Python
for everybody

8. Object Oriented Programming

8.4. Class vs Instance Attributes

Class vs Instance Attributes

A. Class Attributes

- Defined inside class, but outside methods.
- Shared by **all** instances.
- Changes affect everyone (if mutable).

B. Instance Attributes

- Defined inside **`__init__`**.
- Unique to **each** object.
- Changes do not affect other objects.

```python
class Employee:
    company = "TechCorp"        # Class Attribute (Shared)

    def __init__(self, name):
        self.name = name        # Instance Attribute (Unique)

e1 = Employee("Ahmed")
e2 = Employee("Sara")
```

8. Object Oriented Programming

8.5. Methods Types

Instance Methods vs Static Methods

1. Instance Methods: Take **`self`** as the first parameter. They access instance data.

2. Static Methods: Do not take **`self`**. They work like regular functions but belong to the class's namespace. Used for utility tasks.

```python
class MathHelper:

    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def is_even(n):
        return n % 2 == 0

print(MathHelper.add(5, 10))   # 15
print(MathHelper.is_even(8))   # True
```

9. The 4 Pillars of OOP

9. The 4 Pillars of OOP

9.1. Inheritance

Inheritance

Definition

Inheritance is a fundamental concept where a class (called a **Child** or Derived class) inherits attributes and methods from another class (called a **Parent** or Base class).

Why do we need it?

- **Code Reusability:** Write once, use many times.
- **Hierarchies:** Models real-world relationships (e.g., Animal $\rightarrow$ Dog).
- **Maintenance:** Update the parent, and all children are updated.

Implementation in Python

```python
# Parent Class
class Person:
    pass

# Child Class inherits from Person
class Student(Person):
    pass

# Another Child Class
class Teacher(Person):
    pass

s = Student()
# s is both a Student and a Person
```

Multilevel Inheritance

Inheritance can go multiple levels deep.

```
class Person:
    def breathe(self):
        print("Breathing...")

class Employee(Person):
    def work(self):
        print("Working...")

class Manager(Employee):
    def manage(self):
        print("Managing the team...")

m = Manager()
m.breathe()  # From Person (Grandparent)
m.work()     # From Employee (Parent)
m.manage()   # From Manager (Self)
```

9. The 4 Pillars of OOP

9.2. Encapsulation

2. Encapsulation: Definition

Definition

Encapsulation means hiding the internal details of a class and only exposing what is necessary.

Why use it?

- Protects data from unauthorized access (Security).
- Prevents accidental modification.
- Controls how data is accessed using **Getters** and **Setters**.

Encapsulation Example

We use double underscore `__` to make an attribute private.

```python
class BankAccount:
    def __init__(self):
        self.__balance = 0   # Private attribute

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

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

account = BankAccount()
account.deposit(100)
# print(account.__balance)  # Error! Cannot access directly
print(account.get_balance()) # Success: 100
```

9. The 4 Pillars of OOP

9.3. Abstraction

Abstraction

Definition

Abstraction hides complex implementation details and shows only the essential features of an object to the user.

The Concept: The user knows *what* an object does, not *how* it does it.

Example (Process Abstraction):

```
1 my_list.append(5)
```

Internally, Python performs memory allocation and resizing, but you only see append.

Abstract Classes

An **Abstract Class** is a blueprint that enforces subclasses to implement specific methods. We use the **abc** module.

```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def move(self):
        pass

class Car(Vehicle):
    def move(self):
        print("Driving on road")

class Boat(Vehicle):
    def move(self):
        print("Sailing on water")
```

If you try to create a Vehicle() directly, Python raises an error.

9. The 4 Pillars of OOP

9.4. Polymorphism

Polymorphism

Definition

Polymorphism means "many forms". It allows the same method, function, or operator to behave differently depending on the object.

A. Compile-Time (Method Overloading)

Python does not support true method overloading (same name, different arguments), but we can simulate it using **default parameters**.

```
class Welcome:
    @staticmethod
    def say_hi(name="Mr. X"):
        print(f"Hello {name}")

Welcome.say_hi()          # Output: Hello Mr. X
Welcome.say_hi("Ahmed")   # Output: Hello Ahmed
```

B. Runtime (Method Overriding)

Child classes can provide a specific implementation of a method already defined in the parent class.

```
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

animals = [Dog(), Cat(), Animal()]

for a in animals:
    print(a.sound())  # Bark, Meow, Some sound
```

C. Operator Overloading

The same operator (e.g., **+**) behaves differently based on type.

```
print(5 + 10)               # Integer Addition
print("Hello " + "World")   # String Concatenation
```

Customizing the '+' operator:

```
class Vector:
    def __init__(self, x):
        self.x = x

    def __add__(self, other):
        return Vector(self.x + other.x)

v1 = Vector(10)
v2 = Vector(20)
v3 = v1 + v2   # Python calls v1.__add__(v2)
print(v3.x)    # Output: 30
```

10. References

References & Resources

- **Book:** Reuven M. Lerner, *Python Workout (2nd edition)*.
- **Book:** Charles Severance, *Python for Everybody: Exploring Data in Python 3*.
- **Website:** Python for Everybody (PY4E) Lessons
- **Website:** Geeks for Geeks Python Tutorial

Thank You!