Ex. No: 3 26-04-2023	UIT2201 — Programming and Data Structures

Aim:

To execute the following programs and note the output.

PART – A

1. Implement Vector ADT using python and perform appropriate operations. Use special methods and proper docstrings.

Code:

```
"""
    This module provides functionality for representing a vector as ADT.

    It creates a list of n elements and the elements can be modified by
using special functions.

    These special functions support the operations performed on the vector.

    Created on: 19 Apr 2023

    Original Author: Pranesh Kumar
"""


class Vector:
    def __init__(self, size: int):
        """Constructor of Vector Class

        Args:
            size (int): size of the vector

        Sets the default value of the vector as 0

        """
        self.dim = size
        self.coord = [0 for _ in range(size)]

    def __len__(self) -> int:
        """Returns the length (dimension) of the vector

        Returns:
            int: size of the vector
        """
        return self.dim

    def __getitem__(self, index) -> int:
        """Returns the element at a particular index

        Args:
            index (int): index of the element which is required
```

```
        Raises:
            IndexError: If the index is greater than the size of the vector

        Returns:
            int: Element at the particular index
        """
        if index > self.dim:
            raise IndexError("Vector index out of range")
        else:
            return self.coord[index]

    def __setitem__(self, index: int, value) -> None:
        """Sets the element at the specified index with the new value

        Args:
            index (int): index at which the element is to be replaced
            value (int): new value of the element to be replaced

        Raises:
            IndexError: If the index is greater than the size of the vector

        Returns:
            None
        """
        if index > self.dim:
            raise IndexError("Vector index out of range")
        else:
            self.coord[index] = value

    def __eq__(self, other) -> bool:
        """Checks for equality of 2 vector objects

        Args:
            other (Vector): another vector object where the equality is to
be checked

        Returns:
            bool: True if the dimensions and values are the same, else
False
        """
        if self.dim == other.dim and self.coord == other.coord:
            return True
        else:
            return False

    def __str__(self) -> str:
        return str(self.coord)

    def __add__(self, other):
        """Returns the sum of 2 vectors

        Args:
            other (Vector): second vector object to be added

        Raises:
            IndexError: if indices are not the same

        Returns:
            Vector: Vector object of same dimensions but values as the sum
        """
```

```
        if self.dim == other.dim:
            newvector = Vector(self.dim)
            for idx in range(len(newvector)):
                newvector.coord[idx] = self.coord[idx] + other.coord[idx]
            return newvector
        else:
            raise IndexError("Index is not the same")

    def __sub__(self, other):
        """Returns the difference of 2 vectors

        Args:
            other (Vector): second vector object to be subtracted

        Raises:
            IndexError: if indices are not the same

        Returns:
            Vector: Vector object of same dimensions but values as the
difference
        """
        if self.dim == other.dim:
            newvector = Vector(self.dim)
            for idx in range(len(newvector)):
                newvector.coord[idx] = self.coord[idx] - other.coord[idx]
            return newvector
        else:
            raise IndexError("Index is not the same")

    def multiply_by_scalar(self, scalar: int):
        """Returns a vector multiplied by a scalar

        Args:
            scalar (int): scalar value to be multiplied

        Retunrs:
            Vector: new vector after multiplying with scalar
        """
        newvector = Vector(self.dim)
        for idx in range(len(newvector)):
            newvector.coord[idx] *= scalar
        return newvector


# driver code
if __name__ == "__main__":
    n1 = int(input("Enter the size of vector 1: "))
    v1 = Vector(n1)
    print(len(v1))

    for idx in range(len(v1)):
        v1[idx] = int(input("Enter value: "))

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

    for idx in range(len(v1)):
        print(v1[idx])

    print(v1)

    print("====================")
```

```
n2 = int(input("Enter the size of vector 2: "))
v2 = Vector(n2)
print(len(v2))

for idx in range(len(v2)):
    v2[idx] = int(input("Enter value: "))

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

for idx in range(len(v2)):
    print(v2[idx])

print(v2)

print(v1 == v2)
print(v2 == v1)

print(v1 + v2)
print(v1 - v2)
```

Explanation:

This class represents a vector as an abstract data type (ADT) and provides functionality for vector operations. It creates a vector of size n, where elements can be modified using special functions that support the operations performed on the vector. The class has methods for getting and setting elements at specific indices, finding the length of the vector, checking equality of two vectors, adding, and subtracting vectors, and multiplying a vector by a scalar. The methods raise an IndexError if the index is out of range. The class is a simple implementation of a vector ADT in Python.

Inputs and Output:

```
Enter the size of vector 1: 5
5
Enter value: 2
Enter value: 3
Enter value: 5
Enter value: 7
Enter value: 11
====================
2
3
5
7
11
[2, 3, 5, 7, 11]
====================
Enter the size of vector 2: 5
5
Enter value: 13
Enter value: 17
Enter value: 19
Enter value: 27
Enter value: 23
====================
13
17
19
27
23
[13, 17, 19, 27, 23]
False
False
[15, 20, 24, 34, 34]
[-11, -14, -14, -20, -12]
```

2. Implement a Matrix ADT, in the style of the Vector class presented in the course notes. Instances of Matrix should be able to store a matrix of any size. The ADT takes the number of rows and number of columns as arguments with a default value as (0 & 0) , and sets each matrix entry to zero. Realize the matrix as a list of list Generate a function to create a matrix with random numbers, a function that multiplies two Matrix objects (checking for conformity) and returns a Matrix. Also provide functions for Addition/Subtraction of two matrices and return the resultant matrix. Provide the function to find the determinant of the Matrix Object. Pay special attention to the way you use the results of the multiplication functions! Unless you've written __eq__(self, other), you need to be very careful (i.e., use initialization to get your product matrix back to the calling environment)! Create (and submit) a small Test program to demonstrate your ADT functions

Code:

```
"""
    This module provides functionality for implementing matrix as ADT.

    It provides functions for matrix addition, subtraction and
multiplication.

    It also provides with special functions that support various
operations.

    Created on: 19 Apr 2023

    Original Author: Pranesh Kumar
"""


class Matrix:
    def __init__(self, r: int = 0, c: int = 0):
        """Constructor of Matrix class

        Args:
            r (int, optional): Number of rows of matrix. Defaults to 0.
            c (int, optional): Number of cols of matrix. Defaults to 0.
        """
        self.rows = r
        self.cols = c
        self.values = [[0 for _ in range(c)] for _ in range(r)]

    def __len__(self):
        """Returns the length of the matrix (number of rows)

        Returns:
            int: Number of rows of the matrix
        """
        return self.rows

    def __getitem__(self, idx):
        """Returns the element at a particular index

        Args:
            idx (int): index of the element which is required
```

```
        Raises:
            IndexError: If the index is greater than the size of the vector

        Returns:
            int: Element at the particular index
        """
        if idx > self.__len__():
            raise IndexError("Index out of range")
        else:
            return self.values[idx]

    def __setitem__(self, idx, value):
        """Sets the element at the specified index with the new value

        Args:
            idx (int): index at which the element is to be replaced
            value (int): new value of the element to be replaced

        Raises:
            IndexError: If the index is greater than the size of the vector
        """
        if idx > self.__len__():
            raise IndexError("Index out of range")
        else:
            self.values[idx] = value

    def __str__(self):
        matrixasstring = ""
        for rows in range(len(self)):
            for cols in range(len(self[0])):
                matrixasstring += str(self[rows][cols]) + "\t"
            matrixasstring += "\n"
        return matrixasstring

    def __eq__(self, other):
        """Checks for equality of 2 matrix objects

        Args:
            other (Vector): another vector object where the equality is to
be checked

        Returns:
            bool: True if the dimensions and values are the same, else
False
        """
        if self.__len__() == len(other) and self[0].__len__() ==
len(other[0]) and self.values == other.values:
            return True
        else:
            return False

    def __add__(self, other):
        """Returns the sum of 2 matrices

        Args:
            other (Matrix): second matrix object to be added

        Raises:
            IndexError: if indices are not the same

        Returns:
```

```
            Matrix: Matrix object of same dimensions but values as the sum
        """
        if self.__len__() == len(other) and self[0].__len__() ==
len(other[0]):
            newmatrix = Matrix(self.__len__(), self[0].__len__())
            for rows in range(len(newmatrix)):
                for cols in range(len(newmatrix[0])):
                    newmatrix[rows][cols] = self[rows][cols] +
other[rows][cols]
            return newmatrix
        else:
            raise IndexError("Index is not the same")

    def __sub__(self, other):
        """Returns the sum of 2 matrices

        Args:
            other (Matrix): second matrix object to be subtracted

        Raises:
            IndexError: if indices are not the same

        Returns:
            Matrix: Matrix object of same dimensions but values as the
difference
        """
        if self.__len__() == len(other) and self[0].__len__() ==
len(other[0]):
            newmatrix = Matrix(self.__len__(), self[0].__len__())
            for rows in range(len(newmatrix)):
                for cols in range(len(newmatrix[0])):
                    newmatrix[rows][cols] = self[rows][cols] -
other[rows][cols]
            return newmatrix
        else:
            raise IndexError("Index is not the same")

    def __mul__(self, other):
        """Returns the product of 2 matrices

        Args:
            other (Matrix): second matrix object to be multiplied

        Raises:
            IndexError: if indices are not the same

        Returns:
            Matrix: Matrix object of same dimensions but values as the
product
        """
        if self[0].__len__() != len(other):
            raise IndexError("Index is not the same for multiplication")
        else:
            newmatrix = Matrix(self.__len__(), len(other[0]))
            for i in range(self.__len__()):
                for j in range(len(other[0])):
                    for k in range(len(other)):
                        newmatrix[i][j] += self[i][k] * other[k][j]
            return newmatrix

    def multiply_by_scalar(self, scalar: int):
```

```
        """Returns a matrix multiplied by a scalar

        Args:
            scalar (int): scalar value to be multiplied

        Retunrs:
            Matrix: new matrix after multiplying with scalar
        """
        newmatrix = Matrix(self.__len__(), self[0].__len__())
        for rows in range(len(newmatrix)):
            for cols in range(len(newmatrix[0])):
                newmatrix[rows][cols] = self[rows][cols] * scalar
        return newmatrix


# driver code
if __name__ == "__main__":
    m1 = Matrix(3, 3)
    m2 = Matrix(3, 3)
    print(len(m1))
    print(m1[0])
    print(m1[0][0])
    print("Enter matrix elements row-wise:")
    for rows in range(len(m1)):
        for cols in range(len(m1[0])):
            m1[rows][cols] = int(input())
    print(m1)
    print("Enter matrix elements row-wise:")
    for rows in range(len(m2)):
        for cols in range(len(m2[0])):
            m2[rows][cols] = int(input())
    print(m2)
    print(m1 == m2)
    print(m1 + m2)
    print(m1 - m2)
    print(m1.multiply_by_scalar(2))
    print(m1 * m2)
```

Explanation:

This code defines a Matrix class with functionalities for matrix addition, subtraction, and multiplication. It also has a function for multiplying a matrix with a scalar. The class has methods to access and modify the elements of the matrix. The __str__ method returns the matrix as a string. The __eq__ method checks for equality of two matrices. The class has been implemented with error handling for index out of range and index mismatch. The driver code shows an example of using the Matrix class to perform matrix operations.

Output:

```
3
[0, 0, 0]
0
Enter matrix elements row-wise:
1
2
3
4
5
6
7
8
9
1   2   3
4   5   6
7   8   9

Enter matrix elements row-wise:
10
11
12
13
14
15
16
17
18
10  11  12
13  14  15
16  17  18

 False
 11  13  15
 17  19  21
 23  25  27

 -9  -9  -9
 -9  -9  -9
 -9  -9  -9

 2   4   6
 8   10  12
 14  16  18

 84  90  96
 201 216 231
 318 342 366
```

==========