ETGG1803	Lab 8: **Matrices**	Points: **40**
Assigned: **4/5/2017**	Due: **Before noon on 4/14/2017**	

This lab is similar to Lab1 and 2 – I'll give you the test code and you deduce what operators you need to overload. Start early and ask questions! The inverse operation at the end and the string precision are bonus (12 points and 4 points, respectively).

put the MatrixN class in a new module called ***matrix.py***

Your MatrixN should have [only] these attributes (make sure you name them this way[1]):

```
self.mRows          # The number of rows
self.mCols          # The number of columns
self.mData          # A list of VectorN's (one for each row)
```

Sample test code (with desired output in comments)

```
from matrix import *

print("Matrix construction\n====================")
a = MatrixN(4, 3)
b = MatrixN(2, 3, (3.0145, 7.2983, "2.314", 1.9, -2, 4.37562))
# c = MatrixN(4, 3, (3.0145, 7.2983, "2.314", 1.9, -2, 4.37562))     # ValueError: You must pass exactly 12 values
                                                                     # in the data array to populate this 4 x 3 MatrixN
I = identity(3)
print(I)                                                  # /1.0     0.0     0.0\
                                                          # |0.0     1.0     0.0|
                                                          # \0.0     0.0     1.0/

print(a)                                                  # /0.0     0.0     0.0\
                                                          # |0.0     0.0     0.0|
                                                          # |0.0     0.0     0.0|
                                                          # \0.0     0.0     0.0/


print("Item Accessing\n===============")

print("b[0, 0] = " + str(b[0, 0]))                        # b[0, 0] = 3.0145
# print("b[10, 4] = " + str(b[10, 4]))                    # IndexError: list index out of range
print("b[1, 2] = " + str(b[1, 2]) + "\n")                 # b[1, 2] = 4.37562

c = a.copy()

a[0, 2] = 99
a[1, 0] = "100.2"
a[3, 1] = 101.99999
print(a)                                                  # /0.0    0.0    99.0 \
                                                          # |100.2    0.0    0.0 |
                                                          # |0.0    0.0    0.0 |
```

[1] I *probably* will use a unit-tester to grade this one…

```
                                                    # \0.0   101.99999   0.0 /

print(c)                                            # /0.0    0.0    0.0\
                                                    # |0.0    0.0    0.0|
                                                    # |0.0    0.0    0.0|
                                                    # \0.0    0.0    0.0/

v = a.getRow(0)
# v = a.getRow(4)                                   # IndexError: list index out of range
v[0] = 123.4
print("v = " + str(v))                              # v = <Vector3: 123.4, 0.0, 99.0>
# v = a.getColumn(2)                                # IndexError: list assignment index out of range
print(a.getColumn(0))                               # <Vector4: 0.0, 100.2, 0.0, 0.0>


print(a)                                            # /0.0   0.0   99.0 \
                                                    # |100.2   0.0   0.0 |
                                                    # |0.0   0.0   0.0 |
                                                    # \0.0   101.99999   0.0 /

b.setRow(0, VectorN(4, 5, 6))
b.setColumn(2, VectorN(7, 8))
# b.setRow(0, VectorN(4, 5))                        # ValueError: Invalid row argument (must be a VectorN with size = 3)
# b.setRow(2, VectorN(4, 5, 6))                     # IndexError: list index out of range


print("Multiplication\n==============")
print(b * a.transpose())                            # /693.0    400.8   0.0    509.99995\
                                                    # /792.0    190.38  0.0     -203.99998/


print(b.transpose())                                # /4.0   1.9 \
                                                    # |5.0   -2.0 |
                                                    # \7.0   8.0 /

print(b)                                            # /4.0   5.0   7.0 \
                                                    # \1.9   -2.0   8.0 /

print(b * 3)                                        # /12.0   15.0   21.0 \
                                                    # \5.699999999999999   -6.0   24.0 /

print(3 * b)                                        # /12.0   15.0   21.0 \
                                                    # \5.699999999999999   -6.0   24.0 /

# Note: It is assumed that if you multiply a matrix * vector, you are using a right-handed system, so the vector
#                 is actually a n x 1 matrix.
#       If you do vector * matrix, you are assumed to be using a left-handed system, so the vector
#                 is actually a 1 x n matrix
v = b * VectorN(5, 4, 2)                            # Right-handed vector
print(v)                                            # <Vector2: 54.0, 17.5>
```

```
# We now want to support vector * matrix.  But...our VectorN class will (currently) complain if you multiply by
# anything other than a scalar.  We *could* fix this problem by putting matrix-multiplication in VectorN.__mul__, but
# for this lab, I want you to instead change the exception in VectorN.__mul__ from:
#        raise ValueError("....")
# to this:
#        return NotImplemented
# This will still error (see the line right below this), but will allow the __rmul__ method of MatrixN to be called
# v = VectorN(2, 1) * "abc"                                 # TypeError: can't multiply sequence by non-int of type Vector2
v = VectorN(5, 4) * b                                       # Left-handed vector (hint: you'll put this code in MatrixN.__rmul__)
print(v)                                                    # <Vector3: 27.6, 17.0, 67.0>


# @@@@@@@@@@@@@@@@@@@@
# BONUS
# @@@@@@@@@@@@@@@@@@@@
# @@@@@@@@@@@@@@@@@@@@
# BONUS
# @@@@@@@@@@@@@@@@@@@@
print("Inverse and Precision\n======================")
c = MatrixN(2, 2, (1.234567, 2.345678, 3.456789, 4.567891))
print(c)                                                    # /1.234567   2.345678 \
                                                            # \3.456789   4.567891 /

MatrixN.sStrPrecision = 2  # Makes all elements of all MatrixN's display with 2
#    decimals when using MatrixN.__str__
print(c)                                                    # /1.23  2.35\
                                                            # \3.46  4.57/

MatrixN.sStrPrecision = None  # Makes all elements of all MatrixN's display with unlimited
#    decimals when using MatrixN.__str__
print(c)                                                    # /1.234567   2.345678 \
                                                            # \3.456789   4.567891 /

print(b)                                                    # /4.0   5.0   7.0 \
                                                            # \1.9   -2.0   8.0 /
print(b.inverse())                                          # None (non-square matrices don't have an inverse)
c = MatrixN(3, 3, (4, 2, 0, 3, 7, 0, 2, 1, 0))
print(c.inverse())                                          # None (this matrix is square, but fails to find a pivot in col#3)

c = MatrixN(3, 3, (0, 1, 2, 1, 0, 3, 4, -3, 8))
print(c)                                                    # /0.0   1.0   2.0 \
                                                            # |1.0   0.0   3.0 |
                                                            # \4.0   -3.0   8.0/
cI = c.inverse()
print(cI)                                                   # /-4.5   7.0   -1.5 \
                                                            # |-2.0   4.0   -1.0 |
                                                            # \1.5   -2.0   0.5 /
# Test the inverse
print(c * cI)                                               # /1.0   0.0   0.0 \
                                                            # |0.0   1.0   0.0 |
                                                            # \0.0   0.0   1.0 /
```