CECS 229 Programming Assignment #5

Due Date:

Sunday, 11/10 @ 11:59 PM

Submission Instructions:

Complete the programming problems in the file named `pa5.py`. You may test your implementation on your Repl.it workspace by running `main.py`. When you are satisfied with your implementation, submit `pa5.py` to the appropriate CodePost auto-grader folder.

Objectives:

1. Define a matrix data structure with relevant matrix operations.
2. Use a matrix as a transformation function to rotate a 2-dimensional vector.

Notes:

Unless otherwise stated in the FIXME comment, you may not change the outline of the algorithm provided by introducing new loops or conditionals, or by calling any built-in functions that perform the entire algorithm or replaces a part of the algorithm.

Problem 1:

Implement a class `Matrix` that represents $m \times n$ matrix objects with attributes

1. `cols` -columns of the `Matrix` object, as a list of columns (also lists)
2. `rows` -rows of the `Matrix` object, as a list of rows (also lists)

The constructor takes a Python `list` of rows as an argument, and constructs the columns from these rows. If a list is not provided, the parameter defaults to an empty list.

You must implement the following methods in the `Matrix` class:

Setters

- `set_row(self, i, new_row)` - changes the i-th row to be the list `new_row`. If `new_row` is not the same length as the existing rows, then method raises a `ValueError` with the message `Incompatible row length.`
- `set_col(self, j, new_col)` - changes the j-th column to be the list `new_col`. If `new_col` is not the same length as the existing columns, then the method raises a `ValueError` with the message `Incompatible column length.`

- `set_entry(self,i, j, val)` - changes the existing a_{ij} entry in the matrix to `val`. Raises `IndexError` if i does not satisfy $1 \le i \le m$ or j does not satisfy $1 \le j \le n$, where

m = number of rows and n = number of columns.

Getters

- `get_row(self, i)` - returns the i-th row as a list. Raises `IndexError` if i does not satisfy $1 \leq i \leq m$.
- `get_col(self, j)` - returns the j-th column as a list. Raises `IndexError` if j does not satisfy $1 \leq j \leq n$.
- `get_entry(self, i, j)` - returns the existing a_{ij} entry in the matrix. Raises `IndexError` if i does not satisfy $1 \leq i \leq m$ or j does not satisfy $1 \leq j \leq n$, where $m =$ number of rows and $n =$ number of columns.
- `get_columns(self)` - returns the *list of lists* that are the columns of the matrix object
- `get_rows(self)` - returns the *list of lists* that are the rows of the matrix object
- `get_diag(self, k)` - returns the k-th diagonal of a matrix where $k = 0$ returns the main diagonal, $k > 0$ returns the diagonal beginning at $a_{1(k+1)}$, and $k < 0$ returns the diagonal beginning at $a_{(-k+1)1}$. e.g. `get_diag(1)` for an $n \times n$ matrix returns [$a_{12}, a_{23}, a_{34}, \ldots, a_{(n-1)n}$]

Helper methods

- `_construct_rows(self)` - resets the rows of this `Matrix` according to the existing list of lists `self.cols` representing the columns this `Matrix`

- `_construct_cols(self)` - resets the columns of this `Matrix` according to the existing list of lists `self.rows` representing the rows of this `Matrix`

Overloaded operators

In addition to the methods above, the `Matrix` class must also overload the `+`, `-`, and `*` operators to support:

1. `Matrix + Matrix` addition; must return `Matrix` result
2. `Matrix - Matrix` subtraction; must return `Matrix` result
3. `Matrix * scalar` multiplication; must return `Matrix` result
4. `Matrix * Matrix` multiplication; must return `Matrix` result
5. `Matrix * Vec` multiplication; must return `Vec` result
6. `scalar * Matrix` multiplication; must return `Matrix` result

In [ ]:

```
from Vec import Vec

"""-------------------- PROBLEM 1 --------------------"""
class Matrix:

    def __init__(self, rows):
        """
        initializes a Matrix with given rows
        :param rows: the list of rows that this Matrix object has
        """
        self.rows = rows
```

```
        self.cols = []
        self._construct_cols()
        return

    """
  INSERT MISSING SETTERS AND GETTERS HERE
  """

    def _construct_cols(self):
        """
        HELPER METHOD: Resets the columns according to the existing rows
        """
        self.cols = []
        # FIXME: INSERT YOUR IMPLEMENTATION HERE
        return

    def _construct_rows(self):
        """
        HELPER METHOD: Resets the rows according to the existing columns
        """
        self.rows = []
        # FIXME: INSERT YOUR IMPLEMENTATION HERE
        return

    def __add__(self, other):
        """
        overloads the + operator to support Matrix + Matrix
        :param other: the other Matrix object
        :raises: ValueError if the Matrix objects have mismatching dimensions
        :raises: TypeError if other is not of Matrix type
        :return: Matrix type; the Matrix object resulting from the Matrix + Matrix ope
        """
        pass  # FIXME: REPLACE WITH IMPLEMENTATION

    def __sub__(self, other):
        """
        overloads the - operator to support Matrix - Matrix
        :param other:
        :raises: ValueError if the Matrix objects have mismatching dimensions
        :raises: TypeError if other is not of Matrix type
        :return: Matrix type; the Matrix object resulting from Matrix - Matrix operati
        """
        pass  # FIXME: REPLACE WITH IMPLEMENTATION

    def __mul__(self, other):
        """
        overloads the * operator to support
            - Matrix * Matrix
            - Matrix * Vec
            - Matrix * float
            - Matrix * int
        :param other: the other Matrix object
        :raises: ValueError if the Matrix objects have mismatching dimensions
        :raises: TypeError if other is not of Matrix type
        :return: Matrix type; the Matrix object resulting from the Matrix + Matrix ope
        """
        if type(other) == float or type(other) == int:
            print("FIXME: Insert implementation of MATRIX-SCALAR multiplication"
                  )  # FIXME: REPLACE WITH IMPLEMENTATION
        elif type(other) == Matrix:
```

```python
            print("FIXME: Insert implementation of MATRIX-MATRIX multiplication"
                  )  # FIXME: REPLACE WITH IMPLEMENTATION
        elif type(other) == Vec:
            print("FIXME: Insert implementation for MATRIX-VECTOR multiplication"
                  )  # FIXME: REPLACE WITH IMPLEMENTATION
        else:
            raise TypeError(f"Matrix * {type(other)} is not supported.")
        return

    def __rmul__(self, other):
        """
        overloads the * operator to support
            - float * Matrix
            - int * Matrix
        :param other: the other Matrix object
        :raises: ValueError if the Matrix objects have mismatching dimensions
        :raises: TypeError if other is not of Matrix type
        :return: Matrix type; the Matrix object resulting from the Matrix + Matrix ope
        """
        if type(other) == float or type(other) == int:
            print("FIXME: Insert implementation of SCALAR-MATRIX multiplication"
                  )  # FIXME: REPLACE WITH IMPLEMENTATION
        else:
            raise TypeError(f"{type(other)} * Matrix is not supported.")
        return

    '''-------- ALL METHODS BELOW THIS LINE ARE FULLY IMPLEMENTED -------'''

    def dim(self):
        """
        gets the dimensions of the mxn matrix
        where m = number of rows, n = number of columns
        :return: tuple type; (m, n)
        """
        m = len(self.rows)
        n = len(self.cols)
        return (m, n)

    def __str__(self):
        """prints the rows and columns in matrix form """
        mat_str = ""
        for row in self.rows:
            mat_str += str(row) + "\n"
        return mat_str

    def __eq__(self, other):
        """
        overloads the == operator to return True if
        two Matrix objects have the same row space and column space
        """
        if type(other) != Matrix:
            return False
        this_rows = [round(x, 3) for x in self.rows]
        other_rows = [round(x, 3) for x in other.rows]
        this_cols = [round(x, 3) for x in self.cols]
        other_cols = [round(x, 3) for x in other.cols]

        return this_rows == other_rows and this_cols == other_cols

    def __req__(self, other):
```

```
        """
        overloads the == operator to return True if
        two Matrix objects have the same row space and column space
        """
        if type(other) != Matrix:
            return False
        this_rows = [round(x, 3) for x in self.rows]
        other_rows = [round(x, 3) for x in other.rows]
        this_cols = [round(x, 3) for x in self.cols]
        other_cols = [round(x, 3) for x in other.cols]

        return this_rows == other_rows and this_cols == other_cols


"""-------------------- PROBLEM 2 --------------------"""


def rotate_2Dvec(v: Vec, tau: float):
    """
    computes the 2D-vector that results from rotating the given vector
    by the given number of radians
    :param v: Vec type; the vector to rotate
    :param tau: float type; the radians to rotate by
    :return: Vec type; the rotated vector
    """
    pass  # FIXME: REPLACE WITH IMPLEMENTATION
```

Problem 2:

Complete the implementation for the method `rotate_2Dvec(v, tau)` which returns the vector that results from rotating the given 2D-vector `v` by `tau` radians.

INPUT:

- `v` : a `Vec` object representing a 2D vector.
- `tau` : a Python `float` representing the number of radians that the vector `vec` should be rotated.

OUTPUT:

- a `Vec` object that represents the resulting, rotated vector.

In [ ]:

```
def rotate_2Dvec(v: Vec, tau: float):
  """
  computes the 2D-vector that results from rotating the given vector
  by the given number of radians
  :param v: Vec type; the vector to rotate
  :param tau: float type; the radians to rotate by
  :return: Vec type; the rotated vector
  """
  if len(v) != 2:
    raise ValueError(f"rotate_2Dvec is not defined for {len(v)}-D vectors.")

  # FIXME: COMPLETE THE REST OF THE METHOD
```