**1. Classes in Python**

A **class** is a blueprint for creating objects. It defines attributes (variables) and behaviors (methods) that objects created from the class will have.

**Syntax:**

class MyClass:

# Class attributes

class\_variable = "I am a class variable"

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

self.instance\_variable = value # Instance variable

def display(self):

print(f"Class Variable: {MyClass.class\_variable}")

print(f"Instance Variable: {self.instance\_variable}")

# Creating an object

obj = MyClass("Hello")

obj.display()

**2. Objects in Python**

An **object** is an instance of a class. It contains the actual values assigned to attributes defined in the class.

class Car:

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

self.brand = brand

self.model = model

def display(self):

print(f"Car: {self.brand} {self.model}")

# Creating objects

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

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

car1.display()

car2.display()

**3. Inheritance**

Inheritance allows a new class (child class) to inherit properties and methods from an existing class (parent class).

**Example:**

class Animal:

def speak(self):

print("This animal makes a sound")

class Dog(Animal): # Dog inherits from Animal

def speak(self):

print("Dog barks")

dog = Dog()

dog.speak() # Output: Dog barks

**4. Class Attributes vs Instance Attributes**

* **Class attributes** are shared among all instances.
* **Instance attributes** are unique to each instance.

**Example:**

class Employee:

company = "TechCorp" # Class Attribute

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

self.name = name # Instance Attribute

self.salary = salary # Instance Attribute

emp1 = Employee("Alice", 5000)

emp2 = Employee("Bob", 7000)

print(emp1.company) # TechCorp

print(emp2.company) # TechCorp

print(emp1.name) # Alice

print(emp2.name) # Bob

**5. Methods in Python Classes**

* **Instance methods:** Operate on instance variables.
* **Class methods (@classmethod)** operate on class variables.
* **Static methods (@staticmethod)** are independent of class and instance variables.

**Example:**

class MyClass:

class\_variable = "Shared"

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

self.instance\_variable = value

def instance\_method(self):

return f"Instance variable: {self.instance\_variable}"

@classmethod

def class\_method(cls):

return f"Class variable: {cls.class\_variable}"

@staticmethod

def static\_method():

return "I am a static method"

obj = MyClass("Hello")

print(obj.instance\_method())

print(MyClass.class\_method())

print(MyClass.static\_method())

**6. Private and Public Members**

* Public members can be accessed anywhere.
* Private members (prefix with \_ or \_\_) should not be accessed outside the class.

**Example:**

class BankAccount:

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

self.\_balance = balance # Protected variable

self.\_\_secret = "Hidden" # Private variable

def get\_balance(self):

return self.\_balance

account = BankAccount(1000)

print(account.get\_balance()) # Allowed

# print(account.\_\_secret) # Not allowed (Error)

**7. Getters and Setters**

Getters and Setters allow controlled access to private attributes.

**Example:**

class Person:

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

self.\_\_name = name

def get\_name(self): # Getter

return self.\_\_name

def set\_name(self, new\_name): # Setter

self.\_\_name = new\_name

p = Person("Alice")

print(p.get\_name()) # Alice

p.set\_name("Bob")

print(p.get\_name()) # Bob

**8. Subclass and Superclass**

A **subclass** inherits from a **superclass**.

class Parent:

def show(self):

print("Parent class method")

class Child(Parent):

pass

c = Child()

c.show() # Parent class method

**9. Multiple and Multilevel Inheritance**

**Multiple Inheritance (Inheriting from multiple classes):**

class A:

def method\_A(self):

print("Method of class A")

class B:

def method\_B(self):

print("Method of class B")

class C(A, B):

pass

obj = C()

obj.method\_A()

obj.method\_B()

**Multilevel Inheritance (A → B → C)**

class A:

def method\_A(self):

print("Method of class A")

class B(A):

def method\_B(self):

print("Method of class B")

class C(B):

def method\_C(self):

print("Method of class C")

obj = C()

obj.method\_A()

obj.method\_B()

obj.method\_C()

**10. Polymorphism**

Different classes can have methods with the same name.

class Cat:

def speak(self):

print("Meow")

class Dog:

def speak(self):

print("Bark")

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

animal.speak() # Meow, Bark

**11. Special Methods (\_\_repr\_\_, \_\_str\_\_, \_\_add\_\_, \_\_sub\_\_, \_\_mul\_\_)**

* \_\_repr\_\_: Returns an official string representation.
* \_\_str\_\_: Returns a user-friendly string representation.
* \_\_add\_\_: Overloads + operator.
* \_\_sub\_\_: Overloads - operator.

class MyNumber:

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

self.num = num

def \_\_repr\_\_(self):

return f"MyNumber({self.num})"

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

return f"Number: {self.num}"

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

return MyNumber(self.num + other.num)

num1 = MyNumber(10)

num2 = MyNumber(20)

print(num1 + num2) # MyNumber(30)

**12. Built-in Functions and Custom Data Types**

Python has built-in functions like len(), type(), dir(). We can also define custom data types.

print(len("Hello")) # 5

print(type(100)) # <class 'int'>

print(dir(str)) # Shows all string methods

**13. Defining Properties in Classes**

Properties control access to attributes dynamically.

class Student:

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

self.\_name = name

@property

def name(self): # Getter

return self.\_name

@name.setter

def name(self, new\_name): # Setter

self.\_name = new\_name

s = Student("John")

print(s.name) # John

s.name = "Doe"

print(s.name) # Doe

**Conclusion**

These concepts form the foundation of **Object-Oriented Programming (OOP)** in Python. They allow code reuse, modularity, and maintainability.