Python 6: NumPy

금융공학 프로그래밍 I

Intro to NumPy

❖ What is NumPy?

- Fast array processing
- Iteration via loops in interpreted languages (including Python) is relatively slow
- Operations are sent in batches to optimized C and Fortran code

❖ NumPy Arrays

```
In [1]: import numpy as np

In [2]: a = np.zeros(3)

In [3]: a
Out[3]: array([ 0.,  0.,  0.])

In [4]: type(a)
Out[4]: numpy.ndarray
```

NumPy dtype

- NumPy arrays vs. Python lists
  - Data must be homogeneous (all elements of the same type)
  - These types (dtypes) are provided by NumPy

- dtypes
  - float64 (default type)
  - float32
  - int64
  - int32
  - bool

```
In [7]: a = np.zeros(3)

In [8]: type(a[0])
Out[8]: numpy.float64
```

```
In [9]: a = np.zeros(3, dtype=int)

In [10]: type(a[0])
Out[10]: numpy.int32
```

Shape and Dimension

❖ "flat" array

```
In [11]: z = np.zeros(10)

In [12]: z.shape
Out[12]: (10,)  # Note syntax for tuple with one element
```

❖ Changing shape

```
In [13]: z.shape = (10, 1)

In [14]: z
Out[14]:
array([[ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.]])
```

```
In [15]: z = np.zeros(4)

In [16]: z.shape = (2, 2)

In [17]: z
Out[17]:
array([[ 0.,  0.],
       [ 0.,  0.]])
```

Creating Arrays

❖ Empty array

```
In [18]: z = np.empty(3)

In [19]: z
Out[19]: array([  8.90030222e-307,   4.94944794e+173,   4.04144187e-262])
```

❖ Grid of evenly spaced numbers

```
In [20]: z = np.linspace(2, 4, 5)  # From 2 to 4, with 5 elements
```

❖ Identity

```
In [21]: z = np.identity(2)

In [22]: z
Out[22]:
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Creating Array

❖ From Python lists, tuples, etc.

```
In [23]: z = np.array([10, 20])

In [24]: z
Out[24]: array([10, 20])

In [25]: type(z)
Out[25]: numpy.ndarray

In [26]: z = np.array((10, 20), dtype=float)

In [27]: z
Out[27]: array([ 10.,  20.])

In [28]: z = np.array([[1, 2], [3, 4]])

In [29]: z
Out[29]:
array([[1, 2],
       [3, 4]])
```

Indexing

```
In [30]: z = np.linspace(1, 2, 5)

In [31]: z
Out[31]: array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])

In [32]: z[0]
Out[32]: 1.0

In [33]: z[0:2]  # Slice numbering is left closed, right open
Out[33]: array([ 1.  ,  1.25])

In [34]: z[-1]
Out[34]: 2.0
```

2D

```
In [35]: z = np.array([[1, 2], [3, 4]])

In [36]: z
Out[36]:
array([[1, 2],
       [3, 4]])

In [37]: z[0, 0]
Out[37]: 1

In [38]: z[0, 1]
Out[38]: 2

In [39]: z[0,:]
Out[39]: array([1, 2])

In [40]: z[:,1]
Out[40]: array([2, 4])
```

```
In [42]: z
Out[42]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])

In [43]: indices = np.array((0, 2, 3))

In [44]: z[indices]
Out[44]: array([ 2. ,  3. ,  3.5])

In [46]: d = np.array([0, 1, 1, 0, 0], dtype=bool)

In [47]: d
Out[47]: array([False,  True,  True, False, False], dtype=bool)

In [48]: z[d]
Out[48]: array([ 2.5,  3. ])
```

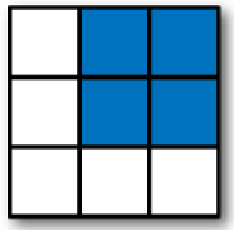
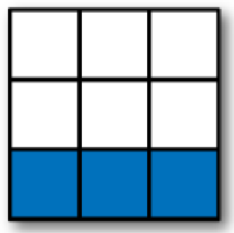
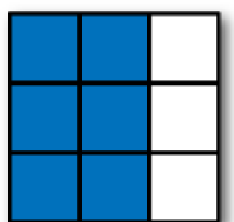
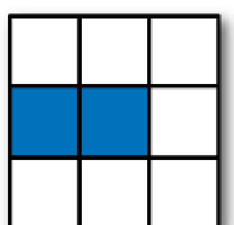
```
In [49]: z = np.empty(3)

In [50]: z
Out[50]: array([ -1.25236750e-041,   0.00000000e+000,   5.45693855e-313])

In [51]: z[:] = 42

In [52]: z
Out[52]: array([ 42.,  42.,  42.])
```

Two-dim slicing

	Expression	Shape
	arr[:2, 1:]	(2, 2)
	arr[2]	(3,)
	arr[2, :]	(3,)
	arr[2:, :]	(1, 3)
	arr[:, :2]	(3, 2)
	arr[1, :2]	(2,)
	arr[1:2, :2]	(1, 2)

Fancy indexing

```
In [102]: arr
Out[102]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

```
In [103]: arr[[4, 3, 0, 6]]
Out[103]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])

In [104]: arr[[-3, -5, -7]]
Out[104]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

```
In [106]: arr
Out[106]:
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],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

```
In [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])

In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[109]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Concatenate

```
In [32]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [33]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [34]: np.concatenate([arr1, arr2], axis=0)
Out[34]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [35]: np.concatenate([arr1, arr2], axis=1)
Out[35]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

```
In [36]: np.vstack((arr1, arr2))
Out[36]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
In [37]: np.hstack((arr1, arr2))
Out[37]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

Stacking helpers: r_ & c_

```
In [45]: arr = np.arange(6)

In [46]: arr1 = arr.reshape((3, 2))

In [47]: arr2 = randn(3, 2)

In [48]: np.r_[arr1, arr2]
Out[48]:
array([[ 0.    ,  1.    ],
       [ 2.    ,  3.    ],
       [ 4.    ,  5.    ],
       [ 0.7258, -1.5325],
       [-0.4696, -0.2127],
       [-0.1072,  1.2871]])
```

```
In [49]: np.c_[np.r_[arr1, arr2], arr]
Out[49]:
array([[ 0.    ,  1.    ,  0.    ],
       [ 2.    ,  3.    ,  1.    ],
       [ 4.    ,  5.    ,  2.    ],
       [ 0.7258, -1.5325,  3.    ],
       [-0.4696, -0.2127,  4.    ],
       [-0.1072,  1.2871,  5.    ]])
```

```
In [50]: np.c_[1:6, -10:-5]
Out[50]:
array([[  1, -10],
       [  2,  -9],
       [  3,  -8],
       [  4,  -7],
       [  5,  -6]])
```

Methods

Attribute		
	dtype	ndim
	shape	size
	T	
Method		
	all	any
	argmax	argmin
	max	min
	mean	
	std	var
	prod	sum
	cumprod	cumsum
	dot	transpose
	diagonal	trace
	sort	copy

```
In [53]: A = np.array((4, 3, 2, 1))

In [54]: A
Out[54]: array([4, 3, 2, 1])

In [55]: A.sort()                # Sorts A in place

In [56]: A
Out[56]: array([1, 2, 3, 4])

In [57]: A.sum()                 # Sum
Out[57]: 10

In [58]: A.mean()                # Mean
Out[58]: 2.5

In [64]: A.std()
Out[64]: 1.1180339887498949

In [65]: A.shape = (2, 2)

In [66]: A.T
Out[66]:
array([[1, 3],
       [2, 4]])
```

Algebraic Operation

```
In [75]: a = np.array([1, 2, 3, 4])

In [76]: b = np.array([5, 6, 7, 8])

In [77]: a + b
Out[77]: array([ 6,  8, 10, 12])

In [78]: a * b
Out[78]: array([ 5, 12, 21, 32])

In [79]: a + 10
Out[79]: array([11, 12, 13, 14])

In [82]: a * 10
Out[82]: array([10, 20, 30, 40])
```

```
In [86]: A = np.ones((2, 2))

In [87]: B = np.ones((2, 2))

In [88]: A + B
Out[88]:
array([[ 2.,  2.],
       [ 2.,  2.]])

In [89]: A + 10
Out[89]:
array([[ 11.,  11.],
       [ 11.,  11.]])

In [90]: A * B
Out[90]:
array([[ 1.,  1.],
       [ 1.,  1.]])
```

In particular, `A * B` is *not* the matrix product, it is an elementwise product

Matrix Multiplication

```
In [137]: A = np.ones((2, 2))

In [138]: B = np.ones((2, 2))

In [139]: np.dot(A, B)
Out[139]:
array([[ 2.,  2.],
       [ 2.,  2.]])
```

```
In [91]: A = np.array([1, 2])

In [92]: B = np.array([10, 20])

In [93]: np.dot(A, B)   # Returns a scalar in this case
Out[93]: 50
```

Comparisons

❖ Elementwise comparisons

```
In [97]: z = np.array([2, 3])

In [98]: y = np.array([2, 3])

In [99]: z == y
Out[99]: array([ True,  True], dtype=bool)

In [100]: y[0] = 5

In [101]: z == y
Out[101]: array([False,  True], dtype=bool)

In [102]: z != y
Out[102]: array([ True, False], dtype=bool)

In [103]: z = np.linspace(0, 10, 5)

In [104]: z
Out[104]: array([  0. ,   2.5,   5. ,   7.5,  10. ])

In [105]: z > 3
Out[105]: array([False, False,  True,  True,  True], dtype=bool)
```

```
In [109]: z[z > 3]
Out[109]: array([  5. ,   7.5,  10. ])
```

Vectorized Functions

```
for i in range(n):
    y[i] = np.sin(z[i])
```

```
In [110]: z = np.array([1, 2, 3])

In [111]: np.sin(z)
Out[111]: array([ 0.84147098,  0.90929743,  0.14112001])

In [113]: (1 / np.sqrt(2 * np.pi)) * np.exp(- 0.5 * z**2)
Out[113]: array([ 0.24197072,  0.05399097,  0.00443185])
```

Universal Functions (ufunc)

❖ Elementwise operations on data in ndarrays

- Unary ufuncs

```
In [120]: arr = np.arange(10)

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [122]: np.exp(arr)
Out[122]:
array([    1.    ,     2.7183,     7.3891,    20.0855,    54.5982,
         148.4132,   403.4288,  1096.6332,  2980.958 ,  8103.0839])
```

- Binary ufuncs

```
In [127]: np.maximum(x, y) # element-wise maximum
Out[127]:
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655,  0.9222,  0.446 ,
       -0.7147])
```

Broadcasting

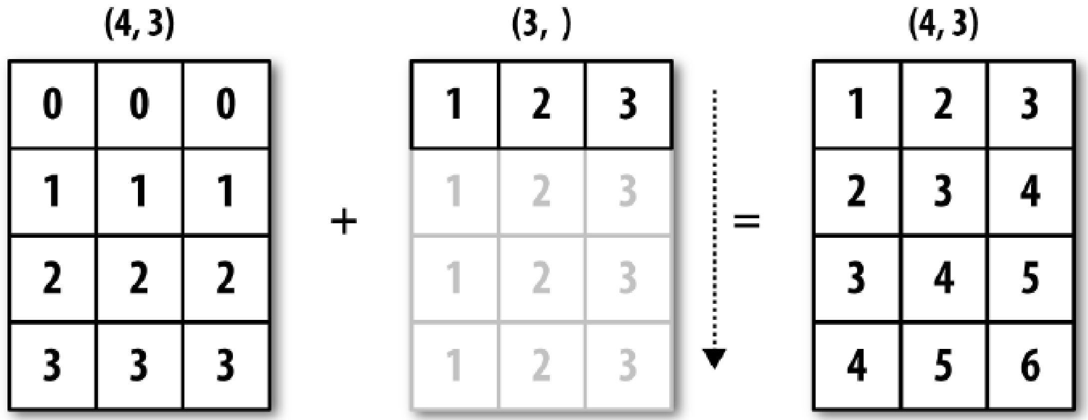
```
In [83]: arr = randn(4, 3)

In [84]: arr.mean(0)
Out[84]: array([ 0.1321,  0.552 ,  0.8571])

In [85]: demeaned = arr - arr.mean(0)

In [86]: demeaned
Out[86]:
array([[ 0.1718, -0.1972, -1.3669],
       [-0.1292,  1.6529, -0.3429],
       [-0.2891, -0.0435,  1.2322],
       [ 0.2465, -1.4122,  0.4776]])
```

```
In [87]: demeaned.mean(0)
Out[87]: array([ 0., -0., -0.])
```



Broadcasting

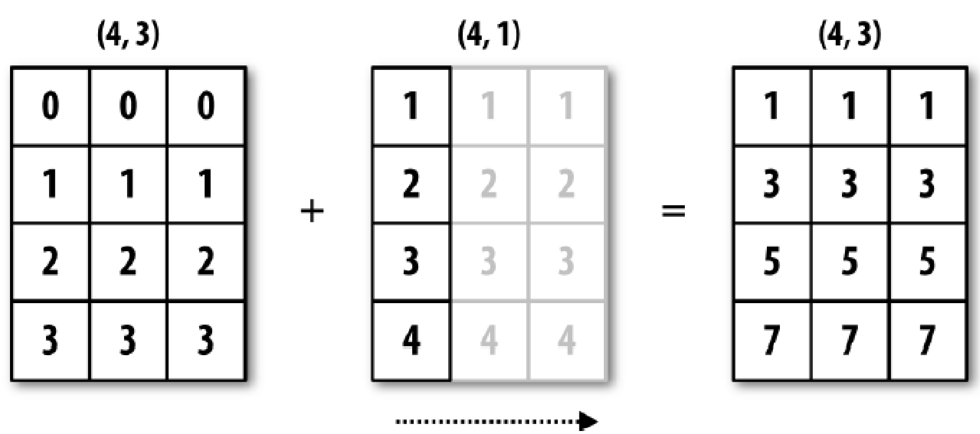
```
In [88]: arr
Out[88]:
array([[ 0.3039,  0.3548, -0.5097],
       [ 0.0029,  2.2049,  0.5142],
       [-0.1571,  0.5085,  2.0893],
       [ 0.3786, -0.8602,  1.3347]])
```

```
In [89]: row_means = arr.mean(1)
```

```
In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.0496],
       [ 0.9073],
       [ 0.8136],
       [ 0.2844]])
```

```
In [91]: demeaned = arr - row_means.reshape((4, 1))
```

```
In [92]: demeaned.mean(1)
Out[92]: array([ 0.,  0.,  0.,  0.])
```



np.where

```
In [114]: import numpy as np

In [115]: x = np.random.randn(4)

In [116]: x
Out[116]: array([-0.25521782,  0.38285891, -0.98037787, -0.083662  ])

In [117]: np.where(x > 0, 1, 0)  # Insert 1 if x > 0 true, otherwise 0
Out[117]: array([0, 1, 0, 0])
```

❖ Vectorized function

```
In [118]: def f(x): return 1 if x > 0 else 0

In [119]: f = np.vectorize(f)

In [120]: f(x)                  # Passing same vector x as previous example
Out[120]: array([0, 1, 0, 0])
```

Other NumPy Functions

```
In [131]: A = np.array([[1, 2], [3, 4]])

In [132]: np.linalg.det(A)            # Compute the determinant
Out[132]: -2.0000000000000004

In [133]: np.linalg.inv(A)            # Compute the inverse
Out[133]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

In [134]: Z = np.random.randn(10000)  # Generate standard normals

In [135]: y = np.random.binomial(10, 0.5, size=1000)
```

Q & A

khhwang78@gmail.com