**AI Day 11 Notes**

**Syed Mansoor ul Hassan Bukhari**

**Python Classes and Objects:**

1. **Definition**: Classes in Python are blueprints for creating objects. An object is an instance of a class. Classes encapsulate data and functions that operate on the data.

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

my\_car = Car("Suzuki", "Skoda", 2021)

print(my\_car.brand)

1. **Class Attributes**: Class attributes are variables that belong to the class itself and are shared among all instances of the class.

class Car:

wheels = 4 # Class attribute

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

print(Car.wheels)

1. **Instance Attributes**: Instance attributes are variables that belong to an instance of the class. Each instance can have different values for these attributes.

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

my\_car = Car("Ford", "Mustang", 1964)

print(my\_car.brand)

1. **Methods**: Methods are functions defined inside a class that describe the behaviors of the objects created from the class.

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

def display\_info(self):

print(f"Brand: {self.brand}, Model: {self.model}, Year: {self.year}")

my\_car = Car("Ford", "Mustang", 1964)

my\_car.display\_info()

1. **The \_\_init\_\_ Method**: The \_\_init\_\_ method is a special method that is called when an object is instantiated. It initializes the object’s attributes.

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

my\_car = Car("Ford", "Mustang", 1964)

print(my\_car.year)

1. **Creating Objects**: Objects are instances of a class. You can create multiple objects from the same class.

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

car1 = Car("Ford", "Mustang", 1964)

car2 = Car("Toyota", "Corolla", 2020)

print(car1.model)

print(car2.model)

1. **Modifying Object Attributes**: You can change the value of an object’s attributes after it has been created.

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

my\_car = Car("Ford", "Mustang", 1964)

my\_car.year = 2020

print(my\_car.year)

1. **Deleting Object Attributes**: You can delete an object’s attributes using the del keyword.

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

my\_car = Car("Ford", "Mustang", 1964)

del my\_car.year

1. **Class Methods**: Class methods are methods that are bound to the class and not the instance of the class. They can modify class state that applies across all instances of the class.

class Car:

wheels = 4

@classmethod

def change\_wheels(cls, num\_wheels):

cls.wheels = num\_wheels

Car.change\_wheels(6)

print(Car.wheels)

1. **Static Methods**: Static methods are methods that do not modify class or instance state. They are defined using the @staticmethod decorator.

class Car:

@staticmethod

def honk():

print("Beep beep!")

Car.honk()

1. **Inheritance**: Inheritance allows a class to inherit attributes and methods from another class. The class that inherits is called the child class, and the class being inherited from is called the parent class.

class Vehicle:

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

self.brand = brand

self.model = model

class Car(Vehicle):

def \_\_init\_\_(self, brand, model, year):

super().\_\_init\_\_(brand, model)

self.year = year

my\_car = Car("Ford", "Mustang", 1964)

print(my\_car.brand)

1. **Polymorphism**: Polymorphism allows methods to do different things based on the object it is acting upon.

class Vehicle:

def start(self):

print("Starting vehicle...")

class Car(Vehicle):

def start(self):

print("Starting car...")

my\_vehicle = Vehicle()

my\_car = Car()

my\_vehicle.start()

my\_car.start()

1. **Encapsulation**: Encapsulation is the concept of wrapping data and methods that work on the data within one unit. This prevents data from being modified directly.

class Car:

def \_\_init\_\_(self, brand, model, year):

self.\_\_brand = brand # Private attribute

self.model = model

self.year = year

def get\_brand(self):

return self.\_\_brand

my\_car = Car("Ford", "Mustang", 1964)

print(my\_car.get\_brand())

1. **Abstraction**: Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object.

from abc import ABC, abstractmethod

class Vehicle(ABC):

@abstractmethod

def start(self):

pass

class Car(Vehicle):

def start(self):

print("Starting car...")

my\_car = Car()

my\_car.start()

1. **Multiple Inheritance**: Multiple inheritance allows a class to inherit from more than one class.

class Engine:

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

self.horsepower = horsepower

class Wheels:

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

self.size = size

class Car(Engine, Wheels):

def \_\_init\_\_(self, brand, model, year, horsepower, size):

Engine.\_\_init\_\_(self, horsepower)

Wheels.\_\_init\_\_(self, size)

self.brand = brand

self.model = model

self.year = year

my\_car = Car("Ford", "Mustang", 1964, 300, 18)

print(my\_car.horsepower)

print(my\_car.size)

1. **Method Overriding**: Method overriding allows a child class to provide a specific implementation of a method that is already defined in its parent class.

class Vehicle:

def start(self):

print("Starting vehicle...")

class Car(Vehicle):

def start(self):

print("Starting car...")

my\_car = Car()

my\_car.start()

1. **Operator Overloading**: Operator overloading allows you to define how operators behave for user-defined types.

class Vector:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(2, 3)

v2 = Vector(4, 5)

v3 = v1 + v2

print(v3.x, v3.y)