In [ ]:
```
import numpy as np
from numpy.linalg import norm
```

Vectors, dot products, norms, and cosine similarity

For our purposes, vectors are ordered lists of numbers or *elements*.

In numpy, an "array" is one way to store what we would consider to be a vector.

In [ ]:
```
u = np.array([0, 1, 0])
v = np.array([1, 2, 2, 4, 5])
w = np.array([5, 4, 3, 2, 1])
```

We are always allowed to multiply a vector by a *scalar*, which is what we call a single number.

In [ ]:
```
3*u
```

Out[ ]:
```
array([0, 3, 0])
```

If they have the same length, sometimes called "dimension," then we can add and subtract vectors.

In [ ]:
```
v + w
```

Out[ ]:
```
array([6, 6, 5, 6, 6])
```

In [ ]:
```
v - w
```

Out[ ]:
```
array([-4, -2, -1,  2,  4])
```

If they have different lengths, addition and subtraction are not defined.

In [ ]:
```
u + v
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[49], line 1
----> 1 u + v

ValueError: operands could not be broadcast together with shapes (3,) (5,)
```

If two vectors have the same length, we can compute the *dot product* by multiplying each corresponding element and summing up the results.

In [ ]:
```
np.dot(v,w)
```

Out[ ]: 32

The *norm* of a vector is the square root of the sum of the squares of all the elements.

In [ ]:
```
norm(v)
```

Out[ ]: 7.0710678118654755

In [ ]:
```
norm(w)
```

Out[ ]: 7.416198487095663

Think for a moment - why are they the same?

The *cosine similarity* of two vectors involves both their dot products and their norms.

In [ ]:
```
np.dot(v,w)/(norm(v)*norm(w))
```

Out[ ]: 0.6102160571171791

Matrices, matrix operations, transpose, and inverse

A matrix is a two-dimensional array of numbers.

In [ ]:
```
A = np.matrix("1 2 3; 4 5 6; 7 8 9; 10 11 12")
A
```

Out[ ]:
```
matrix([[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9],
        [10, 11, 12]])
```

In [ ]:
```
B = np.matrix("1 0 0; 0 1 0; 0 0 1; 1 1 1")
B
```

Out[ ]:
```
matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1],
        [1, 1, 1]])
```

If a matrix has one column, it is also called a *column vector*

In [ ]:
```
colvec = np.matrix("1; 4; 7; 10")
colvec
```

Out[ ]:
```
matrix([[ 1],
        [ 4],
        [ 7],
        [10]])
```

If a matrix has one row, it is also called a *row vector*

In [ ]:
```
rowvec = np.matrix("1 2 3")
rowvec
```

Out[ ]:
```
matrix([[1, 2, 3]])
```

If two matrices have the same number of rows and columns (same shape) then addition and subtraction are defined element-wise.

In [ ]:
```
A.shape
```

Out[ ]:
```
(4, 3)
```

In [ ]:
```
B.shape
```

Out[ ]:
```
(4, 3)
```

In [ ]:
```
A + B
```

Out[ ]:
```
matrix([[ 2,  2,  3],
        [ 4,  6,  6],
        [ 7,  8, 10],
        [11, 12, 13]])
```

The *transpose* of a matrix is another matrix where the rows become columns and the columns become rows. You can imagine this as reflecting the matrix along its diagonal, which runs from top left to bottom right.

In [ ]:
```
A
```

Out[ ]:
```
matrix([[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9],
        [10, 11, 12]])
```

In [ ]:
```
A.T
```

Out[ ]:
```
matrix([[ 1,  4,  7, 10],
        [ 2,  5,  8, 11],
        [ 3,  6,  9, 12]])
```

Matrix multiplication

Matrix multiplication works differently from multiplying scalars (single numbers.)

To multiply two matrices C and D, the number of *columns* of C must be the same as the number of *rows* of D. The resulting matrix has the same number of *rows* of C and the same number of *columns* of D.

In [ ]:
```
C = np.matrix("1 2; 3 4; 5 6")
D = np.matrix("1 2 ; 4 5")
```

In [ ]:
```
C
```

Out[ ]:
```
matrix([[1, 2],
        [3, 4],
        [5, 6]])
```

In [ ]:
```
D
```

Out[ ]:
```
matrix([[1, 2],
        [4, 5]])
```

In [ ]:
```
C*D
```

Out[ ]:
```
matrix([[ 9, 12],
        [19, 26],
        [29, 40]])
```

Element (i,j) of the result is the dot product of row i of the first matrix with column j of the second matrix.

In [ ]:
```
np.dot(C[0,:],D[:,1]) # Numpy indexes from 0
```

Out[ ]:
```
matrix([[12]])
```

In [ ]:
```
(C*D)[0,1]
```

Out[ ]:
```
12
```

If matrices are square, they can be multiplied in either order, but the result will not be the same, in general. (Matrix multiplication is not *commutative*.)

In [ ]:
```
E = np.matrix("1 2; 3 4")
F = np.matrix("5 6; 7 8")
```

In [ ]:
```
E
```

Out[ ]:
```
matrix([[1, 2],
        [3, 4]])
```

In [ ]:
```
F
```

Out[ ]:
```
matrix([[5, 6],
        [7, 8]])
```

In [ ]:
```
E*F
```

Out[ ]:
```
matrix([[19, 22],
        [43, 50]])
```

In [ ]:
```
F*E
```

Out[ ]:
```
matrix([[23, 34],
        [31, 46]])
```

An *identity matrix* is a square matrix, called I, with 1s on the diagonal and 0s elsewhere.

In [ ]:
```
I = np.matrix("1 0 ; 0 1")
I
```

Out[ ]:
```
matrix([[1, 0],
        [0, 1]])
```

Multiplying a matrix by the identity gives the original matrix.

In [ ]:
```
I*F
```

Out[ ]:
```
matrix([[5, 6],
        [7, 8]])
```

In [ ]:
```
E*I
```

Out[ ]:
```
matrix([[1, 2],
        [3, 4]])
```

If mulitplying two matrices produces the identity matrix, then the two matrices are *inverses* of each other.

In [ ]:
```
Einv = np.matrix("-2 1; 1.5, -0.5")
Einv
```

Out[ ]:
```
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

In [ ]:
```
E*Einv
```

Out[ ]:
```
matrix([[1., 0.],
        [0., 1.]])
```

In [ ]:
```
Einv*E
```

Out[ ]:
```
matrix([[1., 0.],
        [0., 1.]])
```

Not all matrices have an inverse, and finding the inverse of a matrix is not obvious or easy to do by hand. If we need it, we use an algorithm/package.

In [ ]:
```
np.linalg.inv(E)
```

Out[ ]:
```
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

Consider the following 3-by-4 matrix:

In [ ]:
```
M = np.matrix("4 5 6 7; 8 10 12 14; 12 15 18 21")
M
```

Out[ ]:
```
matrix([[ 4,  5,  6,  7],
        [ 8, 10, 12, 14],
        [12, 15, 18, 21]])
```

We can actually write it as the product of two much smaller matrices

In [ ]:
```
G = np.matrix("1; 2; 3")
G
```

Out[ ]:
```
matrix([[1],
        [2],
        [3]])
```

In [ ]:
```
H = np.matrix("4 5 6 7")
H
```

Out[ ]:
```
matrix([[4, 5, 6, 7]])
```

In [ ]:
```
G*H
```

Out[ ]:
```
matrix([[ 4,  5,  6,  7],
        [ 8, 10, 12, 14],
        [12, 15, 18, 21]])
```

In this case, the 12 entries in the matrix J can be compressed into the 3 + 4 = 7 entries of matrices G and H.

Also, column j of matrix M is given by the column vector G, weighted the jth element of H.

In [ ]:
```
G * H[:,2]
```

Out[ ]:
```
matrix([[ 6],
        [12],
        [18]])
```

And, row i of matrix M is given by the row vector H, weighted by the ith element of G.

In [ ]:
```
G[1,:] * H
```

Out[ ]:
```
matrix([[ 8, 10, 12, 14]])
```

In general, if M = G*H for matrices M, G, and H, then:

- Each column of M is a weighted sum of columns of G
- Each row of M is a weighted sum of rows of H

By "weighted sum" we mean that for example, a column of M can be written as a sum of the column vectors of G, where each one is "weighted" (multiplied) by a scalar that comes from H. See the examples below.

In [ ]:
```
Q = np.matrix("1 2; 3 4; 5 6")
```

```
Q
```

Out[ ]:
```
matrix([[1, 2],
        [3, 4],
        [5, 6]])
```

In [ ]:
```
R = np.matrix("1 2 3 4; 5 6 7 8")
R
```

Out[ ]:
```
matrix([[1, 2, 3, 4],
        [5, 6, 7, 8]])
```

In [ ]:
```
S = Q*R
S
```

Out[ ]:
```
matrix([[11, 14, 17, 20],
        [23, 30, 37, 44],
        [35, 46, 57, 68]])
```

In [ ]:
```
R[0,:]*3 + R[1,:]*4 # Compute middle row of S as weighted sum of rows of R (the num
```

Out[ ]:
```
matrix([[23, 30, 37, 44]])
```

In [ ]:
```
Q[:,0]*4 + Q[:,1]*8 # Compute last column of S as weighted sum of columns of Q (the
```

Out[ ]:
```
matrix([[20],
        [44],
        [68]])
```

If we are given a matrix, re-writing it as the product of two matrices is called *factorization*. If the two matrices in the product are much smaller, then we are basically compressing the big matrix. This can both save space *and* reveal structure, as we will see next week.