**Installation Instructions**[**¶**](#gjdgxs)

**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**[**¶**](#30j0zll)

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

In [2]:

**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**[**¶**](#1fob9te)

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**[**¶**](#3znysh7)

### **From a Python List**[**¶**](#2et92p0)

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

In [3]:

my\_list = [1,2,3]  
my\_list

Out[3]:

[1, 2, 3]

In [4]:

np.array(my\_list)

Out[4]:

array([1, 2, 3])

In [11]:

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

Out[11]:

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

In [12]:

np.array(my\_matrix)

Out[12]:

array([list([1, 2]), list([4, 5, 6]), list([7, 8, 9])], dtype=object)

**Built-in Methods**[**¶**](#tyjcwt)

There are lots of built-in ways to generate Arrays

**arange**[**¶**](#3dy6vkm)

Return evenly spaced values within a given interval.

In [7]:

np.arange(0,100)

Out[7]:

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, 25, 26, 27, 28, 29, 30, 31, 32, 33,  
 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,  
 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,  
 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,  
 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])

In [23]:

np.arange(0,11,2)

Out[23]:

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

**zeros and ones**[**¶**](#1t3h5sf)

Generate arrays of zeros or ones

In [24]:

np.zeros(3)

Out[24]:

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

In [26]:

np.zeros((5,5))

Out[26]:

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.]])

In [27]:

np.ones(3)

Out[27]:

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

In [28]:

np.ones((3,3))

Out[28]:

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

**linspace**[**¶**](#4d34og8)

Return evenly spaced numbers over a specified interval.

In [29]:

np.linspace(0,10,3)

Out[29]:

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

In [31]:

np.linspace(0,10,50)

Out[31]:

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**[**¶**](#2s8eyo1)

Creates an identity matrix

In [37]:

np.eye(4)

Out[37]:

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

**Random**[**¶**](#17dp8vu)

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

### **rand**[**¶**](#3rdcrjn)

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

In [47]:

np.random.rand(2)

Out[47]:

array([ 0.11570539, 0.35279769])

In [46]:

np.random.rand(5,5)

Out[46]:

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**[**¶**](#26in1rg)

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

In [48]:

np.random.randn(2)

Out[48]:

array([-0.27954018, 0.90078368])

In [45]:

np.random.randn(5,5)

Out[45]:

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**[**¶**](#lnxbz9)

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

In [9]:

np.random.randint(1,100)

Out[9]:

87

In [51]:

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

Out[51]:

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

**Array Attributes and Methods**[**¶**](#35nkun2)

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

In [15]:

arr = np.arange(25)  
ranarr = np.random.randint(0,50,10)

In [16]:

arr

Out[16]:

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])

In [17]:

ranarr

Out[17]:

array([11, 31, 10, 27, 12, 34, 14, 46, 37, 39])

**Reshape**[**¶**](#1ksv4uv)

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

In [18]:

arr.reshape(5,5)

Out[18]:

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**[**¶**](#44sinio)

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

In [19]:

ranarr

Out[19]:

array([11, 31, 10, 27, 12, 34, 14, 46, 37, 39])

In [25]:

arr.min()

Out[25]:

0

In [26]:

arr.argmax()

Out[26]:

24

In [63]:

ranarr.min()

Out[63]:

2

In [60]:

ranarr.argmin()

Out[60]:

5

**Shape**[**¶**](#2jxsxqh)

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

In [65]:

*# Vector*  
arr.shape

Out[65]:

(25,)

In [66]:

*# Notice the two sets of brackets*  
arr.reshape(1,25)

Out[66]:

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]])

In [69]:

arr.reshape(1,25).shape

Out[69]:

(1, 25)

In [70]:

arr.reshape(25,1)

Out[70]:

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]])

In [76]:

arr.reshape(25,1).shape

Out[76]:

(25, 1)

**dtype**[**¶**](#z337ya)

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

In [75]:

arr.dtype

Out[75]:

dtype('int64')