python is the lang")
#object for employee
e = Employee("Aditya", 111)
e.showDetails()
#object for programmer
f = Programmer("Utsav", 112)
f.showDetails()
f.showLang()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day31-40\Day37> python main.py
Name is Aditya whose id is 111
Name is Utsav whose id is 112
Python is the lang
```

Access Modifiers in Python:

What are Access Modifiers?

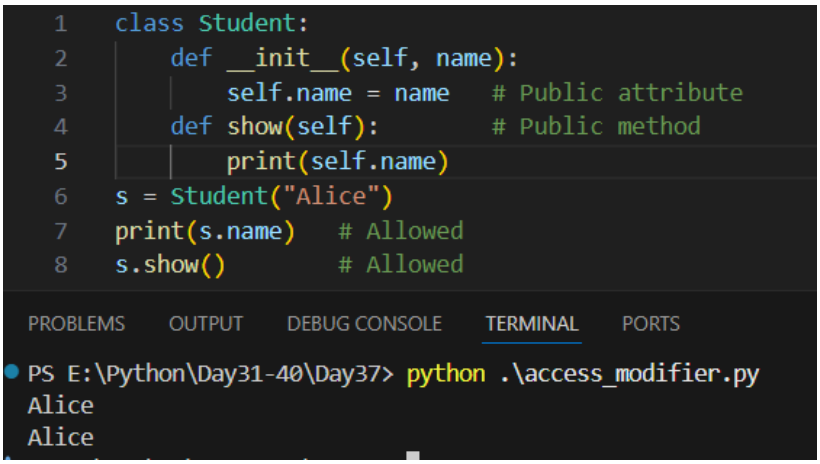
Access modifiers control who can access variables and methods of a class.

Python has three types (by convention):

Modifier	Syntax	Meaning
Public	name	Accessible everywhere
Protected	_name	Accessible within class & subclasses
Private	__name	Accessible only inside the class

Note: Python does not enforce access modifiers like Java or C++. Instead, it uses naming conventions to indicate access level.

Example: Public Access Modifier - Public members are accessible everywhere



Protected Access Modifier (_): Single underscore _name

- Indicates internal use
- Accessible in subclasses
- Still accessible outside (not enforced)

Example:

```python
class Employee:
    def __init__(self, salary):
        self._salary = salary   # Protected attribu
class Manager(Employee):
    def show_salary(self):
        print(self._salary)    # Allowed
m = Manager(50000)
m.show_salary()
print(m._salary)  # Allowed (but discouraged)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day31-40\Day37> python .\access_modifier.py
50000
50000
```

Private Access Modifier (__): Double underscore __name

- Triggers name mangling
- Prevents accidental access
- Strongest form of encapsulation

Exampel:

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance   # Private attribute
    def show_balance(self):
        print(self.__balance)
acc = BankAccount(1000)
acc.show_balance()   # Allowed
# print(acc.__balance)  ❌ AttributeError
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\Python\Day31-40\Day37> python .\access_modifier.py
1000
```

Name Mangling (Important Concept)

Python internally renames:

__balance → _ClassName__balance

So this works (but NOT recommended):

print(acc._BankAccount__balance) # 1000

This exists to avoid accidental access, not to provide true privacy.

Key Takeaways:

- Python access modifiers are conventions, not strict rules.
- _ = protected (internal use)
- __ = private (name mangling)
- Use getters/setters with @property for controlled access.

--The End--