Waleed Akram

20p-0640

Numerical Computing Assignment

Matrix and Numpy

In [123]:

```
## Matrix representation Dense

class Matrix:   # matrix with two dimensions

    def __init__(self, dims, fill):
        self.rows = dims[0]  #set of rows
        self.cols = dims[1]   #set of columns

        self.A = [
            [fill] * self.cols             # for each row, this many columns and fill each
            for i in range(self.rows)      # create this many rows
        ]
```

In [124]:

```
m = Matrix((3, 4), 2.0)   # creating matrix with 3 x 4 rows,columns and each value 2.0
```

In [125]:

```
print(m)  # output shows we need some specific matrix representation
```

```
<__main__.Matrix object at 0x000001C5FE49C2E0>
```

In [126]:

```
# code for matrix output
def __str__(self):
    rows = len(self.A) # Get the number of rows
    ret = ''

    for i in range(rows): # whole loop is for one row
        cols = len(self.A[i])

        for j in range(cols):
            ret += str(self.A[i][j]) + "\t"# output on each column every individual element
        ret += "\n"

    return ret


Matrix.__str__ = __str__
```

In [127]:

```
print(m)
```

```
2.0     2.0     2.0     2.0
2.0     2.0     2.0     2.0
2.0     2.0     2.0     2.0
```

In [128]:

```
%time n = Matrix((100, 100), 0.0) # magic command tells the time to create a matrix
```

```
CPU times: total: 0 ns
Wall time: 999 µs
```

In [129]:

```
from sys import getsizeof# tells the memory taken by a matrix
print(getsizeof(m))
print(getsizeof(n))
```

```
48
48
```

In [130]:

```
!pip install pympler
```

```
Requirement already satisfied: pympler in c:\users\wachattha\anaconda3\lib\s
ite-packages (1.0.1)
```

In [131]:

```
from pympler.asizeof import asizeof
```

In [132]:

```
asizeof(m), asizeof(n)   # tells the memory taken by a matrix
```

Out[132]:

```
(760, 86896)
```

In [133]:

```
n = Matrix((100, 50), 0.0)
```

In [134]:

```
asizeof(m), asizeof(n)  # tells the memory taken by a matrix
```

Out[134]:

```
(760, 46928)
```

In [135]:

```
dim = 5000  # dimensions for a matrix
```

In [136]:

```
%time  m = Matrix((dim, dim), 0.0)  # calculating time taken by a matrix
```

```
CPU times: total: 281 ms
Wall time: 294 ms
```

In [137]:

```
%time  m = Matrix((150, 90), 0.0)   # calculating time taken by following matrix
```

```
CPU times: total: 93.8 ms
Wall time: 102 ms
```

In [138]:

```
size = asizeof(m) / (1024 * 1024)  #

print("{:.2f} MBs".format(size))
```

```
0.11 MBs
```

In [139]:

```
size = asizeof(m) / (1000 * 2000)  #

print("{:.2f} MBs".format(size))
```

```
0.06 MBs
```

In [140]:

```
size = asizeof(m) / (2000 * 2000)  #

print("{:.2f} MBs".format(size))
```

```
0.03 MBs
```

In [141]:

```
# recall that we can get values from our matrix using indices
def get(self, i, j):

    # Error checking  exception checking
    if i < 0 or i > self.rows:
        raise ValueError("Row index out of range.")
    if j < 0 or j > self.cols:
        raise ValueError("Column index out of range.")

    # Value return

    return self.A[i][j]

Matrix.get = get
```

In [142]:

```
m.get(1, 2)
```

Out[142]:

```
0.0
```

In [143]:

```
m.get(15, 0)
```

Out[143]:

```
0.0
```

In [144]:

```
m.get(1, 10)
```

Out[144]:

```
0.0
```

Matrix representation (sparse)

In [145]:

```
class Matrix:

    def __init__(self, dims):
        self.rows = dims[0]
        self.cols = dims[1]
        self.vals = {}  # empty dictionary

        # Let's assume for a minute that fill is 0

    # obviously need a new __str__ here ....
```

In [146]:

```
def set(self, i, j, val):
    self.vals[(i, j)] = val


Matrix.set = set
```

In [147]:

```
# sparse implementation of get
def get(self, i, j):

    # Error checking
    if i < 0 or i > self.rows:
        raise ValueError("Row index out of range.")
    if j < 0 or j > self.cols:
        raise ValueError("Column index out of range.")


    # value return
    if (i, j) in self.vals:
        return self.vals[(i, j)] # if value matching non zero than return present element
    # else return 0.0
    return 0.0

Matrix.get = get
```

In [148]:

```
m = Matrix((5, 5))
```

In [149]:

```
print(m.vals)
```

```
{}
```

In [150]:

```
m.get(1, 1)
```

Out[150]:

```
0.0
```

In [151]:

```
m.get(1, 0)
```

Out[151]:

```
0.0
```

In [152]:

```
m.set(1, 2, 15.0)
```

In [153]:

```
m.get(1, 2)
```

Out[153]:

```
15.0
```

In [154]:

```
m.vals
```

Out[154]:

```
{(1, 2): 15.0}
```

In [155]:

```
m.set(1, 4, 29.9)
```

In [156]:

```
m.get(1, 4)
```

Out[156]:

```
29.9
```

In [157]:

```
dim = 1500 # 5_000_0000_000
m = Matrix((dim, dim))
```

In [158]:

```
asizeof(m)
```

Out[158]:

```
416
```

In [159]:

```
dim =1000
m = Matrix((dim, dim))
```

In [160]:

```
asizeof(m)
```

Out[160]:

```
416
```

Numpy is a python library to perform math functions

Numpy

In [161]:

```
import numpy as np
```

In [162]:

```
np.random.seed(1337)  # to reproduce the same random number again  (reproduceable)
```

In [163]:

```
x = np.array(  [1, 4, 3]  )  # fixed size and fixed data type for efficiency
x
```

Out[163]:

```
array([1, 4, 3])
```

In [164]:

```
y = np.array([ [1, 4, 3],
               [9, 2, 7] ] )# matrix  >> looks like 2-D list
y
```

Out[164]:

```
array([[1, 4, 3],
       [9, 2, 7]])
```

In [165]:

```
x.shape  # it has one dimension with three elements   >> rank one tensor
```

Out[165]:

```
(3,)
```

In [166]:

```
y.shape   # it has 2 x 3 dimension with three elements
```

Out[166]:

```
(2, 3)
```

In [167]:

```
z = np.array([ [1, 4, 3] ])   # can be a row vector
```

In [168]:

```
z.shape
```

Out[168]:

```
(1, 3)
```

In [169]:

```
z = np.arange(1, 2000, 1)  # start, end, step
z[:10]
```

Out[169]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [170]:

```
z.shape
```

Out[170]:

```
(1999,)
```

In [171]:

```
np.arange(0.5, 3, 0.5)
```

Out[171]:

```
array([0.5, 1. , 1.5, 2. , 2.5])
```

In [172]:

```
np.arange(0.5, 10, 1).shape
```

Out[172]:

```
(10,)
```

In [173]:

```
np.arange(0.5, 10, 1).reshape(5, 2).shape
```

Out[173]:

```
(5, 2)
```

In [174]:

```
np.arange(0.5, 5, 1).reshape(5, 3).shape
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [174], in <cell line: 1>()
----> 1 np.arange(0.5, 5, 1).reshape(5, 3).shape

ValueError: cannot reshape array of size 5 into shape (5,3)
```

In [175]:

```
# Evenly spaced but we don't know the step
np.linspace(3, 9, 10)
```

Out[175]:

```
array([3.        , 3.66666667, 4.33333333, 5.        , 5.66666667,
       6.33333333, 7.        , 7.66666667, 8.33333333, 9.        ])
```

In [176]:

```
print(x)
print(x[1])
print(x[1:])
```

```
[1 4 3]
4
[4 3]
```

In [177]:

```
print(y)
y[0, 1]   #
```

```
[[1 4 3]
 [9 2 7]]
```

Out[177]:

```
4
```

In [178]:

```
y[:, 1]
```

Out[178]:

```
array([4, 2])
```

In [179]:

```
y[:, [1, 2]]
```

Out[179]:

```
array([[4, 3],
       [2, 7]])
```

In [180]:

```
import numpy as np # importing Numpy as np
```

In [181]:

```
np.random.seed(1337) # will generate random number sequence everytime
```

In [182]:

```
## Basics of Matrices
```

In [183]:

```
a = np.array(  [561, 564, 5343]  ) # create numpy array homogeneous and fixed
a
```

Out[183]:

```
array([ 561,  564, 5343])
```

In [184]:

```
b = np.array([ [134, 654, 73],
               [92, 52, 754] ] ) # create numpy array homogeneous and fixed
b
```

Out[184]:

```
array([[134, 654,  73],
       [ 92,  52, 754]])
```

In [185]:

```
b.shape # Rank -1 Tensor
```

Out[185]:

```
(2, 3)
```

In [186]:

```
b.shape # can let 2 row 3 column
```

Out[186]:

```
(2, 3)
```

In [187]:

```
c = np.array(  [ [1, 4, 3] ]  )   # list into list
```

In [188]:

```
c.shape # 1 row 3 column
```

Out[188]:

```
(1, 3)
```

In [189]:

```
c = np.arange(1, 200, 1)  # start, end, step
c[:10]
```

Out[189]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [190]:

```
c[:]
```

Out[190]:

```
array([  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,
        14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,
        27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,
        40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,
        53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  65,
        66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,
        79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,
        92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103, 104,
       105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117,
       118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
       131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
       144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156,
       157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169,
       170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182,
       183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195,
       196, 197, 198, 199])
```

In [191]:

```
c.shape
```

Out[191]:

```
(199,)
```

In [192]:

```
np.arange(0.5, 3, 0.6)# start end step
```

Out[192]:

```
array([0.5, 1.1, 1.7, 2.3, 2.9])
```

In [193]:

```
np.arange(0.5, 10, 1).shape # RAnk-1 Tensor
```

Out[193]:

```
(10,)
```

In [194]:

```
np.arange(0.5, 10, 1).reshape(5, 2).shape
```

Out[194]:

```
(5, 2)
```

In [195]:

```
np.arange(0.5, 10, 1).reshape(5, 3).shape  # bcz 10 elements can't reshape into 5X3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [195], in <cell line: 1>()
----> 1 np.arange(0.5, 10, 1).reshape(5, 3).shape

ValueError: cannot reshape array of size 10 into shape (5,3)
```

In [196]:

```
# Evenly spaced but we don't know the step
np.linspace(2, 9, 12) # start , end , step step calculate by self
```

Out[196]:

```
array([2.        , 2.63636364, 3.27272727, 3.90909091, 4.54545455,
       5.18181818, 5.81818182, 6.45454545, 7.09090909, 7.72727273,
       8.36363636, 9.        ])
```

In [197]:

```
print(a) # total a
print(a[1]) # first index
print(a[1:])
```

```
[ 561  564 5343]
564
[ 564 5343]
```

In [198]:

```
print(b)
b[0, 1]   # 0th index row and 1th index column element
```

```
[[134 654  73]
 [ 92  52 754]]
```

Out[198]:

```
654
```

In [199]:

```
b[:, 1] # 1st column
```

Out[199]:

```
array([654,  52])
```

In [200]:

```
b # output full matrix
```

Out[200]:

```
array([[134, 654,  73],
       [ 92,  52, 754]])
```

In [201]:

```
b[:, [1, 2]]
```

Out[201]:

```
array([[654,  73],
       [ 52, 754]])
```

In [202]:

```
## Array Operations
```

In [203]:

```
np.zeros((6, 6)) # numpy built in fill all elements as zero
```

Out[203]:

```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

In [204]:

```
np.ones((3, 6)) # built in function this will fill all ones
```

Out[204]:

```
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

In [205]:

```
a = np.arange(23, 34) # creating an array start,end having 1 step by default
a
```

Out[205]:

```
array([23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33])
```

In [206]:

```
a.shape # Rank -1 tensor
```

Out[206]:

```
(11,)
```

In [207]:

```
a[3] = 7
a
```

Out[207]:

```
array([23, 24, 25,  7, 27, 28, 29, 30, 31, 32, 33])
```

In [208]:

```
a[:3] = 1   # Assign to multiple locations
a
```

Out[208]:

```
array([ 1,  1,  1,  7, 27, 28, 29, 30, 31, 32, 33])
```

In [209]:

```
a[4:7] = [945, 856, 3447]
a
```

Out[209]:

```
array([   1,    1,    1,    7,  945,  856, 3447,   30,   31,   32,   33])
```

In [210]:

```
# This is most useful
b = np.zeros((4, 7)) # 4x7 matrix
b[0, 0] = 111 # will assign value at 0 row 0 column
b[0, 6] = 222 # will assign value at 0 row 6 column
b[1, 4] = 434 # will assign value at 1 row 4 column
b[3, 3] = 423 # will assign value at 3 row 3 column
b[2, 5] = 224 # will assign value at 0 row 0 column
b
```

Out[210]:

```
array([[111.,   0.,   0.,   0.,   0.,   0., 222.],
       [  0.,   0.,   0.,   0., 434.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0., 224.,   0.],
       [  0.,   0.,   0., 423.,   0.,   0.,   0.]])
```

In [211]:

```
b.shape # 4 rows 7 column
```

Out[211]:

```
(4, 7)
```

In [212]:

```
b + 2 # add 2 in every element of 2
```

Out[212]:

```
array([[113.,   2.,   2.,   2.,   2.,   2., 224.],
       [  2.,   2.,   2.,   2., 436.,   2.,   2.],
       [  2.,   2.,   2.,   2.,   2., 226.,   2.],
       [  2.,   2.,   2., 425.,   2.,   2.,   2.]])
```

In [213]:

```
8 * b # multiply every element by 9
```

Out[213]:

```
array([[ 888.,    0.,    0.,    0.,    0.,    0., 1776.],
       [   0.,    0.,    0.,    0., 3472.,    0.,    0.],
       [   0.,    0.,    0.,    0.,    0., 1792.,    0.],
       [   0.,    0.,    0., 3384.,    0.,    0.,    0.]])
```

In [214]:

```
b ** 3 # raise every elemrnt to the power of 3
```

Out[214]:

```
array([[ 1367631.,        0.,        0.,        0.,        0.,        0.,
         10941048.],
       [       0.,        0.,        0.,        0., 81746504.,        0.,
               0.],
       [       0.,        0.,        0.,        0.,        0., 11239424.,
               0.],
       [       0.,        0.,        0., 75686967.,        0.,        0.,
               0.]])
```

In [215]:

```
sum(b) # column wise sum python built in
```

Out[215]:

```
array([111.,   0.,   0., 423., 434., 224., 222.])
```

In [216]:

```
b.sum() # numpy sum all values in matrix
```

Out[216]:

```
1414.0
```

In [217]:

```
b
```

Out[217]:

```
array([[111.,   0.,   0.,   0.,   0.,   0., 222.],
       [  0.,   0.,   0.,   0., 434.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0., 224.,   0.],
       [  0.,   0.,   0., 423.,   0.,   0.,   0.]])
```

In [218]:

```
print(b)
```

```
[[111.   0.   0.   0.   0.   0. 222.]
 [  0.   0.   0.   0. 434.   0.   0.]
 [  0.   0.   0.   0.   0. 224.   0.]
 [  0.   0.   0. 423.   0.   0.   0.]]
```

In [ ]:

```

```

In [219]:

```
b.sum(axis=0).shape # RAnk -1 tensor
```

Out[219]:

```
(7,)
```

In [220]:

```
b.sum(axis=1).shape
```

Out[220]:

```
(4,)
```

In [221]:

```
b = np.array([[2, 2], [3, 4]])
d = np.array([[4, 5], [6, 8]])
```

In [222]:

```
print(b)
print(d)
```

```
[[2 2]
 [3 4]]
[[4 5]
 [6 8]]
```

In [223]:

```
b + d # add both matrix
```

Out[223]:

```
array([[ 6,  7],
       [ 9, 12]])
```

In [224]:

```
b * d
```

Out[224]:

```
array([[ 8, 10],
       [18, 32]])
```

In [225]:

```
b.dot(d)  # multiply both matrix
```

Out[225]:

```
array([[20, 26],
       [36, 47]])
```

In [226]:

```
b ** d
```

Out[226]:

```
array([[   16,    32],
       [  729, 65536]], dtype=int32)
```

In [227]:

```
b.T # Transpose of matrix
```

Out[227]:

```
array([[2, 3],
       [2, 4]])
```

In [228]:

```
a.shape # Rank -1 tensor
```

Out[228]:

```
(11,)
```

In [229]:

```
a.T
```

Out[229]:

```
array([   1,    1,    1,    7,  945,  856, 3447,   30,   31,   32,   33])
```

In [230]:

```
a.T.shape
```

Out[230]:

```
(11,)
```

In [231]:

```
a.reshape(6,1).T.shape
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [231], in <cell line: 1>()
----> 1 a.reshape(6,1).T.shape

ValueError: cannot reshape array of size 11 into shape (6,1)
```

In [232]:

```
# Numpy has "broadcasting" or "mapping" functions
print(np.sqrt(36))

# works on both scalars and arrays
x1 = [1, 4, 9, 16]
np.sqrt(x1)
```

```
6.0
```

Out[232]:

```
array([1., 2., 3., 4.])
```

In [233]:

```
# Checking conditions
x = np.array([1, 2, 4, 5, 9, 3])
y = np.array([0, 2, 3, 1, 2, 3])
```

In [234]:

```
x > 2 # check element one by one and compare by 2
```

Out[234]:

```
array([False, False,  True,  True,  True,  True])
```

In [122]:

```
x > y # element by element comparision
```

Out[122]:

```
array([ True, False,  True,  True,  True, False])
```

In [ ]:

```

```