wbs
WARWICK BUSINESS SCHOOL
THE UNIVERSITY OF WARWICK

For the Change Makers

Programming for Data Analytics

Week 5: Data Processing
Information Systems and Management
Warwick Business School

Advantages of numpy array

- python list/tuple is dynamically typed, it can be mixed with different types of data and the type is not pre-defined.
- numpy array is **statically typed** and **pre-defined**. It can leverage complied language like C to improve the computing efficiency.
- numpy array supports complex mathematical calculations, such as matrix and dot multiplication, which can improve efficiency significantly.

Computation comparison

Problem: multiply each element in a list with the corresponding element in another list of the same length.

Python loop statements

```
c =[]
for i in range(len(a)):
    c.append(a[i]*b[i])
```

Numpy operation

```
c = a * b
```

Computation comparison

```
rng = np.random.RandomState(42)
x = rng.rand(100000) #(100000 ,1)
y = rng.rand(100000)
%timeit x + y
```

2.43 ms ± 52.3 µs per loop

```
%timeit results = [xi + yi for
xi, yi in zip(x, y)]
result = []
for xi, yi in zip(x,y):
    results.append(xi+yi)
```

181 ms ± 2.43 ms per loop

Shape of NumPy arrary

- The **shape** property is used to get the current shape of an array. It returns **tuple** of array dimensions.

>>>larray.shape

(3,) #larray has only one dimension and three elements (also called rank) in the first dimension.

>>>marray.shape

(3, 3) # marray has two dimensions and two elements in the first dimension, 3 elements in the second dimension. In other words, it has three one-dimensional elements, each with three elements.

Reshape your array

- Numpy array can be reshaped.

```
>>>c.reshape(3,2,4)
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8]],
       [[ 9, 10, 11, 12], [13, 14, 15, 16]],
       [[17, 18, 19, 20], [21, 22, 23, 24]]])
```

```
>>>c.reshape(4,3,2)
array([[[ 1,  2], [ 3,  4], [ 5,  6]],
       [[ 7,  8], [ 9, 10], [11, 12]],
       [[13, 14], [15, 16], [17, 18]],
       [[19, 20], [21, 22], [23, 24]]])
```

reshape does not change the original array. It only returns the result of reshaped array.

Special reshape

- You can easily *transpose* a 2-D array with T or transpose().

```
c = np.arange(1,25).shape(4,6)
c = c.T
c = c.transpose()
```

- You can also *flatten* multi-dimensional array into one dimension array with ravel().

```
c = c.ravel()
```

- You can directly *change the shape* of array by resize().

```
c.resize(2,3,4)  #the shape of c is changed.
```

Array stacking

- Multiple arrays can be stacked, vertically and horizontally.

```
a1 = np.array([[1,2],[3,4]])
a2 = np.array([[5,6],[7,8])
a3 = np.vstack((a1,a2))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
a4 = np.hstack((a1,a2))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

More attributes of numpy array

.ndim returns the total number of dimensions of the array.

```
>>>c.ndim
3
```

.size returns the total number of elements in the array.

```
>>>c.size
24
```

Index your array

- Similar to Python nested list, you can index and slice your array.
- For multi-dimensional array, use comma to separate index for each dimension.

```
>>>a = np.arange(1,25).reshape(2,3,4)
array([[[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]],
       [[13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]]])
>>>a[1,2,3] #similar as a[1][2][3] for nested List.
24
```

- If index has been specified for all dimensions, a data value will be returned, otherwise a NumPy array will be returned.

```
>>>a[1,2]
array([21, 22, 23, 24])
>>>a[1]
array([[13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
```

Slice the array

- Slice 1-D array with [start:end:step]

```
>>>a = np.array([1,2,3,4,5,6,7,8])
>>>a[1:5]
array([2, 3, 4, 5])
>>>a[1:6:2]
array([2, 4, 6])
```

Slice multi-dimensional array

- Use comma separating slicing on each dimension. When fewer indices are provided than the number of dimensions, the missing indices are considered **complete** slices

```
>>>a[0:2,0:2,0:2] # three dimensions
array([[[ 1,  2], [ 5,  6]],
       [[13, 14], [17, 18]]])
>>>a[0:2,0:2] # first two dimensions
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8]],
       [[13, 14, 15, 16], [17, 18, 19, 20]]])
>>>a[0:2] # first dimension
array([[[ 1,  2,  3,  4], [ 5,  6,  7,  8], [ 9, 10, 11, 12]],
       [[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]]])
```

Index and slice your array

```
>>>a[1,1:3,3]
array([20, 24])
```

With high dimensional array, it may get harder to index. We can use three dots … to indicate complete slices. For example, x is a 5-D array.

- x[1,2,...] is equivalent to x[1,2,:,:,:],
- x[...,3] to x[:,:,:,:,3] and
- x[4,...,5,:] to x[4,:,:,5,:].

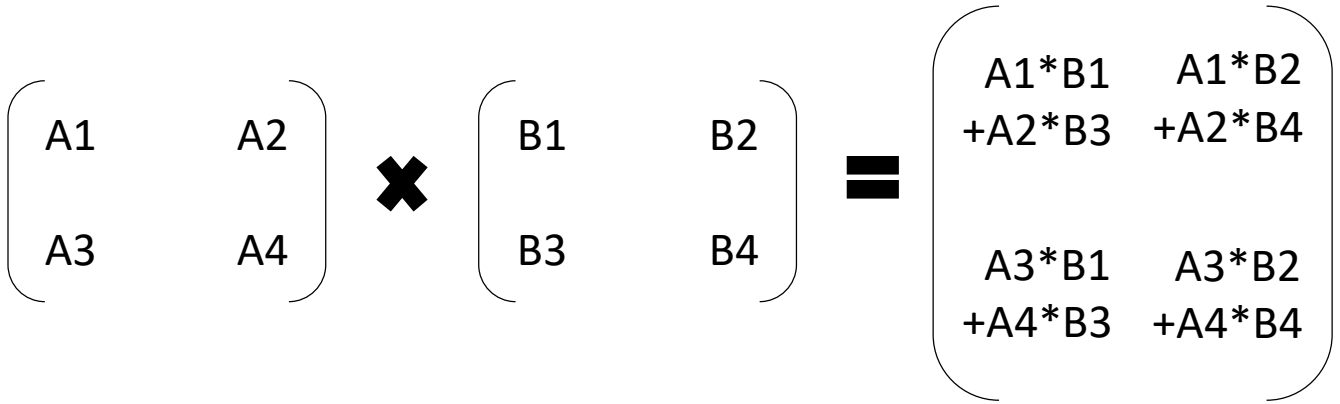
Basic operations

- Arithmetic operators on arrays apply elementwise.

```
>>>a = np.array([1,2,3])
>>>b = np.array([4,5,6])
>>>a + b # array([5, 7, 9])
>>>a – b # array([-3, -3, -3])
>>>a * b # array([ 4, 10, 18])
>>>a / b # array([0.25, 0.4 , 0.5 ])
>>>a ** b # array([  1,  32, 729], dtype=int32)
>>>a < 2 # array([ True, False, False])
```

* is not for matrix product in NumPy.

Matrix product



Matrix product can be performed with @ or dot().

```
>>>a = np.array([[1,2],[3,4]])
>>>b = np.array ([[5,6],[7,8]])
>>>a @ b  # array([[19, 22], [43, 50]])
>>>a.dot(b) # array([[19, 22], [43, 50]])
```

Matrix product

- Make sure you have same number of elements in matching row and column.

```
>>>a = np.array([[1,2,3],[3,4,5]])
>>>b = np.array ([[5,6],[7,8]])
>>>a @ b # ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

- The number of columns (2nd dimension) in the first matrix should equal to the number of rows (1st dimension) in the second matrix

```
>>>c = np.array ([[5,6],[7,8],[9,10]]) #shape(3,2)
>>>a @ b # array([[46,52], [88,100]])
```

For more information

- https://docs.scipy.org/doc/numpy/user/quickstart.html