NumPy

Introduction to Data Science with Python

What is a Vector/Matrix?

- Think of a vector as an array of values
  - Similar to a tuple but even more constrained by type

- Think of a matrix as a "list of lists"
  - Actually you can make a matrix from a list of lists

NumPy

- Numerical Python
- “Foundational” module for scientific computing
- Mimics Matlab interface
- Provides the “ndarray”
  - This is the main class for scientific computing
  - Has a “dtype” attribute
- Provides many functions for matrix/linear algebra we will need

NumPy ndarray

- The Python convention is to import numpy as “np”
- ndarray
  N-dimensional array (alias is array)
  Created with array()

```
import numpy as np

# Generate a NumPy array
arr1 = np.array(range(10))
arr2 = np.array([[1,2,3],[4,5,6]])
```

Python Lists vs NumPy Arrays

- So you might thing, what’s the big deal? Why not just use a Python list?
- They have many of the same properties but there are some important differences
  - Adding an element to a list/array
  - Performing operations over a list/array
- It turns out many operators behave differently
  - NumPy arrays are automatically vectorized
  - You can treat a NumPy array like a vector

Other Ways to Create Arrays

- NumPy provides other functions to create common arrays

  arange()
  ones()   create an array of all ones
  zeros()   create an array of all zeros
  identity() create an identity matrix

```
# More array creation
np.ones((2,4))
np.zeros((2,4))
np.identity(5)
```

Attributes

ndim: number of dimensions

shape: list of the dimension lengths

size: total number of elements

data: elements of the array

dtype: data type of the object

```
# Attributes
arr1.ndim
arr2.ndim
arr2.shape
arr2.dtype
arr2.size
```

Data Types

- Arrays are *fast*, partially due to the data type mapping to lower level calls. This is mostly transparent, but we can specify the dtype if we need to change it

  np.array(*object*[,*dtype*])

  See the full list of dtypes here:

  https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.dtypes.html

  We can change type using array.astype()

```
# Dtype
arr3 = np.array([1,2,3,4,5,6],"float")
arr3.dtype
>> dtype('float64')
# change dtype
arr3 = arr3.astype("int")
```

Test Your Knowledge – Ex 9.1

- Now it's your turn to apply what you have learned.

Write Python code to do the following:

1. Create an np array of the numbers 1-10
   a. Change the dtype to a float

Indexing

- For one dimensional arrays, we index in the usual way
- These slices are views into the data, not copies
- Changes are *broadcasted* if required

```
# Indexing (slicing)
arr = np.arange(1,4*4*4+1)
arr[0]
arr[:4]
arr[-1:-10:-1]
arr[[1,2,1]]
arr[:3] = 2
arr
```

Test Your Knowledge – Ex 9.2

- Now it's your turn to apply what you have learned.

Write Python code to do the following:

1. Create an np array of 10 elements (letters a-j) (hint, see if you can use the string module)
   a. Change the 4^{th} element to the number 1 (does it work, what type is it?)

Reshaping

- You can reshape an array with array.reshape()

```
# Reshaping array
arr = np.arange(1,4*4*4+1)
rec= arr.reshape(4,16)
rec.shape
square= arr.reshape(8,8)
square.shape
cube = arr.reshape(4,4,4)
cube.shape
```

A Note

- Avoid “rank 1” arrays as their behavior can be a little unpredictable
- Go with column vectors or row vectors instead
- Use assert to verify an object’s shape

```
# say I need to create an array
a = np.random.randn(5) # Rank 1 array – don’t use
a.shape() # = (5,) shape of array

# better
a = np.random.randn(5, 1) # column vector
a = np.random.randn(1, 5) # row vector
assert(a.shape == (5, 1)) # verify shape is correct
```

Multi Dimensional Index

- Indexing gets more interesting with multiple dimensions

  List each axis, separate by commas, use colon to designate all elements of a dimension

```
# Indexing Multiple Dimensions
square[0]
square[0,:] # equivalent, implied
rec[3,15]
rec[3][15] # equivalent

cube[0]
cube[0,2,1]
cube[:,0,0]
```

Test Your Knowledge – Ex 9.3

- Now it's your turn to apply what you have learned.

Write Python code to do the following:

1. Create a 2x5 array of the numbers 1-10
   a. Transpose the array, change rows to columns, columns to rows
   b. Reshape the array, make it a 2x2x3 array. Add the numbers 11, 12 to the missing cells

Boolean Index

- We can also index and array using Boolean values
- The Boolean equivalent array will display all elements where the corresponding Boolean location is true
  - Let's demonstrate using a normal random number generator from numpy.random
  - These boolean values can be along an axis, or cover all elements

```
# Boolean Index
np.random.seed(123)
arr_norm = np.random.randn(10).reshape(5,2)
tf = arr_norm > 1

arr_norm[tf]
```

Test Your Knowledge – Ex 9.4

- Now it's your turn to apply what you have learned.

Write Python code to do the following:

1. Create a Boolean index on the array you created in exercise 3. Use it to retrieve elements greater than 5
2. Calculate the mean of the array

Broadcasting

- Let's go back to the difference between a NumPy array and a Python list. When we assign scalar elements to an array, the results are *broadcast* to the entire slice, unlike the list

```
# Broadcasting
arr_norm[arr_norm > 0] = 0
arr_norm
>> output
array([[-0.67888615, -0.09470897],
       [ 0.        , -0.638902  ],
       [-0.44398196, -0.43435128],
       [ 0.        ,  0.        ],
       [ 0.        ,  0.        ]])
```

Vectorized

- We can also perform mathematical functions between arrays and scalars. The scalar will be *broadcast*, also sometimes called vectorized calculations

```
# Vectorized calculations
arr_norm + 1

>>

array([[-0.0856306 ,  1.        ],
       [ 1.        , -0.50629471],
       [ 0.42139975,  1.        ],
       [-1.42667924,  0.57108737],
       [ 1.        ,  0.1332596 ]])
```

NumPy Functions

- NumPy provides many functions you would expect in data science application

```
# NumPy Functions
np.mean(arr_norm)
>> -0.6892857843701543l
np.mean(arr_norm,0)
>> array([-0.81818202, -0.56038955])
np.mean(arr_norm,1)
>> array([-0.5428153 , -0.75314736, -0.28930013, -
1.42779594, -0.4333702 ])

np.var(arr_norm)
```

Random Numbers

- NumPy has a random number generation module

  help(np.random)

  Why do you think we need to Seed sometimes?
  http://docs.scipy.org/doc/numpy/reference/routines.random.html

```
# uniform [0,1)
np.random.rand(4,4)
# normal(mean=0,var=1)
np.random.randn(2,2,2)
4*np.random.randn(2,2,2) + 10

# binomial
np.random.binomial(n=100,p=0.2,size=50)
```

Concatenation / Conversion

- We may wish to “bind” two arrays together
  - Row-wise (vertically) or column-wise (horizontally)
- We may also wish to convert an array to a list for use in a different module function which expects a list

```
# concatenate
ar1 = np.zeros(5)
bin_ar = np.ones(5)
np.vstack((bin_ar,ar1))

# you can run this
np.zeros(10).reshape((10,1))

# or alternatively
ar2 = np.zeros((10,1))
np.hstack((bin_ar,ar2))

# convert to (nested) list
result = bin_ar.tolist()
```

Test Your Knowledge – Ex 9.5

- Now it's your turn to apply what you have learned.

```
Write Python code to do the following:

1. Create an 3 by 3 array of random numbers
   between 1 and 20

2. Convert the array to a nested list
```

Working with Vectors/Matrices

- NumPy lets you perform some relatively sophisticated and useful functions on arrays
  - We can also multiply 2 arrays together
    - You get the elementwise multiplication of the members, as we have already seen
    - Note the arrays must be the same size for this to work
  - We can take the dot product of two arrays
    - There are multiple ways to do this
      - Multiply two arrays
      - Sum the resulting vector
      - Or use the .dot method

Matrix Products

- We are talking about matrix multiplication
  - Note the inner dimensions must match
  - Suppose we have matrix A of size (2, 3) and B of size (3, 3)
  - We can multiple AB since the inner dimension is 3/3
  - We cannot multiply BA since the inner dimension is 3/2

- $C(i,j) = \sum_{k=1}^{K} A(i,k)B(k,j)$
- (i,j)th entry of C is the dot product of row A(I,:) and column B(:,j)
- In NumPy we can also say C = A.dot(B)
- Remember the distinction between elementwise multiplication (*) and matrix multiplication (.dot)

More Matrix Operations

- Let’s look at some additional matrix-specific operations
  - Inverse
  - Determinant
  - Diagonal
  - Outer Product
  - Inner Product (same as the dot product)
  - Matrix trace
  - Transpose
  - Eigenvalues and eigenvectors

Solving a Linear System

- A Linear System is a system of linear equations. Remember the equation for the line:

$$\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n = y$$

- Let's set β_0 (our constant term) to zero (subtract from both sides) to get rid of it
  - We end up with a linear system that looks like this:

$$\beta_1 x_1 + \cdots + \beta_n x_n = y'$$

- This equation comes up a lot in machine learning and while the algorithms mostly do the work for us, we may have to do this for ourselves on occasion
  - So we have a system of n equations with n unknowns

Solving a Linear System

- Using matrices, the problem can be expressed like this:

$$Ax = y$$

- The solution is of this form:

$$A^{-1}Ax = x = A^{-1}y$$

- We can assume A is invertible (we can take its inverse)
  - NumPy already provides the means to do all this
    - Matrix inverse
    - Matrix multiply (dot product)
  - Even better is the solve method

Saving/Loading NumPy Data

- NumPy provides save and load methods for serializing data to/from disk

```
# Numpy specific saving
out_res = np.arange(10).reshape(5,2)
np.save(“./x2.npy", out_res)
np.load(“./x2.npy")
```

NumPy CSV Reader/Writer

- Because CSV file formats are so common, NumPy also provides a CSV interface

```
# Numpy csv interface
out_res = np.arange(10).reshape(5,2)
np.savetxt("./sim1.csv", out_res,delimiter=",")
np.loadtxt("./sim1.csv", delimiter=",")
```

NumPy Lab

- Do the basic Python/NumPy Lab