DATA SCIENCE WITH PYTHON

CS 498 (SPECIAL TOPICS IN COMPUTER SCIENCE)

BASIC ALGEBRA

Dr. Malak Abdullah

Computer Science Department

Jordan University of Science and Technology

NUMPY

- Numpy array creation
- Array access and operations
- Basic linear algebra

NUMPY

- Stands for Numerical Python
- an efficient multi-dimensional container for generic data
- Is the fundamental package required for high performance computing and data analysis
- It provides
  - ndarray for creating multiple dimensional arrays
  - Standard math functions for fast operations on entire arrays of data without having to write loops
  - Tools for reading/writing array data
  - Linear algebra tools
  - etc.

NDARRAY VS LIST OF LISTS

- Say you have grades of three exams (2 midterms and 1 final) in a class of 10 students.
  - grades =

```
[[79, 95, 60],
 [95, 60, 61],
 [99, 67, 84],
 [76, 76, 97],
 [91, 84, 98],
 [70, 69, 96],
 [88, 65, 76],
 [67, 73, 80],
 [82, 89, 61],
 [94, 67, 88]]
```

- How to get final exam grade of student 0?
  - grades[0][2]
- How to get grades of student 2?
  - grades[2]
- How to get grades of all students in midterm 1?
- How to get midterm grades of the first three students (or all female students, or those who failed final)?
- How to get mean grade of each exam?
- How to get (weighted) average exam grade for each student?

propython.py

```python
# easiest way to create an array is by using an array function
import numpy as np        # I am importing numpy as np

scores = [89, 56.34, 76,89, 98]
first_arr =np.array(scores)
print (first_arr)
print (first_arr.dtype)  # .dtype return the data type of the array object

print (scores)
print (type(scores), type(scores[0]), type(scores[1]))


# Nested lists with equal length, will be converted into a multidimensional array
print ("*"*30)
scores_1 = [[34,56,23,89], [11,45,76,34]]
second_arr = np.array(scores_1)
print (second_arr)
print ("without numpy", scores_1)
print (second_arr.ndim)  #.ndim gives you the dimensions of an array.
print (second_arr.shape) #(number of rows, number of columns)
print (second_arr.dtype)

```

```
[89.   56.34 76.   89.   98.  ]
float64
[89, 56.34, 76, 89, 98]
<class 'list'> <class 'int'> <class 'float'>
******************************
[[34 56 23 89]
 [11 45 76 34]]
without numpy [[34, 56, 23, 89], [11, 45, 76, 34]]
2
(2, 4)
int64
```

NDARRAY VS LIST OF LISTS

- gArray = np.array(examGrades)

```
In [3]: gArray
Out[3]:
array([[79, 95, 60],
[95, 60, 61],
[99, 67, 84],
...,
[67, 73, 80],
[82, 89, 61],
[94, 67, 88]])
```

```
In [5]: gArray[0,2].          # gArray[0][2]
Out[5]: 60

In [7]: gArray[2,:]           #gArray[2]
Out[7]: array([99, 67, 84])

In [8]: gArray[:, 0]
Out[8]: array([79, 95, 99, 76, 91, 70, 88, 67, 82, 94])

In [9]: gArray[:3, :2]
Out[9]:
array([[79, 95],
[95, 60],
[99, 67]])
```

NDARRAY

- ndarray is used for storage of homogeneous data
  - i.e., all elements must be the same type
- Every array must have a shape
- And a dtype
- Supports convenient slicing, indexing and efficient vectorized computation
  - Avoid for loops, and much more efficient

```
In [15]: type(gArray)
Out[15]: numpy.ndarray

In [16]: gArray.ndim
Out[16]: 2

In [17]: gArray.shape
Out[17]: (10, 3)

In [18]: gArray.dtype
Out[18]: dtype('int32')
```

propython.py

```
23  print ("*"*30)
24  x = np.zeros(10)       # returns a array of zeros, the same applies for np.ones(10)
25  print (x)
26
27  print ("*"*30)
28  x = np.zeros((3,10))  # returns a array of zeros, the same applies for np.ones(10)
29  print (x)
30
31  print ("*"*30)
32  print (np.arange(15))
33
34  print ("*"*30)
35  print (np.eye(4))
36
37  print ("*"*30)
```

```
******************************
[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. 0. 0. 0. 0.]]
******************************
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
******************************
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
******************************
```

CREATING NDARRAYS

- np.array
- np.zeros
- np.ones
- np.eye
- np.arange
- np.random

```
In [65]: np.array([[0,1,2],[2,3,4]])
Out[65]:
array([[0, 1, 2],
[2, 3, 4]])

In [66]: np.zeros((2,3))
Out[66]:
array([[ 0., 0., 0.],
[ 0., 0., 0.]])

In [67]: np.ones((2,3))
Out[67]:
array([[ 1., 1., 1.],
[ 1., 1., 1.]])
```

```
In [69]: np.eye(3)
Out[69]:
array([[ 1., 0., 0.],
[ 0., 1., 0.],
[ 0., 0., 1.]])

In [70]: np.arange(0, 10, 2)
Out[70]: array([0, 2, 4, 6, 8])
```

```
In [295]: np.random.randint(0, 10, (3,3))
Out[295]:
array([[8, 7, 6],
[0, 8, 9],
[9, 0, 4]])
```

NUMPY DATA TYPES

- int8, int16, int32, int64
- float16, float32, float64, float128
- bool
- object
- String
- Unicode

64 bits

- gArray.astype

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. ,  2. ,  2.5])

>>> x.astype(int)
array([1, 2, 2])
```

```
In [34]: gArray.astype(float64)
Out[34]:
array([[ 79., 95., 60.],
[ 95., 60., 61.],
...,
[ 82., 89., 61.],
[ 94., 67., 88.]])
```

```
In [79]: num_string = array(['1.0', '2.05', '3'])

In [81]: num_string
Out[81]:
array(['1.0', '2.05', '3'], dtype='<U4')

In [82]: num_string.astype(float)
Out[82]: array([ 1. , 2.05, 3. ])
```

propython.py

```
42
43  #Batch operations on data can be performed without using for loops, this is called vectorization
44  scores = [89,56.34, 76,89, 98]
45  first_arr =np.array(scores)
46  print (first_arr)
47  print (first_arr * first_arr)
48  print (first_arr - first_arr)
49  print (1/(first_arr))
50  print (first_arr ** 0.5)
51  print ("*"*30)
52
```

```
**********
[89.   56.34 76.   89.   98.  ]
[7921.     3174.1956 5776.     7921.     9604.    ]
[0. 0. 0. 0. 0.]
[0.01123596 0.01774938 0.01315789 0.01123596 0.01020408]
[9.43398113 7.5059976  8.71779789 9.43398113 9.89949494]
**********
```

ARRAY OPERATIONS

- Between arrays and scalars
- Between equal-sized arrays: elementwise operation

```
In [87]: arr = array([[0,1,2],[3,4,5]])

In [88]: arr * 2
Out[88]:
        array([[ 0, 2, 4],
        [ 6, 8, 10]])

In [90]: arr ** 2
Out[90]:
        array([[ 0, 1, 4],
        [ 9, 16, 25]])

In [91]: 2 ** arr
Out[91]:
        array([[ 1, 2, 4],
        [ 8, 16, 32]], dtype=int32)
```

```
In [94]: arr * arr
Out[94]:
        array([[ 0, 1, 4],
        [ 9, 16, 25]])

In [95]: arr / (arr+1)
Out[95]:
        array([[ 0. , 0.5 , 0.66666667],
        [ 0.75 , 0.8 , 0.83333333]])
```

SPEED DIFFERENCE BETWEEN FOR LOOP AND VECTORIZED COMPUTATION

```
def mySum(inputList):
    s = 0
    for i in range(len(inputList)):
        s += inputList[i]
    return s
```

```
In [118]: a = np.random.rand(1000000,1)
    ...: %timeit a**2
    ...: %timeit [a[i]**2 for i in range(1000000)]
100 loops, best of 3: 4.02 ms per loop
1 loop, best of 3: 1.25 s per loop
```

Vectorization is more than 300 times faster!

```
In [148]: timeit mySum(a)
1 loop, best of 3: 605 ms per loop

In [149]: timeit np.sum(a)
1000 loops, best of 3: 1.15 ms per loop
```

Vectorization is 500 times faster than for loop.

ARRAY INDEXING AND SLICING

- Somewhat similar to python list, but much more flexible

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```

```
In [153]: gArray[0]
Out[153]: array([79, 95, 60])

In [154]: gArray[1:3]
Out[154]:
array([[95, 60, 61],
       [99, 67, 84]])

In [155]: gArray[0][2]
Out[155]: 60

In [156]: gArray[0,2]
Out[156]: 60
```

```
In [157]: gArray[:, 2]
Out[157]: array([60, 61, 84, 97, 98, 96,
76, 80, 61, 88])
```

ARRAY INDEXING AND SLICING (CONT'D)

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```

```
In [160]: gArray[:2, [0, 2]]
Out[160]:
array([[79, 60],
       [95, 61]])
```

```
In [175]: gArray[[0, 2], :]
Out[175]:
array([[79, 95, 60],
       [99, 67, 84]])
```

```
In [200]: gArray[[0,2]][:,[0,2]]
Out[200]:
array([[79, 60],
       [99, 84]])
```

ARRAY SLICES ARE VIEWS

```
In [202]: gArray[0,:]=100

In [203]: gArray
Out[203]:
array([[100, 100, 100],
       [ 95,  60,  61],
       [ 99,  67,  84],
       ...,
       [ 67,  73,  80],
       [ 82,  89,  61],
       [ 94,  67,  88]])
```

```
In [254]: arr2 = gArray.copy()

In [255]: arr2 is gArray
Out[255]: False

In [258]: arr2[1,:]=100

In [260]: gArray[1,:]
Out[260]: array([95, 60, 61])
```

Use .copy() to make a copy of an array explicitly.

```python
# you may want to select a subset of your data, for which Numpy array indexing is really useful
new_arr = np.arange(12)
print (new_arr)
print (new_arr[4:9])
new_arr[4:9] = 99 #assign sequence of values from 4 to 9 as 99
print (new_arr)
modi_arr = new_arr[4:9]
modi_arr[1] = 123456
print (new_arr)                # you can see the changes are refelected in main array.
print (modi_arr[:])            # the sliced variable

# arrays can be treated like matrices
matrix_arr =np.array([[3,4,5],[6,7,8],[9,5,1]])
print (matrix_arr)
print (matrix_arr[1])
print (matrix_arr[0][2]) #first row and third column
print (matrix_arr[0,2]) # This is same as the above operation

matrix_arr =np.array([[3,4,5],[6,7,8],[9,5,1]])
print ("slices the first two rows:{}".format(matrix_arr[:2])) # similar to list slicing. returns first two rows of the array
print ("Slices the first two rows and last two columns:{}".format(matrix_arr[:2, 1:]))
print ("returns 6 and 7: {}".format(matrix_arr[1,:2]))
print ("Returns first column: {}".format(matrix_arr[:,:1])) #Note that a colon by itself means to take the entire axis
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[4 5 6 7 8]
[ 0  1  2  3 99 99 99 99 99  9 10 11]
[     0      1      2      3     99 123456     99     99     99      9
     10     11]
[    99 123456     99     99     99]
[[3 4 5]
 [6 7 8]
 [9 5 1]]
[6 7 8]
5
5
slices the first two rows:[[3 4 5]
 [6 7 8]]
Slices the first two rows and last two columns:[[4 5]
 [7 8]]
returns 6 and 7: [6 7]
Returns first column: [[3]
 [6]
 [9]]
```

BOOLEAN INDEXING

```
# select record for female students
In [262]: female = [ True, False,  True,
True, False,  True, False, False, False,
False]

In [263]: gArray[female, :]
Out[263]:
array([[100, 100, 100],
       [ 99,  67,  84],
       [ 76,  76,  97],
       [ 70,  69,  96]])
```

```
# select record for those who had # <= 70
in final
In [265]: gArray[gArray[:, 2]<70,:]
Out[265]:
array([[95, 60, 61],
       [82, 89, 61]])
```

```
# anything < 70 is changed to 70
In [267]: gArray[gArray < 70] = 70

In [268]: gArray
Out[268]:
array([[100, 100, 100],
       [ 95,  70,  70],
       [ 99,  70,  84],
       ...,
       [ 70,  73,  80],
       [ 82,  89,  70],
       [ 94,  70,  88]])
```

RESHAPE

```
78  my_array= np.arange(6).reshape(2,3)
79  print (my_array)
80  my_array = my_array.reshape(3,2)
81  print (my_array)
82  print (my_array.size   , my_array.min() ,   my_array.max())
83  print (my_array.T)
84
```

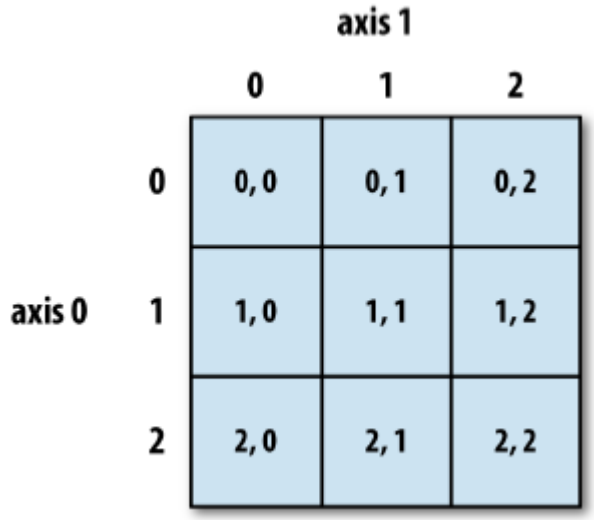
```
[[0 1 2]
 [3 4 5]]
[[0 1]
 [2 3]
 [4 5]]
6 0 5
[[0 2 4]
 [1 3 5]]
```

RESHAPING AND TRANSPOSING

```
In [280]: In [77]: np.arange(6).reshape((2,3))
Out[280]:
array([[0, 1, 2],
       [3, 4, 5]])

In [281]: In [77]: np.arange(6).reshape((2,3), order='F')
Out[281]:
array([[0, 2, 4],
       [1, 3, 5]])
```

```
In [290]: np.arange(6).reshape(2,3).T
Out[290]:
array([[0, 3],
       [1, 4],
       [2, 5]])
```



Fast element-wise functions

print (np.sqrt(my_array))

Table 4-3. Unary ufuncs

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating point, or complex values. Use `fabs` as a faster alternative for non-complex-valued data
`sqrt`	Compute the square root of each element. Equivalent to `arr ** 0.5`
`square`	Compute the square of each element. Equivalent to `arr ** 2`
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
`ceil`	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
`floor`	Compute the floor of each element, i.e. the largest integer less than or equal to each element
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as separate array
`isnan`	Return boolean array indicating whether each value is NaN (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-NaN) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise. Equivalent to `-arr`.

Ufunc: is a function that performs elementwise operations on data in ndarrays

```
>>> a
array([2, 3, 4, 5])
>>> pow(a,2)
array([ 4,  9, 16, 25])
```

TRIGONOMETRIC

```
sin(x)        sinh(x)
cos(x)        cosh(x)
arccos(x)

arccosh(x)
arctan(x)     arctanh(x)
arcsin(x)     arcsinh(x)
arctan2(x,y)
```

OTHERS

```
exp(x)        log(x)
log10(x)      sqrt(x)
absolute(x)   conjugate(x)
negative(x)   ceil(x)
floor(x)      fabs(x)
hypot(x,y)    fmod(x,y)
maximum(x,y)  minimum(x,y)
```

Table 4-4. Binary universal functions

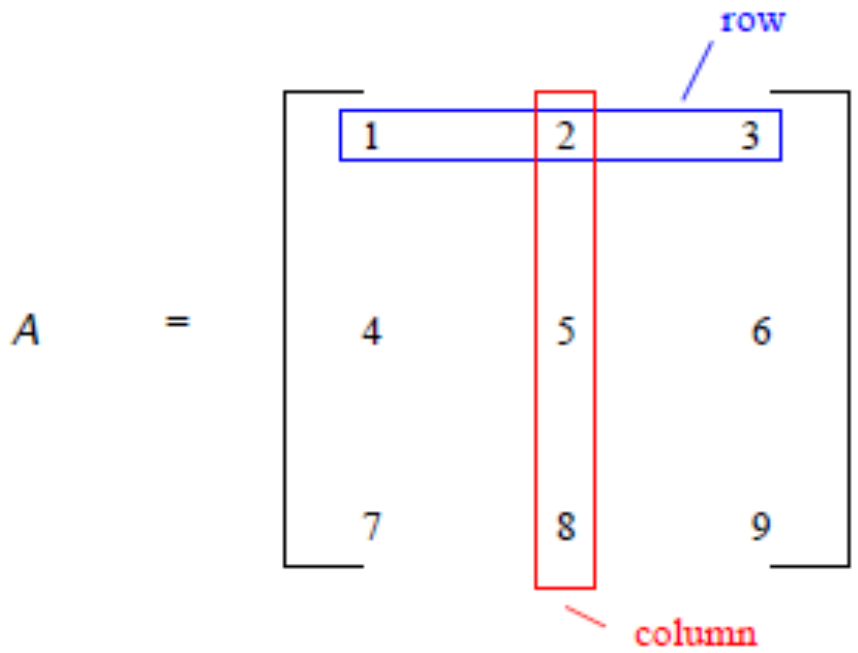
Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum. `fmax` ignores NaN
`minimum, fmin`	Element-wise minimum. `fmin` ignores NaN
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators `>, >=, <, <=, ==, !=`
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation. Equivalent to infix operators `&, |, ^`

```
print (np.add(my_array,my_array))
```

https://docs.scipy.org/doc/numpy/reference/

MATRIX

- A matrix is a rectangular array of numbers organized in rows and columns
- If a matrix A has m rows and n columns, then we say that A is an m x n matrix



MATRIX

- In general, a matrix A of the order $m \times n$ means:
- $$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$
- a_{ij} is the element of A in row i and column j.
- **row** i is the elements $a_{i1}a_{i2}\cdots a_{in}$.
- **column** j is the elements $a_{1j}a_{2j}\cdots a_{mj}$.

VECTORS

- If a matrix has only one row, then it is a **row vector**.
  - Example: $\begin{pmatrix} 1 & 10 & 11 & 12 & 7 \end{pmatrix}$
- If a matrix has only one column, then it is a **column vector**.
  - Example: $\begin{pmatrix} 7 \\ -2 \\ 5 \\ 11 \end{pmatrix}$

IDENTITY MATRIX

- The **identity matrix** is a square matrix that has 1s on the main diagonal and 0s everywhere else.
- An identity matrix of order $n \times n$ is denoted I_n.
- Example: $I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

DIAGONAL MATRIX

- A **diagonal matrix** is a square matrix such that every element that is not on the main diagonal is 0 (elements on the main diagonal can be 0 or non-zero).
- An $n \times n$ diagonal matrix is denoted by D_n.
- Example: $D_3 = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 7 \end{pmatrix}$
- Example: $D_3 = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 5 \end{pmatrix}$

DOT PRODUCT

- A **dot product** is a multiplication of a row vector of order $1 \times n$ with a column vector of order $n \times 1$. The result is a scalar.
- It is obtained by multiplying the ith element of the row vector with the ith element of the row vector and then summing these products.
- $A = (a_1 a_2 \cdots a_n), B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$
- $A \cdot B = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$

```
In [303]: a = b = np.arange(5)
In [305]: a
Out[305]: array([0, 1, 2, 3, 4])
In [306]: b
Out[306]: array([0, 1, 2, 3, 4])
In [307]: a.dot(b)
Out[307]: 30
```

```
a = np.array([[1,2],[3,4]])
b = np.array([[11,12],[13,14]])
np.dot(a,b)
```

It will produce the following output –

```
[[37  40]
 [85  92]]
```

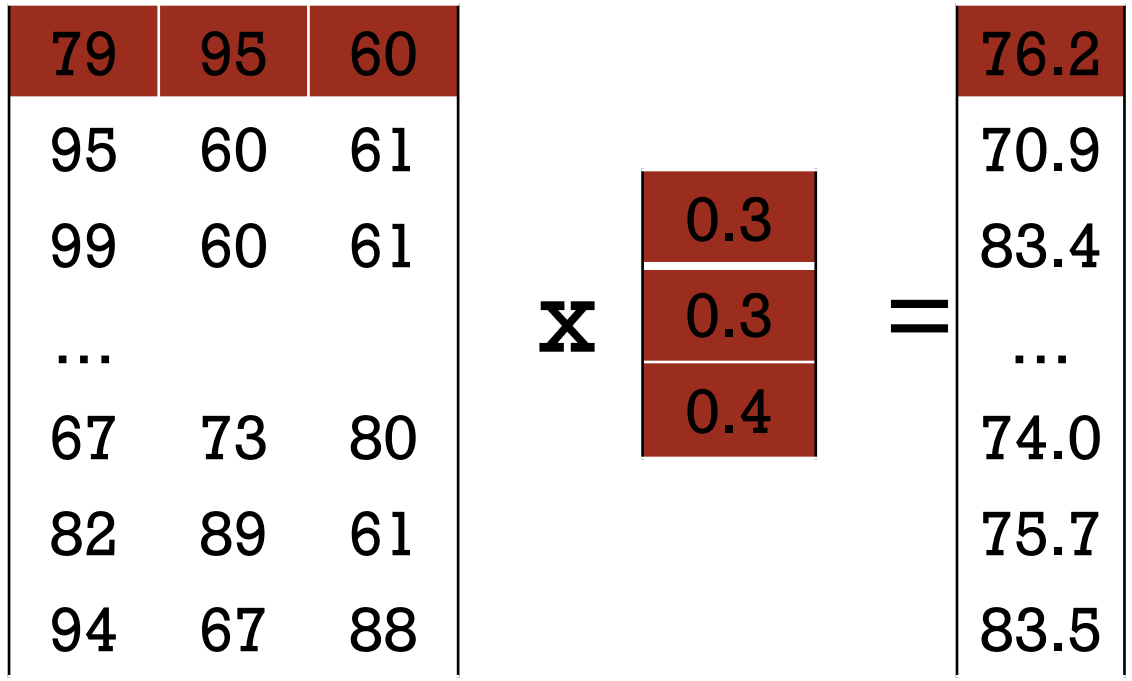
Note that the dot product is calculated as –

```
[[1*11+2*13, 1*12+2*14],[3*11+4*13, 3*12+4*14]]
```

MATRIX MULTIPLICATION

- Suppose A is an $m \times p$ matrix and B is a $p \times n$ matrix (note the number of *columns* of A is the same as the number of *rows* of B). Then the **matrix multiplication** $A \cdot B$ is defined.
- The result is an $m \times n$ matrix (resulting matrix has the same number of rows as A and the same number of columns as B).
- The (i, j)th entry of the resulting matrix is the dot product of row i of A and column j of B.
- Example:
  - $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{pmatrix}$
  - Multiplication is defined because the number of columns of A is the same as the number of rows of B.
  - Result will be a 2×2 matrix.
  - Entry in position $(2, 1)$ in the resulting matrix will be the dot product of the 2nd row in A with the 1st column of B: $a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41}$.

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```




```
In [321]: gArray.dot([0.3, 0.3, 0.4])
Out[321]: array([ 76.2,  70.9,  83.4,
84.4,  91.7,  80.1,  76.3,  74. ,
75.7,  83.5])
```

What is being calculated here?

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```

```
In [329]: scaling = [1.1, 1.05, 1.03]

In [330]: diag(scaling)
Out[330]:
array([[ 1.1 ,  0.  ,  0.  ],
       [ 0.  ,  1.05,  0.  ],
       [ 0.  ,  0.  ,  1.03]])

In [331]: gArray.dot(diag(scaling))
Out[331]:
array([[  86.9 ,   99.75,   61.8 ],
       [ 104.5 ,   63.  ,   62.83],
       [ 108.9 ,   70.35,   86.52],
       ...,
       [  73.7 ,   76.65,   82.4 ],
       [  90.2 ,   93.45,   62.83],
       [ 103.4 ,   70.35,   90.64]])
```

What are we doing here?

NUMPY.SORT()

READ ABOUT IT ☺

```
In [407]: a.sort()

In [408]: a
Out[408]:
array([[1, 2, 4],
       [1, 2, 2],
       [1, 1, 4]])
```

```
In [410]: a.sort(0)

In [411]: a
Out[411]:
array([[1, 1, 2],
       [1, 2, 4],
       [1, 2, 4]])
```