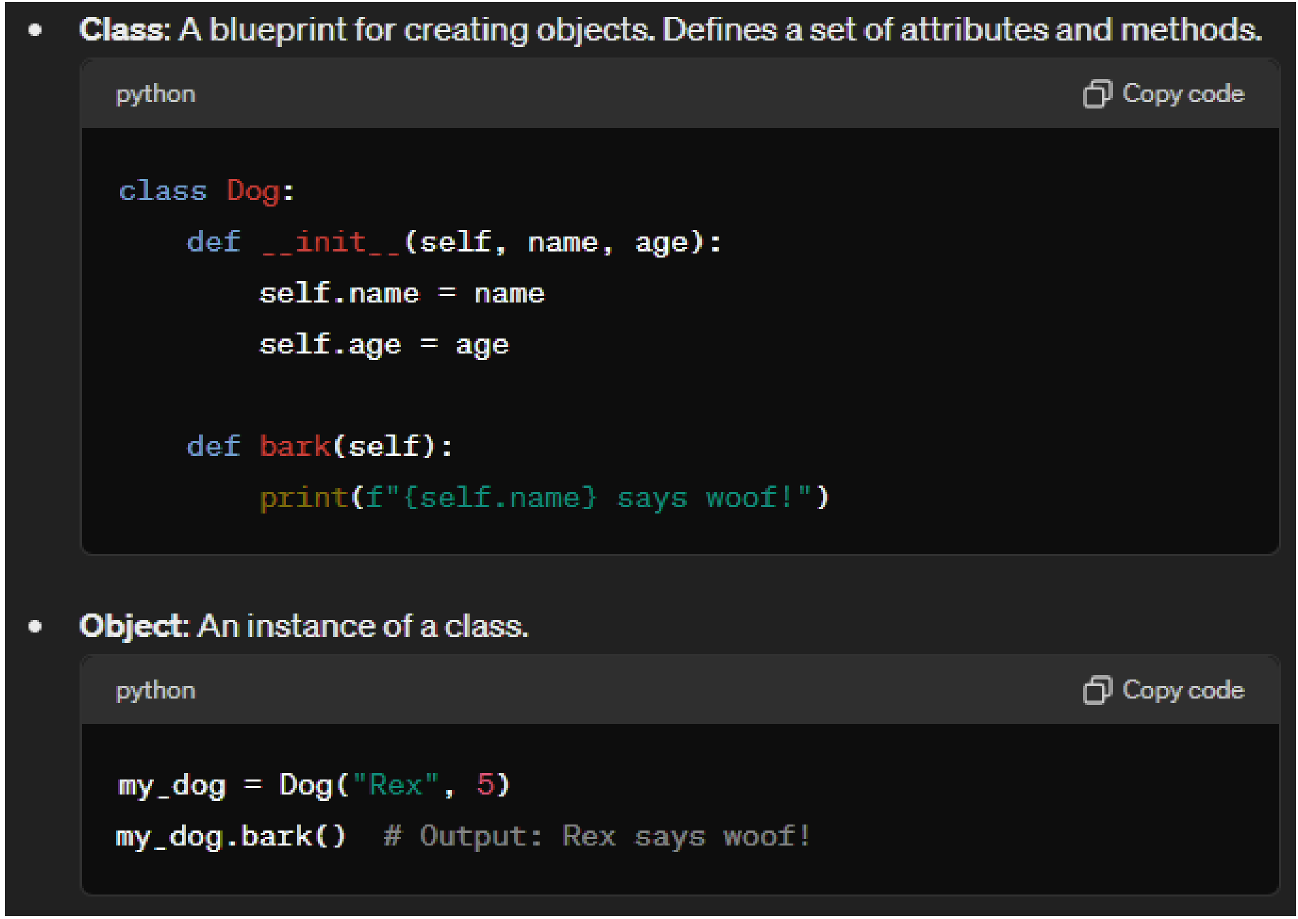
# Introduction to OOP

- **Definition**: Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which can contain data and code to manipulate that data.
- **Core Concepts**:
- Classes and Objects
- Inheritance
- Encapsulation
- Polymorphism
- **Benefits**:
- Modularity
- Reusability
- Scalability

# Classes and Objects



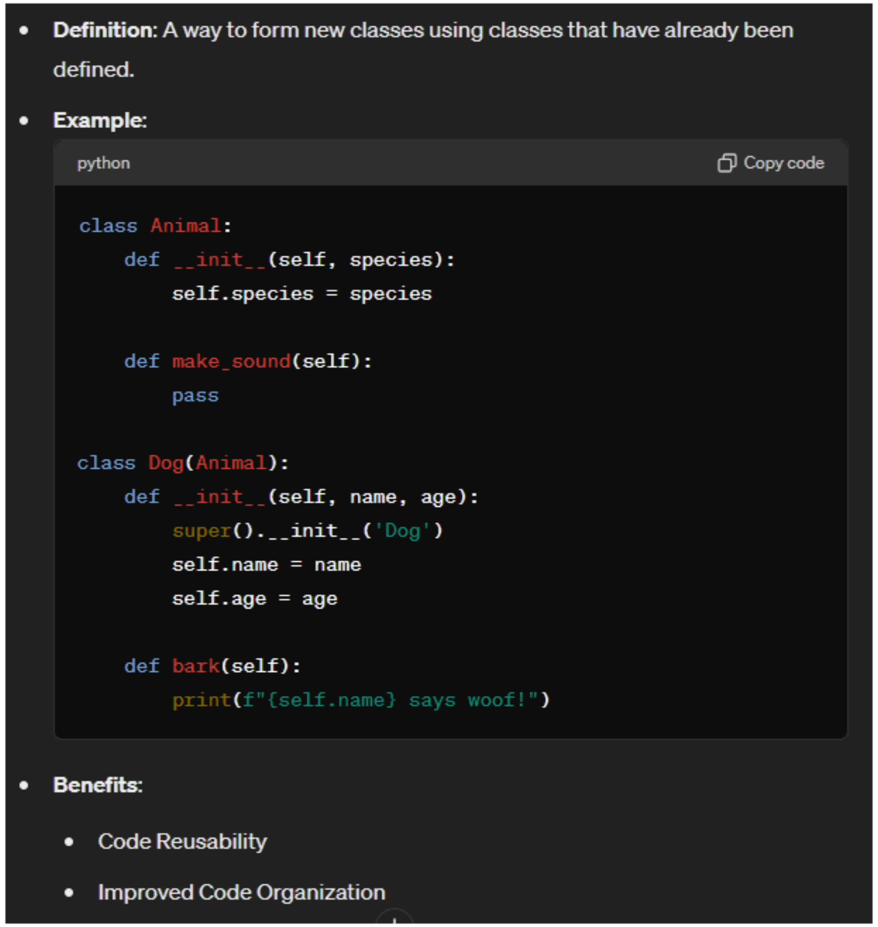
- **Class**: A blueprint for creating objects. Defines a set of attributes and methods.

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} says woof!")
```

- **Object**: An instance of a class.

```python
my_dog = Dog("Rex", 5)
my_dog.bark()  # Output: Rex says woof!
```

# Inheritance

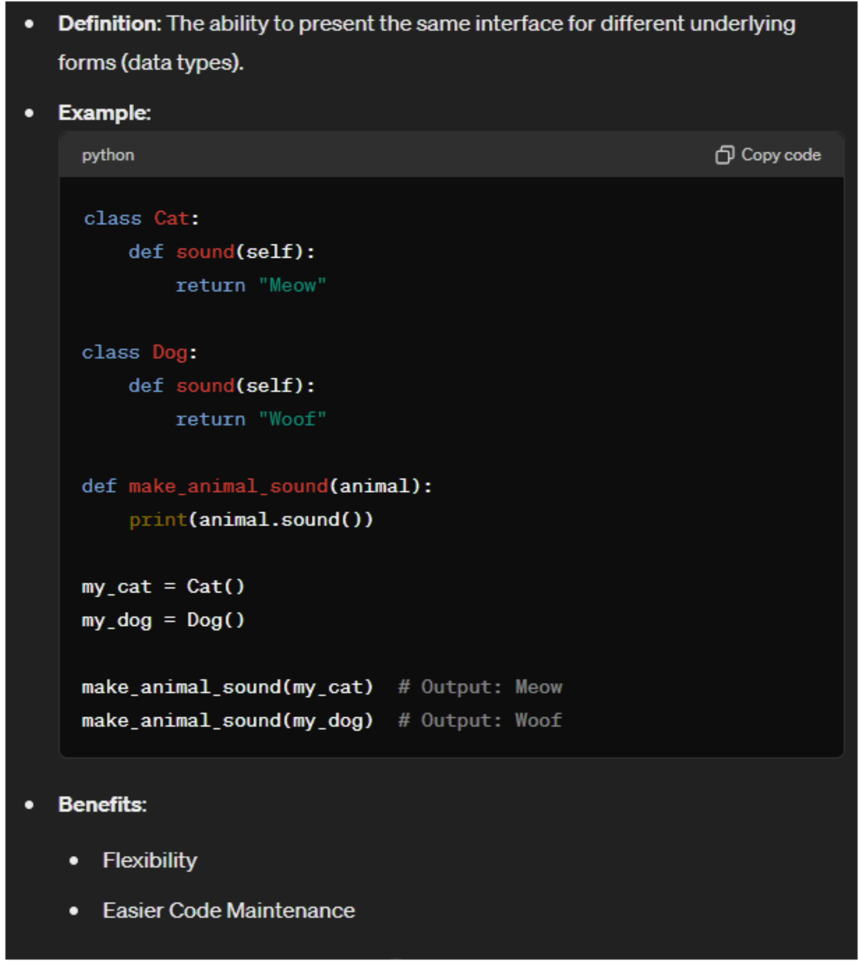


- **Definition:** A way to form new classes using classes that have already been defined.
- **Example:**

```python
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        pass

class Dog(Animal):
    def __init__(self, name, age):
        super().__init__('Dog')
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} says woof!")
```

- **Benefits:**
  - Code Reusability
  - Improved Code Organization

# Polymorphism

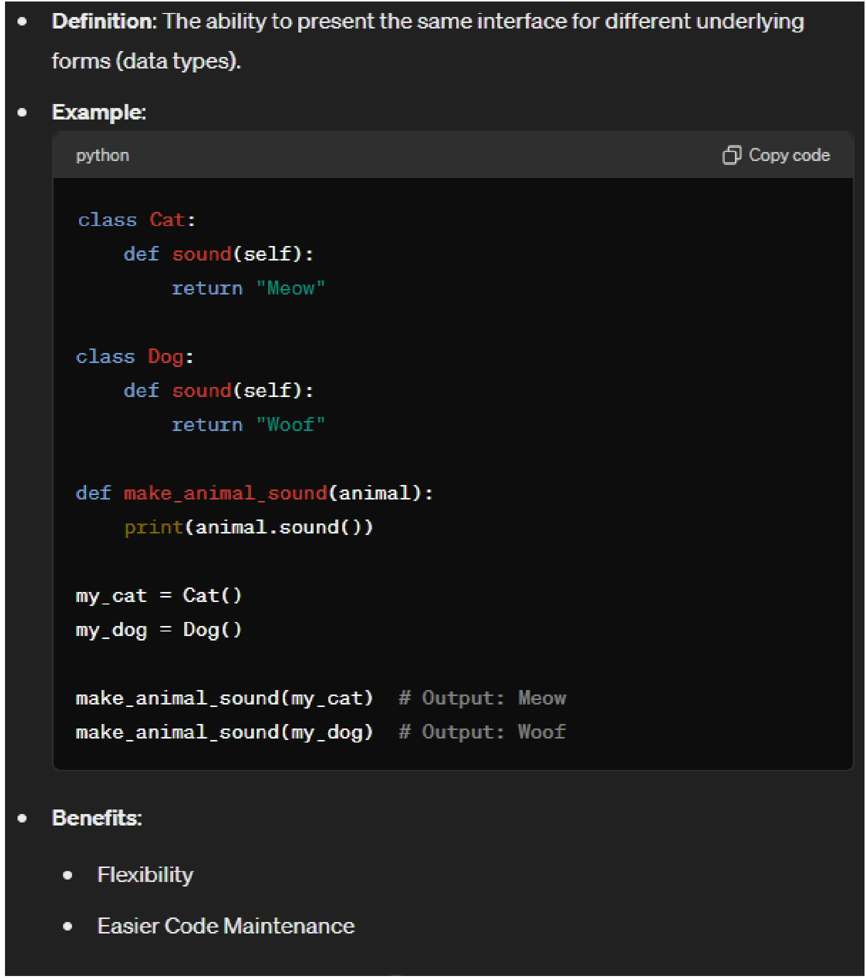


- **Definition:** The ability to present the same interface for different underlying forms (data types).
- **Example:**

```python
class Cat:
    def sound(self):
        return "Meow"

class Dog:
    def sound(self):
        return "Woof"

def make_animal_sound(animal):
    print(animal.sound())

my_cat = Cat()
my_dog = Dog()

make_animal_sound(my_cat)  # Output: Meow
make_animal_sound(my_dog)  # Output: Woof
```

- **Benefits:**
  - Flexibility
  - Easier Code Maintenance

# Polymorphism

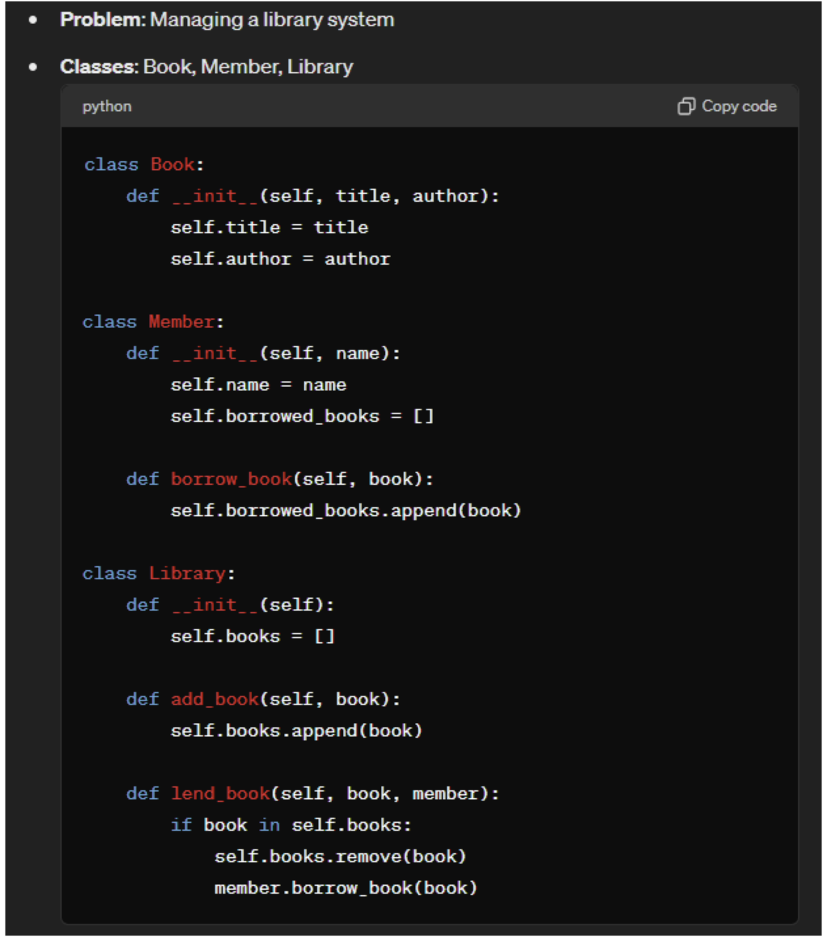


- **Definition:** The ability to present the same interface for different underlying forms (data types).
- **Example:**

```python
class Cat:
    def sound(self):
        return "Meow"

class Dog:
    def sound(self):
        return "Woof"

def make_animal_sound(animal):
    print(animal.sound())

my_cat = Cat()
my_dog = Dog()

make_animal_sound(my_cat)  # Output: Meow
make_animal_sound(my_dog)  # Output: Woof
```

- **Benefits:**
  - Flexibility
  - Easier Code Maintenance

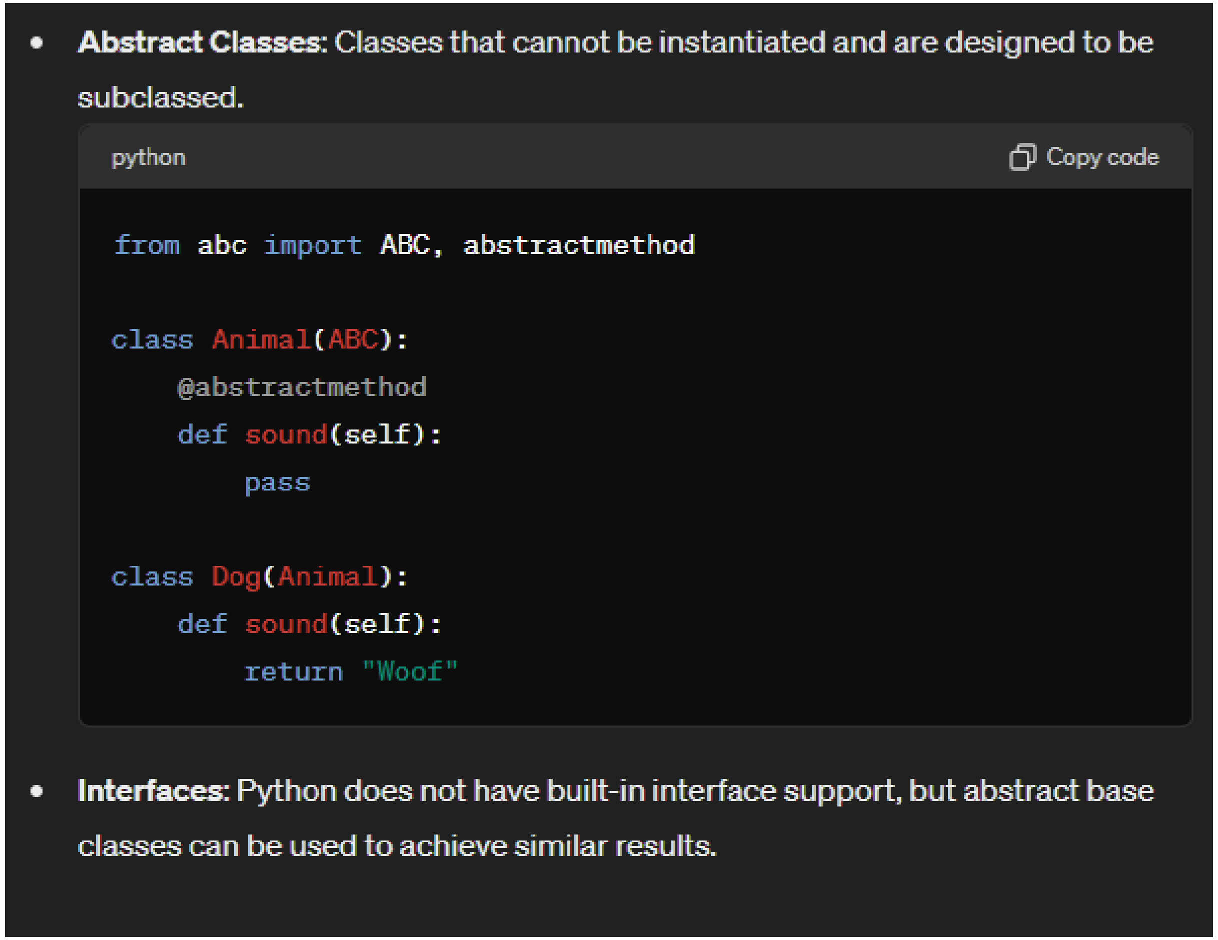
# Real-World Example



- **Problem:** Managing a library system
- **Classes:** Book, Member, Library

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

class Member:
    def __init__(self, name):
        self.name = name
        self.borrowed_books = []

    def borrow_book(self, book):
        self.borrowed_books.append(book)

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def lend_book(self, book, member):
        if book in self.books:
            self.books.remove(book)
            member.borrow_book(book)
```

# Advanced OOP Concepts



- **Abstract Classes:** Classes that cannot be instantiated and are designed to be subclassed.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Woof"
```

- **Interfaces:** Python does not have built-in interface support, but abstract base classes can be used to achieve similar results.

# Summary

- **OOP Principles**: Encapsulation, Inheritance, Polymorphism
- **Benefits**: Modularity, Reusability, Scalability
- **Python Features**: Easy to implement OOP concepts, supports advanced OOP features like abstract classes.

# Q&A

Questions:

**Open the floor for any questions about OOP in Python.**

# Thank you for your time
# Best Regards with
# Eng/Kirollos Gerges