

✓ Lab 5

Matrix Representation: In this lab you will be creating a simple linear algebra system. In memory, we will represent matrices as nested python lists as we have done in lecture. In the exercises below, you are required to explicitly test every feature you implement, demonstrating it works.

1. Create a `matrix` class with the following properties:

- It can be initialized in 2 ways:
 1. with arguments `n` and `m`, the size of the matrix. A newly instantiated matrix will contain all zeros.
 2. with a list of lists of values. Note that since we are using lists of lists to implement matrices, it is possible that not all rows have the same number of columns. Test explicitly that the matrix is properly specified.
- Matrix instances `M` can be indexed with `M[i][j]` and `M[i,j]`.
- Matrix assignment works in 2 ways:
 1. If `M_1` and `M_2` are matrix instances `M_1=M_2` sets the values of `M_1` to those of `M_2`, if they are the same size. Error otherwise.
 2. In example above `M_2` can be a list of lists of correct size.

```
class Matrix:
    def __init__(self, *args):
        if len(args) == 2: # Initializing with size n and m
            n, m = args
            if n <= 0 or m <= 0:
                raise ValueError("Matrix dimensions must be positive integers.")
            self.data = [[0 for _ in range(m)] for _ in range(n)]
        elif len(args) == 1: # Initializing with a list of lists
            data = args[0]
            if not all(isinstance(row, list) for row in data):
                raise ValueError("Input must be a list of lists.")
            row_length = len(data[0])
            if any(len(row) != row_length for row in data):
                raise ValueError("All rows must have the same number of columns.")
            self.data = data
        else:
            raise ValueError("Invalid number of arguments for Matrix initialization.")

    def __getitem__(self, key):
        if isinstance(key, tuple):
            return self.data[key[0]][key[1]]
        elif isinstance(key, int):
            return self.data[key]
        else:
            raise TypeError("Index must be an integer or a tuple of integers.")

    def __setitem__(self, key, value):
        if isinstance(key, tuple):
            self.data[key[0]][key[1]] = value
        elif isinstance(key, int):
            self.data[key] = value
        else:
            raise TypeError("Index must be an integer or a tuple of integers.")

    def __copy__(self):
        return Matrix([row[:] for row in self.data]) # Deep copy of the matrix

    def assign(self, other):
        if isinstance(other, Matrix):
            if len(self.data) != len(other.data) or len(self.data[0]) != len(other.data[0]):
                raise ValueError("Matrices must be the same size for assignment.")
            self.data = [row[:] for row in other.data] # Deep copy
        elif isinstance(other, list):
            if len(self.data) != len(other) or len(self.data[0]) != len(other[0]):
                raise ValueError("Provided list must match the size of the matrix.")
            self.data = [row[:] for row in other] # Deep copy
        else:
            raise TypeError("Assigned value must be a Matrix or a list of lists.")

    def __str__(self):
        return '\n'.join(['\t'.join(map(str, row)) for row in self.data])
```

```

def main():
    # Test 1: Creating a matrix with dimensions
    matrix1 = Matrix(3, 4)
    print("Matrix 1 (3x4):")
    print(matrix1)

    # Test 2: Creating a matrix with a list of lists
    matrix2 = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    print("\nMatrix 2 (list of lists):")
    print(matrix2)

    # Test 3: Accessing elements
    print("\nAccessing element at (1, 2):", matrix2[1, 2]) # Output: 6

    # Test 4: Modifying an element
    matrix2[1, 2] = 10
    print("\nMatrix 2 after modification:")
    print(matrix2)

    # Test 5: Assignment from one matrix to another
    matrix3 = Matrix(3, 3)
    print("\nMatrix 3 before assignment:")
    print(matrix3)

    matrix3.assign(matrix2)
    print("\nMatrix 3 after assignment from Matrix 2:")
    print(matrix3)

    # Test 6: Attempt to assign incompatible sizes
    try:
        matrix4 = Matrix(2, 2)
        matrix4.assign(matrix2)
    except ValueError as e:
        print("\nError during assignment of incompatible sizes:", e)

    # Test 7: Copying a matrix
    matrix_copy = matrix2.__copy__()
    print("\nCopy of Matrix 2:")
    print(matrix_copy)

if __name__ == "__main__":
    main()

```

→ Matrix 1 (3x4):

0	0	0	0
0	0	0	0
0	0	0	0

Matrix 2 (list of lists):

1	2	3
4	5	6
7	8	9

Accessing element at (1, 2): 6

Matrix 2 after modification:

1	2	3
4	5	10
7	8	9

Matrix 3 before assignment:

0	0	0
0	0	0
0	0	0

Matrix 3 after assignment from Matrix 2:

1	2	3
4	5	10
7	8	9

Error during assignment of incompatible sizes: Matrices must be the same size for assignment.

Copy of Matrix 2:

1	2	3
4	5	10
7	8	9

2. Add the following methods:

- `shape()`: returns a tuple `(n,m)` of the shape of the matrix.
- `transpose()`: returns a new matrix instance which is the transpose of the matrix.
- `row(n)` and `column(n)`: that return the `n`th row or column of the matrix `M` as a new appropriately shaped matrix object.
- `to_list()`: which returns the matrix as a list of lists.
- `block(n_0,n_1,m_0,m_1)` that returns a smaller matrix located at the `n_0` to `n_1` columns and `m_0` to `m_1` rows.
- (Extra credit) Modify `__getitem__` implemented above to support slicing.

```
class Matrix:
    def __init__(self, *args):
        if len(args) == 2: # Initializing with size n and m
            n, m = args
            if n <= 0 or m <= 0:
                raise ValueError("Matrix dimensions must be positive integers.")
            self.data = [[0 for _ in range(m)] for _ in range(n)]
        elif len(args) == 1: # Initializing with a list of lists
            data = args[0]
            if not all(isinstance(row, list) for row in data):
                raise ValueError("Input must be a list of lists.")
            row_length = len(data[0])
            if any(len(row) != row_length for row in data):
                raise ValueError("All rows must have the same number of columns.")
            self.data = data
        else:
            raise ValueError("Invalid number of arguments for Matrix initialization.")

    def __getitem__(self, key):
        if isinstance(key, tuple):
            return self.data[key[0]][key[1]]
        elif isinstance(key, int):
            return self.data[key]
        elif isinstance(key, slice): # Support for slicing
            return [row[key] for row in self.data]
        else:
            raise TypeError("Index must be an integer, tuple of integers, or a slice.")

    def __setitem__(self, key, value):
        if isinstance(key, tuple):
            self.data[key[0]][key[1]] = value
        elif isinstance(key, int):
            self.data[key] = value
        else:
            raise TypeError("Index must be an integer or a tuple of integers.")

    def __copy__(self):
        return Matrix([row[:] for row in self.data]) # Deep copy of the matrix

    def assign(self, other):
        if isinstance(other, Matrix):
            if len(self.data) != len(other.data) or len(self.data[0]) != len(other.data[0]):
                raise ValueError("Matrices must be the same size for assignment.")
            self.data = [row[:] for row in other.data] # Deep copy
        elif isinstance(other, list):
            if len(self.data) != len(other) or len(self.data[0]) != len(other[0]):
                raise ValueError("Provided list must match the size of the matrix.")
            self.data = [row[:] for row in other] # Deep copy
        else:
            raise TypeError("Assigned value must be a Matrix or a list of lists.")

    def __str__(self):
        return '\n'.join(['\t'.join(map(str, row)) for row in self.data])

    def shape(self):
        return (len(self.data), len(self.data[0]) if self.data else 0)

    def transpose(self):
        return Matrix([[self.data[j][i] for j in range(len(self.data))] for i in range(len(self.data[0]))])

    def row(self, n):
        if n < 0 or n >= len(self.data):
            raise IndexError("Row index out of bounds.")
        return Matrix([self.data[n]])
```

```

def column(self, n):
    if n < 0 or n >= len(self.data[0]):
        raise IndexError("Column index out of bounds.")
    return Matrix([[self.data[i][n] for i in range(len(self.data))]])

def to_list(self):
    return [row[:] for row in self.data]

def block(self, n_0, n_1, m_0, m_1):
    if n_0 < 0 or n_1 >= len(self.data) or m_0 < 0 or m_1 >= len(self.data[0]):
        raise IndexError("Block indices out of bounds.")
    return Matrix([row[m_0:m_1 + 1] for row in self.data[n_0:n_1 + 1]])

def main():
    # Test cases for the updated Matrix class
    matrix = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

    print("Original Matrix:")
    print(matrix)

    # Test shape
    print("\nShape of the matrix:", matrix.shape())

    # Test transpose
    transposed = matrix.transpose()
    print("\nTransposed Matrix:")
    print(transposed)

    # Test row
    row2 = matrix.row(1)
    print("\nRow 2 of the matrix:")
    print(row2)

    # Test column
    col1 = matrix.column(1)
    print("\nColumn 1 of the matrix:")
    print(col1)


    # Test to_list
    list_representation = matrix.to_list()
    print("\nList representation of the matrix:")
    print(list_representation)

    # Test block
    block_matrix = matrix.block(0, 1, 0, 1)
    print("\nBlock (0 to 1, 0 to 1):")
    print(block_matrix)

    # Test slicing
    print("\nSlicing the matrix (1:3):")
    sliced = matrix[:2]
    print(sliced)

if __name__ == "__main__":
    main()

```

 Original Matrix:

```

1   2   3
4   5   6
7   8   9

```

Shape of the matrix: (3, 3)

Transposed Matrix:

```

1   4   7
2   5   8
3   6   9

```

Row 2 of the matrix:

```

4   5   6

```

Column 1 of the matrix:

```

2   5   8

```

List representation of the matrix:

```

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```

```
Block (0 to 1, 0 to 1):
```

```
1      2
4      5
```

```
Slicing the matrix (1:3):
```

```
[[1, 2], [4, 5], [7, 8]]
```

3. Write functions that create special matrices (note these are standalone functions, not member functions of your `matrix` class):

- `constant(n,m,c)`: returns a n by m matrix filled with floats of value c .
- `zeros(n,m)` and `ones(n,m)`: return n by m matrices filled with floats of value 0 and 1 , respectively.
- `eye(n)`: returns the n by n identity matrix.

```
def constant(n, m, c):
    """Returns an n by m matrix filled with floats of value c."""
    return [[float(c) for _ in range(m)] for _ in range(n)]

def zeros(n, m):
    """Returns an n by m matrix filled with floats of value 0."""
    return constant(n, m, 0)

def ones(n, m):
    """Returns an n by m matrix filled with floats of value 1."""
    return constant(n, m, 1)

def eye(n):
    """Returns the n by n identity matrix."""
    return [[1.0 if i == j else 0.0 for j in range(n)] for i in range(n)]

# Example usage
if __name__ == "__main__":
    print("Constant Matrix (3x4 filled with 5):")
    print(constant(3, 4, 5))

    print("\nZeros Matrix (3x3):")
    print(zeros(3, 3))

    print("\nOnes Matrix (2x5):")
    print(ones(2, 5))

    print("\nIdentity Matrix (4x4):")
    print(eye(4))
```

```
↳ Constant Matrix (3x4 filled with 5):
[[5.0, 5.0, 5.0, 5.0], [5.0, 5.0, 5.0, 5.0], [5.0, 5.0, 5.0, 5.0]]

Zeros Matrix (3x3):
[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]

Ones Matrix (2x5):
[[1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0]]

Identity Matrix (4x4):
[[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [0.0, 0.0, 0.0, 1.0]]
```

4. Add the following member functions to your class. Make sure to appropriately test the dimensions of the matrices to make sure the operations are correct.

- `M.scalar_mul(c)`: a matrix that is scalar product cM , where every element of M is multiplied by c .
- `M.add(N)`: adds two matrices M and N . Don't forget to test that the sizes of the matrices are compatible for this and all other operations.
- `M.sub(N)`: subtracts two matrices M and N .
- `M.mat_mult(N)`: returns a matrix that is the matrix product of two matrices M and N .
- `M.element_mult(N)`: returns a matrix that is the element-wise product of two matrices M and N .
- `M.equals(N)`: returns true/false if $M=N$.

```
class Matrix:
    def __init__(self, *args):
        if len(args) == 2: # Initializing with size n and m
            n, m = args
            if n <= 0 or m <= 0:
```

```

        raise ValueError("Matrix dimensions must be positive integers.")
    self.data = [[0 for _ in range(m)] for _ in range(n)]
    elif len(args) == 1: # Initializing with a list of lists
        data = args[0]
        if not all(isinstance(row, list) for row in data):
            raise ValueError("Input must be a list of lists.")
        row_length = len(data[0])
        if any(len(row) != row_length for row in data):
            raise ValueError("All rows must have the same number of columns.")
        self.data = data
    else:
        raise ValueError("Invalid number of arguments for Matrix initialization.")

def __getitem__(self, key):
    if isinstance(key, tuple):
        return self.data[key[0]][key[1]]
    elif isinstance(key, int):
        return self.data[key]
    else:
        raise TypeError("Index must be an integer or a tuple of integers.")

def __setitem__(self, key, value):
    if isinstance(key, tuple):
        self.data[key[0]][key[1]] = value
    elif isinstance(key, int):
        self.data[key] = value
    else:
        raise TypeError("Index must be an integer or a tuple of integers.")

def scalarmul(self, c):
    """Returns a new matrix that is the scalar product of c and the matrix."""
    return Matrix([[c * self.data[i][j] for j in range(len(self.data[i]))] for i in range(len(self.data))])

def add(self, other):
    """Adds two matrices."""
    if isinstance(other, Matrix):
        if len(self.data) != len(other.data) or len(self.data[0]) != len(other.data[0]):
            raise ValueError("Matrices must be the same size for addition.")
        return Matrix([[self.data[i][j] + other.data[i][j] for j in range(len(self.data[i]))] for i in range(len(self.data))])
    else:
        raise TypeError("The added object must be a Matrix.")

def sub(self, other):
    """Subtracts two matrices."""
    if isinstance(other, Matrix):
        if len(self.data) != len(other.data) or len(self.data[0]) != len(other.data[0]):
            raise ValueError("Matrices must be the same size for subtraction.")
        return Matrix([[self.data[i][j] - other.data[i][j] for j in range(len(self.data[i]))] for i in range(len(self.data))])
    else:
        raise TypeError("The subtracted object must be a Matrix.")

def mat_mult(self, other):
    """Returns the matrix product of two matrices."""
    if isinstance(other, Matrix):
        if len(self.data[0]) != len(other.data):
            raise ValueError("The number of columns in the first matrix must equal the number of rows in the second matrix.")
        result = [[0 for _ in range(len(other.data[0]))] for _ in range(len(self.data))]
        for i in range(len(self.data)):
            for j in range(len(other.data[0])):
                result[i][j] = sum(self.data[i][k] * other.data[k][j] for k in range(len(other.data)))
        return Matrix(result)
    else:
        raise TypeError("The multiplied object must be a Matrix.")

def element_mult(self, other):
    """Returns the element-wise product of two matrices."""
    if isinstance(other, Matrix):
        if len(self.data) != len(other.data) or len(self.data[0]) != len(other.data[0]):
            raise ValueError("Matrices must be the same size for element-wise multiplication.")
        return Matrix([[self.data[i][j] * other.data[i][j] for j in range(len(self.data[i]))] for i in range(len(self.data))])
    else:
        raise TypeError("The multiplied object must be a Matrix.")

def equals(self, other):
    """Checks if two matrices are equal."""
    if isinstance(other, Matrix):
        return self.data == other.data
    else:
        return False

```

```

    return False

def __str__(self):
    return '\n'.join(['\t'.join(map(str, row)) for row in self.data])

# Example usage and testing

if __name__ == "__main__":
    matrix_a = Matrix([[1, 2], [3, 4]])
    matrix_b = Matrix([[5, 6], [7, 8]])

    print("Matrix A:")
    print(matrix_a)

    print("\nMatrix B:")
    print(matrix_b)

    # Test scalar multiplication
    matrix_c = matrix_a.scalar_mul(2)
    print("\nMatrix A scaled by 2:")
    print(matrix_c)

    # Test addition
    matrix_d = matrix_a.add(matrix_b)
    print("\nMatrix A + Matrix B:")
    print(matrix_d)

    # Test subtraction
    matrix_e = matrix_a.sub(matrix_b)
    print("\nMatrix A - Matrix B:")
    print(matrix_e)

    # Test matrix multiplication
    matrix_f = matrix_a.mat_mult(matrix_b)
    print("\nMatrix A * Matrix B:")
    print(matrix_f)

    # Test element-wise multiplication
    matrix_g = matrix_a.element_mult(matrix_b)
    print("\nElement-wise product of Matrix A and Matrix B:")
    print(matrix_g)

    # Test equality
    print("\nMatrix A equals Matrix B:", matrix_a.equals(matrix_b))
    print("Matrix A equals itself:", matrix_a.equals(matrix_a))

```

```

→ Matrix A:
1      2
3      4

Matrix B:
5      6
7      8

Matrix A scaled by 2:
2      4
6      8

Matrix A + Matrix B:
6      8
10     12

Matrix A - Matrix B:
-4     -4
-4     -4

Matrix A * Matrix B:
19     22
43     50

Element-wise product of Matrix A and Matrix B:
5      12
21     32

Matrix A equals Matrix B: False
Matrix A equals itself: True

```

5. Overload python operators to appropriately use your functions in 4 and allow expressions like:

- $2 \times M$
- $M \times 2$
- $M \times N$
- $M - N$
- $M \times N$
- $M == N$
- $M = N$

6. Demonstrate the basic properties of matrices with your matrix class by creating two 2 by 2 example matrices using your Matrix class and illustrating the following:

$$\begin{aligned}(AB)C &= A(BC) \\ A(B+C) &= AB+AC \\ AB &\neq BA \\ AI &= A\end{aligned}$$

```
class Matrix:
    def __init__(self, *args):
        if len(args) == 2: # Initializing with size n and m
            n, m = args
            if n <= 0 or m <= 0:
                raise ValueError("Matrix dimensions must be positive integers.")
            self.data = [[0 for _ in range(m)] for _ in range(n)]
        elif len(args) == 1: # Initializing with a list of lists
            data = args[0]
            if not all(isinstance(row, list) for row in data):
                raise ValueError("Input must be a list of lists.")
            row_length = len(data[0])
            if any(len(row) != row_length for row in data):
                raise ValueError("All rows must have the same number of columns.")
            self.data = data
        else:
            raise ValueError("Invalid number of arguments for Matrix initialization.")

    def __getitem__(self, key):
        if isinstance(key, tuple):
            return self.data[key[0]][key[1]]
        elif isinstance(key, int):
            return self.data[key]
        else:
            raise TypeError("Index must be an integer or a tuple of integers.")

    def __setitem__(self, key, value):
        if isinstance(key, tuple):
            self.data[key[0]][key[1]] = value
        elif isinstance(key, int):
            self.data[key] = value
        else:
            raise TypeError("Index must be an integer or a tuple of integers.")

    def mat_mult(self, other):
        """Returns the matrix product of two matrices."""
        if isinstance(other, Matrix):
            if len(self.data[0]) != len(other.data):
                raise ValueError("The number of columns in the first matrix must equal the number of rows in the second matrix.")
            result = [[0 for _ in range(len(other.data[0]))] for _ in range(len(self.data))]
            for i in range(len(self.data)):
                for j in range(len(other.data[0])):
                    result[i][j] = sum(self.data[i][k] * other.data[k][j] for k in range(len(other.data)))
            return Matrix(result)
        else:
            raise TypeError("The multiplied object must be a Matrix.")

    def add(self, other):
        """Adds two matrices."""
        if isinstance(other, Matrix):
            if len(self.data) != len(other.data) or len(self.data[0]) != len(other.data[0]):
                raise ValueError("Matrices must be the same size for addition.")
            return Matrix([[self.data[i][j] + other.data[i][j] for j in range(len(self.data[i]))] for i in range(len(self.data))])
        else:
            raise TypeError("The added object must be a Matrix.")
```



```

def equals(self, other):
    """Checks if two matrices are equal."""
    if isinstance(other, Matrix):
        return self.data == other.data
    return False

def __str__(self):
    return '\n'.join(['\t'.join(map(str, row)) for row in self.data])

# Define matrices A, B, and C
A = Matrix([[1, 2], [3, 4]])
B = Matrix([[5, 6], [7, 8]])
C = Matrix([[9, 10], [11, 12]])

# Display matrices
print("Matrix A:")
print(A)
print("\nMatrix B:")
print(B)
print("\nMatrix C:")
print(C)

# Property 1: (AB)C = A(BC)
AB = A.mat_mult(B)
AB_C = AB.mat_mult(C)
BC = B.mat_mult(C)
A_BC = A.mat_mult(BC)

print("\n(AB)C:")
print(AB_C)
print("\nA(BC):")
print(A_BC)

# Check if (AB)C == A(BC)
print("\n(AB)C == A(BC):", AB_C.equals(A_BC))

# Property 2: A(B+C) = AB + AC
B_plus_C = B.add(C)
A_B_plus_C = A.mat_mult(B_plus_C)
AB_plus_AC = A.mat_mult(B).add(A.mat_mult(C))

print("\nA(B + C):")
print(A_B_plus_C)
print("\nAB + AC:")
print(AB_plus_AC)

# Check if A(B + C) == AB + AC
print("\nA(B + C) == AB + AC:", A_B_plus_C.equals(AB_plus_AC))

# Property 3: AB ≠ BA
BA = B.mat_mult(A)
print("\nAB:")
print(AB)
print("\nBA:")
print(BA)
print("\nAB == BA:", AB.equals(BA))

# Property 4: Identity Matrix I
I = Matrix([[1, 0], [0, 1]]) # 2x2 Identity Matrix
AI = A.mat_mult(I)

print("\nAI (A * Identity):")
print(AI)
print("\nA equals AI:", A.equals(AI))

```

↗ Matrix A:

1	2
3	4

Matrix B:

5	6
7	8

Matrix C:

9	10
---	----

```
11      12
```

```
(AB)C:  
413      454  
937      1030
```

```
A(BC):  
413      454  
937      1030
```

```
(AB)C == A(BC): True
```

```
A(B + C):  
50        56  
114       128
```

```
AB + AC:  
50        56  
114       128
```

```
A(B + C) == AB + AC: True
```

```
AB:  
19        22  
43        50
```

```
BA:  
23        34  
31        46
```

```
AB == BA: False
```

```
AI (A * Identity):  
1          2  
3          4
```

```
A equals AI: True
```