CSU Online DSCI 369 Lab 1

Original lab written by: Emily J. King

Goals: Get comfortable using numpy and loading/manipulating vectors and matrices, including how to create special arrays, like increasing sequences of integers and all-ones/all-zeros. Apply vector and matrix notation in mathematical and programming form. Judge when vectors/matrices are equal mathematically and on computer

To do linear algebra in Python / Jupyter, we will always need to load packages at the beginning of each lab. We will often assign the package names common abbreviations (e.g., "np" for "numpy") to decrease the number of characters we need to type to use the functions.

Numpy is a package that will be used for math functions in every lab this course. Inside numpy, linalg contains other useful functions that will arise in certain labs.

In [ ]:
```
import numpy as np
```

We begin by entering in a numpy one-way array and play around with some display options. One-way arrays will be the main way that we represent vectors in R^d on a computer.

In [ ]:
```
x = np.array([0.1,-0.05, 0.2, -0.77])
print(x)
print("The vector x is",x,".")
```

```
[ 0.1  -0.05  0.2  -0.77]
The vector x is [ 0.1  -0.05  0.2  -0.77] .
```

It will often be useful to manipulate a vector where all of the entries are constant. The most basic version is all zeros.

In [ ]:
```
y=np.zeros(3)
print("The vector y is",y,".")
```

```
The vector y is [0. 0. 0.] .
```

On a computer, we will typically represent a constant vector with non-zero entries by first creating a one way array with all ones as entries before then scaling each entry.

In [ ]:
```
y = np.ones(3)
print("The vector y is",y,".")
```

```
The vector y is [1. 1. 1.] .
```

Now let's make a different constant array.

In [ ]:
```python
y = 2*np.ones(3)
print("The vector y is",y,".")
```

```
The vector y is [2. 2. 2.] .
```

Notice that 2*np.ones(3) creates an array with three entries, all 1's. Play around the with code above to make an array with all entries equal to -3.2.

It will also be useful to create one way arrays with arithmetic sequences as entries, like the numbers 1 to 10. In Python, this is acheived by "arange" (not: "arrange"!).

In [ ]:
```python
z=np.arange(1,11)
print("The vector z is",z,".")
```

```
The vector z is [ 1  2  3  4  5  6  7  8  9 10] .
```

Play around with the above command to generate a one-way array with entries -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5.

Determine the dimensions of the arrays and compare to the number of elements.

In [ ]:
```python
print("The dimension(s) (in Python) of x is",x.shape,".")
print("The number of elements of x is",len(x),".")

print("The dimension(s) (in Python) of y is",y.shape,".")
print("The number of elements of y is",len(y),".")

print("The dimension(s) (in Python) of z is",z.shape,".")
print("The number of elements of z is",len(z),".")
```

```
The dimension(s) (in Python) of x is (4,) .
The number of elements of x is 4 .
The dimension(s) (in Python) of y is (3,) .
The number of elements of y is 3 .
The dimension(s) (in Python) of z is (10,) .
The number of elements of z is 10 .
```

We will see later in the course that it is at times advantageous mathematically to view vectors with d entries as dx1 or 1xd matrices. Although it is possible to force Python to create a dx1 or 1xd two-dimensional array that "looks" like a vector, the code is clunky and makes other operations not work correctly.

Now let's do some basic vector manipulations. First, we display some parts of the given vectors, remembering that counting starts with 0 in Python. So, the first entry indexed by 0, the second by 1, and so on. This is similar to time, where the second hour of the day starts at 1 A.M., the third hour at 2 A.M., etc.

Also, colons fill in gaps. E.g., 0:2 extracts entries indexed by 0 and 1, but also so does :2.

Finally, one can access entries at the end of arrays using negative indices.

```
In [ ]: print("The first entry of z is",z[0],".")
        print("The first two entries of z are",z[0:2],".")
        print("The first two entries of z are",z[:2],".")
        print("The middle four entries of z are",z[3:7],".")
        print("The last entry of z is",z[-1],".")
        print("The last two entries of z are",z[-2:],".")
```

```
The first entry of z is 1 .
The first two entries of z are [1 2] .
The first two entries of z are [1 2] .
The middle four entries of z are [4 5 6 7] .
The last entry of z is 10 .
The last two entries of z are [ 9 10] .
```

Play around with the commands above to access different entries and ranges of entries in the arrays.

Now let's change entries of the vectors.

```
In [ ]: y[0] = 5
        print("Changing the first entry of y to be 5, we get",y,".")

        y[1] = 2*x[0]
        print("Next, changing the second entry of y to be twice the first entry of x
```

```
Changing the first entry of y to be 5, we get [5. 2. 2.] .
Next, changing the second entry of y to be twice the first entry of x, we ge
t [5.  0.2 2. ] .
```

Now let's change multiple entries at once.

```
In [ ]: z[3:7]=5*np.ones(4)
        print("Chaning the middle four entries of z to all be equal to 5,\n we get",
```

```
Chaning the middle four entries of z to all be equal to 5,
 we get [ 1  2  3  5  5  5  5  8  9 10] .
```

```
In [ ]: z[3:7]=x
        print("Chaning the middle four entries of z to be equal to the entries x,\n
```

```
Chaning the middle four entries of z to be equal to the entries x,
 we get [ 1  2  3  0  0  0  0  8  9 10] .
```

Oh no! Somehow when we tried to change the middle four entries of z to [0.1 -0.05 0.2 -0.77], they all changed to 0. What happened? The problem was that z and x are different data types. For some operations, Python implicitly upcasts (e.g., views the integers as floating point numbers), but here it has implicitly downcast (e.g., rounded the floating point numbers to integers).

It's always important to check your work to ensure the computer is doing what you want it to do!

```
In [ ]: print("The data type of x is",x.dtype,".")
```

```
print("The data type of z is",z.dtype,".")
```

```
The data type of x is float64 .
The data type of z is int64 .
```

We can fix this issue by recasting z to be floating point.

In [ ]:
```
z=z.astype(float)
z[3:7]=x
print("Changing the middle four entries of z to be equal to the entries x,\r

z[7:]=-3*y
print("Next, changing the last three entries of z to be -3 times the entries
```

```
Changing the middle four entries of z to be equal to the entries x,
 we get [ 1.    2.    3.    0.1  -0.05  0.2  -0.77  8.    9.   10.   ] .
Next, changing the last three entries of z to be -3 times the entries of y,
 we have [  1.     2.     3.     0.1   -0.05   0.2   -0.77 -15.    -0.6   -
6.   ] .
```

Now, let's do some similar operations for two-way arrays, the standard way we will represent matrices on a computer. In Python, two-way arrays are represented as one-way arrays of one-way arrays. In mathematical parlance, we will list the rows of the matrix in order top-to-bottom.

As with one-way arrays, the indices start at 0.

In [ ]:
```
A = np.array([[1, 2], [3, 4]])
print("The matrix A is\n",A,".")

print("The (1,2) entry of A is",A[0,1],".")

print("The (1,2) entry of A is",A[0][1],".")

print("The first row of A is",A[0],".")

print("The second column of A is",A[:,1],".")
```

```
The matrix A is
 [[1 2]
 [3 4]] .
The (1,2) entry of A is 2 .
The (1,2) entry of A is 2 .
The first row of A is [1 2] .
The second column of A is [2 4] .
```

Notice how the second column is written by Python horizontally because it is treating is after extraction as a directionless list of numbers.

Now let's create some constant two-way array and also start changing entries of two-way arrays.

In [ ]:
```
B=np.zeros([6,6])
print("The matrix B is\n",B,".")
```

```
C=np.ones([3,3])
print("The matrix C is\n",C,".")

B[:2,:2]=A
print("Replacing the upper left 2x2 entries of B with A results in\n",B,".")

B[-3:,-3:]=-2.5*C
print("Next, replacing the lower right 3x3 entries of B with -2.5 times each
```

```
The matrix B is
 [[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.]] .
The matrix C is
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]] .
Replacing the upper left 2x2 entries of B with A results in
 [[1. 2. 0. 0. 0. 0.]
 [3. 4. 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.]] .
Next, replacing the lower right 3x3 entries of B with -2.5 times each entry
in C results in
 [[ 1.   2.   0.   0.   0.   0. ]
 [ 3.   4.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.  -2.5 -2.5 -2.5]
 [ 0.   0.   0.  -2.5 -2.5 -2.5]
 [ 0.   0.   0.  -2.5 -2.5 -2.5]] .
```

Let's quickly make a three-way array to represent a valence three tensor.

In [ ]:

```
T=np.array([[[1,2],[3,4],[5,6],[7,8]],np.zeros([4,2]),-np.ones([4,2])])
print("The dimensions of T are",T.shape,".")
print("The (1,3,2) entry of T is",T[0,2,1],".")
print("The first slice of T is\n",T[0],".")
print("The second 'column' of the first slice is",T[0,:,1],".")
```

```
The dimensions of T are (3, 4, 2) .
The (1,3,2) entry of T is 6.0 .
The first slice of T is
 [[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]] .
The second 'column' of the first slice is [2. 4. 6. 8.] .
```

Play around with various commands to modify the three-way array more. Practice making a four-way array.

Finally, we test "equality" of arrays in Python. Due to floating point arithmetic issues, typically array representations of vectors/matrices/tensors which are mathematically equal are not exactly equal on the computer. Thus, we must test that all of the entries are really close.

Define a random one-way array u with 6 entries. (If you don't know probability, just take "random" to mean something that a human wouldn't likely write down if asked for a vector. If you know probability, each entry is a draw independently from the uniform [0,1] probability distribution.)

In [ ]:
```
u = np.random.rand(7)
print(u)
```

```
[0.58914103 0.21503728 0.63454744 0.82377393 0.50032267 0.2644693
 0.98890927]
```

Now, we're doing to make a new array v which is equal to u.

In [ ]:
```
v=u
print("v is",v,".")
print("u is",u,".")
```

```
v is [0.58914103 0.21503728 0.63454744 0.82377393 0.50032267 0.2644693
 0.98890927] .
u is [0.58914103 0.21503728 0.63454744 0.82377393 0.50032267 0.2644693
 0.98890927] .
```

Now let's modify u.

In [ ]:
```
u[0]=1
print("v is",v,".")
print("u is",u,".")
```

```
v is [1.         0.21503728 0.63454744 0.82377393 0.50032267 0.2644693
 0.98890927] .
u is [1.         0.21503728 0.63454744 0.82377393 0.50032267 0.2644693
 0.98890927] .
```

Oh no! v changed as well. This is because v=u created a reference, not a copy.

In [ ]:
```
v=np.copy(u)
print("v is",v,".")
print("u is",u,".")
```

```
v is [1.         0.21503728 0.63454744 0.82377393 0.50032267 0.2644693
 0.98890927] .
u is [1.         0.21503728 0.63454744 0.82377393 0.50032267 0.2644693
 0.98890927] .
```

```
In [ ]: u=2.2*u
        print("v is",v,".")
        print("u is",u,".")
```

```
v is [1.         0.21503728 0.63454744 0.82377393 0.50032267 0.2644693
 0.98890927] .
u is [2.2        0.47308202 1.39600438 1.81230264 1.10070988 0.58183247
 2.1756004 ] .
```

So, now changing u (here, by multiplying each entry by 2.2) doesn't change v.

Let's modify v now.

```
In [ ]: v=v/np.pi
```

And let's modify some more.

```
In [ ]: v=2.2*v*np.pi
```

Note that we multiplied each entry of u by 2.2. On the other hand, we first divided each entry of v by pi and then multiplied the result of that by 2.2*pi. Mathematically, both result in the same outcome. However, floating point arithmetic on the computer isn't perfect.

```
In [ ]: u==v
```

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

Most likely, when you ran the above command, you received a mixed list of True's and False's, saying that some of the entries were exactly equal and some not. However, the False's are due to floating point arithmetic error. Thus, we will NEVER use == in this class! The np.allclose command will instead tell us when one-way arrays are within floating point arithmetic error (and thus likely represent vectors which are mathematically equal).

```
In [ ]: np.allclose(u,v)
```

```
Out[ ]: True
```

The same command works for multiway arrays.

```
In [ ]: np.allclose(A,A)
```

```
Out[ ]: True
```

```
In [ ]: np.allclose(T,T)
```

```
Out[ ]: True
```

Exercises

1. Generate a random array with 10 entries. Call that array p.

```
In [ ]: p = np.random.rand(10)
```

2. Make an all-zeros array with 10 entries. Call that array q.

```
In [ ]: q = np.zeros(10)
```

3. Make the first entry of q be equal to the second entry of p. (Remember where indexing starts in Python.)

```
In [ ]: q[0] = p[1]
```

4. Make a two-way array with first "row" equal to p and second "row" equal to the q.

```
In [ ]: a = np.array([p, q])
```

5. Make r be a copy of p.

```
In [ ]: r = np.copy(p)
```

6. Multiply each entry in r by 5.1 times square root of two. (np.sqrt(a) returns the square root of a.)

```
In [ ]: r = r*5.1*np.sqrt(2)
```

7. Divide each entry in r by 5.1 times square root of two.

```
In [ ]: r = r/5.1*np.sqrt(2)
```

8. Test if p and r are equal within floating point arithmetic.

```
In [ ]: np.allclose(p,r)

Out[ ]: False
```