Intro to NumPy and Matplotlib

- NumPy is the main scientific computing package for Python - it allows you to easily work with large arrays of data and supports functionality for many common operations (including linear algebra)
- All about doing computations on large data sets all at once - can do many many things without looping! Much more effecient
- based on this numpy quickstart guide
- NumPy main page
- NumPY and SciPy doc page

```
# import numpy and other stuff for this tutorial
import numpy as np

# import a specific function from NumPy cause we'll use it a lot
from numpy import pi

# functionality for plotting
import matplotlib.pyplot as plt
```

Initialize array and a few basic operations

- np.arange method works just like the built in range function
- the interval includes `start` but excludes `stop`, overall interval [start...stop-1]

```
# set up an array and figure out shape...
my_array = np.arange(10)
print(my_array)

# note that its 1D (a vector...)
my_array.shape
```

```
# can specify start, stop and step
seq_array = np.arange(0,30,5)     # start, stop (stop at < X), step size
print(seq_array)
# note that 30 is not in there...
```

Reshape array - in this case a 1D vector to a 2D matrix

```
my_array = np.arange(36)
my_array = my_array.reshape(6,6)    # 3,12,  9,4
print(my_array.shape)
print(my_array)
# why is (6,6) and (12,3) ok but (5,5) not ok?
```

Reshape array - more complex...

- 1D, 2D, ND arrays
- Notice how the dims stack on top of each other!

```
my_array = np.arange(100)
my_array = my_array.reshape(5,5,4)   # 2,5,10
my_array.shape
print(my_array)

# NOTICE how the dims stack on top of each other! there are 5, 5x4 matrices
```

Data types (and remember - strong typed language)

```
print('Dims of data:', my_array.ndim)          # number of dims
print('Name of data type:', my_array.dtype)    # name of data type (float, int32, int6
print('Size of each element (bytes):', my_array.itemsize)          # size of each ele
print('Total number of elements in array:', my_array.size)         # total number of
```

Infer data types upon array creation

- Use np.array to initialize an array and fill it with numbers
- Can use lists or tuples (or any array-like input of numerical values)
- Can specify data type upon array creation...complex, float32, float64, int32, uint32 (unsigned int32), etc

```
# will infer data type based on input values...here we have 1 float so the whole thing
float_array = np.array([1.2,2,3])
float_array.dtype             # or np.dtype
```

Can also specify type upon array creation

- What happens if you initialize with floating point numbers but you declare an int data type?
- e.g. type casting upon array creation, as we discussed with pandas
- doesn't round, it truncates!

```
int_array = np.array([1.1,7.5], dtype = 'int32')
int_array

# truncation!
```

Allocate arrays of zeros, ones or rand to reserve the memory before filling up later

- Handy when you know what size you need, but you're not ready to fill it up yet...saves you from dynamically resizing the matrix during analysis, which is VERY,VERY slow (e.g. the 'append' method)

```
# note the () around the dims because here we're specifying as a tuple...
# default type is float64...can also pass in a list
arr = np.zeros( (3,4) )
print(arr)
arr.dtype
```

Init an array of ones

- Can use this method to init an array of any value...see next cell below

```
# ones
# note the 3D output below...4, 4x4 squares of floating point 1s...
arr = np.ones( (4,4,4) )
print(arr)
```

What if you want to initialize an array of 10s?

```
arr = np.ones( (4,4,4) ) * 10
print(arr)
```

Random numbers - generate all at once as opposed to looping like we did earlier in the class

```
arr = np.random.random( (5,4) )
print(arr)
```

Empty

- Because you're not initializing to a specific value (like zeros), can by marginally faster when allocating a large array
- However, this is a bit dangerous because exact values in an 'empty' array are based on current state of memory and can vary...
- Need to make sure that you are overwriting ALL of the values and that you remember that the values are NOT 0!!! (or 1)

```
# and empty...not really 'empty' but initialized with varible output determined
# by current state of memory
arr = np.empty( (2,2) )
print(arr)
```

Fill up an array at init with any value, include NaNs! (very handy for error checking!)

```
# an alternate way to initialize an array with arbitrary values
# note that 'full' will guess best data type given init value
arr = np.full( (2,2), np.nan)
print(arr)
```

NumPy Part II: Simple elementwise arithmetic operations like + and - work on corresponding elements of arrays.

- MASSIVE speed up over looping!

```
# set up two sets of data...
N=1440
x = np.linspace(0,2*pi,N)
y = np.sin(x)
```

First add each element of x with the corresponding element of y using the old method...

```
sum_lst = []
%timeit for i in range(N): sum_lst.append(x[i]+y[i])
```

Now do it the "NumPy" way...it goes much much faster!

- often goes from milliseconds to microseconds

```
%timeit sum_lst = x-y
```

Another timing test - see how much it helps to pre-allocate a matrix to store the output

- For example: make a matrix of 'zeros' to store the output of element-by-element multiplication of two other matrices
- Allocating the matrix of zeros takes some time, but its still faster that asking python to dynamically allocate the matrix at the time of the operation

```
# python way - although helping python way out a bit by preallocating z (so real)
# python way even slower...
N = 100000
x = np.arange(0,N)
y = np.arange(0,N)
z=np.zeros(N)

%timeit for i in range(N): z[i] = x[i]*y[i]
```

```
# numpy way

N = 100000
x = np.arange(0,N)
y = np.arange(0,N)
z=np.zeros(N)

%timeit z=x*y
```

When dealing with muliple arrays of different data types, resulting array will take the form of the highest precision input array (upcasting)!

```
# declare dtype as int32
x = np.arange(10, dtype='int32')

# this will default to float64
y = np.random.randn(1,10)

# now multiply the int32 array with the float64 array and answer should be the
# higher precision of the two (float64)
z = x * y
print('z data type: ', z.dtype)
```

Unary operations implemented as methods of the ndarray class

```
# note the method chain...
x = np.arange(10).reshape(2,5)   # 2 x 5 matrix

print(x.sum())                   # sum of all elements
print(x.sum(axis=0))             # sum of each column (across 1st dim)
print(x.sum(axis=1))             # sum of each row (across 2nd dim)
print(x.sum(0))                  # don't need the axis arg, can just specify
```

Set logic....

```
x = np.arange(20)
y = np.linspace(0, 20, 21)
print(x.size)
print(y.size)

z = np.union1d(x,y)
print(z, z.size)

# z = np.intersect1d(x,y)
# print(z)

# z = np.unique([np.append(x,y)])
# print(z)
```

Slicing...

```
# create a 1d array
x = np.linspace(0,9,10)
print(x)
x[1]                     # just the second entry, remember 0 based indexing

# specific start and stop points (exclusive)
x[0:2]                   # the first and second entries in the array, so N>=0 and N<2

# assign the 2nd - 4th element to 100 (index 1,2,3)
x[1:4] = 100
print(x[1:4])
```

Step through a ndarray - similar to a list

```
# start, stop, step interval
print(x[0:8:2])

# reverse x
print(x[::-1])
```

Iterate over elements in a ndarray...also similar to a list

```
# init an ndarray - in this case a 2x2 array
x = np.array([(2,4),(5,6)])

# iterate over all elements in 1D array x
# iterating goes over the first dim (rows), and on each
# loop it will print the entire row
cnt = 0
for i in x:
  cnt+=1
  print(i)    # then i takes the value of each element in x

# only loops twice (once for each row)
print(cnt)
```

Can also flatten a ND array to print it out like 1D array

```
# init an ndarray - in this case a 3x2 array (3 row, 2 columns)
x = np.array([(2,4),(5,6),(12,19)])
print(x)

# then iterate over all entries in the array using 'flat'
# will proceed along 1st row, then to 2nd row, etc.
for i in x.flat:
    print(i)
```

Multidimentional array indexing, slicing etc

```
# generate a matrix of uniformly distributed random numbers over 0:10
x = np.round(np.random.random((10,5))*10)
print(x)

x[0,0]     # first row, first column
x[2,3]     # third row, 4th column

x[:, 3]    # all entries in the 4th column
x[3, :]    # all entries in the 4th row
x[0:2, 4]  # first two entries of the 5th column
x[6, 2:4]  # 7th row, 3rd and 4th entries.

# if not all dims specified then missing values are considered complete slices
# these three ways of writing all do the same thing...
x[6]
x[6,]
x[6,:]

# tricks...
print('last row: ', x[-1,:])      # last row
print('last column: ', x[:,-1])   # last column
print('last entry: ', x[-1,-1])   # last value
```

Pull out subsets of rows and columns

```
# generate a matrix of random numbers over 0-1
x = np.random.rand(4,3)
print(x)

# first two rows - note that you don't have to specify the 2nd dim - and note that
# '2' here means rows 0 and 1 (not 0 through 2!)
y = x[:2]
print('\n', y)

# can also take the last two rows...in the same manner...in this case rows 3 and 4
y = x[2:]
print('\n', y)

# first two rows, 1st column
y = x[:2,0]
print('\n', y)

# rows 3 - end, columns 2 - end
y = x[2:,1:]
print('\n', y)
```

Important - slicing an array and re-assigning the output creates a view of the data, not a copy!

- Recall that a 'view' is when two variables are both referencing the same data in memory
- Because both variables are referencing the same data, changing one variable will also change the other...

Init an array to demonstrate...in this case a 3x2 array

```
x = np.array([ [2,4], [6,7], [5,4] ])
print('Initial values in x:\n', x)
```

Then reassign all values in the 3rd row of x to a new variable z

- z will be a 'view' of the data in the 3rd row of x

```
z = x[2,]
print('Shape of z:', z.shape, 'Values in z:', z)
```

Now change all values in z to 100 (or whatever you want)

- use the syntax z[:], which indicates "all values in z"
- if you change data in z it will also change the corresponding elements in x because z references the same data (or chunk of memory)

```
z[:]=100
print('New values in z:', z)
```

Notice that x has now changed even though you never directly changed it!

```
print('x also changed!!!\n', x)
```

If you want two independent variables that do not reference the same data, use the copy method

```
# re-initialize x
x = np.array([ [2,4], [6,7], [5,4] ])

# make a copy
z = x[2,].copy()

# now you can modify z
z[:] = 100

# and it won't change x
print(x)
```

Logical indexing.

- Just like with Pandas, we in NumPy we use '&' for comparisons instead of 'and' and 'or'

```
# using logical indexing to grab out subsets of data...
x = np.arange(0,10)
y = x[(x>3) & (x<7)]
print(y)
```

Fancy indexing...using arrays to index arrays - used all the time in data analysis...

- Fancy indexing always makes a COPY of the data (unlike normal slicing which creates a view)!!!

```
# define an array to play around with...
x = np.random.rand(3,4)

# define another array (a tuple) to use as an index into the first array
y = (2,3)

# index
print(x)
print('\n x indexed at tuple y: ', x[y])
```

Can use fancy indexing to extract elements in a particular order

```
print(x)

# this will extract the 3rd row, then the 2nd row, then the first row
x[[2,1,0]]

# and this will extract all rows from the 2nd, 3rd and then 1st column.
x[:,[1,2,0]]
```

Or can pass in multiple arrays...will return a 1D array corresponding to each array [1,1] and [2,2] in this case

```
print(x)
x[[1,2],[1,2]]
```

Because of machine precision issues, sometimes hard to predict how many elements will end up in an array when initialized using arange...

- Often better to specify a sequence based on start point, stop point, and the exact number of elements that you want (or the number of steps between start and stop). linspace (linear spacing) is the function to do this, and note that unlike arange that ends < stop point, linspace will always end exactly at the specified stop point.

```
seq_array = np.arange(0,10,.56788)    # decimal input is ok too
# (and again - stop is NOT included)
print(seq_array)
```

Linspace: start, stop, number of linearly spaced steps between start and stop...note that start AND stop included (inclusive)

```
# n evenly spaced steps from start to stop...
n = 9
lin_array = np.linspace(0,180,n)
print(lin_array)
```

Common use of linspace...eval a function over an interval. quick intro to basic plotting here too...

```
# eval sin function over an interval from 0 to 2*pi
n = 18
lin = np.linspace(0, 2*pi, n)
sw = np.sin(lin)

# plotting - can play with formatting here...change line color and other
# properties
# note we assign a handle (h), or a unique identifyer...will use this in a bit to inte
h = plt.plot(lin*180/pi, sw)    # specify x,y data...convert rad to deg for x-axis

# update the plot output window (show the plot)
plt.show()
```

Can change appearance of plots in several ways - first start with inline params at time of plotting, then modify the axes labels, etc.

- marker types
- linestyles

```
# change color, marker type, line type
h = plt.plot(lin*180/pi, sw, 'ro-', linewidth = 2)

# a few other examples [comment/uncomment each line to check]
#h = plt.plot(lin*180/pi, sw, 'kv:', linewidth = 2)
#h = plt.plot(lin*180/pi, sw, 'go-', linewidth = 2)

# label each axis
plt.xlabel('Angle (deg)')
plt.ylabel('Amplitude')

# give the plot a title
plt.title('Sin Wave')
plt.grid(1)

# show the plot!
plt.show()
```

Figure out all properties that you can set on a figure

```
# figure out all settings that you can tweak...
plt.setp(h)
```

Use 'setp' method to change some of these features like markersize and alpha (opacity)

```
# plot
h = plt.plot(lin*180/pi, sw, 'k1-', linewidth = 2)    # specify x,y data...convert rad

# axes labels, etc...
plt.xlabel('Angle (deg)')
plt.ylabel('Amplitude')
plt.title('Sin Wave')
plt.grid(1)

# set marker size
plt.setp(h, 'markersize', 15)

# opacity!
plt.setp(h, 'alpha', .5)

plt.show()
```

Scatterplots and legends

- main scatterplot page
- more about legends

- note that for scatter plots, color is specified by keyword...i.e. c='green', r='red', etc.

```
# Scatter plots..
N = 30
x = np.linspace(0,9,N)

# random method! - randn like rand but draws from N(0,var)
# What does the *3 do here?
y = x + np.random.randn(1,N)*3   # make a second vector x + some randn noise

# make a scatter plot of x vs. y
plt.scatter(x, y, s=50, c='green', alpha=.5)   # note alpha or transparency
plt.xlabel("X")
plt.ylabel("Y")

# add a legend! First text, then location...
plt.legend(['x versus y'], loc=3)    # 1-4 for each corner of the plot

# show the plot
plt.show()
```

Error bar example

```
# x axis, mean and standard deviation of a data set
x = np.linspace(1,8,8)
print(x.shape)
m = [1,2,4,3,3.5,6,7.2,9]
sd = [.2, .25, .75, 1, .9, 1.2, 1.1, .97]

plt.errorbar(x, m, yerr=sd, c='black')
plt.xlabel('Day of training')
plt.ylabel('Performance')

plt.show()
```