Lecture 7 Introduction to Numpy

[NumPy -- *Numerical Python* (https://numpy.org/)](https://numpy.org/) provides the building-blocks for the entire ecosystem of data science tools in Python, serving as the efficient tool to store and manipulate data, and [friendly to Matlab users (https://numpy.org/doc/stable/user/numpy-for-matlab-users.html)](https://numpy.org/doc/stable/user/numpy-for-matlab-users.html).

In [ ]:
```
import numpy as np

my_arr = np.arange(1000000)
my_list = list(range(1000000))
```

In [ ]:
```
%time for _ in range(10): my_arr2 = my_arr * 2
```

In [ ]:
```
%time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

Difference between ndarray and list : Data Memory Perspective

[Intuitively speaking (https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html)](https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html), the built-in list object in Python can be viewed as the "address book" that store multiple pointers to heterogeneous objects in Python as its elements. On the other, the Numpy array object in Python stored the pointer to a consecutive memory block (data buffer) implemented in C language -- that's why the elements in Numpy array should be fixed-type, and the implementation is more efficient than list.

In [ ]:
```
a = np.array([1,2,3,4]) #numpy 1-d array, initialization with list
l = [1,2,3,4]  # python built-in list
```

Slicing of Numpy array creates *View* instead of *Copy*. The view object shares the same data buffer with the original one.

In [ ]:
```
b = a[0:2] # creating view by slicing
```

In [ ]:
```
print(b)
b.base # view has the base object because its memory is from some other object.
```

We can also check the `flags` to see whether the array has its "own data".

In [ ]:
```
b.flags
```

In [ ]:
```
a.flags
```

This mechanism may cause unexpected outcomes for beginners.

In [ ]:
```
b[0] = 1000
a
```

This is very different with the Python built-in list.

In [ ]:
```
c = l[0:2]
c[0] = 100
l
```

Many other methods/functions in Numpy creates view instead of copy (in fact view is far more efficient than copy).

For example, Reshape creates the view whenever possible (for most of the case with consistent dimensions).

In [ ]:
```
a_mat = a.reshape(2,2)
```

In [ ]:
```
a_mat.base
```

In [ ]:
```
a_mat[0,0] = 2000 # same as a_mat[0][0]
a
```

Transpose also creates the view.

In [ ]:
```
a_t = a_mat.T # attribute
a_tt = a_mat.transpose() # method
```

In [ ]:
```
a_t.base
```

In [ ]:
```
a_t[0,0] = 0
a
```

By the way, once the "base" is changed, all the associated "view" objects are changed!

In [ ]:
```
a_mat
```

In [ ]:
```
b
```

Use the copy method to create the new data buffer

In [ ]:
```
a_copy = a.copy()
a_copy.base
```

In [ ]:
```
a_copy.flags
```

In [ ]:
```
a_mat_copy = a_mat.copy()
```

In [ ]:
```
a_mat_copy.flags
```

Numpy ndarray as object

As the object created by Numpy, the ndarray has identity, type, value, attributes and methods.

In [ ]:
```
type(a)
```

In [ ]:
```
dir(a)
```

In [ ]:
```
dir(a)
```

In [ ]:
```
a.shape # 1-d array with length 4 -- different with 4x1 2-d array!
```

In [ ]:
```
a_mat.shape
```

In [ ]:
```
a_mat.tolist()
```

In [ ]:
```
a.mean()
```

In [ ]:
```
help(a.mean)
```

In [ ]:
```
np.mean(a)
```

In [ ]:
```
help(a.reshape)
```

Dimension and Axis of ndarray

Numpy use the term *dimension* and *axis* (indexing from 0) to describe the degree of freedom of array. [See the illustrations here. (https://www.cs.ubc.ca/~pcarter/cs189/cs189_ch7s3.html)](https://www.cs.ubc.ca/~pcarter/cs189/cs189_ch7s3.html)

In [ ]:
```
a = np.arange(24).reshape(2,3,4) # 3-d array, or tensor
```

In [ ]:
```
a
```

In [ ]:
```
help(np.arange) # note the difference with
```

In [ ]:
```
a.T
```

In [ ]:
```
a_1d = np.array([1,2,3,4])
a_1d.shape
```

In [ ]:
```
a_1d.T.shape
```

In [ ]:
```
a_2d = a_1d[:,np.newaxis]
a_2d.shape
```

In [ ]:
```
a_2d
```

In [ ]:
```
print(a_1d.ndim)
print(a_2d.ndim)
```

Indexing of ndarray

1. Slicing: Similiar to the list indexing

Always remember that slicing creates the view instead of copy!

In [ ]:
```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3] # create the view instead of copy
print(a[0, 1])
b[0, 0] = 77
print(a[0, 1])
```

Be cautions with the difference between simple indexing (one integer index) and slicing.

In [ ]:
```
a[:,0] # 1-d array
```

In [ ]:
```
a[:,0:1] # 2-d array
```

For more exercise: See Figure 4-2 in [this material (https://www.oreilly.com/library/view/python-for-data/9781449323592/ch04.html)](https://www.oreilly.com/library/view/python-for-data/9781449323592/ch04.html).

2. Boolean Indexing

In [ ]:
```
a[a<5] = 0
```

In [ ]:
```
a
```

Boolean indexing can create new numpay ndarray instead of the view.

In [ ]:
```
x = np.arange(10)
y = x[(x>4) & (x<8)] # just for your information: do not use keyword "and" here
```

In [ ]:
```
y.flags
```

3. Integer Array Indexing (Fancy Indexing)

General rule: `arr[[ind1,ind2]]` just means `np.array([arr[ind1],arr[ind2]])`

In [ ]:
```
ind = np.array([1,0,2]) # no problem for list [1,0,2]
x = np.arange(10)
x[ind] # equivalently, x[[1,0,2]]
```

In [ ]:
```
a = np.arange(12).reshape(3,4)
a
```

In [ ]:
```
a[[1,0,2],:]
```

In [ ]:
```
a[2,[1,0,2]]
```