Python for X-Informatics – NumPy I

Overview

NumPy is a python package which can be used to represent data in a vectorized format. A number of other packages are built on top of numpy. These packages use the data structures and operations used in numpy.The express version of canopy (the one recommended for this class) comes with its own version of NumPy.

Download(If you are not using Canopy)

If you do not wish to use the canopy IDE numpy can be downloaded from - https://pypi.python.org/pypi/numpy. But for the purpose of the class we will be using the version of NumPy provided with Canopy.

Importing Numpy Package

```
import numpy as np
```

Importing numpy as np is a standard convention which will be used in the unit and the course. If this runs without any errors which indicates that numpy is present.

Introduction to Arrays using Numpy

N Dimensional Array(ndArray)

ndArray or an n-dimensional array is the most important data structure used in numpy. It is the data structure which gives all of the vectorization power to numpy.

Creating 1 dimensional ndArray

The **np.array()** method is used to create ndarrays. To it we pass a a sequence(like a list) of data elements.

```
s = np.array([1,2,3,4])
s
Out[100]: array([1, 2, 3, 4])
```

Creating 2 dimensional ndArray

For creating a 2-d array we need to pass a sequence of sequences (or a list of lists) where each sequence denotes a row.

```
import numpy as np
s = np.array([[2,3,4],[2,3,5],[1,4,5]])
s
```

```
Out[103]:
array([[2, 3, 4],
       [2, 3, 5],
       [1, 4, 5]])
```

We can extend this to create arrays of higher dimensions. Also it is essential that the number of elements in each of these inner lists be equal for it to be converted to a higher dimensional array.

Important Attributes and Methods of ndarray

Dimensionality of Array (ndim)

The ndim attribute of the ndarray can be used to find the dimensionality. The array s we created was a 2 dimensional array.

```
s.ndim
Out[104]: 2
```

Shape of Array(shape):

The shape attribute can be used to find the number of elements of the ndarray. s we created was a 3x3 array.

```
s.shape
Out[105]: (3, 3)
```

Size of the Array (size):

The size property tells the size of the array. The size is essentially the product of elements of the shape. Since our array s was a 3x3 array, the size is 9

```
s.size
Out[106]: 9
```

Data Type of Array Elements (dtype)

The dtype property stores the data type of the array elements. Our array was an int array and it used int32 datatype. A list of all data types can be found in the documentation - http://docs.scipy.org/doc/numpy/user/basics.types.html

```
s.dtype
Out[107]: dtype('int32')
```

Reshaping the array (reshape())

The reshape() method can be used to change the shape of the array. You can pass a tuple which indicates the new shape array and an array with the new shape is returned. The shape of the ***original array is however unaltered***. Here we change the reshape s as a 9x1 array and store it in d. However s is unaltered

```
d = s.reshape((9,1))
d
Out[109]:
array([[2],
       [3],
       [4],
       [2],
       [3],
       [5],
       [1],
       [4],
       [5]])
s
Out[110]:
array([[2, 3, 4],
       [2, 3, 5],
       [1, 4, 5]])
```

Few other methods of creating arrays

np.zeros():

To create an array with all elements set to zero use np.zeros() method. For higher dimensions pass the dimensions as a tuple.

```
np.zeros(5)
Out[111]: array([ 0.,  0.,  0.,  0.,  0.])
np.zeros((2,2))
Out[112]:
array([[ 0.,  0.],
       [ 0.,  0.]])
```

np.ones():

This method works similar to the np.zeros() method except that the array has all elements set to 1.

```
np.ones((2,2))
Out[113]:
array([[ 1.,  1.],
```

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

np.arange()

It takes the input start, end, stepsize and creates an array accordingly. And as it always in python the ***end is not included***. So it starts from start and increments by the step size and stops when value becomes larger than or equal to end.

```
np.arange(1,11,2)
Out[114]: array([1, 3, 5, 7, 9])
```

If the step size is not specified a default step size of 1 is taken.

```
np.arange(1,11)
Out[116]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

If the start is not specified a default value of 0 is taken.

```
np.arange(11)
Out[118]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

linspace()

You can specify the start, end and the number of elements needed. It create an array with first element as start, last element as end***(here end is included),*** and the array has a total number of elements as specified.

```
np.linspace(1,10,10)
Out[119]: array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

The default value for the number of elements is 50.

```
np.linspace(1,10)
Out[120]:
array([ 1.        ,  1.18367347,  1.36734694,  1.55102041,
        1.73469388,  1.91836735,  2.10204082,  2.28571429,
        2.46938776,  2.65306122,  2.83673469,  3.02040816,
        3.20408163,  3.3877551 ,  3.57142857,  3.75510204,
        3.93877551,  4.12244898,  4.30612245,  4.48979592,
        4.67346939,  4.85714286,  5.04081633,  5.2244898 ,
        5.40816327,  5.59183673,  5.7755102 ,  5.95918367,
        6.14285714,  6.32653061,  6.51020408,  6.69387755,
        6.87755102,  7.06122449,  7.24489796,  7.42857143,
        7.6122449 ,  7.79591837,  7.97959184,  8.16326531,
        8.34693878,  8.53061224,  8.71428571,  8.89795918,
```

```
    9.08163265, 9.26530612, 9.44897959, 9.63265306,
    9.81632653, 10.     ])
```

Arithmetic operations(between 2 arrays)

When we perform an operation using +,-,*,/,%,**(power or exponent) between two arrays of the same shape, it results in an element-wise operation between the two arrays.

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

Addition

```
a+b
Out[123]:
array([[3, 5],
    [5, 7]])
```

Subtraction

```
a-b
Out[124]:
array([[1, 1],
    [1, 1]])
```

Product

```
a*b
Out[125]:
array([[ 2,  6],
    [ 6, 12]])
```

This is different from a dot product which can be obtained using the **dot()** method

```
a.dot(b)
Out[127]:
array([[ 8, 13],
    [11, 18]])
```

Division

```
a/b
Out[126]:
array([[2, 1],
    [1, 1]])
```

For division as in python, ***floating point division is performed only if atleast one array has a floating data type***

Arithmetic Operations with scalars

When a scalar is added(true for any other arithmetic operation), the scalar is added to each element in the array.

Addition

```
a + 2
Out[128]:
array([[4, 5],
       [5, 6]])
```

Subtraction

```
a - 2
Out[129]:
array([[0, 1],
       [1, 2]])
```

Product

```
a * 2
Out[130]:
array([[4, 6],
       [6, 8]])
```

Division

Here as all elements/scalars involved are int values the result is an int.

```
a / 2
Out[132]:
array([[1, 1],
       [1, 2]])
```

Here as 2.0 is float, we get float precision.

```
a / 2.0
Out[133]:
array([[ 1. ,  1.5],
       [ 1.5,  2. ]])
```

Exponent:

Here the "**" operator is used to indicate that an array a is raised in an element-wise fashion by a power of 2

```
a ** 2
Out[31]:
array([[ 4,  9],
       [ 9, 16]])
```