1: shape interface

"""

1.Create an interface called Shape with abstract methods area() and perimeter().

"""

from abc import ABC, abstractmethod

# Define the Shape interface with abstract methods

class Shape(ABC):

@abstractmethod

def area(self):

pass

@abstractmethod

def perimeter(self):

pass

"""

Create an interface called Shape with abstract methods area() and perimeter().

Implement concrete classes Circle, Rectangle, and Triangle that implement the Shape interface.

"""

from abc import ABC, abstractmethod

import math

# Define the Shape interface with abstract methods

class Shape(ABC):

@abstractmethod

def area(self):

pass

@abstractmethod

def perimeter(self):

pass

# Concrete class: Circle

class Circle(Shape):

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

self.radius = radius

def area(self):

return math.pi \* self.radius\*\*2

def perimeter(self):

return 2 \* math.pi \* self.radius

# Concrete class: Rectangle

class Rectangle(Shape):

def \_\_init\_\_(self, length, width):

self.length = length

self.width = width

def area(self):

return self.length \* self.width

def perimeter(self):

return 2 \* (self.length + self.width)

# Concrete class: Triangle

class Triangle(Shape):

def \_\_init\_\_(self, side1, side2, side3):

self.side1 = side1

self.side2 = side2

self.side3 = side3

def area(self):

s = (self.side1 + self.side2 + self.side3) / 2

return math.sqrt(s \* (s - self.side1) \* (s - self.side2) \* (s - self.side3))

def perimeter(self):

return self.side1 + self.side2 + self.side3

# Example usage

circle = Circle(radius=5)

rectangle = Rectangle(length=4, width=6)

triangle = Triangle(side1=3, side2=4, side3=5)

# Output results

print(f"Circle - Area: {circle.area()}, Perimeter: {circle.perimeter()}")

print(f"Rectangle - Area: {rectangle.area()}, Perimeter: {rectangle.perimeter()}")

print(f"Triangle - Area: {triangle.area()}, Perimeter: {triangle.perimeter()}")

2. vehicle interface

"""

Create an interface called Vehicle with abstract methods start(), stop(), and move().

"""

from abc import ABC, abstractmethod

# Define the Vehicle interface with abstract methods

class Vehicle(ABC):

@abstractmethod

def start(self):

pass

@abstractmethod

def stop(self):

pass

@abstractmethod

def move(self):

pass

"""

Create an interface called Vehicle with abstract methods start(), stop(), and move().

Implement concrete classes Car, Bike, and Plane that implement the Vehicle interface.

"""

from abc import ABC, abstractmethod

# Define the Vehicle interface with abstract methods

class Vehicle(ABC):

@abstractmethod

def start(self):

pass

@abstractmethod

def stop(self):

pass

@abstractmethod

def move(self):

pass

# Concrete class: Car

class Car(Vehicle):

def start(self):

print("Car starting")

def stop(self):

print("Car stopping")

def move(self):

print("Car moving")

# Concrete class: Bike

class Bike(Vehicle):

def start(self):

print("Bike starting")

def stop(self):

print("Bike stopping")

def move(self):

print("Bike moving")

# Concrete class: Plane

class Plane(Vehicle):

def start(self):

print("Plane starting")

def stop(self):

print("Plane stopping")

def move(self):

print("Plane moving")

# Example usage

car = Car()

bike = Bike()

plane = Plane()

# Output results

print("=== Car ===")

car.start()

car.move()

car.stop()

print("\n=== Bike ===")

bike.start()

bike.move()

bike.stop()

print("\n=== Plane ===")

plane.start()

plane.move()

plane.stop()

calculator interfaces:

"""

Calculator Interfaces:

1. Create interfaces for basic arithmetic operations: Adder, Subtractor, Multiplier, and Divider.

2. Implement a class Calculator that can perform these operations using different strategies

(e.g., simple implementation, optimized algorithms).

"""

from abc import ABC, abstractmethod

# Interface for addition

class Adder(ABC):

@abstractmethod

def add(self, x, y):

pass

# Interface for subtraction

class Subtractor(ABC):

@abstractmethod

def subtract(self, x, y):

pass

# Interface for multiplication

class Multiplier(ABC):

@abstractmethod

def multiply(self, x, y):

pass

# Interface for division

class Divider(ABC):

@abstractmethod

def divide(self, x, y):

pass

# Simple implementation of the Calculator class

class SimpleCalculator(Adder, Subtractor, Multiplier, Divider):

def add(self, x, y):

return x + y

def subtract(self, x, y):

return x - y

def multiply(self, x, y):

return x \* y

def divide(self, x, y):

if y == 0:

raise ValueError("Cannot divide by zero")

return x / y

# Optimized implementation of the Calculator class (using built-in operators)

class OptimizedCalculator(Adder, Subtractor, Multiplier, Divider):

def add(self, x, y):

return x + y

def subtract(self, x, y):

return x - y

def multiply(self, x, y):

return x \* y

def divide(self, x, y):

if y == 0:

raise ValueError("Cannot divide by zero")

return x / y

# Example usage

simple\_calculator = SimpleCalculator()

optimized\_calculator = OptimizedCalculator()

# Perform operations using the simple calculator

print("Simple Calculator:")

print("Addition:", simple\_calculator.add(5, 3))

print("Subtraction:", simple\_calculator.subtract(8, 4))

print("Multiplication:", simple\_calculator.multiply(2, 6))

print("Division:", simple\_calculator.divide(10, 2))

# Perform operations using the optimized calculator

print("\nOptimized Calculator:")

print("Addition:", optimized\_calculator.add(5, 3))

print("Subtraction:", optimized\_calculator.subtract(8, 4))

print("Multiplication:", optimized\_calculator.multiply(2, 6))

print("Division:", optimized\_calculator.divide(10, 2))

**"""**

**Q1. Create interfaces for basic arithmetic operations: Adder, Subtractor, Multiplier, and Divider.**

**Q2. Implement a class Calculator that can perform these operations using different strategies**

**(e.g., simple implementation, optimized algorithms).**

**"""**

**from abc import ABC, abstractmethod**

**# Interface for addition**

**class Adder(ABC):**

**@abstractmethod**

**def add(self, x, y):**

**pass**

**# Interface for subtraction**

**class Subtractor(ABC):**

**@abstractmethod**

**def subtract(self, x, y):**

**pass**

**# Interface for multiplication**

**class Multiplier(ABC):**

**@abstractmethod**

**def multiply(self, x, y):**

**pass**

**# Interface for division**

**class Divider(ABC):**

**@abstractmethod**

**def divide(self, x, y):**

**pass**

**# Simple implementation of the Calculator class**

**class SimpleCalculator(Adder, Subtractor, Multiplier, Divider):**

**def add(self, x, y):**

**return x + y**

**def subtract(self, x, y):**

**return x - y**

**def multiply(self, x, y):**

**return x \* y**

**def divide(self, x, y):**

**if y == 0:**

**raise ValueError("Cannot divide by zero")**

**return x / y**

**# Optimized implementation of the Calculator class (using built-in operators)**

**class OptimizedCalculator(Adder, Subtractor, Multiplier, Divider):**

**def add(self, x, y):**

**return x + y**

**def subtract(self, x, y):**

**return x - y**

**def multiply(self, x, y):**

**return x \* y**

**def divide(self, x, y):**

**if y == 0:**

**raise ValueError("Cannot divide by zero")**

**return x / y**

**# Example usage**

**simple\_calculator = SimpleCalculator()**

**optimized\_calculator = OptimizedCalculator()**

**# Perform operations using the simple calculator**

**print("Simple Calculator:")**

**print("Addition:", simple\_calculator.add(5, 3))**

**print("Subtraction:", simple\_calculator.subtract(8, 4))**

**print("Multiplication:", simple\_calculator.multiply(2, 6))**

**print("Division:", simple\_calculator.divide(10, 2))**

**# Perform operations using the optimized calculator**

**print("\nOptimized Calculator:")**

**print("Addition:", optimized\_calculator.add(5, 3))**

**print("Subtraction:", optimized\_calculator.subtract(8, 4))**

**print("Multiplication:", optimized\_calculator.multiply(2, 6))**

**print("Division:", optimized\_calculator.divide(10, 2))**

from abc import ABC, abstractmethod

# Interface for addition

class Adder(ABC):

@abstractmethod

def add(self, x, y):

pass

# Interface for subtraction

class Subtractor(ABC):

@abstractmethod

def subtract(self, x, y):

pass

# Interface for multiplication

class Multiplier(ABC):

@abstractmethod

def multiply(self, x, y):

pass

# Interface for division

class Divider(ABC):

@abstractmethod

def divide(self, x, y):

pass

"""

Question 2:

Implement a class Calculator that can perform these operations using different strategies

(e.g., simple implementation, optimized algorithms).

"""

from abc import ABC, abstractmethod

# Interface for addition

class Adder(ABC):

@abstractmethod

def add(self, x, y):

pass

# Interface for subtraction

class Subtractor(ABC):

@abstractmethod

def subtract(self, x, y):

pass

# Interface for multiplication

class Multiplier(ABC):

@abstractmethod

def multiply(self, x, y):

pass

# Interface for division

class Divider(ABC):

@abstractmethod

def divide(self, x, y):

pass

# Simple implementation of the Calculator class

class SimpleCalculator(Adder, Subtractor, Multiplier, Divider):

def add(self, x, y):

return x + y

def subtract(self, x, y):

return x - y

def multiply(self, x, y):

return x \* y

def divide(self, x, y):

if y == 0:

raise ValueError("Cannot divide by zero")

return x / y

# Optimized implementation of the Calculator class (using built-in operators)

class OptimizedCalculator(Adder, Subtractor, Multiplier, Divider):

def add(self, x, y):

return x + y

def subtract(self, x, y):

return x - y

def multiply(self, x, y):

return x \* y

def divide(self, x, y):

if y == 0:

raise ValueError("Cannot divide by zero")

return x / y

# Example usage

simple\_calculator = SimpleCalculator()

optimized\_calculator = OptimizedCalculator()

# Perform operations using the simple calculator

print("Simple Calculator:")

print("Addition:", simple\_calculator.add(5, 3))

print("Subtraction:", simple\_calculator.subtract(8, 4))

print("Multiplication:", simple\_calculator.multiply(2, 6))

print("Division:", simple\_calculator.divide(10, 2))

# Perform operations using the optimized calculator

print("\nOptimized Calculator:")

print("Addition:", optimized\_calculator.add(5, 3))

print("Subtraction:", optimized\_calculator.subtract(8, 4))

print("Multiplication:", optimized\_calculator.multiply(2, 6))

print("Division:", optimized\_calculator.divide(10, 2))

Abstract clases:

1. Animal class
2. from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def eat(self):

pass

@abstractmethod

def make\_sound(self):

pass

ii. class Dog(Animal):

def eat(self):

return "Dog is eating"

def make\_sound(self):

return "Woof!"

class Cat(Animal):

def eat(self):

return "Cat is eating"

def make\_sound(self):

return "Meow!"

class Bird(Animal):

def eat(self):

return "Bird is eating"

def make\_sound(self):

return "Tweet!"

Employ class:

from abc import ABC, abstractmethod

class Employee(ABC):

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

self.name = name

self.salary = salary

def get\_name(self):

return self.name

def get\_salary(self):

return self.salary

@abstractmethod

def calculate\_pay(self):

pass

"""

Question: Create an abstract class Employee with methods for get\_name(), get\_salary(), and an abstract method calculate\_pay().

"""

from abc import ABC, abstractmethod

class Employee(ABC):

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

self.name = name

self.salary = salary

def get\_name(self):

return self.name

def get\_salary(self):

return self.salary

@abstractmethod

def calculate\_pay(self):

pass

class FullTimeEmployee(Employee):

def calculate\_pay(self):

return self.salary / 12 # Monthly salary for full-time employee

class PartTimeEmployee(Employee):

def \_init\_(self, name, hourly\_rate, hours\_worked):

super().\_init\_(name, 0) # Part-time employees don't have a fixed salary

self.hourly\_rate = hourly\_rate

self.hours\_worked = hours\_worked

def calculate\_pay(self):

return self.hourly\_rate \* self.hours\_worked # Pay for part-time employee based on hours worked

# Example usage:

full\_time\_employee = FullTimeEmployee("John Doe", 60000)

part\_time\_employee = PartTimeEmployee("Jane Smith", 15, 20)

print(f"{full\_time\_employee.get\_name()}'s monthly pay: ${full\_time\_employee.calculate\_pay()}")

print(f"{part\_time\_employee.get\_name()}'s pay for the week: ${part\_time\_employee.calculate\_pay()}")

"""

Question: Create an abstract class Shape with abstract methods area() and perimeter().

"""

from abc import ABC, abstractmethod

import math

class Shape(ABC):

@abstractmethod

def area(self):

pass

@abstractmethod

def perimeter(self):

pass

class Circle(Shape):

def \_init\_(self, radius):

self.radius = radius

def area(self):

return math.pi \* self.radius\*\*2

def perimeter(self):

return 2 \* math.pi \* self.radius

class Rectangle(Shape):

def \_init\_(self, length, width):

self.length = length

self.width = width

def area(self):

return self.length \* self.width

def perimeter(self):

return 2 \* (self.length + self.width)

class Triangle(Shape):

def \_init\_(self, side1, side2, side3):

self.side1 = side1

self.side2 = side2

self.side3 = side3

def area(self):

# Using Heron's formula for the area of a triangle

s = (self.side1 + self.side2 + self.side3) / 2

return math.sqrt(s \* (s - self.side1) \* (s - self.side2) \* (s - self.side3))

def perimeter(self):

return self.side1 + self.side2 + self.side3

# Example usage:

circle = Circle(5)

rectangle = Rectangle(4, 6)

triangle = Triangle(3, 4, 5)

print(f"Circle - Area: {circle.area()}, Perimeter: {circle.perimeter()}")

print(f"Rectangle - Area: {rectangle.area()}, Perimeter: {rectangle.perimeter()}")

print(f"Triangle - Area: {triangle.area()}, Perimeter: {triangle.perimeter()}")