**NumPy**

NumPy (or Numpy) is a Linear Algebra Library for Python, the reason it is so important for Data Science with Python is that almost all of the libraries in the PyData Ecosystem rely on NumPy as one of their main building blocks.

Numpy is also incredibly fast, as it has bindings to C libraries. For more info on why you would want to use Arrays instead of lists, check out this great [StackOverflow post](http://stackoverflow.com/questions/993984/why-numpy-instead-of-python-lists).

We will only learn the basics of NumPy, to get started we need to install it!

**Installation Instructions**

**It is highly recommended you install Python using the Anaconda distribution to make sure all underlying dependencies (such as Linear Algebra libraries) all sync up with the use of a conda install. If you have Anaconda, install NumPy by going to your terminal or command prompt and typing:**

conda install numpy

**If you do not have Anaconda and can not install it, please refer to** [**Numpy's official documentation on various installation instructions.**](http://docs.scipy.org/doc/numpy-1.10.1/user/install.html)

**Using NumPy**

Once you've installed NumPy you can import it as a library:

import numpy as np

Numpy has many built-in functions and capabilities. We won't cover them all but instead we will focus on some of the most important aspects of Numpy: vectors,arrays,matrices, and number generation. Let's start by discussing arrays.

**Numpy Arrays**

NumPy arrays are the main way we will use Numpy throughout the course. Numpy arrays essentially come in two flavors: vectors and matrices. Vectors are strictly 1-d arrays and matrices are 2-d (but you should note a matrix can still have only one row or one column).

Let's begin our introduction by exploring how to create NumPy arrays.

**Creating NumPy Arrays**

**From a Python List**

We can create an array by directly converting a list or list of lists:

#declaration of list

my\_list = [1,2,3]

my\_list

[1, 2, 3]

#converting a list into array using np

np.array(my\_list)

array([1, 2, 3])

#creating a matrix list

my\_matrix = [[1,2,3],[4,5,6],[7,8,9]]

my\_matrix

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

#covering matrix into an array

np.array(my\_matrix)

array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

**Built-in Methods**

There are lots of built-in ways to generate Arrays

**arange**

Return evenly spaced values within a given interval.

#range function of any array in numpy

np.arange(0,10)

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

#range at an interval of 2

np.arange(0,11,2)

array([ 0, 2, 4, 6, 8, 10])

**zeros and ones**

Generate arrays of zeros or ones

#null vector

np.zeros(3)

array([ 0., 0., 0.])

#null array

np.zeros((5,5))

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.]])

#singleton vector

np.ones(3)

array([ 1., 1., 1.])

#singelton matrix

np.ones((3,3))

array([[ 1., 1., 1.],

[ 1., 1., 1.],

[ 1., 1., 1.]])

**linspace**

**# first it calculates the desity – 1/b-a and then multiply with the index**

Return evenly spaced numbers over a specified interval.

np.linspace(0,10,3)

array([ 0., 5., 10.])

np.linspace(0,10,50)

array([ 0. , 0.20408163, 0.40816327, 0.6122449 ,

0.81632653, 1.02040816, 1.2244898 , 1.42857143,

1.63265306, 1.83673469, 2.04081633, 2.24489796,

2.44897959, 2.65306122, 2.85714286, 3.06122449,

3.26530612, 3.46938776, 3.67346939, 3.87755102,

4.08163265, 4.28571429, 4.48979592, 4.69387755,

4.89795918, 5.10204082, 5.30612245, 5.51020408,

5.71428571, 5.91836735, 6.12244898, 6.32653061,

6.53061224, 6.73469388, 6.93877551, 7.14285714,

7.34693878, 7.55102041, 7.75510204, 7.95918367,

8.16326531, 8.36734694, 8.57142857, 8.7755102 ,

8.97959184, 9.18367347, 9.3877551 , 9.59183673,

9.79591837, 10. ])

**eye**

Creates an identity matrix

#identity matrix

np.eye(4)

array([[ 1., 0., 0., 0.],

[ 0., 1., 0., 0.],

[ 0., 0., 1., 0.],

[ 0., 0., 0., 1.]])

**Random**

Numpy also has lots of ways to create random number arrays:

**rand**

Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).

np.random.rand(2)

array([ 0.11570539, 0.35279769])

np.random.rand(5,5)

array([[ 0.66660768, 0.87589888, 0.12421056, 0.65074126, 0.60260888],

[ 0.70027668, 0.85572434, 0.8464595 , 0.2735416 , 0.10955384],

[ 0.0670566 , 0.83267738, 0.9082729 , 0.58249129, 0.12305748],

[ 0.27948423, 0.66422017, 0.95639833, 0.34238788, 0.9578872 ],

[ 0.72155386, 0.3035422 , 0.85249683, 0.30414307, 0.79718816]])

**randn**

Return a sample (or samples) from the "standard normal" distribution. Unlike rand which is uniform:

np.random.randn(2)

array([-0.27954018, 0.90078368])

np.random.randn(5,5)

array([[ 0.70154515, 0.22441999, 1.33563186, 0.82872577, -0.28247509],

[ 0.64489788, 0.61815094, -0.81693168, -0.30102424, -0.29030574],

[ 0.8695976 , 0.413755 , 2.20047208, 0.17955692, -0.82159344],

[ 0.59264235, 1.29869894, -1.18870241, 0.11590888, -0.09181687],

[-0.96924265, -1.62888685, -2.05787102, -0.29705576, 0.68915542]])

**randint**

Return random integers from low (inclusive) to high (exclusive).

np.random.randint(1,100)

44

np.random.randint(1,100,10)

array([13, 64, 27, 63, 46, 68, 92, 10, 58, 24])

**Array Attributes and Methods**

Let's discuss some useful attributes and methods or an array:

arr = np.arange(25)

ranarr = np.random.randint(0,50,10)

arr

array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,

17, 18, 19, 20, 21, 22, 23, 24])

ranarr

array([10, 12, 41, 17, 49, 2, 46, 3, 19, 39])

**Reshape**

Returns an array containing the same data with a new shape.

arr.reshape(5,5)

array([[ 0, 1, 2, 3, 4],

[ 5, 6, 7, 8, 9],

[10, 11, 12, 13, 14],

[15, 16, 17, 18, 19],

[20, 21, 22, 23, 24]])

**max,min,argmax,argmin**

These are useful methods for finding max or min values. Or to find their index locations using argmin or argmax

ranarr

array([10, 12, 41, 17, 49, 2, 46, 3, 19, 39])

ranarr.max()

49

ranarr.argmax()

4

ranarr.min()

2

ranarr.argmin()

5

**Shape**

Shape is an attribute that arrays have (not a method):

# Vector

arr.shape

(25,)

# Notice the two sets of brackets

arr.reshape(1,25)

array([[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,

17, 18, 19, 20, 21, 22, 23, 24]])

arr.reshape(1,25).shape

(1, 25)

arr.reshape(25,1)

array([[ 0],

[ 1],

[ 2],

[ 3],

[ 4],

[ 5],

[ 6],

[ 7],

[ 8],

[ 9],

[10],

[11],

[12],

[13],

[14],

[15],

[16],

[17],

[18],

[19],

[20],

[21],

[22],

[23],

[24]])

arr.reshape(25,1).shape

(25, 1)

**dtype**

You can also grab the data type of the object in the array:

arr.dtype

dtype('int64')

**Great Job!**[**¶**](http://localhost:8888/notebooks/Documents/Study%20Materials/UDEMY_PYTHON_COURSE/02-Python-for-Data-Analysis-NumPy/01-NumPy%20Arrays.ipynb#Great-Job!)

indexing:

# NumPy Indexing and Selection

In this lecture we will discuss how to select elements or groups of elements from an array.

import numpy as np

#Creating sample array

arr = np.arange(0,11)

#Show

arr

array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

## Bracket Indexing and Selection

The simplest way to pick one or some elements of an array looks very similar to python lists:

#Get a value at an index

arr[8]

8

#Get values in a range

arr[1:5]

array([1, 2, 3, 4])

#Get values in a range

arr[0:5]

array([0, 1, 2, 3, 4])

## Broadcasting

Numpy arrays differ from a normal Python list because of their ability to broadcast:

#Setting a value with index range (Broadcasting)

arr[0:5]=100

​

#Show

arr

array([100, 100, 100, 100, 100, 5, 6, 7, 8, 9, 10])

# Reset array, we'll see why I had to reset in a moment

arr = np.arange(0,11)

​

#Show

arr

array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

#Important notes on Slices

slice\_of\_arr = arr[0:6]

​

#Show slice

slice\_of\_arr

array([0, 1, 2, 3, 4, 5])

#Change Slice

slice\_of\_arr[:]=99

​

#Show Slice again

slice\_of\_arr

array([99, 99, 99, 99, 99, 99])

Now note the changes also occur in our original array!

arr

array([99, 99, 99, 99, 99, 99, 6, 7, 8, 9, 10])

Data is not copied, it's a view of the original array! This avoids memory problems!

#To get a copy, need to be explicit

arr\_copy = arr.copy()

​

arr\_copy

array([99, 99, 99, 99, 99, 99, 6, 7, 8, 9, 10])

## Indexing a 2D array (matrices)

The general format is **arr\_2d[row][col]** or **arr\_2d[row,col]**. I recommend usually using the comma notation for clarity.

arr\_2d = np.array(([5,10,15],[20,25,30],[35,40,45]))

​

#Show

arr\_2d

array([[ 5, 10, 15],

[20, 25, 30],

[35, 40, 45]])

#Indexing row

arr\_2d[1]

​

array([20, 25, 30])

# Format is arr\_2d[row][col] or arr\_2d[row,col]

​

# Getting individual element value

arr\_2d[1][0]

20

# Getting individual element value

arr\_2d[1,0]

20

# 2D array slicing

​

#Shape (2,2) from top right corner

arr\_2d[:2,1:]

array([[10, 15],

[25, 30]])

#Shape bottom row

arr\_2d[2]

array([35, 40, 45])

#Shape bottom row

arr\_2d[2,:]

array([35, 40, 45])

### Fancy Indexing

Fancy indexing allows you to select entire rows or columns out of order,to show this, let's quickly build out a numpy array:

#Set up matrix

arr2d = np.zeros((10,10))

#Length of array

arr\_length = arr2d.shape[1]

#Set up array

​

for i in range(arr\_length):

arr2d[i] = i

arr2d

array([[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],

[ 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],

[ 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.],

[ 4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],

[ 5., 5., 5., 5., 5., 5., 5., 5., 5., 5.],

[ 6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],

[ 7., 7., 7., 7., 7., 7., 7., 7., 7., 7.],

[ 8., 8., 8., 8., 8., 8., 8., 8., 8., 8.],

[ 9., 9., 9., 9., 9., 9., 9., 9., 9., 9.]])

Fancy indexing allows the following

arr2d[[2,4,6,8]]

array([[ 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],

[ 4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],

[ 6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],

[ 8., 8., 8., 8., 8., 8., 8., 8., 8., 8.]])

#Allows in any order

arr2d[[6,4,2,7]]

array([[ 6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],

[ 4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],

[ 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],

[ 7., 7., 7., 7., 7., 7., 7., 7., 7., 7.]])

## More Indexing Help

Indexing a 2d matrix can be a bit confusing at first, especially when you start to add in step size. Try google image searching NumPy indexing to fins useful images, like this one:

## Selection

Let's briefly go over how to use brackets for selection based off of comparison operators.

arr = np.arange(1,11)

arr

array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

arr > 4

array([False, False, False, False, True, True, True, True, True, True], dtype=bool)

bool\_arr = arr>4

bool\_arr

array([False, False, False, False, True, True, True, True, True, True], dtype=bool)

arr[bool\_arr]

array([ 5, 6, 7, 8, 9, 10])

arr[arr>2]

array([ 3, 4, 5, 6, 7, 8, 9, 10])

x = 2

arr[arr>x]

array([ 3, 4, 5, 6, 7, 8, 9, 10])

# Great Job!