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 [1]:
```
import numpy as np

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

In [11]:
```
print(type(my_arr))
```

```
<class 'numpy.ndarray'>
```

In [9]:
```
%time for _ in range(10): my_arr2 = my_arr * 2
```

```
Wall time: 18 ms
```

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

```
Wall time: 1.01 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 [12]:
```
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 [13]:
```
b = a[0:2] # creating view by slicing
```

In [14]:
```python
print(b)
b.base # view has the base object because its memory is from some other object.
```

```
[1 2]
```

Out[14]:
```
array([1, 2, 3, 4])
```

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

In [15]:
```python
b.flags
```

Out[15]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [16]:
```python
a.flags
```

Out[16]:
```
  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 [17]:
```python
b[0] = 1000 # change the first element of b (which is the slice of a -- view)
a
```

Out[17]:
```
array([1000,    2,    3,    4])
```

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

In [18]:
```python
c = l[0: 2] #slicing in list, c is a copy of l
c[0] = 100
l
```

Out[18]:
```
[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 [19]:
```python
a_mat = a.reshape(2,2)
print(a_mat)
```

```
[[1000    2]
 [   3    4]]
```

In [20]:
```
print(a_mat[1,0]) #vertical index (top to bottom)
print(a_mat[0,1]) #horizontal index (left to right)
```

```
3
2
```

In [21]:
```
a_mat.base
```

Out[21]:
```
array([1000,    2,    3,    4])
```

In [22]:
```
a_mat[0,0] = 2000 # same as a_mat[0][0]
a
```

Out[22]:
```
array([2000,    2,    3,    4])
```

Transpose also creates the **view**.

In [23]:
```
a_t = a_mat.T # attribute
a_tt = a_mat.transpose() # method
print(a_t) #swapped row and column
print(a_tt)
```

```
[[2000    3]
 [   2    4]]
[[2000    3]
 [   2    4]]
```

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

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

In [24]:
```
a_t[0,0] = 0 # change the view -- change the data buffer -- the base a is also changed!
a
```

Out[24]:
```
array([0, 2, 3, 4])
```

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

In [25]:
```
a_mat # reshape of a -- view, changed!
```

Out[25]:
```
array([[0, 2],
       [3, 4]])
```

In [26]:
```
b # slicing of a -- view, changed!
```

Out[26]:
```
array([0, 2])
```

Use the copy method to create the new data buffer

In [28]:
```
a_copy = a.copy()
print(a_copy)
print(a_copy.base)
```

```
[0 2 3 4]
None
```

In [29]:
```
a_copy.flags
```

Out[29]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [30]:
```
a_mat_copy = a_mat.copy()
```

In [31]:
```
a_mat_copy.flags
```

Out[31]:
```
  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 [32]:
```
type(a)
```

Out[32]:
```
numpy.ndarray
```

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

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

In [35]:
```
a = np.arange(4)
print(a.shape) # 1-d array with length 4 -- different with 4x1 2-d array!
print(a)
```

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

In [42]:
```
b = a.reshape(-1,1) #similar to how -1 is used as an index for the end of a list.
print(b.shape)
print(b)
```

```
(4, 1)
[[0]
 [1]
```

```
 [2]
 [3]]
```

In [43]:

```python
a_mat.shape
```

Out[43]:

```
(2, 2)
```

In [47]:

```python
L = a_mat.tolist() #each row of matrix is an element of the new list
print(L)
print(L[1])
print(L[1][0])
```

```
[[0, 2], [3, 4]]
[3, 4]
3
```

In [48]:

```python
a.mean()
```

Out[48]:

```
1.5
```

In [49]:

```python
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, *, where=True)

    Returns the average of the array elements along given axis.

    Refer to `numpy.mean` for full documentation.

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

In [51]:

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

Out[51]:

```
1.5
```

In [39]:

```python
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 uses the terms *dimension* and *axis* (indexing from 0) to describe the degree of freedom of an array. See the illustrations here.

In [43]:
```
a = np.arange(24).reshape(2,3,4) # 3-d array, or tensor
a
```

Out[43]:
```
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 [47]:
```
print(a[1,0,0]) #first index is depth (from front to back)
print(a[0,1,0]) #second index is vertical (from top to bottom)
print(a[0,0,1]) #third index is horizontal (from left to right)
```

```
12
4
1
```

In [42]:
```
help(np.arange) # note the difference with range()
```

```
Help on built-in function arange in module numpy:

arange(...)
    arange([start,] stop[, step,], dtype=None)

    Return evenly spaced values within a given interval.

    Values are generated within the half-open interval ``[start, stop)``
    (in other words, the interval including `start` but excluding `stop`).
    For integer arguments the function is equivalent to the Python built-in
    `range` function, but returns an ndarray rather than a list.

    When using a non-integer step, such as 0.1, the results will often not
```

```
    be consistent.  It is better to use `numpy.linspace` for these cases.

    Parameters
    ----------
    start : number, optional
        Start of interval.  The interval includes this value.  The default
        start value is 0.
    stop : number
        End of interval.  The interval does not include this value, except
        in some cases where `step` is not an integer and floating point
        round-off affects the length of `out`.
    step : number, optional
        Spacing between values.  For any output `out`, this is the distance
        between two adjacent values, ``out[i+1] - out[i]``.  The default
        step size is 1.  If `step` is specified as a position argument,
        `start` must also be given.
    dtype : dtype
        The type of the output array.  If `dtype` is not given, infer the data
        type from the other input arguments.

    Returns
    -------
    arange : ndarray
        Array of evenly spaced values.

        For floating point arguments, the length of the result is
        ``ceil((stop - start)/step)``.  Because of floating point overflow,
        this rule may result in the last element of `out` being greater
        than `stop`.

    See Also
    --------
    numpy.linspace : Evenly spaced numbers with careful handling of endpoints.
    numpy.ogrid: Arrays of evenly spaced numbers in N-dimensions.
    numpy.mgrid: Grid-shaped arrays of evenly spaced numbers in N-dimensions.

    Examples
    --------
    >>> np.arange(3)
    array([0, 1, 2])
    >>> np.arange(3.0)
    array([ 0.,  1.,  2.])
    >>> np.arange(3,7)
    array([3, 4, 5, 6])
    >>> np.arange(3,7,2)
    array([3, 5])
```

In [48]:

```
print(a.T) #Exercise: Describe what happens here
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[48]: (4, 3, 2)

In [64]:
```
a_1d = np.array([1,2,3,4])
a_1d.shape
```

Out[64]: (4,)

In [65]:
```
a_1d.T.shape # transpose is still 1-D array! this is very different with Matlab!
```

Out[65]: (4,)

In [66]:
```
a_2d = a_1d[:,np.newaxis] # increase dimension
a_2d.shape
```

Out[66]: (4, 1)

In [67]:
```
a_2d #recall the first index is vertical!
```

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

In [68]:
```
a_1d
```

Out[68]: 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 [69]:
```
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 [70]:
```
a_mat_ravel
```

Out[70]: array([0., 0., 0., 0.])

In [71]:
```
a_mat_flatten
```

Out[71]: 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 [74]:
```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3] # creates a 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 [75]:
```
a[:,0] # 1-d array
```

Out[75]: array([1, 5, 9])

In [76]:
```
a[:,0:1] # 2-d array
```

Out[76]:
```
array([[1],
       [5],
       [9]])
```

In [77]:
```
a[0:1,:] # 2-d array
```

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

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

2. Boolean Indexing

In [78]:
```
a[a<5] = 0 # In Numpy terms, a<5 creates the "mask" containg true or false values
```

In [79]:
```
a
```

Out[79]:
```
array([[ 0, 77,  0,  0],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [80]:
```
b = a[a>2]
b
```

Out[80]:
```
array([77,  5,  6,  7,  8,  9, 10, 11, 12])
```

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

In [81]:
```
x = np.arange(10)
y = x[(x>4) & (x<8)] # just for your information: do not use keyword "and" here
```

In [82]:
```
print(y)
```

```
[5 6 7]
```

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]:
```python
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]:
```python
a = np.arange(6).reshape(2,3)
a
```

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

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

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

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

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

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

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

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

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

In [70]:
```python
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 [72]:
```python
help(np.dot)
```

```
Help on function dot in module numpy:

dot(...)
    dot(a, b, out=None)

    Dot product of two arrays. Specifically,

    - If both `a` and `b` are 1-D arrays, it is inner product of vectors
      (without complex conjugation).

    - If both `a` and `b` are 2-D arrays, it is matrix multiplication,
      but using :func:`matmul` or ``a @ b`` is preferred.

    - If either `a` or `b` is 0-D (scalar), it is equivalent to :func:`multiply`
      and using ``numpy.multiply(a, b)`` or ``a * b`` is preferred.

    - If `a` is an N-D array and `b` is a 1-D array, it is a sum product over
      the last axis of `a` and `b`.

    - If `a` is an N-D array and `b` is an M-D array (where ``M>=2``), it is a
      sum product over the last axis of `a` and the second-to-last axis of `b`::

        dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

    Parameters
    ----------
    a : array_like
        First argument.
    b : array_like
        Second argument.
    out : ndarray, optional
        Output argument. This must have the exact kind that would be returned
        if it was not used. In particular, it must have the right type, must be
        C-contiguous, and its dtype must be the dtype that would be returned
        for `dot(a,b)`. This is a performance feature. Therefore, if these
        conditions are not met, an exception is raised, instead of attempting
        to be flexible.

    Returns
    -------
    output : ndarray
        Returns the dot product of `a` and `b`.  If `a` and `b` are both
        scalars or both 1-D arrays then a scalar is returned; otherwise
        an array is returned.
        If `out` is given, then it is returned.

    Raises
    ------
    ValueError
```

```
        If the last dimension of `a` is not the same size as
        the second-to-last dimension of `b`.

    See Also
    --------
    vdot : Complex-conjugating dot product.
    tensordot : Sum products over arbitrary axes.
    einsum : Einstein summation convention.
    matmul : '@' operator as method with out parameter.

    Examples
    --------
    >>> np.dot(3, 4)
    12

    Neither argument is complex-conjugated:

    >>> np.dot([2j, 3j], [2j, 3j])
    (-13+0j)

    For 2-D arrays it is the matrix product:

    >>> a = [[1, 0], [0, 1]]
    >>> b = [[4, 1], [2, 2]]
    >>> np.dot(a, b)
    array([[4, 1],
           [2, 2]])

    >>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
    >>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
    >>> np.dot(a, b)[2,3,2,1,2,2]
    499128
    >>> sum(a[2,3,2,:] * b[1,2,:,2])
    499128
```

In [73]:

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

```
Help on function vdot in module numpy:

vdot(...)
    vdot(a, b)

    Return the dot product of two vectors.

    The vdot(`a`, `b`) function handles complex numbers differently than
    dot(`a`, `b`).  If the first argument is complex the complex conjugate
    of the first argument is used for the calculation of the dot product.

    Note that `vdot` handles multidimensional arrays differently than `dot`:
    it does *not* perform a matrix product, but flattens input arguments
    to 1-D vectors first. Consequently, it should only be used for vectors.

    Parameters
    ----------
    a : array_like
        If `a` is complex the complex conjugate is taken before calculation
        of the dot product.
    b : array_like
        Second argument to the dot product.

    Returns
    -------
    output : ndarray
```

```
    Dot product of `a` and `b`.  Can be an int, float, or
    complex depending on the types of `a` and `b`.

See Also
--------
dot : Return the dot product without using the complex conjugate of the
      first argument.

Examples
--------
>>> a = np.array([1+2j,3+4j])
>>> b = np.array([5+6j,7+8j])
>>> np.vdot(a, b)
(70-8j)
>>> np.vdot(b, a)
(70+8j)

Note that higher-dimensional arrays are flattened!

>>> a = np.array([[1, 4], [5, 6]])
>>> b = np.array([[4, 1], [2, 2]])
>>> np.vdot(a, b)
30
>>> np.vdot(b, a)
30
>>> 1*4 + 4*1 + 5*2 + 6*2
30
```

In [ ]:

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