Object-Oriented Programming (OOP) in Python

Presented by Suman Kumar Gochhayat

What is OOP?

Object-Oriented Programming (OOP) is a powerful paradigm that structures code around “objects” rather than functions and logic. In Python, this means creating self-contained units that bundle both data (attributes) and the actions that can be performed on that data (methods).

This approach leads to applications that are more intuitive, easier to manage, and robust.

Core Pillars of OOP

1

Encapsulation

Bundling data and methods within a single unit, controlling access to prevent direct manipulation.

2

Abstraction

Hiding complex implementation details, presenting only essential features for interaction.

3

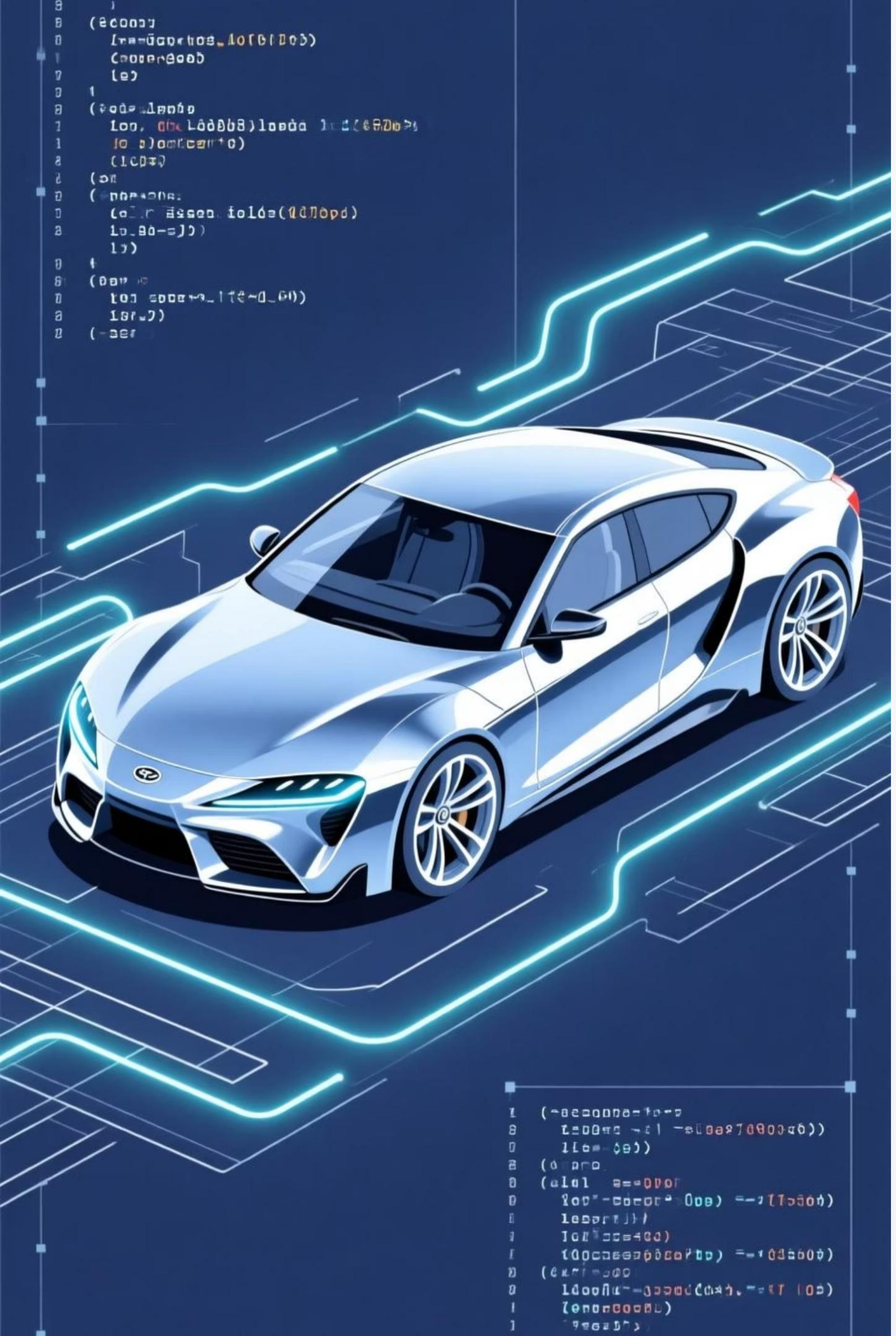
Inheritance

Creating new classes (child classes) that reuse and extend properties and behaviors from existing classes (parent classes).

4

Polymorphism

Allowing objects of different classes to be treated as objects of a common type, enabling a single interface for diverse actions.



Classes and Objects

A **class** is like a blueprint or a template for creating objects. It defines the common attributes and methods that all objects of that type will have.

An **object** is a concrete instance of a class, with its own unique data for the defined attributes. You can create multiple objects from a single class.

```
class Car:
def __init__(self, brand, model):
        self.brand = brand
        self.model = model
# Creating objects (instances of the Car class)
my_car = Car("Toyota", “Audi")
your_car = Car("Honda", “Suzuki")
```

Here, `Car` is the class, and `my_car` and `your_car` are objects.

Inheritance: Building on What Exists

Inheritance allows a new class (the **child** or **derived** class) to inherit attributes and methods from an existing class (the **parent** or **base** class).

This promotes code reusability and establishes a natural hierarchy among classes. Child classes can also add their own unique features or override inherited ones.

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal): # Dog inherits from Animal
    def speak(self): # Overrides speak() method
        print("Woof!")

class Cat(Animal): # Cat also inherits from Animal
    def speak(self):
        print("Meow!")
my_dog = Dog()
my_dog.speak() # Output: Woof!
```

Polymorphism: Many Forms, One Interface

Polymorphism, meaning "many forms," allows objects of different classes to be treated as objects of a common superclass. This means you can use a single interface (method call) to perform different actions depending on the object's actual type.

It makes code more flexible and extensible.

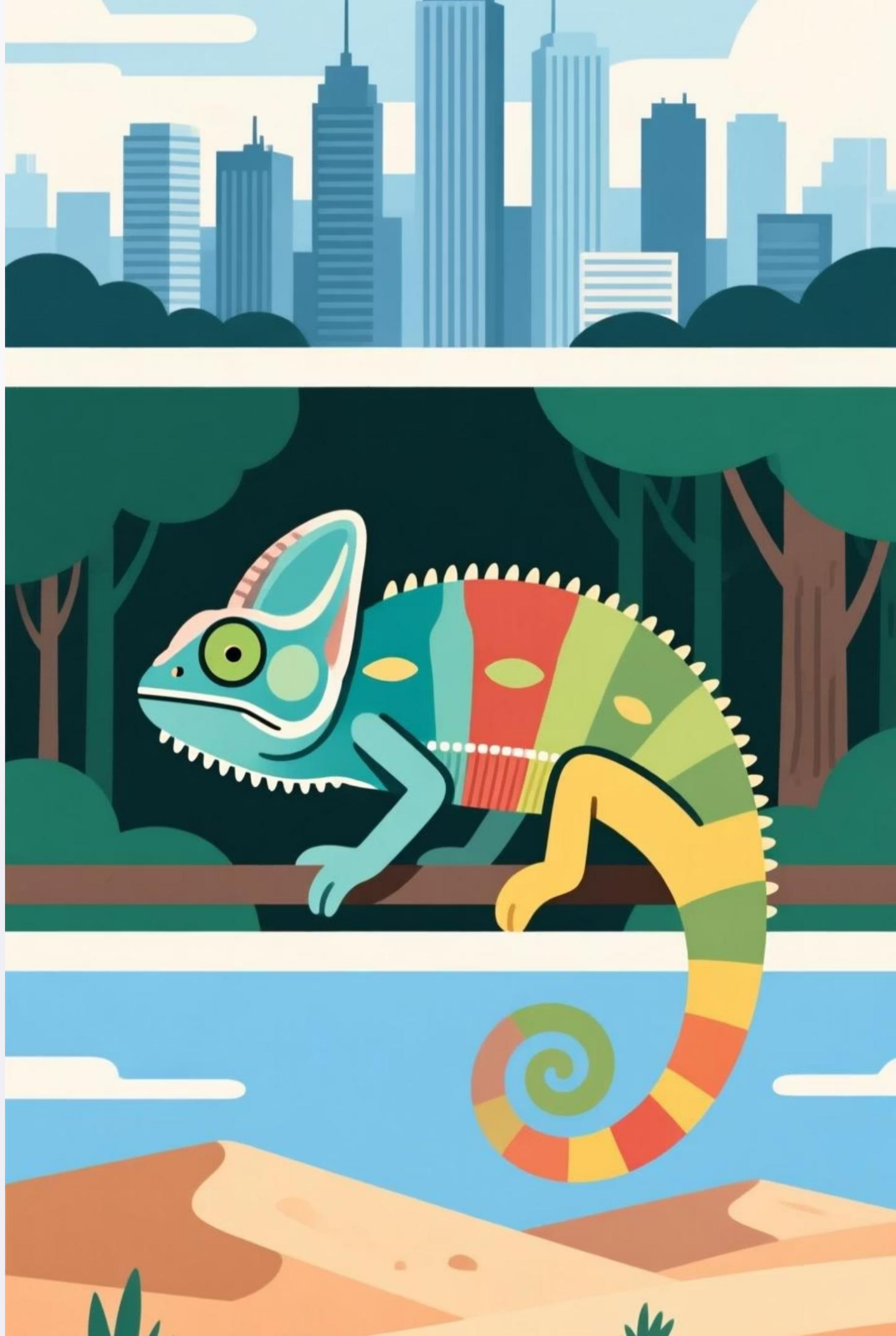
```
# (Using Animal, Dog, Cat classes from
previous slide)

def make_them_speak(animal):
    animal.speak()

my_dog = Dog()
my_cat = Cat()

make_them_speak(my_dog) # Output: Woof!
make_them_speak(my_cat) # Output: Meow!
```

Both Dog and Cat objects respond to the speak() method, but in their own unique way.



Encapsulation: Protecting Your Data

Encapsulation involves bundling the data (attributes) and methods that operate on the data within a single unit (a class). It also means restricting direct access to some of an object's components, which is often referred to as "data hiding."

In Python, this is achieved through naming conventions, like using a double underscore `__` for "private" attributes, signaling they shouldn't be accessed directly from outside the class.

```
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance # "Private" attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}. New balance: {self.__balance}")

    def get_balance(self):
        return self.__balance

account = BankAccount(100)
account.deposit(50)
# print(account.__balance) # This would raise an AttributeError!
```

Abstraction: Simplifying Complexity

Abstraction focuses on showing only essential information and hiding the complex implementation details. It allows you to work with high-level concepts without needing to understand the intricate inner workings.

In Python, abstract classes and methods (using the `abc` module) are used to define an interface that concrete subclasses must implement.

```
from abc import ABC, abstractmethod

class Shape(ABC): # Abstract Base Class
    @abstractmethod
    def area(self):
        pass # Must be implemented by subclasses

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

# shape = Shape() # This would raise a TypeError!
circle = Circle(5)
print(circle.area())
```

The Power of Object-Oriented Design

Modularity & Reusability

Break down complex systems into smaller, independent, and reusable components.

Readability & Debugging

Organized code is easier to understand, maintain, and troubleshoot when issues arise.

Scalability & Maintainability

Easily add new features or modify existing ones without disrupting the entire codebase.

Real-World Mapping

Model complex problems by mapping real-world entities directly into code objects.

Mastering Python OOP

Object-Oriented Programming in Python is a fundamental skill for any aspiring developer. By understanding and applying its core principles—**Encapsulation, Abstraction, Inheritance, and Polymorphism**—you gain the ability to write code that is not just functional, but also:

- Clean and organized
- Modular and reusable
- Easier to debug and maintain
- Scalable for complex applications

> "The journey to becoming a proficient Python developer is incomplete without a solid grasp of OOP."

THANK YOU