Intro to Numpy

What is Numpy and why do we use it?

It's an awesome python package that adds support for large, multi-dimensional arrays. Really good for vector operations, matrix operations because its super parallelized so its super fast!

Why not Python arrays?

Python arrays has certain limitations: they don't support "vectorized" operations like elementwise addition and multiplication, and the fact that they can contain objects of differing types mean that Python must store type information for every element, and must execute type dispatching code when operating on each element. This also means that very few list operations can be carried out by efficient C loops – each iteration would require type checks and other Python API bookkeeping.

Importing numpy

Functions for numerical computiing are provided by a separate module called `numpy` which we must import.

By convention, we import numpy using the alias `np` .

Once we have done this we can prefix the functions in the numpy library using the prefix `np` .

In [1]:
```
# This is the de facto way to import NumPy. You probably don't want to
write numpy.whatever every time
import numpy as np
```

Numpy Arrays

NumPy arrays are the workhorse of the library. A NumPy array is essentially a bunch of data coupled with some metadata:

type: the type of objects in the array. This will typically be floating-point numbers for our purposes, but other types can be stored. The type of an array can be accessed via the `dtype` attribute.

shape: the dimensions of the array. This is given as a tuple, where element i of the tuple tells you how the "length" of the array in the ith dimension. For example, a 10-dimensional vector would have shape (10,), a 32-by-100 matrix would have shape (32,100), etc. The shape of an array can be accessed via the `shape` attribute.

Let's see some examples! There are number of ways to construct arrays. One is to pass in a Python sequence (such as list or tuple) to the `np.array` function:

In [2]:
```
np.array([1, 2.3, -6])
```

Out[2]:
```
array([ 1. ,  2.3, -6. ])
```

We can also easily create ordered numerical lists!

In [3]:
```
# Remember we zero index so you will actually get 0 to 6!
print(np.arange(7))
# Remember the list wont include 9
print(np.arange(3, 9))
```

```
[0 1 2 3 4 5 6]
[3 4 5 6 7 8]
```

We can also customize these list with a third paramter that specifices step size

In [4]:
```
np.arange(0.0, 100.0, 10.0)
```

Out[4]:
```
array([ 0., 10., 20., 30., 40., 50., 60., 70., 80., 90.])
```

To create a multi-dimensional array, we'll need to nest the sequences:

In [5]:
```
np.array([[1, 2.3, -6], [7, 8, 9]])
```

Out[5]:
```
array([[ 1. ,  2.3, -6. ],
       [ 7. ,  8. ,  9. ]])
```

Neat!

There are also many convenience functions for constructing special arrays. Here are some that might be useful:

In [6]:
```
# The identity matrix of given size
np.eye(7)
```

Out[6]:
```
array([[1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 1.]])
```

In [7]:
```
# A matrix with the given vector on the diagonal
np.diag([1.1,2.2,3.3])
```

Out[7]:
```
array([[1.1, 0. , 0. ],
       [0. , 2.2, 0. ],
       [0. , 0. , 3.3]])
```

In [8]:
```
#An array of all zeros or ones with the given shape
np.zeros((8,4)), np.ones(3)
```

Out[8]:
```
(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.]]), array([1., 1., 1.]))
```

In [9]:
```
# An array with a given shape full of a specified value
np.full((3,4), 2.1)
```

Out[9]:
```
array([[2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1]])
```

In [10]:
```
# A random (standard normal distribution) array with the given shape
np.random.randn(5,6)
```

Out[10]:
```
array([[-0.21012433, -0.395064  ,  0.58699609,  0.21996799, -0.63239
225,
         0.26370822],
       [-0.46016365, -1.18521576, -0.40156712, -0.01302791,  1.16057
806,
         0.75917215],
       [-0.1697336 ,  1.30926073,  0.68306209,  0.05048018, -0.25175
731,
         1.06677987],
       [-0.42961748, -0.34984805, -0.7626864 , -0.65096765,  0.66512
512,
        -0.2647765 ],
       [ 0.10989286,  0.4199001 ,  0.28404666, -0.66111752,  1.26021
329,
         0.25438349]])
```

Okay your turn! In the cell belows try and create:

A diagonal matrix with values from 1-20 (try and create this and only type two numbers!)

In [11]:
```
#Your answer here
np.diag(np.arange(1,21))
```

Out[11]:
```
array([[ 1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  4,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  5,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  7,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  8,  0,  0,  0,  0,  0,  0,  0,
```

```
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  9,  0,  0,  0,  0,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0, 10,  0,  0,  0,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 11,  0,  0,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 12,  0,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 13,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 14,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 15,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
16,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         17,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
          0, 18,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
          0,  0, 19,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
          0,  0,  0, 20]])
```

Okay now let's suppose we have some data in an array so we can start doing stuff with it.

In [12]:
```
A = np.random.randn(10,5); x = np.random.randn(5)
A
```

Out[12]:
```
array([[-2.92468280e-01,  8.79546762e-01, -1.31717614e+00,
         1.08877423e+00, -1.13145551e+00],
       [ 1.32223145e+00,  6.74591595e-01, -6.81043247e-01,
        -7.36051389e-01, -8.34278291e-02],
       [-4.68542890e-01,  4.42117298e-01,  1.21137837e+00,
        -1.33352354e+00, -1.38875897e+00],
       [ 2.02622100e-01, -5.24428790e-01, -1.04213330e+00,
         3.22529310e-01, -9.88326861e-01],
       [-1.20211711e-03,  1.02885550e+00,  5.90903166e-01,
        -6.08259125e-01, -1.58422186e+00],
       [-3.67900439e-01,  1.46486428e+00,  9.47578215e-01,
        -8.01376683e-01,  1.64676644e-01],
       [-4.11347485e-01, -8.57052284e-01, -1.46004739e+00,
        -3.91742223e-01,  6.33541639e-02],
       [-2.27954001e-01, -6.15088897e-01,  8.01566650e-01,
         7.83904053e-01,  4.73235355e-01],
       [-7.31756484e-01,  1.52676976e-01,  1.08942979e+00,
        -4.37079083e-01, -5.59649993e-01],
       [-1.63605277e+00, -3.19578330e-01, -3.16311099e-01,
        -2.49333922e+00,  7.77339143e-02]])
```

One useful thing that NumPy lets us do efficiently is apply the same function to every element in an array. You'll often need to e.g. exponentiate a bunch of values, but if you use a list comprehension or map with the builtin Python math functions it may be really slow. Instead just write

In [13]:
```
# log, sin, cos, etc. work similarly
np.exp(A)
```

Out[13]:
```
array([[0.74641892, 2.40980724, 0.26789072, 2.97063053, 0.32256342],
       [3.75178395, 1.96323102, 0.50608874, 0.47900158, 0.91995748],
       [0.62591363, 1.55599825, 3.3581102 , 0.263547  , 0.24938461],
       [1.2246096 , 0.59189336, 0.35270146, 1.38061536, 0.37219891],
       [0.99879861, 2.79786185, 1.80561845, 0.5442976 , 0.20510733],
       [0.69218609, 4.32695593, 2.5794552 , 0.44871081, 1.17901182],
       [0.66275659, 0.42441128, 0.23222527, 0.67587832, 1.0654041 ],
       [0.79616088, 0.54059284, 2.22903031, 2.1900055 , 1.60517913],
       [0.48106327, 1.16494861, 2.97257859, 0.64592035, 0.57140903],
       [0.19474724, 0.7264553 , 0.72883268, 0.08263357, 1.08083503]]
)
```

We can take the sum/mean/standard deviation/etc. of all the elements in an array:

In [14]:
```
np.sum(x), np.mean(x), np.std(x)
```

Out[14]:
```
(-0.762267691287962, -0.1524535382575924, 0.4153529425783442)
```

You can also specify an axis over which to compute the sum if you want a vector of row/column sums (again, sum here can be replaced with mean or other operations):

In [15]:
```
# Create an array with numbers in the range 0,...,3 (similar to the no
rmal Python range function,
# but it returns a NumPy array) and then reshape it to a 2x2 matrix
B = np.arange(4).reshape((2,2))

# Original matrix, column sum, row sum
B, np.sum(B, axis=0), np.sum(B, axis=1)
```

Out[15]:
```
(array([[0, 1],
        [2, 3]]), array([2, 4]), array([1, 5]))
```

Linear Algebra

By now we have a pretty good idea of how data is stored and accessed within NumPy arrays. But we typically want to do something more "interesting", which for our ML purposes usually means linear algebra operations. Fortunately, numpy has good support for such routines. Let's see some examples!

In [16]:
```
# Matrix-vector product. The dimensions have to match, of course
A.dot(x)
# Note that in Python3 there is also a slick notation A @ x which does
the same thing
```

Out[16]:
```
array([ 0.99288794, -1.16042446, -0.21681476,  0.92161511, -0.083960
16,
       -1.17795819,  0.34539557,  0.57268078,  0.14905389, -0.825641
11])
```

In [17]:
```
# Transpose a matrix
A.T
```

Out[17]:
```
array([[-2.92468280e-01,  1.32223145e+00, -4.68542890e-01,
         2.02622100e-01, -1.20211711e-03, -3.67900439e-01,
        -4.11347485e-01, -2.27954001e-01, -7.31756484e-01,
        -1.63605277e+00],
       [ 8.79546762e-01,  6.74591595e-01,  4.42117298e-01,
        -5.24428790e-01,  1.02885550e+00,  1.46486428e+00,
        -8.57052284e-01, -6.15088897e-01,  1.52676976e-01,
        -3.19578330e-01],
       [-1.31717614e+00, -6.81043247e-01,  1.21137837e+00,
        -1.04213330e+00,  5.90903166e-01,  9.47578215e-01,
        -1.46004739e+00,  8.01566650e-01,  1.08942979e+00,
        -3.16311099e-01],
       [ 1.08877423e+00, -7.36051389e-01, -1.33352354e+00,
         3.22529310e-01, -6.08259125e-01, -8.01376683e-01,
        -3.91742223e-01,  7.83904053e-01, -4.37079083e-01,
        -2.49333922e+00],
       [-1.13145551e+00, -8.34278291e-02, -1.38875897e+00,
        -9.88326861e-01, -1.58422186e+00,  1.64676644e-01,
         6.33541639e-02,  4.73235355e-01, -5.59649993e-01,
         7.77339143e-02]])
```

Now that you're familiar with numpy feel free to check out the documentation and see what else you can do! Documentation can be found here: https://docs.scipy.org/doc/ (https://docs.scipy.org/doc/)

Exercises

Lets try out all the new numpy stuff we just learned! Even if you have experience in numpy we suggest trying these out.

1) Create a vector of size 10 containing zeros

In [18]:
```
## FILL IN YOUR ANSWER HERE ##
v = np.zeros(10)
v
```

Out[18]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

2) Now change the fith value to be 5

In [20]:
```
## FILL IN YOUR ANSWER HERE ##
v[4] = 5
v
```

Out[20]:
```
array([0., 0., 0., 0., 5., 0., 0., 0., 0., 0.])
```

3) Create a vector with values ranging from 10 to 49

In [22]:
```
## FILL IN YOUR ANSWER HERE ##
v = np.arange(10, 50)
v
```

Out[22]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2
5, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 4
2, 43,
       44, 45, 46, 47, 48, 49])
```

4) Reverse the previous vector (first element becomes last)

In [23]:
```
## FILL IN YOUR ANSWER HERE ##
v = v[::-1]
v
```

Out[23]:
```
array([49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 3
4, 33,
       32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 1
7, 16,
       15, 14, 13, 12, 11, 10])
```

5) Create a 3x3 matrix with values ranging from 0 to 8. Create a 1D array first and then resshape it. </b<

In [24]:
```
## FILL IN YOUR ANSWER HERE ##
np.arange(9).reshape((3,3))
```

Out[24]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

6) Create a 3x3x3 array with random values

In [25]:
```
## FILL IN YOUR ANSWER HERE ##
np.random.randn(3,3,3)
```

Out[25]:
```
array([[[ 0.10553989, -1.44918101,  1.76692187],
        [ 0.9600631 , -2.12465232,  0.07983749],
        [ 0.88609752,  0.47117387, -0.55992636]],

       [[ 1.09676838,  1.59762001,  1.27709261],
        [ 0.75616915, -1.07806943,  0.18685633],
        [-0.62523703, -1.09455755, -0.23230448]],

       [[ 0.09512103, -3.39568338,  1.290432  ],
        [-0.97324084,  0.99904464,  0.37482322],
        [-0.84228701, -1.41113008, -0.81834712]]])
```

7) Create a random array and find the sum, mean, and standard deviation

In [26]:
```
## FILL IN YOUR ANSWER HERE ##
v = np.random.randn(100)
np.sum(v), np.mean(v), np.std(v)
```

Out[26]:
```
(0.20254356081693747, 0.0020254356081693746, 1.0827826632876085)
```

In [ ]:
```

```