STA141C: Big Data & High Performance Statistical Computing

Lecture 1: Python programming (2)

Cho-Jui Hsieh
UC Davis

April 6, 2017

Numpy

Numpy

Numpy provides:

- The definition of multi-dimensional arrays in python
- Efficient operations for manipulating the arrays/matrix
- These things are widely used for statistical computing

Defining an np.array

- Import numpy:

```
>>> import numpy as np
```

- "np.array" is a basic numpy multi-dimensional array
- Shape: a tuple of integers giving the size of each dimension

```
>>> a = np.array([1, 2, 3])
>>> type(a)
<type 'numpy.ndarray'>
>>> a.ndim
1
>>> a.shape
(3,)
>>> a = np.array([[1,2], [3,4]])
>>> a.ndim
2
>>> a.shape
(2, 2)
```

Creating Arrays

- Using np.arange:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- Commonly used arrays:

```
>>> a = np.ones((2, 2))
>>> a
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> c = np.eye(2)
>>> c
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Creating Arrays

```
>>> d = np.diag(np.array([1, 2]))
>>> d
array([[1, 0],
       [0, 2]])
```

- Creating a random array:

```
>>> a = np.random.rand(2,2)
>>> a
array([[ 0.86291659,  0.55238008],
       [ 0.69545018,  0.74234538]])
```

Array types

- Check the type of your array:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a.dtype
dtype('int64')
>>> b = np.random.random((2, 2))
>>> b.dtype
dtype('float64')
>>> a = a+b
>>> a.dtype
dtype('float64')
```

- Types:
  - int, bool, float (default), complex, str, $\cdots$

Convert types

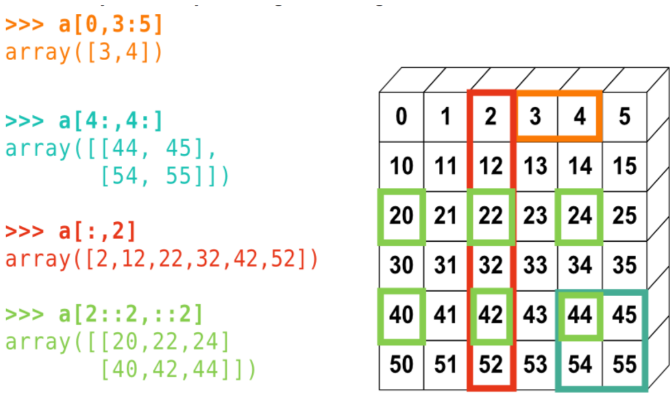
```
>>> a = np.random.rand(2,2) * 5
>>> a
array([[ 0.13924165,  2.19114316],
       [ 1.87070166,  3.49563945]])
>>> b = a.astype(int)
>>> b
array([[0, 2],
       [1, 3]])
>>> b.dtype
dtype('int64')
```

Indexing

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
1
>>> a[1][1]
1
>>> a[2, 1] = 10
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
array([0, 1, 0])
```

Slicing

Access a subset of the array



(Figure from `http://www.scipy-lectures.org/intro/numpy/array_object.html#indexing-and-slicing`)

Slicing

Important:

- Slicing will just create a “view” of the array, so the array is not copied in memory.

```
>>> a = np.diag([1, 2, 3])
>>> a
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
>>> b = a[1,:]
>>> b
array([0, 2, 0])
>>> b[0]=1
>>> b
array([1, 2, 0])
>>> a
array([[1, 0, 0],
       [1, 2, 0],
       [0, 0, 3]])
>>> b = b+1
```

Slicing

```
>>> b
array([2, 3, 1])
>>> a
array([[1, 0, 0],
       [1, 2, 0],
       [0, 0, 3]])
>>> np.may_share_memory(a,b)
False
```

More on indexing

- Using boolean masks

```
>>> a = np.array([1,2], [3,4], [5,6]])
>>> bool_idx = (a > 2)
>>> bool_idx
array([[False, False],
       [ True,  True],
       [ True,  True]], dtype=bool)
>>> b = a[bool_idx]
>>> b
array([3, 4, 5, 6])
>>> np.may_share_memory(a,b)
False
```

- Using integer array indexing:

```
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> a = np.array([1,2], [3, 4], [5, 6]])
>>> b = a[[0, 1, 2], [0, 1, 0]]
>>> b
array([1, 4, 5])
```

Elementwise Operators

- +, -, *, /, ** are elementwise operators.

```
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> b
array([[4, 5, 6],
       [1, 2, 3]])
>>> a+b
array([[5, 7, 9],
       [5, 7, 9]])
>>> a-b
array([[-3, -3, -3],
       [ 3,  3,  3]])
>>> a*b
array([[ 4, 10, 18],
       [ 4, 10, 18]])
>>> a/b
array([[0, 0, 0],
       [4, 2, 2]])
>>> a**b
array([[  1,  32, 729],
       [  4,  25, 216]])
```

Elementwise Operators

- Element-wise operator works if the size matches
- Sometimes it's also possible to do element-wise operators if Numpy can transform these arrays to the same size.

```
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a + 1
array([[2, 3, 4],
       [5, 6, 7]])
>>> a + np.array([1, 2, 3])
array([[2, 4, 6],
       [5, 7, 9]])
>>> a + np.array([[1], [2]])
array([[2, 3, 4],
       [6, 7, 8]])
```

Matrix Multiplication

- Using "dot" instead of "*"

```
>>> a
array([[1, 2],
       [3, 4]])
>>> b = a
>>> a.dot(b)
array([[ 7, 10],
       [15, 22]])
```

reduction

- Sum: sum over the rows/columns or the whole matrix.

```
>>> a
array([[1, 2],
       [3, 4]])
>>> a.sum()
10
>>> a.sum(axis=0)
array([4, 6])
>>> a.sum(axis=1)
array([3, 7])
```

- min/max: find min or max over rows/columns or the whole matrix.

```
>>> a.min()
1
>>> a.min(axis=0)
array([1, 2])
>>> a.min(axis=1)
array([1, 3])
```

Reshape

- ravel: flatering the array

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

- reshape: reshape the array (the dimensions have to match)

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

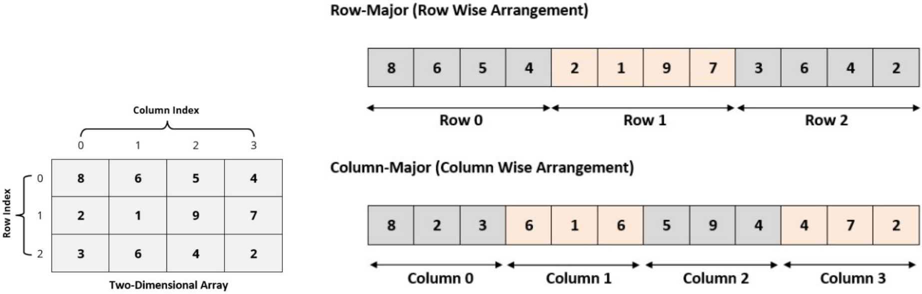
- transpose:

```
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Python uses row-major storage

- When storing the elements of a 2-D array in memory, these are allocated contiguous memory locations

  $\Rightarrow$ A 2-D array must be linearized to 1-D in torage
- Row major vs Column major
- Python uses row major



Try to use matrix operations

- Matrix operations in numpy are highly optimized (often using C/Fortran)
- Try to use matrix operations when possible
- Excesses: compare the element-wise product:

```
n = 1000
a = np.ones(n, n))
for i in range(n):
    for j in range(n):
        a[i][j] = a[i][j]*4.0
```

- Compare with

```
a = a*4.0
```

Matrix product

- Calling numpy matrix product is much faster than hand-written loops
- Numpy matrix product is using Blas library (will go back later)
- Three levels of blas:
  - Level 1: vector operations
  - Level 2: Matrix-vector product
  - Level 3: Matrix-matrix product (much faster than hand-written loops in C++)
- Exercise: compare the computation of matrix product with loops and with $a.dot(b)$

Coming up

- Basic algorithms and data structure.

Questions?