LEARN CODING

ale66

github.com/ale66/learn-coding

NUMPY

MOTIVATIONS

Python does not cover the data structures normally used in science/engineering

Numpy comes in to support data manipulation of n-dimensional arrays.

Extensive library of functions to reshape data.

Comprehensive collection of mathematical operations.

```
1 pip install numpy
```

ARRAYS

A computer version of vectors and matrices: sequence of uniform-type values with indexing mechanism by integers.

Numpy arrays have methods, applied element-wise, and functions that take into account the position of each element in the array.

```
1 import numpy as np
```

```
1 # nr from 2 to 20 (excl.) with step 2
2
3 b = np.arange(2, 20, 2)
4
5 b
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
1  # element-wise operations
2
3  2*b
```

```
array([ 4,  8, 12, 16, 20, 24, 28, 32, 36])
```

```
1 # cumulative step-by-step sum
2 b.cumsum()
```

```
array([ 2,  6, 12, 20, 30, 42, 56, 72, 90])
```

LISTS VS. ARRAYS

Same indexing notation:

```
1 mylist[0]
2
3 mylistoflists[0][1]
```

A list is a generic sequence of heterogenous objects.

So, strings, numbers, characters, file name, URLs can be all mixed up!

An array is a sequence of strictly-homogenous objects, normally `int` or `float`

```
1 myarray[1]
2
3 mymatrix[1][3]
```

NOTATION

1-dimension: an array (sequence of numbers): `[1, 23, …]`

2-dimensions: a matrix (table of numbers) `[ [1, 23, …], [14, 96, …], ...]`

3-dimensions: a *tensor* (box/cube/cuboid) of numbers: `[ [ [1, 23, …], [14, 96, …], …], ...]`

2-D NUMPY ARRAYS

```
1  c = np.arange(8)
2
3  c
```

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

```
1  # build a 2-dimensional array from a 1-d one
2  d = np.array([c, c*2])
3
4  d
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 0,  2,  4,  6,  8, 10, 12, 14]])
```

```
1  # count elements
2
3  d.size
```

```
16
```

```
1  #  size along each dimension
2
3  d.shape
```

```
(2, 8)
```

AXES

Numpy arrays can have multiple dimensions

```
1 # operations along columns
2 d
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 0,  2,  4,  6,  8, 10, 12, 14]])
```

```
1 # operations along columns
2 d.sum(axis=0)
```

```
array([ 0,  3,  6,  9, 12, 15, 18, 21])
```

```
1  # summing by row
2  d.sum(axis=1)
```

```
array([28, 56])
```

N. B. unlike Pandas, not specifying the axis will apply a function to the entire array.

```
1  # sum the whole content
2  d.sum()
```

```
np.int64(84)
```

SHAPES

Using information about the shape we can create/manipulate (or reshape, or transpose) Numpy variables.

```
1  # Create 2x3 Numpy array and initialise it to 0s
2  e = np.zeros((2, 3), dtype = 'i')
3
4  e
```

```
array([[0, 0, 0],
       [0, 0, 0]], dtype=int32)
```

```
1  # Change the shape
2  e.reshape(3, 2)
```

```
array([[0, 0],
       [0, 0],
       [0, 0]], dtype=int32)
```

```
1 # a new array with exactly the same shape as 'e' and type integer
2 f = np.ones_like(e, dtype = 'i')
3 f
```

```
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

```
1 # Transposition
2 g = np.arange(6).reshape(3,2)
3 g
```

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

```
1 g.T
```

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

STACKING

2-D arrays with the same dimensions can be merged

```
1  # Create an identity matrix of order 5
2  i = np.eye(5)
3
4  i
```

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

```
1  # stacking combines two 2-d arrays: vertically
2  np.vstack((i, i))
```

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

```
1  # stacking combines two 2-d arrays: horizontally
2  np.hstack((i, i))
```

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

DETOUR: N-DIMENSIONAL ARRAYS

Numpy can handle multiple dimensions.

This is useful when dealing with multivariate data, from time series to documents.

```
1  # N-dimensional array
2
3  g = np.zeros((2, 3, 4))
4
5  g
```

```
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.]]])
```

Two samples, each with three rows and four columns.

SLICING BY BOOLEAN FILTERS

Data can be selected according to specific conditions.

The Boolean filter itself can be represented by a Numpy array

```
1 l = np.array(np.arange(9))
2
3 l
```

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

```
1 l.reshape(3, 3)
2
3 l
```

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

MORE SLICING

```
1  # Let's apply a high-pass filter
2
3  l[l>4]
```

```
array([5, 6, 7, 8])
```

```
1  # Generate a Boolean array
2
3  (l>4)
```

```
array([False, False, False, False, False,  True,  True,  True,  True])
```

```
1  # now with integers: False=0, True=1)
2
3  (l>4).astype(int)
```

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