Section 6 Introduction to Numpy

NumPy -- *Numerical Python* 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.

Unfortunately, the native numpy does not support GPU operations. For arrays on GPU, we have some popular substitutions, such as tensors in TensorFlow and jax (by Google), PyTorch (by Facebook) or arrays in CuPy (by Nvidia) -- while they all have close relations/ similar interface with Numpy. Therefore, learning the basic concepts about Numpy is crucial for doing data science with Python.

In [6]:

```
import numpy as np

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

In [2]:

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

```
CPU times: user 18.4 ms, sys: 10.1 ms, total: 28.6 ms
Wall time: 29.4 ms
```

In [3]:

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

```
CPU times: user 968 ms, sys: 283 ms, total: 1.25 s
Wall time: 1.34 s
```

Difference between ndarray and list : Data Memory Perspective

Intuitively speaking, 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 [7]:

```
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 [8]:

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

In [9]:

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

```
[1 2]
```

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

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

In [10]:
```
b.flags
```

Out[10]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [11]:
```
a.flags
```

Out[11]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

This mechanism may cause unexpected outcomes for beginners.

In [12]:
```
b[0] = 1000 # change the first element of b (which is the slice of a -- view)
a
```

Out[12]:
```
array([1000,    2,    3,    4])
```

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

In [13]:
```
c = l[0: 2] #slicing in list
c[0] = 100
l
```

Out[13]:
```
[1, 2, 3, 4]
```

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 [14]:
```
a_mat = a.reshape(2,2)
```

In [15]:
```
a_mat.base
```

Out[15]:
```
array([1000,    2,    3,    4])
```

In [16]:
```
a_mat[0,0] = 2000 # same as a_mat[0][0]
a
```

Out[16]:
```
array([2000,    2,    3,    4])
```

Transpose also creates the **view**.

In [17]:
```
a_t = a_mat.T # attribute
a_tt = a_mat.transpose() # method
```

In [18]:
```
a_t.base
```

Out[18]:
```
array([2000,    2,    3,    4])
```

In [19]:
```
a_t[0,0] = 0 # change the view -- change the data buffer -- the base a is also changed!
a
```

Out[19]:
```
array([0, 2, 3, 4])
```

Conversely, once the "base" is changed, **all** the associated "view" objects are changed!

In [20]:
```
a_mat # reshape of a -- view, changed!
```

Out[20]:
```
array([[0, 2],
       [3, 4]])
```

In [21]:
```
b # slicing of a -- view, changed!
```

Out[21]:
```
array([0, 2])
```

Use the copy method to create the new data buffer

In [23]:
```
a_copy = a.copy()
print(a_copy.base)
```

```
None
```

In [24]:
```
a_copy.flags
```

Out[24]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [25]:
```
a_mat_copy = a_mat.copy()
```

In [26]:
```
a_mat_copy.flags
```

Out[26]:
```
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
```

Numpy ndarray as object

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

In [27]:
```
type(a)
```

Out[27]:
```
numpy.ndarray
```

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

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

In [30]:
```
a = np.arange(4)
a.shape # 1-d array with length 4 -- different with 4x1 2-d array!
```

Out[30]:
```
(4,)
```

In [33]:
```
b = a.reshape(-1,1)
b.shape
```

Out[33]:
```
(4, 1)
```

In [34]:
```
a_mat.shape
```

Out[34]:
```
(2, 2)
```

In [35]:
```
a_mat.tolist()
```

Out[35]:
```
[[0, 2], [3, 4]]
```

In [36]:
```
a.mean()
```

Out[36]:
```
1.5
```

In [37]:
```
help(a.mean)
```

```
Help on built-in function mean:

mean(...) method of numpy.ndarray instance
```

```
    a.mean(axis=None, dtype=None, out=None, keepdims=False)

    Returns the average of the array elements along given axis.

    Refer to `numpy.mean` for full documentation.

    See Also
    --------
    numpy.mean : equivalent function
```

In [38]:

```
np.mean(a)
```

Out[38]:

```
1.5
```

In [39]:

```
help(a.reshape)
```

```
Help on built-in function reshape:

reshape(...) method of numpy.ndarray instance
    a.reshape(shape, order='C')

    Returns an array containing the same data with a new shape.

    Refer to `numpy.reshape` for full documentation.

    See Also
    --------
    numpy.reshape : equivalent function

    Notes
    -----
    Unlike the free function `numpy.reshape`, this method on `ndarray` allows
    the elements of the shape parameter to be passed in as separate arguments.
    For example, ``a.reshape(10, 11)`` is equivalent to
    ``a.reshape((10, 11))``.
```

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.

In [40]:

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

Out[40]:

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

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

In the method `reshape`, you can also pass value -1 to let Numpy calculate the number for you.

In [41]:

```
np.arange(24).reshape(2,-1,4)
```

Out[41]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

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

In [43]:
```
print(a.T)
a.T.shape
```

```
[[[ 0 12]
  [ 4 16]
  [ 8 20]]

 [[ 1 13]
  [ 5 17]
  [ 9 21]]

 [[ 2 14]
  [ 6 18]
  [10 22]]

 [[ 3 15]
  [ 7 19]
  [11 23]]]
```

Out[43]:
```
(4, 3, 2)
```

In [44]:
```
a_1d = np.array([1,2,3,4])
a_1d.shape
```

Out[44]:
```
(4,)
```

In [45]:
```
a_1d.T.shape # transpose is still 1-D array! this is very different with Matlab!
```

Out[45]:
```
(4,)
```

In [46]:
```
a_2d = a_1d[:,np.newaxis] # increase dimension
a_2d.shape
```

Out[46]:
```
(4, 1)
```

In [47]:
```
a_2d
```

Out[47]:
```
array([[1],
       [2],
       [3],
       [4]])
```

In [48]:
```
a_1d
```

Out[48]: array([1, 2, 3, 4])

In [49]:
```
print(a_1d.ndim)
print(a_2d.ndim)
```

```
1
2
```

To change the multi-dimension array to 1-d array, in addition to `reshape` (create view), we can also choose `ravel` (create view) or `flatten` (create copy).

In [50]:
```
a_mat = np.zeros((2,2)) # note the parentheses here
a_mat_reshape = a_mat.reshape(-1) # -1 means default length -- create view
a_mat_ravel =  a_mat.ravel()
a_mat_flatten = a_mat.flatten()
```

In [51]:
```
a_mat_reshape
```

Out[51]: array([0., 0., 0., 0.])

In [52]:
```
a_mat_ravel.base
```

Out[52]:
```
array([[0., 0.],
       [0., 0.]])
```

In [53]:
```
a_mat_flatten.flags
```

Out[53]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

Indexing of ndarray

1. Slicing: Similiar to the list indexing

Always remember that slicing creates the view instead of copy!

In [55]:
```
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])
```

```
2
77
```

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

In [56]:
```
a[:,0] # 1-d array
```

Out[56]: array([1, 5, 9])

In [57]:
```
a[:,0:1] # 2-d array
```

Out[57]:
```
array([[1],
       [5],
       [9]])
```

In [5]:
```
a[0:1,:] # 2-d array
```

Out[5]: array([[1, 77, 3, 4]])

For more exercise: See Figure 4-2 in this material.

2. Boolean Indexing

In [58]:
```
a[a<5] = 0 # In Numpy terms, a<5 creates the "mask" contaning true or false values
```

In [59]:
```
a
```

Out[59]:
```
array([[ 0, 77,  0,  0],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [60]:
```
b = a[a>2]
b
```

Out[60]: array([77, 5, 6, 7, 8, 9, 10, 11, 12])

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

In [61]:
```
x = np.arange(10)
y = x[(x>4) & (x<8)] # just for your information: do not use keyword "and" here
```

In [62]:
```
y.flags
```

Out[62]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

3. Integer Array Indexing (Fancy Indexing)

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

In [63]:
```
ind = np.array([1,0,2]) # no problem for list [1,0,2]
x = np.arange(10)
x[ind] # equivalently, x[[1,0,2]]
```

Out[63]: array([1, 0, 2])

In [64]:
```
a = np.arange(12).reshape(3,4)
a
```

Out[64]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [13]:
```
a[[1,0,2],:]
```

Out[13]:
```
array([[ 4,  5,  6,  7],
       [ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
```

In [14]:
```
a[2,[1,0,2]]
```

Out[14]:
```
array([ 9,  8, 10])
```

Numpy Universal Functions (ufuncs) and Aggregate Function

Similar to Matlab, the built-in loops in Python can be very slow for large-scale problems. To solve this issue, Numpy adopts vectorized methods (uses vectorization) written in optimized C-language codes, and provides the interface as Numpy universal functions (ufuncs).

Numpy ufuncs operates on ndarrays in an element-by-element fashion. You can find all the ufuncs in the documentation.

In [66]:
```
x = np.arange(1000000)
np.log(1+x)
```

Out[66]:
```
array([ 0.        ,  0.69314718,  1.09861229, ..., 13.81550856,
       13.81550956, 13.81551056])
```

We can also iterate the numpy array through elements just as Python built-in list (of course you can always get elements through iterating the index), although it is not very recommended for large-scale problems.

In [16]:
```
a = np.arange(6)
for elem in a:
    print(elem, end =" " )
```

```
0 1 2 3 4 5
```

In [17]:
```
a = a.reshape(2,-1)
for row in a:
    print(row, end =" " )
```

```
[0 1 2] [3 4 5]
```

In [18]:
```
for row in a:
    for elem in row:
        print(elem, end =" " )
```

```
0 1 2 3 4 5
```

In [19]:
```
for elem in np.nditer(a):
    print(elem, end =" " )
```

```
0 1 2 3 4 5
```

In [20]:
```
for (idx, elem) in np.ndenumerate(a):
    print([idx, elem])
```

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

Numpy also provides some useful aggregate functions.

In [67]:
```
a = np.arange(6).reshape(2,3)
a
```

Out[67]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [22]:
```
a.sum(axis=0)
```

Out[22]:
```
array([3, 5, 7])
```

In [23]:
```
a.sum(axis=1)
```

Out[23]:
```
array([ 3, 12])
```

In [68]:
```
a.sum()
```

Out[68]:
```
15
```

In [69]:
```
a.min(axis=1)
```

Out[69]:
```
array([0, 3])
```

In [70]:
```
b = np.arange(24).reshape(2,3,-1)
b
```

Out[70]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
```

```
[[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]]])
```

In [26]:

```python
b.sum(axis=1)
```

Out[26]:

```
array([[12, 15, 18, 21],
       [48, 51, 54, 57]])
```

In [71]:

```python
b.max(axis=0)
```

Out[71]:

```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

Numpy Linear Algebra Functions

See the reference here and compare it with Matlab. Be cautious with operators like `*` , `@` (only available after Python 3.5) and functions/methods `dot` , `vdot` and `matmul` .

In [ ]:

```python
help(np.dot)
```

In [ ]:

```python
help(np.vdot)
```

In [ ]:

```python
help(np.matmul)
```