Introduction to OOP

1. Introduction to OOP

**Introduction to OOP: Classes, Objects, Attributes, and Methods (1 hour) 🏫**

**What’s OOP?**  
Imagine everything as objects — a car 🚗, a book 📚, or even a cute little puppy 🐶. OOP helps us model complex systems by treating real-world entities as objects within the program. We assign them **classes** (their blueprint), **attributes** (their characteristics), and **methods** (their actions)!

**Key Concepts:**

* **Class**: The blueprint or prototype, like a recipe 🍲! Defines the structure for creating objects.
* **Object**: A specific instance of a class. Think of it as a cake made from that recipe 🎂.
* **Attributes**: Characteristics or properties of an object, like the color of a car.
* **Methods**: Actions or behaviors that the object can perform (e.g., drive() for a car).

**Example: Create Your First Class!**

# Defining a class

class Car:

color = "red" # Attribute

# Method

def drive(self):

print("The car is driving 🚗")

# Creating an object

my\_car = Car()

print(my\_car.color)

my\_car.drive()

1. Constructors in Python OOP

**Constructors: The \_\_init\_\_ Method and Instance Variables 🚀**

**What’s a Constructor?** Every time you create a new object, you want it to have its unique attributes (color, model, etc.). Constructors (\_\_init\_\_ methods) allow each object to start with specific values. It’s like building a pizza 🍕 with the toppings you want!

**Instance Variables**  
Instance variables are specific to each object and can vary across objects. For example, two Car objects can have different colors, models, and speeds.

**Example: Setting Up Your Constructor!**

class Car:

def \_\_init\_\_(self, color, model):

self.color = color # Instance variable

self.model = model # Instance variable

# Creating objects with unique attributes

car1 = Car("blue", "Sedan")

car2 = Car("red", "SUV")

print(car1.color) # Output: blue

print(car2.model) # Output: SUV

1. OOP Principles

**OOP Principles: Inheritance, Polymorphism, and Encapsulation (1.5 hours) 👩‍🏫👨‍🏫**

* **Inheritance** 👨‍👩‍👧‍👦: Just like humans inherit traits from their parents, classes can inherit attributes and methods from other classes. This helps reduce code repetition and create a natural hierarchy in your code!
  + **Example:** Imagine a Vehicle class with general features (like wheels). We can create subclasses like Car and Bike that inherit those features!

python

class Vehicle:

def \_\_init\_\_(self, wheels):

self.wheels = wheels

class Car(Vehicle):

pass

car = Car(4)

print(car.wheels) # Output: 4

**Polymorphism** 🦄: Derived classes can behave differently for the same method inherited from a base class. With polymorphism, a method name can mean different things across multiple classes.

* + **Example:** Imagine a speak() method. Dogs bark(), while cats meow(), even though both use speak()!

class Dog:

def speak(self):

return "Woof!"

class Cat:

def speak(self):

return "Meow!"

# Polymorphism in action

for animal in [Dog(), Cat()]:

print(animal.speak())

**Encapsulation** 🔐: This is the practice of keeping attributes and methods private to prevent unwanted interference from outside the class. It’s like hiding your chocolate stash 🍫 from everyone else!

class SecretStash:

def \_\_init\_\_(self):

self.\_\_chocolates = 10 # Private attribute

def take\_chocolate(self):

if self.\_\_chocolates > 0:

self.\_\_chocolates -= 1

print("One chocolate taken!")

else:

print("No chocolates left 😢")

stash = SecretStash()

stash.take\_chocolate()

**In Summary**  
OOP allows you to organize code in a way that’s fun, reusable, and efficient! As you practice, imagine the real-world objects around you and think of how they could become classes in your code. Whether you’re designing a Smartphone, Pet, or Superhero, OOP gives you the power to build programs that feel like real-world systems.

Happy coding! 🎉