# TP00-IntroductionToNumPy

September 18, 2019

## 1 Introduction to Numpy

NumPy is THE fundamental library for scientific calculation in python. You can play with this library to do deeplearning but NumPy is not the best choice... Nevertheless, most scientific libs rely on NumPy conventions and APIs so it is important to have some knowledges about it. If you are familiar with NumPy, you can skip this section and go to the section *Introduction to Tensorflow*

### 1.1 The `ndarray` class

To use NumPy you should import it with the following command

[1]:
```
import numpy as np
```

Now you can use Numpy with the shortcut `np`.

The fundamental class of NumPy is `ndarray`. It represents table of items, with the following constraints:

- It's *multidimensional* (1d, 2d, 3d, ..., nd),
- It's *homogeneous*, that is, all items inside the table should belong to the same type.

[2]:
```
# Ndarray instanciation from known values
a = np.array([[1., 2., 3.], [3., 4., .5]])
a
```

[2]:
```
array([[1. , 2. , 3. ],
       [3. , 4. , 0.5]])
```

[3]:
```
# Type of a
type(a)
```

[3]:
```
numpy.ndarray
```

[4]:
```
# 'Rank' as mention in NumPy doc or number of dimensions
a.ndim
```

[4]:
```
2
```

[5]:
```
# Shape of the ndarray
a.shape
```

[5]:
```
(2, 3)
```

[6]:
```
# Total number of items
a.size
```

[6]:
```
6
```

[7]:
```
# Item type
a.dtype
```

[7]:
```
dtype('float64')
```

[8]:
```
# Actual data of the table
a.data
```

[8]:
```
<memory at 0x7f2010a27120>
```

### 1.2 Creation of a `ndarray`

The basic constructors of `ndarray`are :

- `numpy.array(object, dtype=None, copy=True, order=K, subok=False, ndmin=0)` Create an array from known values
- `numpy.zeros(shape, dtype=float, order=C)` Create an array full of zeros
- `numpy.ones(shape, dtype=None, order=C)` Create an array full of ones

`dtype` determines the type of each element, `order` indicates how elements are organized into `data` .

**Take Care !** `dtype` is determined at instanciation and can not be changed after.

[9]:
```
np.array([[1., 2., 3.], [3., 4., .5]])
```

[9]:
```
array([[1. , 2. , 3. ],
       [3. , 4. , 0.5]])
```

[10]:
```
np.zeros((5, 3))
```

[10]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[11]:
```
np.ones((2, 2, 3), dtype='int')
```

[11]:
```
array([[[1, 1, 1],
        [1, 1, 1]],

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

Some other useful methods are :

- `numpy.arange([start, ]stop, [step, ]dtype=None)`
- `numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)`

- `numpy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)`
- `numpy.eye(N, M=None, k=0, dtype=float)`
- `numpy.random.randn(d0, d1, ..., dn)`

More creation routines are available here.

[12]:
```
np.arange(5, 10, 1)
```

[12]:
```
array([5, 6, 7, 8, 9])
```

[13]:
```
np.arange(0, 10, 2)
```

[13]:
```
array([0, 2, 4, 6, 8])
```

[14]:
```
np.linspace(0, 10, 20)
```

[14]:
```
array([ 0.        ,  0.52631579,  1.05263158,  1.57894737,  2.10526316,
        2.63157895,  3.15789474,  3.68421053,  4.21052632,  4.73684211,
        5.26315789,  5.78947368,  6.31578947,  6.84210526,  7.36842105,
        7.89473684,  8.42105263,  8.94736842,  9.47368421, 10.        ])
```

[15]:
```
np.logspace(1, 10, 20)
```

[15]:
```
array([1.00000000e+01, 2.97635144e+01, 8.85866790e+01, 2.63665090e+02,
       7.84759970e+02, 2.33572147e+03, 6.95192796e+03, 2.06913808e+04,
       6.15848211e+04, 1.83298071e+05, 5.45559478e+05, 1.62377674e+06,
       4.83293024e+06, 1.43844989e+07, 4.28133240e+07, 1.27427499e+08,
       3.79269019e+08, 1.12883789e+09, 3.35981829e+09, 1.00000000e+10])
```

[16]:
```
np.eye(3)
```

[16]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

[17]:
```
np.eye(3, 4)
```

[17]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

[18]:
```
np.random.randn(3, 4)
# !!! shape is given dimension by dimension as arguments not in one tuple
```

[18]:
```
array([[-0.0435123 ,  2.12342016, -0.3517692 , -0.37803699],
       [-0.6603126 ,  1.47938205,  0.74353841,  0.01077219],
       [ 1.25235323,  0.17773648, -1.34153894,  0.83950519]])
```

## 1.3 Indexation / Slicing

### 1.3.1 Monodimensional indexation

Indexing and slicing are done with the operator `[]` as for list.

[19]:
```
a = np.random.randn(10)
a
```

[19]:
```
array([ 0.38561388,  0.1312874 , -1.77371446,  0.20922443,  1.99598316,
       -1.10727819,  1.07983388,  1.12979408,  0.8988875 , -2.33405206])
```

[20]:
```
# First item
a[0]
```

[20]:
```
0.3856138828937419
```

[21]:
```
# Last item
a[-1]
```

[21]:
```
-2.334052056277316
```

[22]:
```
# From item 2 to item 5 (excluded !)
a[2:5]
```

[22]:
```
array([-1.77371446,  0.20922443,  1.99598316])
```

[23]:
```
# Eliptic formulation
# 3 first items
a[:3]
```

[23]:
```
array([ 0.38561388,  0.1312874 , -1.77371446])
```

[24]:
```
# Starting from the 4th item
a[3:]
```

[24]:
```
array([ 0.20922443,  1.99598316, -1.10727819,  1.07983388,  1.12979408,
        0.8988875 , -2.33405206])
```

[25]:
```
# All items
a[:]
```

[25]:
```
array([ 0.38561388,  0.1312874 , -1.77371446,  0.20922443,  1.99598316,
       -1.10727819,  1.07983388,  1.12979408,  0.8988875 , -2.33405206])
```

[26]:
```
# With a step
a[2:8:2]
```

[26]:
```
array([-1.77371446,  1.99598316,  1.07983388])
```

[27]:
```
# Reverse
a[::-1]
```

[27]:
```
array([-2.33405206,  0.8988875 ,  1.12979408,  1.07983388, -1.10727819,
        1.99598316,  0.20922443, -1.77371446,  0.1312874 ,  0.38561388])
```

### 1.3.2 Multidimensional indexation

[28]:
```
b = np.random.randn(3, 4, 5)
b
```

[28]:
```
array([[[-1.45499873,  0.60968565,  0.96500901, -1.56774256,
         -0.5823915 ],
        [ 0.94823332,  0.24173161, -0.24645756,  0.26872042,
```

```
        1.67306428],
       [ 0.04360488,  0.55531735, -1.24950238, -0.25182346,
        -0.88974734],
       [ 0.11518298, -1.43411316,  1.50391982, -0.56323472,
        -0.48075885]],

      [[ 0.20507796, -0.42739154,  0.08603242, -1.20745744,
         0.01711814],
       [-1.29099347,  0.02268427,  1.05075633, -0.44586989,
         1.45876102],
       [-0.19055536,  1.36780152,  0.0881775 ,  1.29125986,
        -0.18357063],
       [ 2.23198325,  0.5520259 ,  0.7517003 , -2.42938987,
         0.20634189]],

      [[-2.73825075,  1.02797841, -1.7735629 ,  0.66537866,
        -0.39395895],
       [-1.69521282, -0.19509658,  0.68660841, -0.15368767,
        -0.89636285],
       [ 1.39049358, -0.65005792, -0.36788816, -0.34044015,
        -0.31725071],
       [-0.18828673, -0.18759523, -0.87914565, -1.65259218,
        -0.92335552]]])
```

[29]:
```
# First item on each axis
b[0, 0, 0]
```

[29]:
```
-1.4549987250944822
```

[30]:
```
# With an interval and ann elipse
b[:, 1, 2:5]
```

[30]:
```
array([[-0.24645756,  0.26872042,  1.67306428],
       [ 1.05075633, -0.44586989,  1.45876102],
       [ 0.68660841, -0.15368767, -0.89636285]])
```

[31]:
```
# a[2] is equivalent to a[2,:,:]
b[2]
```

[31]:
```
array([[-2.73825075,  1.02797841, -1.7735629 ,  0.66537866, -0.39395895],
       [-1.69521282, -0.19509658,  0.68660841, -0.15368767, -0.89636285],
       [ 1.39049358, -0.65005792, -0.36788816, -0.34044015, -0.31725071],
       [-0.18828673, -0.18759523, -0.87914565, -1.65259218, -0.92335552]])
```

[32]:
```
# Multiple elipses : c[1,...,2] is equivalent to c[1,:,:,2] on 4-D array
c = np.random.randn(2, 2, 2, 3)
c
```

[32]:
```
array([[[[-0.35956823,  0.65123647, -0.79485964],
         [ 1.38711688, -0.72095591, -0.31142876]],
```

```
         [[ 0.11768983, -0.60527716, -1.64752649],
          [ 0.92145919, -0.6481817 , -0.85347846]]],


        [[[ 0.30409495, -0.0871556 , -1.46601627],
          [ 1.28863748,  0.18944056, -0.79280346]],

         [[ 0.06680389,  1.32417265,  0.11993329],
          [-0.07648969,  0.95043854,  1.4697187 ]]]])
```

[33]:
```
c[1, ..., 2]
```

[33]:
```
array([[-1.46601627, -0.79280346],
       [ 0.11993329,  1.4697187 ]])
```

[34]:
```
c[1, :, :, 2]
```

[34]:
```
array([[-1.46601627, -0.79280346],
       [ 0.11993329,  1.4697187 ]])
```

### Indexation implies cut in dimension ! (Warning for Matlab users)
Important for matrix operation (multiplication...)

[35]:
```
a = np.random.randn(4, 3)
a
```

[35]:
```
array([[ 0.97215472, -0.78910336,  0.44128341],
       [ 0.18547472, -1.22244959,  0.9214553 ],
       [ 0.17964464,  1.33731686,  0.13527983],
       [-1.13387243,  0.64657783, -0.0359727 ]])
```

[36]:
```
b = a[:, 0]
b
```

[36]:
```
array([ 0.97215472,  0.18547472,  0.17964464, -1.13387243])
```

[37]:
```
# b has shape (4,) not (4,1)
b.shape
```

[37]:
```
(4,)
```

[38]:
```
c = a[0, :]
c
```

[38]:
```
array([ 0.97215472, -0.78910336,  0.44128341])
```

[39]:
```
# c has shape (3,) not (1,3)
c.shape
```

[39]:
```
(3,)
```

[40]:
```
# Meanwhile using slice and not index preserves dimension
d = a[0:1, :]
d
```

[40]:
```
array([[ 0.97215472, -0.78910336,  0.44128341]])
```

[41]:
```
d.shape
```

[41]:
```
(1, 3)
```

### 1.4 Assignation

Assignation is performed by the operator =. Item or a sub-array can be targeted.

[42]:
```
a = np.array([[1, 2, 3], [4, 5, 6]], dtype=int)
a
```

[42]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[43]:
```
a[0, 0] = 10
a
```

[43]:
```
array([[10,  2,  3],
       [ 4,  5,  6]])
```

[44]:
```
a[0:2, 1:3] = np.ones((2, 2))
a
```

[44]:
```
array([[10,  1,  1],
       [ 4,  1,  1]])
```

**Take Care !** `dtype` is determined at instanciation and can not be changed after.

[45]:
```
# 1.75 will be downcast before assignation
a[1, 0] = 1.75
a
```

[45]:
```
array([[10,  1,  1],
       [ 1,  1,  1]])
```

### 1.5 Resize operation

Arrays can be reshaped by the `resize` method. That's an in-place operation:

[46]:
```
a.resize((3, 2))
a
```

[46]:
```
array([[10,  1],
       [ 1,  1],
       [ 1,  1]])
```

### 1.6 References, view and copy

If `a` and `b` reference the same `ndarray`, all operation on aalso applied to `b`. They share both data and metadata.

If `c` is a view of `a`, they share the same data but not the metadata. For example shapes can be modified separately. But if we change the first element of `c`, the first element of `a` is also changed.

If `d` is a copy of `a`, all data and metadata are separated.

[47]:
```
a = np.random.randn(4, 3)
a
```

[47]:
```
array([[-0.27481389,  0.59862034,  0.01464082],
       [ 2.22461135, -0.16125853, -0.04968828],
       [-0.02298794, -0.41436418, -0.92506087],
       [ 1.08093282, -0.66716209, -0.62792629]])
```

[48]:
```
# b is a reference to a
b = a
b[0, 0] = 1
a
```

[48]:
```
array([[ 1.        ,  0.59862034,  0.01464082],
       [ 2.22461135, -0.16125853, -0.04968828],
       [-0.02298794, -0.41436418, -0.92506087],
       [ 1.08093282, -0.66716209, -0.62792629]])
```

[49]:
```
# c is a view of a
c = a.view()
c.resize(3, 4)
c
```

[49]:
```
array([[ 1.        ,  0.59862034,  0.01464082,  2.22461135],
       [-0.16125853, -0.04968828, -0.02298794, -0.41436418],
       [-0.92506087,  1.08093282, -0.66716209, -0.62792629]])
```

[50]:
```
# Shape of a is not affected
a
```

[50]:
```
array([[ 1.        ,  0.59862034,  0.01464082],
       [ 2.22461135, -0.16125853, -0.04968828],
       [-0.02298794, -0.41436418, -0.92506087],
       [ 1.08093282, -0.66716209, -0.62792629]])
```

[51]:
```
# But if we modify the last element of c, the last element of a is changed
c[2, 3] = 0
a
```

[51]:
```
array([[ 1.        ,  0.59862034,  0.01464082],
       [ 2.22461135, -0.16125853, -0.04968828],
       [-0.02298794, -0.41436418, -0.92506087],
       [ 1.08093282, -0.66716209,  0.        ]])
```

[52]:
```
# d is a copy of a
d = a.copy()
d
```

[52]:
```
array([[ 1.        ,  0.59862034,  0.01464082],
       [ 2.22461135, -0.16125853, -0.04968828],
       [-0.02298794, -0.41436418, -0.92506087],
       [ 1.08093282, -0.66716209,  0.        ]])
```

[53]:
```
d[0, 0] = 3
d
```

[53]:
```
array([[ 3.        ,  0.59862034,  0.01464082],
       [ 2.22461135, -0.16125853, -0.04968828],
       [-0.02298794, -0.41436418, -0.92506087],
       [ 1.08093282, -0.66716209,  0.        ]])
```

[54]:
```
# a was not modified by the assigniation on d
a
```

[54]:
```
array([[ 1.        ,  0.59862034,  0.01464082],
       [ 2.22461135, -0.16125853, -0.04968828],
       [-0.02298794, -0.41436418, -0.92506087],
       [ 1.08093282, -0.66716209,  0.        ]])
```

### 1.7 Shape manipulation

- `ndarray.resize(new shape, refcheck=True)` Resize in-place
- `ndarray.reshape(shape, order=C)` Return a view with a new shape
- `ndarray.ravel(order=C)` Return a flatten view
- `ndarray.flatten(order=C)` Return a flatten copy
- `numpy.concatenate((a1, a2, ...), axis=0)` Return a concatenation of arrays along an existing axis
- `numpy.stack((a1, a2, ...), axis=0)` Return a stack of arrays along a new axis

[ ]:
```

```

### 1.8 Operations

Simple operations +, -, *, **, / operate item by item

[55]:
```
a = np.random.randn(4, 3)
a
```

[55]:
```
array([[ 0.92405372,  0.09652336, -0.31998292],
       [ 0.18227558, -0.64323192,  2.00728865],
       [ 0.38859642, -0.15633407,  0.42904298],
       [ 0.59067084, -0.46896047, -1.29174562]])
```

[56]:
```
b = np.random.randn(4, 3)
b
```

[56]:
```
array([[ 0.30297353, -0.99762754, -0.73577338],
       [ 0.0567732 , -0.90825298, -2.00021683],
       [-0.31508253,  0.31644956,  1.93990276],
       [-1.11142293,  0.23842539,  0.30297604]])
```

[57]:
```
a + b
```

[57]:
```
array([[ 1.22702725, -0.90110419, -1.0557563 ],
       [ 0.23904878, -1.5514849 ,  0.00707182],
       [ 0.0735139 ,  0.1601155 ,  2.36894574],
       [-0.52075209, -0.23053509, -0.98876958]])
```

[58]:
```
a * b
```

[58]:
```
array([[ 0.27996382, -0.09629436,  0.23543491],
       [ 0.01034837,  0.58421731, -4.01501255],
       [-0.12243994, -0.04947185,  0.83230166],
       [-0.65648512, -0.11181208, -0.39136797]])
```

[59]:
```
a ** 2
```

[59]:
```
array([[0.85387528, 0.00931676, 0.10238907],
       [0.03322439, 0.4137473 , 4.02920773],
       [0.15100718, 0.02444034, 0.18407788],
       [0.34889205, 0.21992392, 1.66860674]])
```

Arrays can be viewed as set where we can get min/max of all items.

[60]:
```
a.min()
```

[60]:
```
-1.2917456189907488
```

[61]:
```
a.max()
```

[61]:
```
2.007288651529005
```

[62]:
```
# Position of the min in the flatten view of the array
a.argmin()
```

[62]:
```
11
```

[63]:
```
a.argmax()
```

[63]:
```
5
```

If you want to compute an extremum along a particular axis, you should precise `axis` in argument. As indexing, this reduce the dimension of the array. If you want to keep the same number of dimension, you should set the `keepdims` argument to `True`.

[64]:
```
a.min(axis=1)
```

[64]:
```
array([-0.31998292, -0.64323192, -0.15633407, -1.29174562])
```

[65]:
```
a.min(axis=1, keepdims=True)
```

[65]:
```
array([[-0.31998292],
       [-0.64323192],
       [-0.15633407],
       [-1.29174562]])
```

### 1.9 Broadcasting

Broadcasting is a mechanism to automatically tile arrays of incompatible dimensions before an operation. This is a powerfull mechanism (you don't have to use `repmat` as in Matlab) but it can

hide dimensionality errors.

[66]:
```
a = np.array([[1, 2, 3, 4]])
a.shape
```

[66]:
```
(1, 4)
```

[67]:
```
b = np.array([[5], [6], [7]])
b.shape
```

[67]:
```
(3, 1)
```

[68]:
```
# a and b are tiled, line by line for a, column by column for b, to enable the␣
 ↪add operation
c = a+b
c
```

[68]:
```
array([[ 6,  7,  8,  9],
       [ 7,  8,  9, 10],
       [ 8,  9, 10, 11]])
```

[69]:
```
c.shape
```

[69]:
```
(3, 4)
```

## 1.10 Linear Algebra

### 1.10.1 1D arrays

- `numpy.inner(a, b)` Return the inner/scalar product of 2 vectors
- `numpy.outer(a, b)` Return the outer product of 2 vectors

[70]:
```
a = np.array([1, 2, 3, 4])
a
```

[70]:
```
array([1, 2, 3, 4])
```

[71]:
```
b = np.array([5, 6, 7, 8])
b
```

[71]:
```
array([5, 6, 7, 8])
```

[72]:
```
np.inner(a, b)
```

[72]:
```
70
```

[73]:
```
np.outer(a, b)
```

[73]:
```
array([[ 5,  6,  7,  8],
       [10, 12, 14, 16],
       [15, 18, 21, 24],
       [20, 24, 28, 32]])
```

### 1.10.2 2D arrays

- `a.T` is the transposition of `a`

- `numpy.dot(a, b)` return the matrix product between a and b.

[74]:
```
a = np.random.randn(3, 5)
a.T
```

[74]:
```
array([[ 0.35970533,  0.82933675, -0.28593392],
       [ 0.11714446,  0.48988585, -0.21440799],
       [ 0.37182741, -0.94559446,  0.80436999],
       [ 0.04891257,  1.02279121,  0.33140726],
       [ 0.48148513, -1.26135326,  0.7512605 ]])
```

[75]:
```
a = np.random.randn(3, 5)
b = np.random.randn(5, 2)
np.dot(a, b)
```

[75]:
```
array([[-1.93791225, -0.80052914],
       [-1.0058203 ,  1.46348968],
       [-1.82776987, -1.80662118]])
```

[76]:
```
# Equivalent notation
a.dot(b)
```

[76]:
```
array([[-1.93791225, -0.80052914],
       [-1.0058203 ,  1.46348968],
       [-1.82776987, -1.80662118]])
```

[77]:
```
# Since python 3.5, the @ symbol can be used for matrix multiplication
a @ b
```

[77]:
```
array([[-1.93791225, -0.80052914],
       [-1.0058203 ,  1.46348968],
       [-1.82776987, -1.80662118]])
```

## 1.11 Saving and loading data

### 1.11.1 Input

- `numpy.load(file, mmap_mode=None, allow_pickle=True, fix_imports=True, encoding='ASCII')`

load a `npy` or `npz` file, * `numpy.loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0)`
load a `txt` file.

### 1.11.2 Output

- `numpy.save(file, arr, allow_pickle=True, fix_imports=True)`

save ONE array into a `npy` file,

- `numpy.savez(file, *args, **kwds)`

save many arrays into an `npz` file,

- `numpy.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', footer='', comments='# ')`

  save ONE array into a `txt` file,

### 1.12 Your turn

Try to answer each following questions by a small snippet of code.

1. How to reverse a vector (1d array) ?

[78]:
```
z = np.random.randn(10,)
print(z)
print(z[::-1])
```

```
[ 2.54928284 -0.89357957 -0.39332602  0.55477184  0.30195226 -0.40189679
  2.09471715  0.65917889  1.11062961  1.56091121]
[ 1.56091121  1.11062961  0.65917889  2.09471715 -0.40189679  0.30195226
  0.55477184 -0.39332602 -0.89357957  2.54928284]
```

2. How to keep dimension consistency when slicing a matrix (2d array) ?

[79]:
```
z = np.random.randn(5, 4)
z[2:3]
```

[79]:
```
array([[-1.64901303, -1.45377755,  0.77134663, -0.61366275]])
```

3. How to create a (5,5) array with random values and find the extrema values ?

[80]:
```
z = np.random.randn(5, 5)
print(z.min())
print(z.max())
```

```
-2.7425843188801458
2.3099695446346757
```

4. With the help of broadcasting, how to produce a matrix A where A[i,j] = 2i + j ? (no for loop allowed)

[81]:
```
a = np.arange(0, 5).reshape((5, 1))
A = 2*a+a.T
A
```

[81]:
```
array([[ 0,  1,  2,  3,  4],
       [ 2,  3,  4,  5,  6],
       [ 4,  5,  6,  7,  8],
       [ 6,  7,  8,  9, 10],
       [ 8,  9, 10, 11, 12]])
```

5. A is a (4,4) int array, I want to change the last element of A to 1.5 without loosing any information. How can I do it ?

[82]:
```
A = np.zeros((4, 4), dtype=int)
B = np.array(A, dtype=float)
B[-1, -1] = 1.5
B
```

[82]:
```
array([[0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 1.5]])
```