**1. Basic Class and Object**

class Car:

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

self.brand = brand

self.model = model

def display(self):

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

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

car1.display()

**2. Class with Encapsulation**

class BankAccount:

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

self.\_\_account\_number = account\_number # Private variable

self.\_\_balance = balance

def deposit(self, amount):

self.\_\_balance += amount

def withdraw(self, amount):

if amount <= self.\_\_balance:

self.\_\_balance -= amount

else:

print("Insufficient balance!")

def get\_balance(self):

return self.\_\_balance

account = BankAccount(12345, 5000)

account.deposit(2000)

print(account.get\_balance())

**3. Inheritance**

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

print("Dog barks")

d = Dog()

d.speak()

**4. Multiple Inheritance**

class Father:

def skill(self):

print("Father: Knows driving")

class Mother:

def skill(self):

print("Mother: Knows cooking")

class Child(Father, Mother):

def skill(self):

super().skill()

Mother.skill(self)

c = Child()

c.skill()

**5. Multilevel Inheritance**

class Animal:

def move(self):

print("Animals move")

class Mammal(Animal):

def feed\_milk(self):

print("Mammals feed milk")

class Dog(Mammal):

def bark(self):

print("Dog barks")

d = Dog()

d.move()

d.feed\_milk()

d.bark()

**6. Hierarchical Inheritance**

class Parent:

def function(self):

print("This is a parent class.")

class Child1(Parent):

def function1(self):

print("This is Child1.")

class Child2(Parent):

def function2(self):

print("This is Child2.")

c1 = Child1()

c1.function()

c2 = Child2()

c2.function()

**7. Method Overriding**

class Vehicle:

def speed(self):

print("Vehicles have different speeds.")

class Car(Vehicle):

def speed(self):

print("Cars can go up to 200 km/h.")

c = Car()

c.speed()

**8. Method Overloading using Default Arguments**

class Calculator:

def add(self, a, b=0, c=0):

return a + b + c

calc = Calculator()

print(calc.add(5))

print(calc.add(5, 10))

print(calc.add(5, 10, 15))

**9. Operator Overloading**

class ComplexNumber:

def \_\_init\_\_(self, real, imag):

self.real = real

self.imag = imag

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

return ComplexNumber(self.real + other.real, self.imag + other.imag)

def display(self):

print(f"{self.real} + {self.imag}i")

c1 = ComplexNumber(3, 4)

c2 = ComplexNumber(1, 2)

c3 = c1 + c2

c3.display()

**10. Static Methods**

class MathOperations:

@staticmethod

def add(a, b):

return a + b

print(MathOperations.add(10, 20))

**11. Class Methods**

class Company:

company\_name = "TechCorp"

@classmethod

def change\_name(cls, new\_name):

cls.company\_name = new\_name

Company.change\_name("InnovateTech")

print(Company.company\_name)

**12. Abstract Class**

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

class Square(Shape):

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

self.side = side

def area(self):

return self.side \*\* 2

s = Square(4)

print(s.area())

**13. Interface Implementation**

class Animal:

def make\_sound(self):

pass

class Dog(Animal):

def make\_sound(self):

print("Woof Woof")

d = Dog()

d.make\_sound()

**14. Composition (Has-a Relationship)**

class Engine:

def start(self):

print("Engine started")

class Car:

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

self.engine = Engine()

def start(self):

self.engine.start()

print("Car started")

car = Car()

car.start()

**15. Polymorphism**

class Bird:

def fly(self):

print("Birds can fly")

class Sparrow(Bird):

def fly(self):

print("Sparrows can fly fast")

class Penguin(Bird):

def fly(self):

print("Penguins cannot fly")

for bird in [Sparrow(), Penguin()]:

bird.fly()

**16. Property Decorators**

class Person:

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

self.\_\_name = name

@property

def name(self):

return self.\_\_name

p = Person("Alice")

print(p.name)

**17. Singleton Pattern**

class Singleton:

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super(Singleton, cls).\_\_new\_\_(cls)

return cls.\_instance

s1 = Singleton()

s2 = Singleton()

print(s1 is s2)

**18. Factory Pattern**

class Animal:

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "Bark"

class Cat(Animal):

def speak(self):

return "Meow"

def animal\_factory(animal\_type):

return Dog() if animal\_type == "dog" else Cat()

animal = animal\_factory("dog")

print(animal.speak())

**19. Class and Instance Variables**

class Student:

school\_name = "ABC School" # Class variable

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

self.name = name # Instance variable

s1 = Student("John")

print(s1.school\_name, s1.name)

**20. Object Cloning (Deep Copy)**

import copy

class Person:

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

self.name = name

p1 = Person("John")

p2 = copy.deepcopy(p1)

print(p1.name, p2.name)