**CA-2 GENAI**

**NAME- PIYUSH KHUSHALANI**

**PRN- 21070521051**

**QUESTION 1-**

Generate a model to represent a mathematical equation, write a program to parse the

equation, and ask for input for each parameter.

**CODE-**

import math

# Function to parse the equation and solve it

def solve\_quadratic\_equation(a, b, c):

# Calculate the discriminant

discriminant = b \*\* 2 - 4 \* a \* c

# Check if the discriminant is positive, negative, or zero

if discriminant > 0:

root1 = (-b + math.sqrt(discriminant)) / (2 \* a)

root2 = (-b - math.sqrt(discriminant)) / (2 \* a)

return f"Two distinct real roots: {root1:.2f} and {root2:.2f}"

elif discriminant == 0:

root = -b / (2 \* a)

return f"One real root: {root:.2f}"

else:

real\_part = -b / (2 \* a)

imaginary\_part = math.sqrt(-discriminant) / (2 \* a)

return f"Two complex roots: {real\_part:.2f} ± {imaginary\_part:.2f}i"

# Function to ask for input and parse the equation

def input\_and\_parse\_equation():

print("Quadratic equation format: ax^2 + bx + c = 0")

# Input coefficients a, b, c

a = float(input("Enter coefficient a: "))

b = float(input("Enter coefficient b: "))

c = float(input("Enter constant c: "))

# Solve the quadratic equation

solution = solve\_quadratic\_equation(a, b, c)

print(solution)

# Call the function to run the program

input\_and\_parse\_equation()

**CODE EXPLAINATION-**

Parsing the Equation: The equation is parsed as ax2+bx+c=0ax^2 + bx + c = 0ax2+bx+c=0, where the user is prompted to input values for aaa, bbb, and ccc.

Discriminant: The discriminant Δ=b2−4ac\Delta = b^2 - 4acΔ=b2−4ac determines the nature of the roots:

* If Δ>0\Delta > 0Δ>0, there are two real and distinct roots.
* If Δ=0\Delta = 0Δ=0, there is one real root.
* If Δ<0\Delta < 0Δ<0, there are two complex roots.

Quadratic Formula: The program applies the quadratic formula to find the roots, either real or complex, and prints the solution accordingly.

**OUTPUT-**

Quadratic equation format: ax^2 + bx + c = 0

Enter coefficient a: 1

Enter coefficient b: -3

Enter constant c: 2

Two distinct real roots: 2.00 and 1.00

**QUESTION 2-**

Generate a model in Python for representation of a bank account of type savings and

balance along with transactions of deposit and withdrawals and currently create a program to

generate 100 accounts with Random balance and transactions for no. of months and no. of

transactions with a seed value of amount. Print all 100 accounts with the last balance and

organize them by lowest to highest balance.

**CODE-**

import random

import pandas as pd

# Class to represent a bank account

class BankAccount:

def \_\_init\_\_(self, account\_id, balance=0):

self.account\_id = account\_id

self.balance = balance

self.transactions = []

# Method to deposit money

def deposit(self, amount):

self.balance += amount

self.transactions.append(f"Deposited: {amount}")

# Method to withdraw money

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

self.transactions.append(f"Withdrew: {amount}")

else:

self.transactions.append(f"Failed withdrawal: {amount} (Insufficient balance)")

# Method to return the last balance

def get\_balance(self):

return self.balance

# Method to print account details

def print\_account(self):

print(f"Account ID: {self.account\_id}, Balance: {self.balance}")

for transaction in self.transactions:

print(transaction)

# Function to generate 100 random accounts

def generate\_accounts(num\_accounts, num\_months, num\_transactions, seed\_value):

random.seed(seed\_value) # Set a seed for reproducibility

accounts = []

for i in range(1, num\_accounts + 1):

# Random initial balance between 1000 and 5000

initial\_balance = random.uniform(1000, 5000)

account = BankAccount(f"ACC{i:03}", initial\_balance)

for \_ in range(num\_months):

for \_ in range(num\_transactions):

transaction\_type = random.choice(["deposit", "withdraw"])

amount = random.uniform(100, 1000) # Transaction amounts between 100 and 1000

if transaction\_type == "deposit":

account.deposit(amount)

else:

account.withdraw(amount)

accounts.append(account)

return accounts

# Generate 100 accounts

num\_accounts = 100

num\_months = 12

num\_transactions\_per\_month = 10

seed\_value = 42

accounts = generate\_accounts(num\_accounts, num\_months, num\_transactions\_per\_month, seed\_value)

# Create a dataframe to store account details and sort by final balance

account\_data = {

"Account ID": [account.account\_id for account in accounts],

"Final Balance": [account.get\_balance() for account in accounts]

}

df = pd.DataFrame(account\_data)

df\_sorted = df.sort\_values(by="Final Balance")

# Print sorted accounts by final balance (lowest to highest)

print(df\_sorted)

**CODE EXPLAINATION-**

BankAccount Class:

The BankAccount class represents each bank account with an account\_id and an initial balance. It has methods to deposit and withdraw money, and a method to retrieve the current balance.

Generate Accounts:

The function generate\_accounts() creates 100 random accounts. Each account starts with a random initial balance between ₹1000 and ₹5000.

Over the course of 12 months, each account performs 10 random transactions (either a deposit or a withdrawal) per month. The transaction amounts are randomly selected between ₹100 and ₹1000.

Sorting:

After generating the accounts and simulating the transactions, the program creates a pandas DataFrame to store the account IDs and their final balances. It then sorts the accounts by final balance (lowest to highest).

Output-

The final output is a table of 100 accounts sorted by their final balance, showing each account ID and its last balance.

**OUTPUT-**

Account ID Final Balance

99 ACC100 1050.45

23 ACC024 1253.78

85 ACC086 1342.12

... ... ...

45 ACC046 11328.45

67 ACC068 11457.32

7 ACC008 12489.65