COMP41680

Numerical Computing

Slides by Derek Greene

UCD School of Computer Science

Spring 2019

Overview

- NumPy Array Basics
  - 1-dimensional Arrays
  - Multidimensional Arrays
- Array Creation and Reshaping
- Array Operations
- Basic Statistics on Arrays
- Storing NumPy Data
- Using Matplotlib with NumPy
- Pandas v NumPy

Introduction to NumPy

- Standard Python containers are convenient but not designed for large scale data analysis.
- NumPy is the standard Python package for scientific computing:
  - Provides support for multidimensional arrays (i.e. matrices).
  - Implemented closer to hardware for efficiency.
  - Designed for scientific computation, useful for linear algebra and data analysis.
- NumPy can "turn Python into the equivalent of a free and more powerful version of Matlab".

http://www.numpy.org

- NumPy included in the Anaconda distribution. General convention to import numpy is using:

```
import numpy as np
```

NumPy Arrays

- The fundamental NumPy data structure is an array: a memory-efficient container that provides fast numerical operations.
- Unlike standard Python lists, a NumPy array can only contain a single type of value (e.g. only floats; only integers etc).
- The simplest type of array is 1-dimensional - i.e. a vector.
- We can manually create an array from an existing Python list:

```
a = np.array([1,2,3,4])
a
```

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

```
a.dtype
```

```
dtype('int64')
```

```
a.shape
```

```
(4,)
```

A 1-dimensional array of 4 ints

```
b = np.array([0.1,1.45,0.04])
b
```

```
array([ 0.1 ,  1.45,  0.04])
```

```
b.dtype
```

```
dtype('float64')
```

```
b.shape
```

```
(3,)
```

A 1-dimensional array of 3 floats

Basic Numerical Operations

- We can apply standard numerical operations to arrays using scalars (numbers). The operations are applied element-wise - i.e. applied separately to every element (entry) in the array.
- The operations are much faster than if run in pure Python on a standard list structure.

```
c = np.array([2,4,6,8])
c
```

```
array([2, 4, 6, 8])
```

```
c + 1
```

```
array([3, 5, 7, 9])
```

```
c - 2
```

```
array([0, 2, 4, 6])
```

```
c * 10
```

```
array([20, 40, 60, 80])
```

```
c / 10
```

```
array([ 0.2,  0.4,  0.6,  0.8])
```

These numerical operations create a new array of the same size as the original.

Accessing Values in 1D Arrays

- We can access entries in a 1-dimensional array using the same position-based notation as standard Python lists.

```
d = np.array([3.5, 6.7, 1.1, 0.6, 0.0])
```

3.5	6.7	1.1	0.6	0.0
0	1	2	3	4

Access individual entries in the array

```
d[1]
```
```
6.7
```

```
d[4]
```
```
0.0
```

```
d[-1]
```
```
0.0
```

Apply slicing to an array using the using the `[i:j]` notation

```
d[:2]
```
```
array([ 3.5,  6.7])
```

```
d[0:2]
```
```
array([ 3.5,  6.7])
```

```
d[2:]
```
```
array([ 1.1,  0.6,  0. ])
```

Accessing Values in 1D Arrays

- Important: Slicing creates a "view" on the original array, not a copy.

	4	7	3	5	1
Pos	0	1	2	3	4

[2:4]   Start at position 2, end before 4

```
a = np.array([4,7,3,5,1])
print( a[2:4] )
```

```
[3 5]
```

	4	7	3	5	1
Pos	0	1	2	3	4

[1:]   From position 1 onwards

```
print( a[1:] )
```

```
[7 3 5 1]
```

	4	7	3	5	1
Pos	0	1	2	3	4

[:3]   Stop before position 3

```
print( a[:3] )
```

```
[4 7 3]
```

Multidimensional Arrays

- An array can have > 1 dimension. A 2-dimensional array can be viewed as a matrix, with rows and columns. It has these properties:
  - Rank of array: Number of dimensions it has.
  - Shape of array: A tuple of integers giving the length of the array in each dimension.
  - Size of array: Total number of entries it contains.

0.4	2.3	4.5
1.5	0.1	1.3
3.2	0.4	3.2
2.7	2.3	6.3
0.1	0.1	0.9

Example:

Rank = 2
    i.e. 2 dimensions: rows, columns

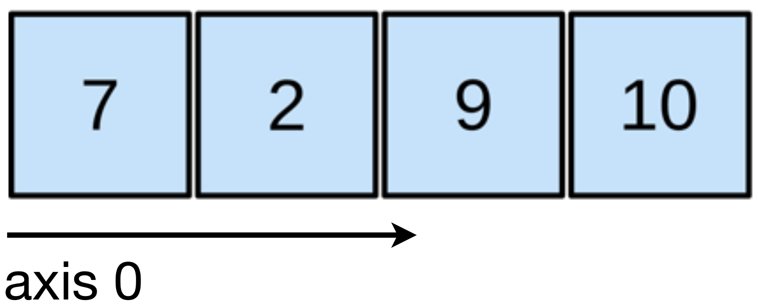
Shape = 5x3
    i.e. 5 rows x 3 columns

Size = 15
    i.e. 5 x 3 = 15 total elements

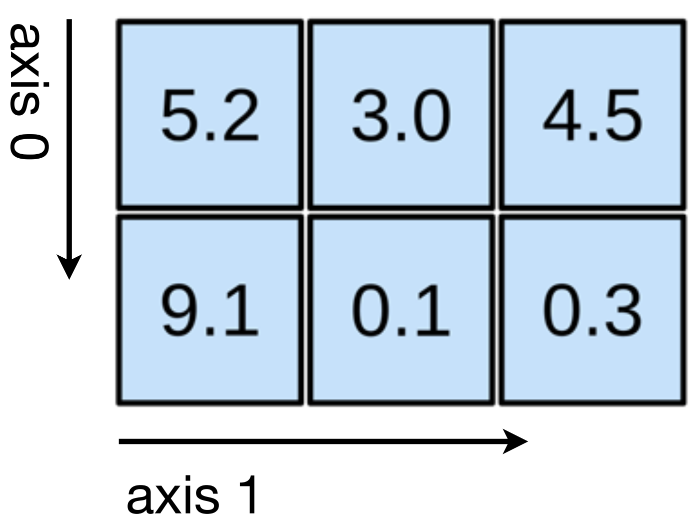
Multidimensional Arrays

- As well as creating 1-dimensional and 2-dimensional arrays, we can also create arrays with > 2 dimensions.
- Axes are defined for arrays with more than one dimension - e.g. a 2-dimensional array has two corresponding axes.

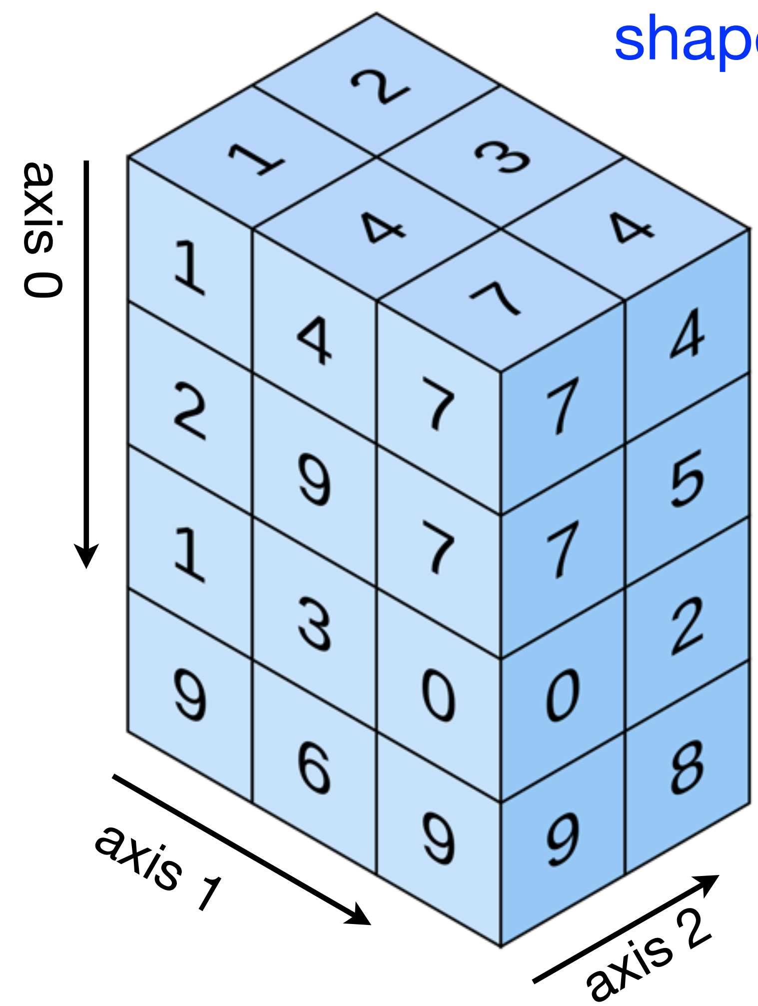
1D array: shape = (4,)



2D array: shape = (2, 3)



3D array:
shape = (4, 3, 2)



(Dashnow et al, 2017)

Multidimensional Arrays

- We can create 2D arrays from a list of Python lists.
- Note: Make sure to include the outer [ ] brackets!

```
d = np.array([[0,4,3], [9,8,6]])
print(d)
```

```
[[0 4 3]
 [9 8 6]]
```

Pass in a list containing 2 nested lists

Create a 2D array, with 2 rows and 3 columns. Total of 2x3 = 6 values

```
m = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(m)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Pass in a list containing 3 nested lists

Create a 2D array, with 3 rows and 4 columns. Total of 3x4 = 12 values

```
m.ndim
```

```
2
```

```
m.shape
```

```
(3, 4)
```

```
m.size
```

```
12
```

We can can check the rank, shape, and size of the new array.

Array Creation Functions

- Rather than using Python lists, a variety of functions are available for conveniently creating and populating arrays.

- Use the `zeros()` function to create an array full of zeros with required shape

```
np.zeros(4)
```

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

```
np.zeros((2,3))
```

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

Default type is float. For multi-dimensional arrays, specify shape as a tuple.

- Use the `ones()` function to create an array full of ones with required shape

```
np.ones((2,4))
```

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

```
np.ones((2,4),dtype=int)
```

```
array([[ 1,  1,  1,  1],
       [ 1,  1,  1,  1]])
```

Use the `dtype` parameter to tell NumPy we want an array of ints, not floats.

Array Creation Functions

- We can create an array corresponding to a sequence using the `arange()` function.

Start at 2, end before 7

```
np.arange(2,7)
```

```
array([2, 3, 4, 5, 6])
```

- We can also specify a step size for the values. The default step is 1.

```
np.arange(2,7,2)
```

```
array([2, 4, 6])
```

- The range and step sizes do not have to be integers. We can also specify floats:

```
x = np.arange(0.5, 9.4, 1.3)
print(x)
```

```
[0.5  1.8  3.1  4.4  5.7  7.   8.3]
```

Start at 0.5, increment in steps of 1.3, end before 9.4

- The `linspace()` function creates an array with a specified number of evenly-spaced samples in a given range:

```
y = np.linspace(1, 10, 4)
print(y)
```

```
[  1.   4.   7.  10.]
```

Divides up the range [1,10] into 4 evenly-spaced values, including the endpoints.

Array Shape Manipulation

- The previous functions all created 1D arrays. What if we want to create multidimensional arrays?
- We can change array shape. The original values are copied to a new array with the specified shape.

```
x = np.arange(2,8)
print(x)
```

```
[2 3 4 5 6 7]
```

Original 1D array with 6 values

```
m1 = x.reshape(3,2)
print(m1)
```

```
[[2 3]
 [4 5]
 [6 7]]
```

New 2D array with 3 rows and 2 columns, same values.

```
m2 = x.reshape(2,3)
print(m2)
```

```
[[2 3 4]
 [5 6 7]]
```

New 2D array with 2 rows and 3 columns, same values.

- The size of the reshaped array has to be same as the original. e.g. we cannot reshape a 1D array with 6 values into 2D arrays of size 2x2, 4x2, etc.

Accessing Multidimensional Arrays

- To access a value in a 1D array, specify the <u>position</u> `[i]` counting from 0, just like a Python list.
- We can also use this notation to change the values in an existing array.

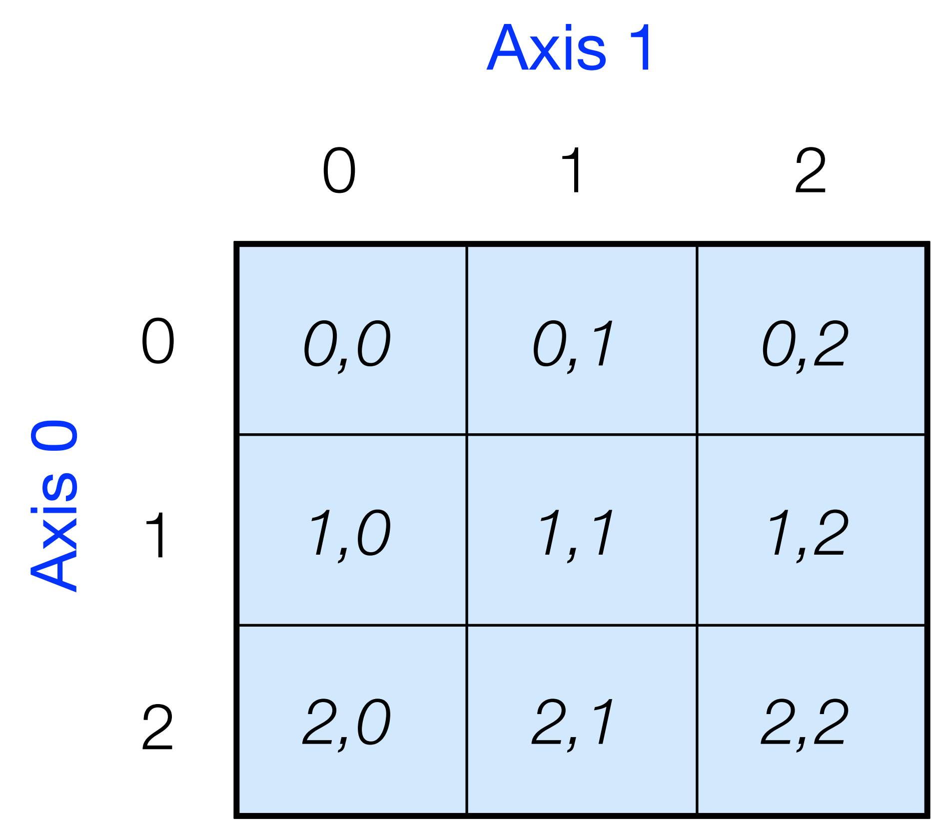
```
a = np.array([8,6,12])
a[1]
```

```
6
```

```
a[2] = 25
print(a)
```

```
[ 8  6 25]
```

- When working with arrays with more than one dimension, use the notation `[i,j]`, where the <u>position</u> in each dimension is separated by commas.
- Axis 0 refers to the rows, Axis 1 refers to the columns.

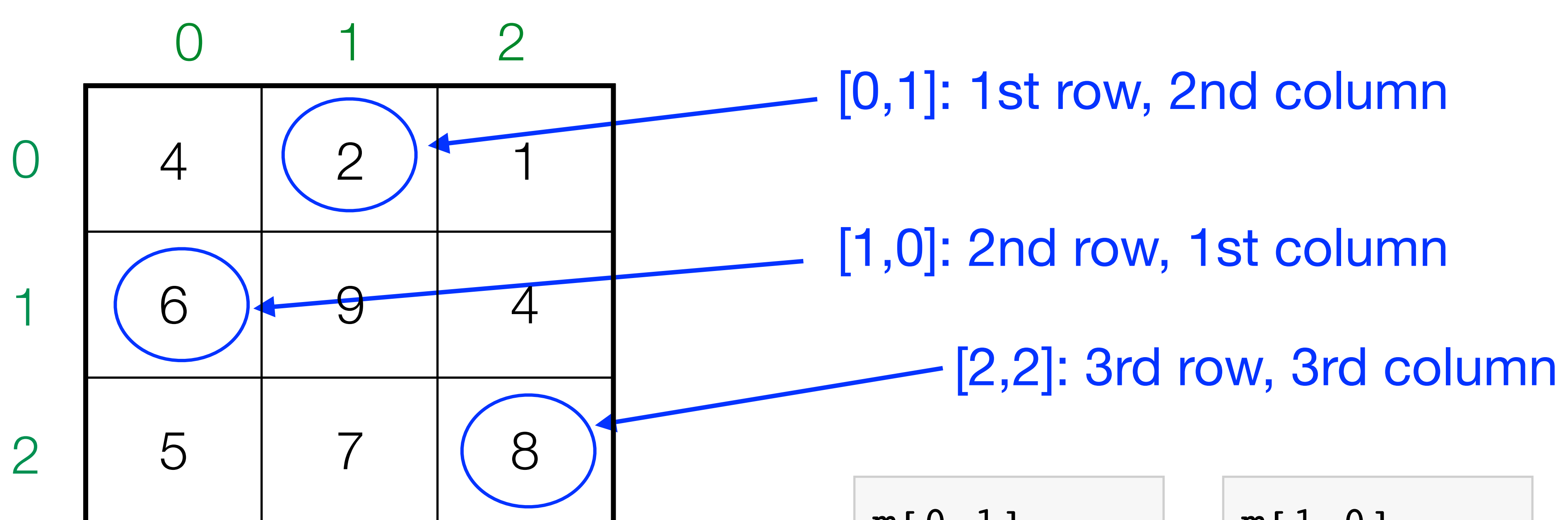


Accessing Multidimensional Arrays

- When working with arrays with more than 1 dimension, use the notation `[i,j]`, where the position in each dimension is separated by commas.

```
np.array([[4,2,1],[6,9,4],[5,7,8]])
```

Create 3x3 array




```
m[0,1]
```
```
2
```

```
m[1,0]
```
```
6
```

```
m[2,2]
```
```
8
```

- Same notation applies to higher dimensional arrays (e.g. 3D, etc).

Slicing Multidimensional Arrays

- For multidimensional arrays, we specify the slices for each dimension, separated by commas - e.g. for 2D `[i:j,p:q]`

```
d = np.array([[4,2,1],[6,9,4],[5,7,8]])
```

0	4	2	1
1	6	9	4
2	5	7	8
	0	1	2

```
d[0:2,1:3]
```

```
array([[2, 1],
       [9, 4]])
```

Rows: Start at 0, end before 2
Columns: Start at 1, end before 3

0	4	2	1
1	6	9	4
2	5	7	8
	0	1	2

```
d[1:3,0:2]
```

```
array([[6, 9],
       [5, 7]])
```

Rows: Start at 1, end before 3
Columns: Start at 0, end before 2

```
d[1,:]
```

```
array([6, 9, 4])
```

Full row at position 1

```
d[:,2]
```

```
array([1, 4, 8])
```

Full column at position 2

Iterating Over Arrays

- Generally, we want to avoid iterating over the individual elements of arrays as it is extremely slow.
- NumPy arrays are designed to be used for vectorised operations i.e. applying one operation to every value in an array at once.
- Sometimes it might be unavoidable.
  We can use a `for` loop...

```
v = np.array([1,2,3,4])
for x in v:
    print( x * 10 )
```

```
10
20
30
40
```

```
M = np.array([[1,2], [3,4]])
for row in M:
    print("row", row)
    for x in row:
        print(x)
```

```
row [1 2]
1
2
row [3 4]
3
4
```

- But, better to apply an operation to all values in an array whenever possible, unless running time does not matter.

Numerical Operations

- We can run batch operations on multidimensional arrays without for loops. These operations create a new copy of the original array.

```
d = np.array([[1,4,2], [9,8,2]])
d
```

```
array([[1, 4, 2],
       [9, 8, 2]])
```

```
d * 2
```

```
array([[ 2,  8,  4],
       [18, 16,  4]])
```

```
1.0/d
```

```
array([[1.,   0.25,   0.5      ],
       [0.11111111, 0.125, 0.5 ]])
```

```
d * d
```

```
array([[ 1, 16,  4],
       [81, 64,  4]])
```

Note that * between arrays multiplies corresponding elements together, does not perform matrix multiplication.

- We can also apply functions to all elements in an array.

```
np.log(d)
```

```
array([[ 0.        , 1.38629436, 0.69314718],
       [ 2.19722458, 2.07944154, 0.69314718]])
```

Function np.log() is applied to every element in the array.

Comparison Operations

- We can use standard boolean expressions in batch to all elements in an array. The result is a new boolean array of the same shape.

```
d = np.array([[5,2], [1,3]])
d
```

```
array([[5, 2],
       [1, 3]])
```

```
d > 2
```

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

Is each element > 2?

```
d == 1
```

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

Which elements are equal to 1?

```
d != 1
```

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

Which elements are not equal to 1?

- We can also use this approach to change certain values in arrays.

```
d[d < 5] = 0
d
```

```
array([[5, 0],
       [0, 0]])
```

Modify the original array to change all elements < 5 to 0

Basic Statistics

- NumPy arrays also have basic descriptive statistics functions.

```
a = np.array([0.1,0,1.4,0.04])
a.sum()
```

```
1.54
```

```
a.min()
```

```
0.0
```

```
a.max()
```

```
1.4
```

```
a.mean()
```

```
0.38500000000000001
```

```
a.std()
```

```
0.58709028266528129
```

Can compute the mean (average), standard deviation (std), and min/max

- For multidimensional arrays, the above can also take an optional `axis` parameter. If this is specified, calculations are only performed along that axis (dimension) and the result is a new array.

```
d = np.array([[5,4,0],[0,1,2]])
d.mean()
```

```
2.0
```

Mean based on all elements in the array

```
d.mean(axis=0)
```

```
array([ 2.5,  2.5,  1. ])
```

Average over rows, to get mean for each of the 3 columns

```
d.mean(axis=1)
```

```
array([ 3.,   1.])
```

Average over columns, to get Mean for each of the 2 rows

Storing NumPy Data

- The `np.savetxt()` function can be used to save CSV formatted version of NumPy arrays.

```
x = np.arange(1,4,0.5)
m1 = x.reshape(3,2)
print(m1)
```

```
[[ 1.    1.5]
 [ 2.    2.5]
 [ 3.    3.5]]
```

```
np.savetxt("out.txt",m1)
```

File: out.txt

```
1.00000000000000e+00 1.50000000000000e+00
2.00000000000000e+00 2.50000000000000e+00
3.00000000000000e+00 3.50000000000000e+00
```

Output file shows full precision for float values

We can also specify extra parameters specifying a format string to control the output and a separator for values.

```
np.savetxt("res.csv",m1,"%.1f",",")
```

File: out2.txt

```
1.0,1.5
2.0,2.5
3.0,3.5
```

Output file shows values are comma-separated, and only written to 1 decimal place.

Storing NumPy Data

- The `np.loadtxt()` function can be used to read CSV data from a file and create a multidimensional NumPy array from the data.
- Each line is a row, and should contain the same number of values. By default values are separated by spaces.

File: numbers.txt

```
0.8 0.8 0.9
0.7 0.1 0.2
0.6 0.4 0.1
1.0 0.8 0.2
0.9 0.1 0.4
```

```
a = np.loadtxt("numbers.txt")
a
```

```
array([[ 0.8,  0.8,  0.9],
       [ 0.7,  0.1,  0.2],
       [ 0.6,  0.4,  0.1],
       [ 1. ,  0.8,  0.2],
       [ 0.9,  0.1,  0.4]])
```

- We can also load files with other separators, by specifying the `delimiter` parameter.

File: scores.csv

```
0.74,0.63,0.58,0.89
0.91,0.89,0.78,0.99
0.43,0.35,0.34,0.45
0.56,0.61,0.66,0.58
0.50,0.49,0.76,0.72
0.88,0.75,0.61,0.78
```

```
a = np.loadtxt("scores.csv",delimiter=",")
a
```

```
array([[ 0.74,  0.63,  0.58,  0.89],
       [ 0.91,  0.89,  0.78,  0.99],
       [ 0.43,  0.35,  0.34,  0.45],
       [ 0.56,  0.61,  0.66,  0.58],
       [ 0.5 ,  0.49,  0.76,  0.72],
       [ 0.88,  0.75,  0.61,  0.78]])
```

Using Matplotlib with NumPy

- Matplotlib can be used in conjunction with NumPy arrays to visualise numeric data, in the same way we saw with Python lists.
- For example, a scatter plot of one 1D array against another:

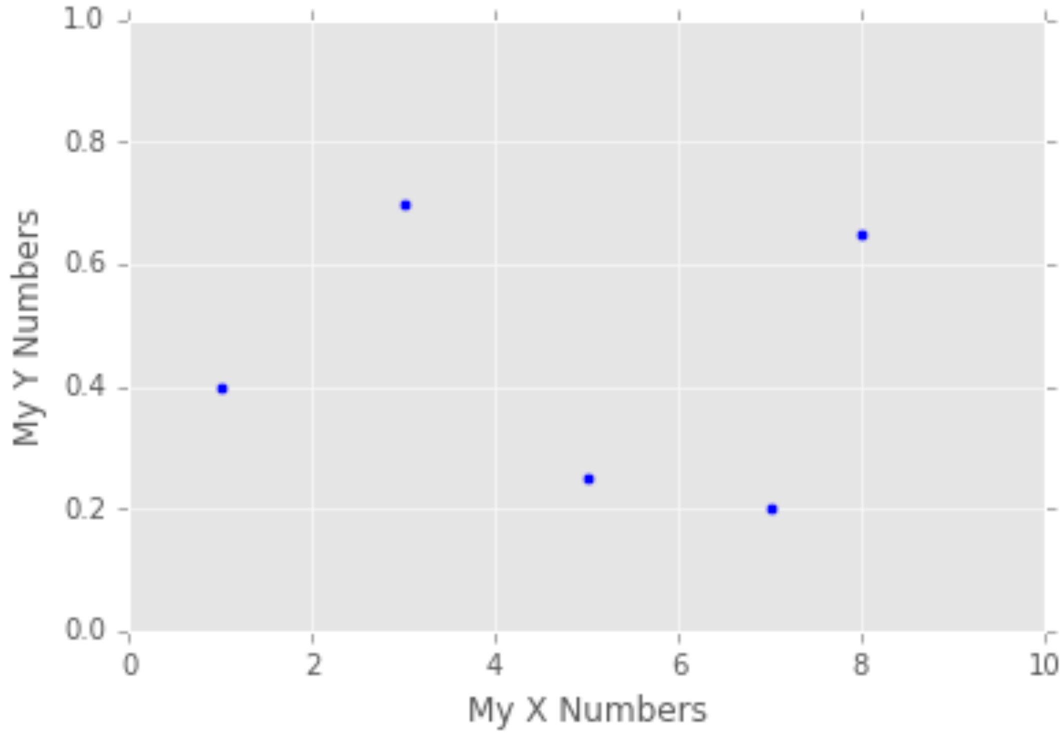
Create the X and Y values

```
xvalues = np.array([1, 5, 8, 3, 7])
yvalues = np.array([0.4, 0.25, 0.65, 0.7, 0.2])
```

Create a scatter plot of X versus Y values

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

```
plt.figure(figsize=(8,5))
plt.scatter(xvalues,yvalues)
plt.axis([0,10,0,1])
plt.xlabel("My X Numbers")
plt.ylabel("My Y Numbers")
```



Using Matplotlib with NumPy

- For 2D NumPy arrays, a common type of visualisation is a colour plot, which can be produced using `plt.pcolor()`.

Create a 5x4 2D array of random numbers

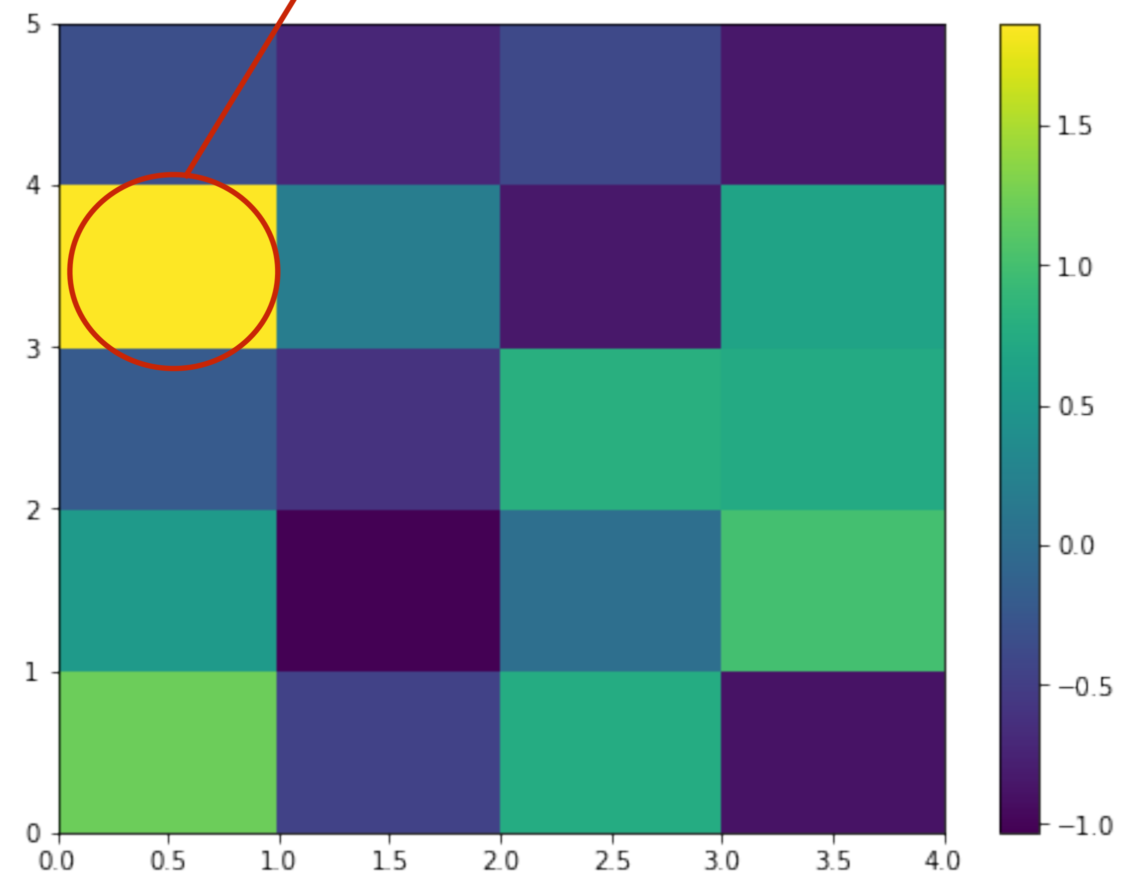
```
v = np.random.randn(20)
a = v.reshape(5,4)
```

Create the figure

```
plt.figure(figsize=(8,6))
plt.pcolor(a)
plt.colorbar()
```

Each entry in the coloured matrix corresponds to an entry in the original 2D arra.y

```
array([[ 1.2, -0.5,  0.7, -0.9],
       [ 0.5, -1. ,  0. ,  1. ],
       [-0.2, -0.6,  0.8,  0.7],
       [ 1.9,  0.2, -0.8,  0.6],
       [-0.3, -0.7, -0.4, -0.8]])
```



Pandas and NumPy

- NumPy is primarily useful for working with arrays. Highly optimised for efficient operations on numeric arrays.
- Pandas provides higher level data manipulation tools built on top of NumPy arrays, along with more semantics (e.g. indexes).
- Some operations are not as efficient, but Pandas provides additional functionality - e.g dictionary-style access via row or column index to tabular data.
- Since Pandas is built on top of NumPy, we can easily convert values between a NumPy array and a Pandas Series or Data Frame.

Create a 1D array, then construct a Series from it.

```
import numpy as np
import pandas as pd
a = np.array([0.1,0,1.4,0.04])
s = pd.Series(a)
print(s)
```

```
0     0.10
1     0.00
2     1.40
3     0.04
dtype: float64
```

Pandas and NumPy

- Since NumPy arrays do not have row or column indices, we may need to specify these if we convert an array to a Data Frame.

Create a 4x3 2D array of random numbers

```
v = np.random.randn(12)
m = v.reshape(4,3)
```

```
[[ 0.78016711 -1.53057067  0.67719719]
 [ 0.78171014  1.29939974 -0.47013332]
 [-1.28672265 -0.55481209 -0.9546214 ]
 [ 1.33540396  1.81220164
-2.05715548]]
```

Create a corresponding number of row and column index labels

```
col_index = ["A","B","C"]
row_index = ["r1","r2","r3","r4"]
```

Now use this to create a Data Frame

```
df = pd.DataFrame(m,
columns=col_index,
index=row_index )
```

	A	B	C
r1	0.780167	-1.530571	0.677197
r2	0.781710	1.299400	-0.470133
r3	-1.286723	-0.554812	-0.954621
r4	1.335404	1.812202	-2.057155