**Classes in Python Programming**

A **class** in Python is a blueprint for creating objects. It provides a way to bundle data and functionality together. A class defines the structure and behavior of objects, and an object is an instance of a class.

**1. Defining a Class**

A class is defined using the class keyword.

**Example:**

class Car:

# Class attribute

wheels = 4

# Constructor (initializer)

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

self.brand = brand # Instance attribute

self.color = color # Instance attribute

# Method to display car details

def display\_info(self):

print(f"Brand: {self.brand}, Color: {self.color}, Wheels: {self.wheels}")

**2. Creating Objects (Instances)**

To create an object, call the class name as a function:

**Example:**

# Creating objects of the Car class

car1 = Car("Toyota", "Red")

car2 = Car("Honda", "Blue")

# Accessing methods

car1.display\_info()

car2.display\_info()

**Output:**

Brand: Toyota, Color: Red, Wheels: 4

Brand: Honda, Color: Blue, Wheels: 4

**3. Instance vs. Class Attributes**

* **Instance attributes**: Defined inside the constructor (\_\_init\_\_) and are unique to each object.
* **Class attributes**: Shared across all instances of the class.

**Example:**

class Dog:

species = "Canine" # Class attribute

def \_\_init\_\_(self, name, breed):

self.name = name # Instance attribute

self.breed = breed # Instance attribute

dog1 = Dog("Buddy", "Labrador")

dog2 = Dog("Charlie", "Poodle")

print(dog1.species) # Output: Canine

print(dog2.species) # Output: Canine

print(dog1.name) # Output: Buddy

print(dog2.name) # Output: Charlie

**4. Methods in Classes**

**Types of Methods:**

1. **Instance Method**: Operates on instance attributes.
2. **Class Method (@classmethod)**: Works with class attributes.
3. **Static Method (@staticmethod)**: Does not depend on class or instance attributes.

**Example:**

class Example:

class\_attr = "I am a class attribute"

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

self.instance\_attr = value

def instance\_method(self):

return f"Instance Method: {self.instance\_attr}"

@classmethod

def class\_method(cls):

return f"Class Method: {cls.class\_attr}"

@staticmethod

def static\_method():

return "Static Method: Independent of class or instance"

obj = Example("Instance Value")

print(obj.instance\_method()) # Accessing instance method

print(Example.class\_method()) # Accessing class method

print(Example.static\_method()) # Accessing static method

**5. Inheritance (Reusing Code)**

Inheritance allows a class to inherit attributes and methods from another class.

**Example:**

# Parent class

class Animal:

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

self.name = name

def speak(self):

return "I make sounds"

# Child class

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

dog = Dog("Buddy")

cat = Cat("Whiskers")

print(dog.name, "says:", dog.speak()) # Output: Buddy says: Woof!

print(cat.name, "says:", cat.speak()) # Output: Whiskers says: Meow!

**6. Encapsulation (Data Hiding)**

Encapsulation restricts direct access to data by using **private variables** (\_\_variable).

**Example:**

class BankAccount:

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

self.\_\_balance = balance # Private variable

def deposit(self, amount):

self.\_\_balance += amount

def get\_balance(self):

return self.\_\_balance

account = BankAccount(1000)

account.deposit(500)

print(account.get\_balance()) # Output: 1500

# print(account.\_\_balance) # This will raise an AttributeError

**7. Polymorphism (Multiple Forms)**

Different classes can define the same method in different ways.

**Example:**

class Bird:

def fly(self):

return "Birds can fly"

class Airplane:

def fly(self):

return "Airplanes fly with engines"

def flying\_ability(entity):

print(entity.fly())

bird = Bird()

plane = Airplane()

flying\_ability(bird) # Output: Birds can fly

flying\_ability(plane) # Output: Airplanes fly with engines

**8. Special Methods (Magic/Dunder Methods)**

Python classes have special methods that start and end with double underscores (\_\_).

**Common Dunder Methods:**

| **Method** | **Description** |
| --- | --- |
| \_\_init\_\_ | Constructor (initializes objects) |
| \_\_str\_\_ | String representation of the object |
| \_\_len\_\_ | Defines behavior for len() |
| \_\_eq\_\_ | Defines behavior for == operator |

**Example:**

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def \_\_str\_\_(self):

return f"Person(Name: {self.name}, Age: {self.age})"

p = Person("Alice", 25)

print(p) # Output: Person(Name: Alice, Age: 25)

**9. Abstract Classes (Using ABC Module)**

Abstract classes cannot be instantiated and must be inherited.

**Example:**

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def make\_sound(self):

pass

class Dog(Animal):

def make\_sound(self):

return "Bark"

dog = Dog()

print(dog.make\_sound()) # Output: Bark

**Conclusion**

* **Classes** define objects with attributes (data) and methods (behavior).
* **Objects** are instances of classes.
* **Encapsulation** hides data from direct modification.
* **Inheritance** allows one class to inherit from another.
* **Polymorphism** enables different classes to use the same method name in different ways.
* **Special Methods** customize class behavior.