**1. Class and Object Creation**

class Car:

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

self.brand = brand

self.model = model

def display(self):

return f"Car: {self.brand} {self.model}"

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

print(car1.display())

**2. Class with Default and Parameterized Constructor**

class Employee:

def \_\_init\_\_(self, name="John Doe", salary=50000):

self.name = name

self.salary = salary

emp1 = Employee()

emp2 = Employee("Alice", 70000)

print(emp1.name, emp1.salary)

print(emp2.name, emp2.salary)

**3. Private and Public Attributes**

class BankAccount:

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

self.\_\_balance = balance # Private Attribute

def deposit(self, amount):

self.\_\_balance += amount

def get\_balance(self):

return self.\_\_balance

acc = BankAccount(1000)

acc.deposit(500)

print(acc.get\_balance()) # Accessing via method

# print(acc.\_\_balance) # This will raise an error

**4. Getter and Setter Methods**

class Student:

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

self.\_\_name = name

def get\_name(self):

return self.\_\_name

def set\_name(self, name):

self.\_\_name = name

s = Student("Bob")

print(s.get\_name())

s.set\_name("Alice")

print(s.get\_name())

**5. Class Method and Static Method**

class MathOperations:

@classmethod

def add(cls, a, b):

return a + b

@staticmethod

def multiply(a, b):

return a \* b

print(MathOperations.add(5, 3))

print(MathOperations.multiply(4, 2))

**6. Inheritance Example**

class Animal:

def speak(self):

return "I make a sound"

class Dog(Animal):

def speak(self):

return "Bark"

dog = Dog()

print(dog.speak())

**7. Multiple Inheritance**

class A:

def method\_a(self):

return "A's Method"

class B:

def method\_b(self):

return "B's Method"

class C(A, B):

def method\_c(self):

return "C's Method"

obj = C()

print(obj.method\_a())

print(obj.method\_b())

print(obj.method\_c())

**8. Multilevel Inheritance**

class GrandParent:

def method\_gp(self):

return "Grandparent Method"

class Parent(GrandParent):

def method\_p(self):

return "Parent Method"

class Child(Parent):

def method\_c(self):

return "Child Method"

obj = Child()

print(obj.method\_gp())

print(obj.method\_p())

print(obj.method\_c())

**9. Polymorphism with Method Overriding**

class Bird:

def fly(self):

return "Some birds can fly"

class Sparrow(Bird):

def fly(self):

return "Sparrows can fly"

bird = Bird()

sparrow = Sparrow()

print(bird.fly())

print(sparrow.fly())

**10. Polymorphism with Method Overloading (Using Default Parameters)**

class MathOps:

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

return a + b + c

obj = MathOps()

print(obj.add(2, 3)) # 2 arguments

print(obj.add(2, 3, 4)) # 3 arguments

**11. Abstract Base Class**

from abc import ABC, abstractmethod

class Vehicle(ABC):

@abstractmethod

def fuel\_efficiency(self):

pass

class Car(Vehicle):

def fuel\_efficiency(self):

return "Car: 15 km/l"

c = Car()

print(c.fuel\_efficiency())

**12. Operator Overloading**

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(f"Vector: ({v3.x}, {v3.y})")

**13. Encapsulation using Property Decorators**

class Product:

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

self.\_\_price = price

@property

def price(self):

return self.\_\_price

@price.setter

def price(self, value):

if value >= 0:

self.\_\_price = value

else:

raise ValueError("Price cannot be negative")

p = Product(100)

print(p.price)

p.price = 150

print(p.price)

**14. Composition Example**

class Engine:

def start(self):

return "Engine started"

class Car:

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

self.engine = Engine()

def start(self):

return self.engine.start()

c = Car()

print(c.start())

**15. Aggregation Example**

class Address:

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

self.city = city

class Person:

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

self.name = name

self.address = address

addr = Address("New York")

p = Person("John", addr)

print(p.name, "lives in", p.address.city)

**16. Method Chaining**

class Car:

def set\_brand(self, brand):

self.brand = brand

return self

def set\_model(self, model):

self.model = model

return self

car = Car().set\_brand("Toyota").set\_model("Camry")

print(car.brand, car.model)

**17. Singleton Design 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) # True

**18. Duck Typing**

class Bird:

def fly(self):

return "Bird is flying"

class Airplane:

def fly(self):

return "Airplane is flying"

def let\_it\_fly(obj):

print(obj.fly())

let\_it\_fly(Bird())

let\_it\_fly(Airplane())

**19. Class Variable vs Instance Variable**

class Counter:

count = 0 # Class variable

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

Counter.count += 1

c1 = Counter()

c2 = Counter()

print(Counter.count) # 2

**20. Mixin Class**

class LogMixin:

def log(self, message):

print(f"Log: {message}")

class User(LogMixin):

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

self.name = name

def display(self):

self.log(f"User {self.name} displayed.")

u = User("Alice")

u.display()