#1a

class Students:

pass

class Marks:

pass

# Creating instances of the classes

student\_instance = Students()

marks\_instance = Marks()

# Checking if instances are of the specified classes

is\_student\_instance = isinstance(student\_instance, Students)

is\_marks\_instance = isinstance(marks\_instance, Marks)

# Checking if classes are subclasses of the built-in object class

is\_students\_subclass = issubclass(Students, object)

is\_marks\_subclass = issubclass(Marks, object)

# Printing the results

print(f"Is student\_instance an instance of Students? {is\_student\_instance}")

print(f"Is marks\_instance an instance of Marks? {is\_marks\_instance}")

print(f"Is Students a subclass of object? {is\_students\_subclass}")

print(f"Is Marks a subclass of object? {is\_marks\_subclass}")

#1b

class Complex:

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

self.real = real

self.imag = imag

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

# Overloading the + operator to perform addition of complex numbers

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

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

# String representation of the complex number

return f"{self.real} + {self.imag}j"

# Creating instances of the Complex class

c1 = Complex(3, 5)

c2 = Complex(1, 2)

# Adding c1 and c2 to get c3

c3 = c1 + c2

# Displaying the results

print(f"c1: {c1}")

print(f"c2: {c2}")

print(f"c3 (c1 + c2): {c3}")

#2a

class Song:

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

self.lyrics = lyrics

def sing\_me\_a\_song(self):

for line in self.lyrics:

print(line)

# Define the lyrics for the "Happy Birthday" song

happy\_bday\_lyrics = [

"May God bless you,",

"Have a sunshine on you,",

"Happy birthday to you"

]

# Create an instance of the Song class with the "Happy Birthday" lyrics

happy\_bday = Song(happy\_bday\_lyrics)

# Call the sing\_me\_a\_song method on the happy\_bday instance

happy\_bday.sing\_me\_a\_song()

#2b

class Dog:

def \_\_init\_\_(self, name, size, breed='Unknown', dob='Unknown'):

self.name = self.set\_name(name)

self.size = size

self.breed = breed

self.dob = dob

def bark(self):

print("Woof!")

def get\_name(self):

return self.name

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

if 2 <= len(new\_name) <= 30 and new\_name.isalpha():

return new\_name.title()

else:

print("Invalid name. Set a name between 2 and 30 alphabetical characters.")

return "Unknown"

def dog\_years(self):

# Assuming 1 year is equivalent to 7 dog years

return 7 \* (2024 - int(self.dob[-4:]))

# Example usage:

dog1 = Dog("Buddy", "Medium", "Labrador", "01/01/2010")

dog2 = Dog("Max", "Small")

# Accessing attributes

print(f"Dog 1's name: {dog1.get\_name()}")

print(f"Dog 1's size: {dog1.size}")

print(f"Dog 1's breed: {dog1.breed}")

print(f"Dog 1's date of birth: {dog1.dob}")

# Performing methods

dog1.bark()

print(f"Dog 1's age in dog years: {dog1.dog\_years()}")

# Trying to set an invalid name

dog1.set\_name("Invalid123")

# Setting a valid name

dog1.set\_name("Rover")

# Checking the updated name

print(f"Dog 1's updated name: {dog1.get\_name()}")

#3a

class Employee:

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

self.name = name

self.\_\_salary = salary # Using double underscore to make it private

def work(self):

print(f"{self.name} is working.")

def show(self):

print(f"Employee Name: {self.name}")

print(f"Salary: ${self.\_\_salary}")

# Example usage:

employee1 = Employee("John Doe", 50000)

# Accessing public attributes and methods

employee1.work()

# Accessing private attribute indirectly using a public method

employee1.show()

#3b

class CreditCard:

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

self.\_\_limit = limit

self.\_\_balance = 0

def get\_balance(self):

return self.\_\_balance

def withdraw(self, amount):

if amount <= 0:

print("Invalid withdrawal amount.")

return

if amount > self.\_\_balance:

print("Insufficient funds for withdrawal.")

return

self.\_\_balance -= amount

print(f"Withdrawal successful. Remaining balance: ${self.\_\_balance}")

def deposit(self, amount):

if amount <= 0:

print("Invalid deposit amount.")

return

self.\_\_balance += amount

print(f"Deposit successful. New balance: ${self.\_\_balance}")

# Example usage:

credit\_card = CreditCard(limit=1000)

# Accessing the balance using get\_balance method

current\_balance = credit\_card.get\_balance()

print(f"Current Balance: ${current\_balance}")

# Attempting to directly modify the balance (will not work)

# credit\_card.\_\_balance = 500 # This line will cause an AttributeError

# Depositing and withdrawing

credit\_card.deposit(200)

credit\_card.withdraw(50)

# Trying an invalid withdrawal

credit\_card.withdraw(1000)

4b

# Parent class 1

class Person:

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

self.name = name

# Parent class 2

class Company:

def \_\_init\_\_(self, company\_name):

self.company\_name = company\_name

# Child class inheriting from both Person and Company

class Employee(Person, Company):

def \_\_init\_\_(self, name, company\_name, employee\_id):

# Call constructors of both parent classes

Person.\_\_init\_\_(self, name)

Company.\_\_init\_\_(self, company\_name)

self.employee\_id = employee\_id

# Create an Employee object and access attributes from both parent classes

employee\_instance = Employee(name="John Doe", company\_name="ABC Corp", employee\_id="E12345")

print("Name:", employee\_instance.name)

print("Company:", employee\_instance.company\_name)

print("Employee ID:", employee\_instance.employee\_id)

4afrom abc import ABC, abstractmethod

# Define abstract base class

class Car(ABC):

@abstractmethod

def milcagc0(self):

pass

# Subclass 1

class Sedan(Car):

def milcagc0(self):

return "Sedan implementation of milcagc0"

# Subclass 2

class SUV(Car):

def milcagc0(self):

return "SUV implementation of milcagc0"

# Subclass 3

class Truck(Car):

def milcagc0(self):

return "Truck implementation of milcagc0"

sedan\_instance = Sedan()

suv\_instance = SUV()

truck\_instance = Truck()

print(sedan\_instance.milcagc0()) # Output: Sedan implementation of milcagc0

print(suv\_instance.milcagc0()) # Output: SUV implementation of milcagc0

print(truck\_instance.milcagc0()) # Output: Truck implementation of milcagc0

6A

class WrongAge(Exception):

pass

class AgeInvalid(Exception):

pass

class Father:

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

if age < 0:

raise WrongAge("Age cannot be negative")

self.age = age

class Son(Father):

def \_\_init\_\_(self, father\_age, son\_age):

super().\_\_init\_\_(father\_age)

if son\_age >= father\_age:

raise AgeInvalid("Son's age should be less than father's age")

self.son\_age = son\_age

# Example usage:

try:

father\_age = int(input("Enter father's age: "))

son\_age = int(input("Enter son's age: "))

father = Father(father\_age)

son = Son(father\_age, son\_age)

print("Father's age:", father.age)

print("Son's age:", son.son\_age)

except ValueError:

print("Invalid input! Please enter valid ages.")

except WrongAge as e:

print("Error:", e)

except AgeInvalid as e:

print("Error:", e)

6(B)

class FormulaError(Exception):

pass

def calculate(formula):

try:

values = formula.split()

if len(values) != 3:

raise FormulaError("Invalid formula: Formula must consist of a number, an operator (+ or -), and another number separated by white space")

num1 = float(values[0])

operator = values[1]

num2 = float(values[2])

if operator not in ['+', '-']:

raise FormulaError("Invalid operator: Operator must be + or -")

if operator == '+':

result = num1 + num2

elif operator == '-':

result = num1 - num2

print("Result:", result)

except ValueError:

raise FormulaError("Invalid value: Unable to convert to float")

except FormulaError as e:

print("FormulaError:", e)

def main():

while True:

user\_input = input("Enter formula (number operator number), or type 'quit' to exit: ")

if user\_input.lower() == 'quit':

print("Exiting...")

break

try:

calculate(user\_input)

except FormulaError as e:

print("FormulaError:", e)

if \_\_name\_\_ == "\_\_main\_\_":

main()

7.

import pygame

import sys

# Initialize Pygame

pygame.init()

# Set up window

WIDTH, HEIGHT = 400, 400

WINDOW = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption("Shapes Drawing")

# Define colors

BLUE = (0, 0, 255)

GREEN = (0, 255, 0)

BLACK = (0, 0, 0)

# Main loop

while True:

WINDOW.fill((255, 255, 255)) # Fill background with white

# Draw polygon

pygame.draw.polygon(WINDOW, BLUE, [(146, 0), (291, 106), (236, 277), (56, 277), (0, 106)])

# Draw line

pygame.draw.line(WINDOW, GREEN, (60, 300), (120, 300), 4)

# Draw circle

pygame.draw.circle(WINDOW, GREEN, (300, 50), 20)

# Draw ellipse

pygame.draw.ellipse(WINDOW, BLACK, (300, 250, 40, 80), 1)

# Draw rectangle

pygame.draw.rect(WINDOW, BLACK, (150, 300, 100, 50))

# Event handling

for event in pygame.event.get():

if event.type == pygame.QUIT:

pygame.quit()

sys.exit()

pygame.display.update()

8.

import threading

import random

import time

from queue import Queue

# Function to generate a random integer every 1 second

def generate\_random(q):

while True:

random\_int = random.randint(1, 100)

q.put(random\_int)

time.sleep(1)

# Function to compute the square of even numbers

def compute\_square(q):

while True:

num = q.get()

if num % 2 == 0:

print("Square of", num, "is", num\*\*2)

q.task\_done()

# Function to compute the cube of odd numbers

def compute\_cube(q):

while True:

num = q.get()

if num % 2 != 0:

print("Cube of", num, "is", num\*\*3)

q.task\_done()

def main():

# Create a queue to communicate between threads

q = Queue()

# Create and start the threads

thread1 = threading.Thread(target=generate\_random, args=(q,))

thread2 = threading.Thread(target=compute\_square, args=(q,))

thread3 = threading.Thread(target=compute\_cube, args=(q,))

thread1.start()

thread2.start()

thread3.start()

# Join the threads to the main thread

thread1.join()

thread2.join()

thread3.join()

if \_\_name\_\_ == "\_\_main\_\_":

main()

9.

import threading

import time

import random

from queue import Queue

# Vegetable types and their production times

VEGETABLES = ['Tomato', 'Cucumber', 'Carrot']

PRODUCTION\_TIMES = {'Tomato': 3, 'Cucumber': 4, 'Carrot': 5}

# Market capacity

MARKET\_CAPACITY = 10

# Lock for accessing the market

market\_lock = threading.Lock()

# Queue to represent the market

market\_queue = Queue(maxsize=MARKET\_CAPACITY)

# Function for farmers to produce vegetables

def produce\_vegetables():

while True:

vegetable = random.choice(VEGETABLES)

production\_time = PRODUCTION\_TIMES[vegetable]

time.sleep(production\_time)

print(f"Farmer produced {vegetable}")

market\_queue.put(vegetable)

# Function for consumers to purchase vegetables

def purchase\_vegetables():

while True:

vegetable = random.choice(VEGETABLES)

with market\_lock:

if vegetable in list(market\_queue.queue):

market\_queue.get()

print(f"Consumer purchased {vegetable}")

else:

print(f"No {vegetable} available. Consumer is waiting.")

time.sleep(2)

# Start farmer and consumer threads

farmer\_thread = threading.Thread(target=produce\_vegetables)

consumer\_thread = threading.Thread(target=purchase\_vegetables)

farmer\_thread.start()

consumer\_thread.start()

10.

import threading

import time

# Global variable for synchronization

lock = threading.Lock()

# Function to print numbers from 1 to 5

def function\_A():

for num in range(1, 6):

time.sleep(1) # Simulate time-consuming task

with lock:

print(num)

# Function to print letters from 'A' to 'E'

def function\_B():

for char in ['A', 'B', 'C', 'D', 'E']:

time.sleep(1) # Simulate time-consuming task

with lock:

print(char)

# Create two threads, one for each function

thread\_A = threading.Thread(target=function\_A)

thread\_B = threading.Thread(target=function\_B)

# Start both threads

thread\_A.start()

thread\_B.start()

# Wait for both threads to complete their tasks

thread\_A.join()

thread\_B.join()

print("Both threads have completed their tasks.")