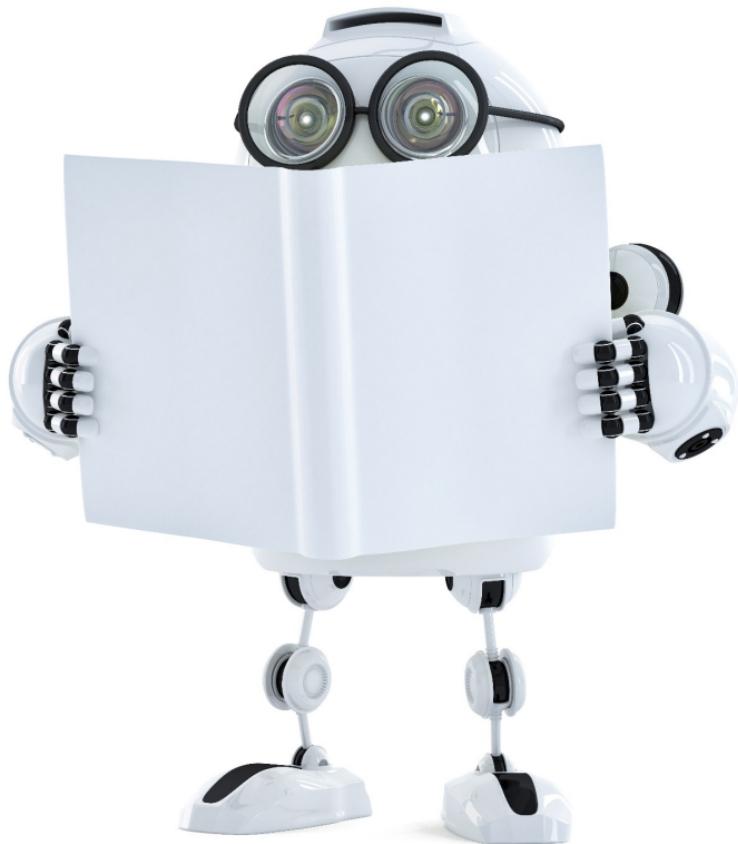


Numerical Python - numpy

Data handling section

Director of TEAMLAB
Sungchul Choi



어떻게 수식을
코드로 표현할 것인가?

코드로 방정식 표현하기

$$2x_1 + 2x_2 + x_3 = 9$$

$$2x_1 - x_2 + 2x_3 = 6$$

$$x_1 - x_2 + 2x_3 = 5$$

$$\begin{bmatrix} 2 & 2 & 1 & 9 \\ 2 & -1 & 2 & 6 \\ 1 & -1 & 2 & 5 \end{bmatrix}$$

```
coefficient_matrix = [[2, 2, 1], [2, -1, 2], [1, -1, 2]]  
constant_vector = [9,6,5]
```

코드로 방정식 표현하기

```
coefficient_matrix = [[2, 2, 1], [2, -1, 2], [1, -1, 2]]  
constant_vector = [9, 6, 5]
```

- 다양한 Matrix 계산을 어떻게 만들 것인가?
- 굉장히 큰 Matrix에 대한 표현
- 처리 속도 문제 – 파이썬은 Interpreter 언어

적절한 패키지를 활용하는 건 좋은 방법

파이썬 과학 처리 패키지

Numpy

Numpy

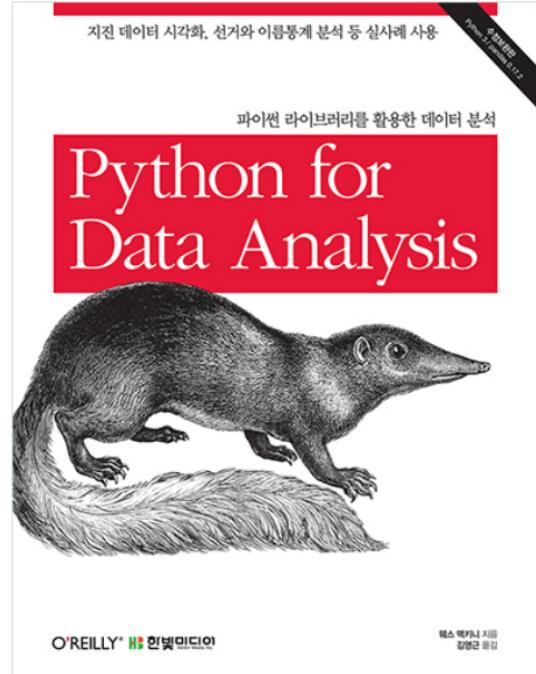
- Numerical Python
- 파이썬의 고성능 과학 계산용 패키지
- Matrix와 Vector와 같은 Array 연산의 사실상의 표준
- 한글로 넘파이로 주로 통칭, 넘피/눔파이라고 부르기도 함

Numpy 특징

- 일반 List에 비해 빠르고, 메모리 효율적
- 반복문 없이 데이터 배열에 대한 처리를 지원함
- 선형대수와 관련된 다양한 기능을 제공함
- C, C++, 포트란 등의 언어와 통합 가능

References

- cs231 - <http://cs231n.github.io/python-numpy-tutorial/#numpy>
- <https://docs.scipy.org/doc/numpy-dev/user/quickstart.html>
- 데이터 사이언스 스쿨 (파이썬 버전) - <https://goo.gl/3hsjbS>
- Numpy - <https://goo.gl/7Nwjw>
- 파이썬 라이브러리를 활용한 데이터 분석



Numpy Install

```
activate ml_scratch  
conda install numpy
```

- Windows 환경에선 conda로 패키지 관리 필요
(C 패키지 핸들링 등)
- jupyter 등을 설치한 상태에서는 추가 설치 필요 없음

ndarray

import

```
import numpy as np
```

- numpy의 호출 방법
- 일반적으로 numpy는 np라는 alias(별칭) 이용해서 호출함
- 특별한 이유는 없음 세계적인 약속 같은 것

Array creation

```
test_array = np.array([1, 4, 5, 8], float)  
print(test_array)  
type(test_array[3])
```

- numpy는 np.array 함수를 활용하여 배열을 생성함 → ndarray
- numpy는 하나의 데이터 type만 배열에 넣을 수 있음
- List와 가장 큰 차이점, Dynamic typing not supported
- C의 Array를 사용하여 배열을 생성함

Array creation

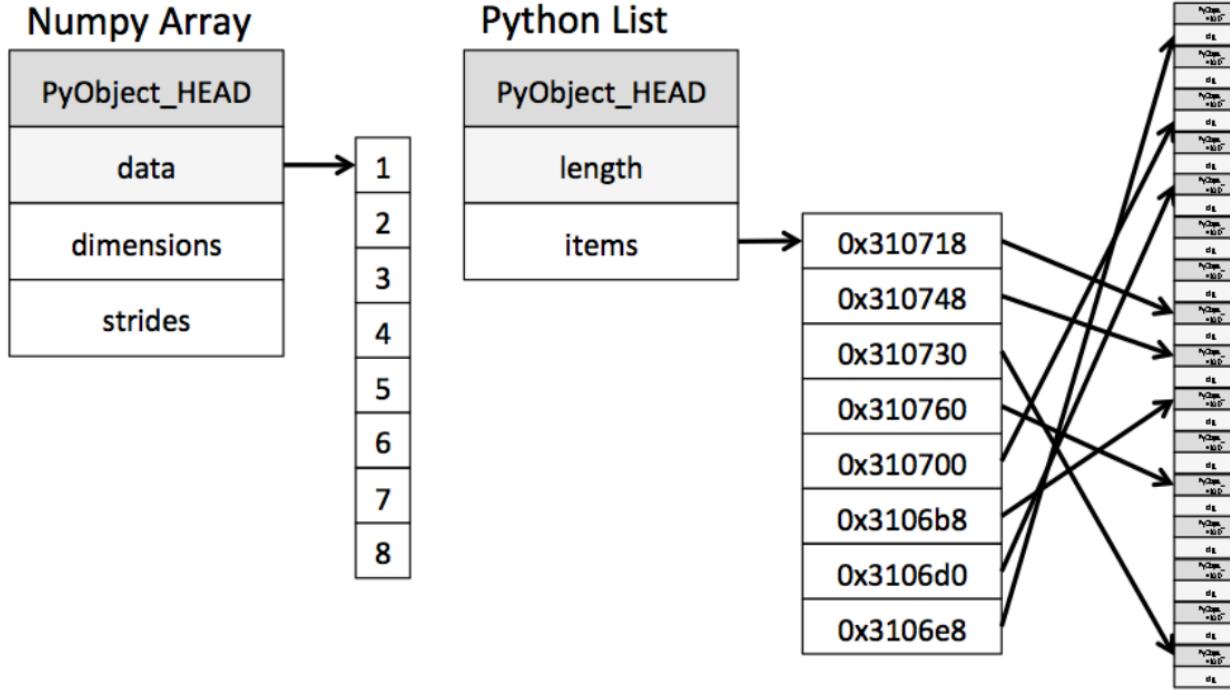
```
test_array = np.array([1, 4, 5, 8], float)  
test_array
```

```
array([ 1., 4., 5., 8.])
```

```
type(test_array[3])
```

```
numpy.float64
```

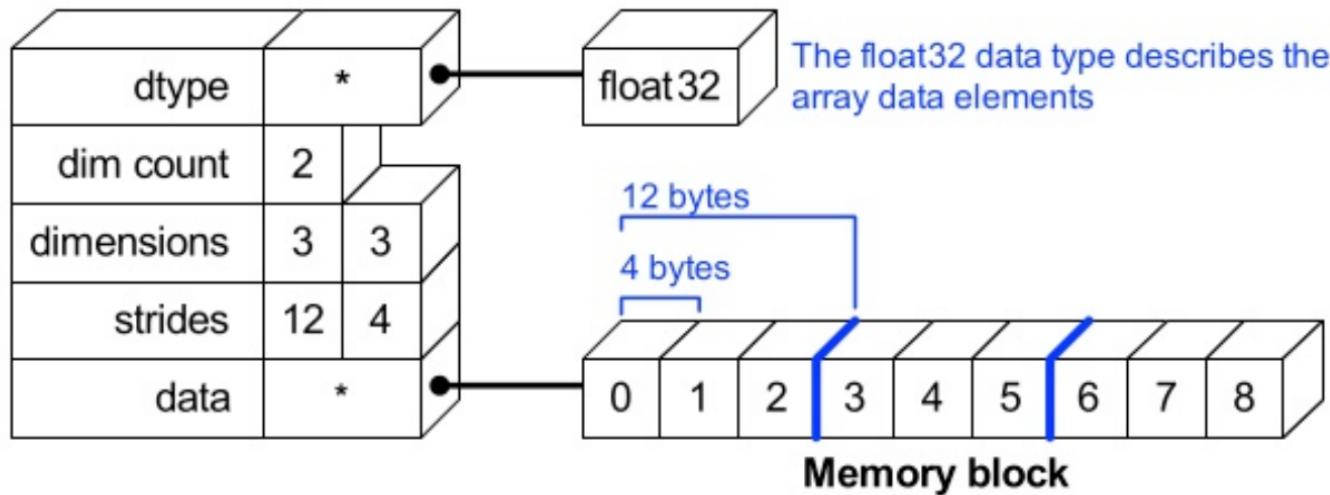
Array creation



<https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/>

Array creation

NDArray Data Structure



Array creation

```
test_array = np.array([1, 4, 5, "8"], float)      # String Type의 데이터를 입력해도  
print(test_array)  
print(type(test_array[3]))                      # Float Type으로 자동 형변환을 실시  
print(test_array.dtype)                         # Array(배열) 전체의 데이터 Type을 반환함  
print(test_array.shape)                        # Array(배열) 의 shape을 반환함
```

- **shape** : numpy array의 object의 dimension 구성을 반환함
- **dtype** : numpy array의 데이터 type을 반환함

Array creation

```
test_array = np.array([1, 4, 5, "8"], float) # String Type의 데이터를 입력해도  
test_array
```

```
array([ 1., 4., 5., 8.])
```

```
type(test_array[3]) # Float Type으로 자동 형변환을 실시
```

```
numpy.float64
```

```
test_array.dtype # Array(배열) 전체의 데이터 Type을 반환함
```

```
dtype('float64')
```

```
test_array.shape # Array(배열) 의 shape을 반환함
```

(4,) 4

Array shape (vector)

- Array (vector, matrix, tensor)의 크기, 형태 등에 대한 정보

```
test_array = np.array([1, 4, 5, "8"], float) # String Type의 데이터를 입력해도  
test_array  
  
array([ 1., 4., 5., 8.])
```



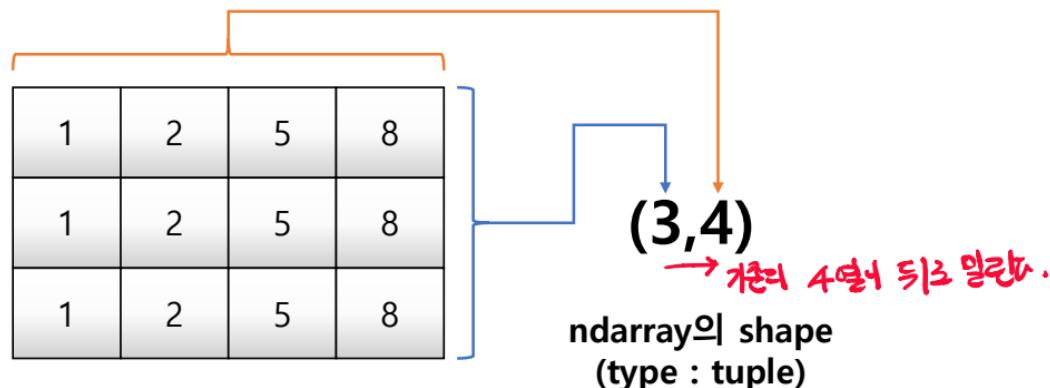
ndarray의 구성

ndarray의 shape
(type : tuple)

Array shape (matrix)

```
matrix = [[1,2,5,8],[1,2,5,8],[1,2,5,8]]  
np.array(matrix, int).shape
```

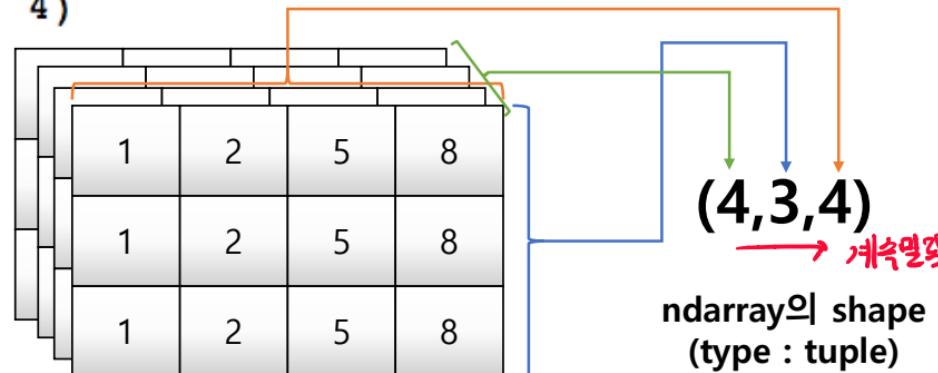
(3, 4)



Array shape (3rd order tensor)

```
tensor = [[[1,2,5,8],[1,2,5,8],[1,2,5,8]],
          [[1,2,5,8],[1,2,5,8],[1,2,5,8]],
          [[1,2,5,8],[1,2,5,8],[1,2,5,8]],
          [[1,2,5,8],[1,2,5,8],[1,2,5,8]]]
np.array(tensor, int).shape
```

(4, 3, 4)



차원이 늘어날 때 마다 ↗ 증가함.

Array shape – ndim & size

- **ndim** – number of dimension
- **size** – data의 개수 *전체 데이터 개수*

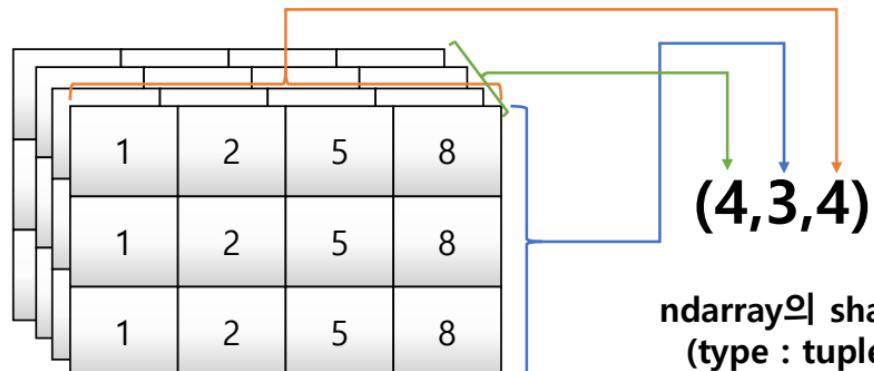
```
tensor = [[[1,2,5,8],[1,2,5,8],[1,2,5,8]],
          [[1,2,5,8],[1,2,5,8],[1,2,5,8]],
          [[1,2,5,8],[1,2,5,8],[1,2,5,8]],
          [[1,2,5,8],[1,2,5,8],[1,2,5,8]]]
```

```
np.array(tensor, int).ndim
```

3

```
np.array(tensor, int).size
```

48



Array dtype

- Ndarray의 single element가 가지는 data type
- 각 element가 차지하는 memory의 크기가 결정됨

```
np.array([[1, 2, 3], [4.5, 5, 6]], dtype=int)
```

```
array([[1, 2, 3],  
       [4, 5, 6]])
```

Data type을 integer로 선언

```
np.array([[1, 2, 3], [4.5, "5", "6"]], dtype=np.float32)
```

```
array([[ 1. ,  2. ,  3. ],  
       [ 4.5,  5. ,  6. ]], dtype=float32)
```

Data type을 float로 선언

Array dtype

- C의 data type과 compatible

Basic Type	Available NumPy types
Boolean	bool
Integer	int8, int16, int32, int64, int128, int
Unsigned Integer	uint8, uint16, uint32, uint64, uint128, uint
Float	float32, float64, float, longfloat, <i>(with)</i> .
Complex	complex64, complex128, complex
Strings	str, unicode
Object	object
Records	void

Data Types	Size in Bytes	Can contain:
boolean	1	true (1) or false (0)
char	1	ASCII character or signed value between -128 and 127
unsigned char, byte, uint8_t	1	ASCII character or unsigned value between 0 and 255
int, short	2	signed value between -32,768 and 32,767
unsigned int, word, uint16_t	2	unsigned value between 0 and 65,535
long	4	signed value between -2,147,483,648 and 2,147,483,647
unsigned long, uint32_t	4	unsigned value between 0 and 4,294,967,295
float, double	4	floating point value between - 3.4028235E+38 and 3.4028235E+38 (Note that double is the same as a float on this platform.)

Array dtype

- nbytes – ndarray object의 메모리 크기를 반환함

```
np.array([[1, 2, 3], [4.5, "5", "6"]], dtype=np.float32). nbytes
```

```
np.array([[1, 2, 3], [4.5, "5", "6"]], dtype=np.int8). nbytes
```

6 bits = 1 byte → 6 * 1 byte

```
np.array([[1, 2, 3], [4.5, "5", "6"]], dtype=np.float64). nbytes
```

48 **64bits = 8bytes → 6 * 48bytes**

Handling shape

reshape

- Array의 shape의 크기를 변경함 (element의 갯수는 동일)

data 개수만 맞추면댐.

1	2	5	8
1	2	5	8

2x4

(2,4)



1	2	5	8	1	2	5	8
---	---	---	---	---	---	---	---

vector

(8,)

reshape

- Array의 shape의 크기를 변경함 (element의 갯수는 동일)

```
test_matrix = [[1,2,3,4], [1,2,5,8]]  
np.array(test_matrix).shape
```

```
(2, 4)
```

```
np.array(test_matrix).reshape(8,)
```

```
array([1, 2, 3, 4, 1, 2, 5, 8])
```

```
np.array(test_matrix).reshape(8,).shape
```

```
(8, )
```

reshape

- Array의 size만 같다면 다차원으로 자유로이 변형가능

```
np.array(test_matrix).reshape(2,4).shape
```

(2, 4) → 8

```
np.array(test_matrix).reshape(-1,2).shape
```

(4, 2)

-1: size를 기반으로 row 개수 선정

```
np.array(test_matrix).reshape(2,2,2)
```

array([[[1, 2],
 [3, 4]],

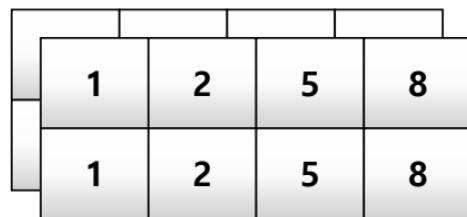
[[1, 2],
 [5, 8]]])

```
np.array(test_matrix).reshape(2,2,2).shape
```

(2, 2, 2)

flatten

- 다차원 array를 1차원 array로 변환



(2,2,4)



다차원을 1차원으로 평면화

1	2	5	8	1	2	5	8	1	2	5	8	1	2	5	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(16,)

flatten

- 다차원 array를 1차원 array로 변환

```
test_matrix = [[[1,2,3,4], [1,2,5,8]], [[1,2,3,4], [1,2,5,8]]]  
np.array(test_matrix).flatten()
```

```
array([1, 2, 3, 4, 1, 2, 5, 8, 1, 2, 3, 4, 1, 2, 5, 8])
```

Indexing & slicing

indexing

```
a = np.array([[1, 2, 3], [4.5, 5, 6]], int)
print(a)
print(a[0,0]) # Two dimensional array representation #1
print(a[0][0]) # Two dimensional array representation #2
a[0,0] = 12 # Matrix 0,0 에 12 할당
print(a)
a[0][0] = 5 # Matrix 0,0 에 12 할당
print(a)
```

- List와 달리 이차원 배열에서 [0,0] 과 같은 표기법을 제공함
- Matrix 일경우 앞은 row 뒤는 column을 의미함

indexing

```
test_exmaple = np.array([[1, 2, 3], [4.5, 5, 6]], int)  
test_exmaple
```

```
array([[1, 2, 3],  
       [4, 5, 6]])
```

```
test_exmaple[0][0]
```

```
1
```

```
test_exmaple[0,0]
```

```
1
```

indexing

```
test_exmaple[0,0] = 12 # Matrix 0,0 에 12 할당  
test_exmaple
```

```
array([[12, 2, 3],  
       [ 4, 5, 6]])
```

```
test_exmaple[0][0] = 5 # Matrix 0,0 에 12 할당  
test_exmaple[0,0]
```

slicing

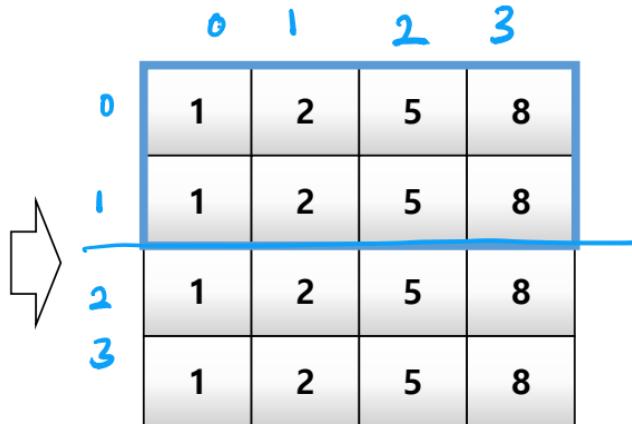
```
a = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]], int)  
a[:,2:] # 전체 Row의 2열 이상  
a[1,1:3] # 1 Row의 1열 ~ 2열  
a[1:3] # 1 Row ~ 2Row의 전체
```

for가 필요없이 데이터를 가져올수있음.

- List와 달리 행과 열 부분을 나눠서 slicing이 가능함
- Matrix의 부분 집합을 추출할 때 유용함

slicing

1	2	5	8
1	2	5	8
1	2	5	8
1	2	5	8



$[:2,:]$

Row - 0~1 까지 column - 전체

slicing

```
test_exmaple = np.array([
    [1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]], int)
test_exmaple[:2,:]
```

```
array([[1, 2, 5, 8],
       [1, 2, 5, 8]])
```

```
test_exmaple[:,1:3]
test_exmaple[1,:2]
```



1	2	5	8
1	2	5	8
1	2	5	8
1	2	5	8

slicing

```
test_exmaple = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]], int)
test_exmaple[:,2:] # 전체 Row의 2열 이상
```

```
array([[ 3,  4,  5],
       [ 8,  9, 10]])
```

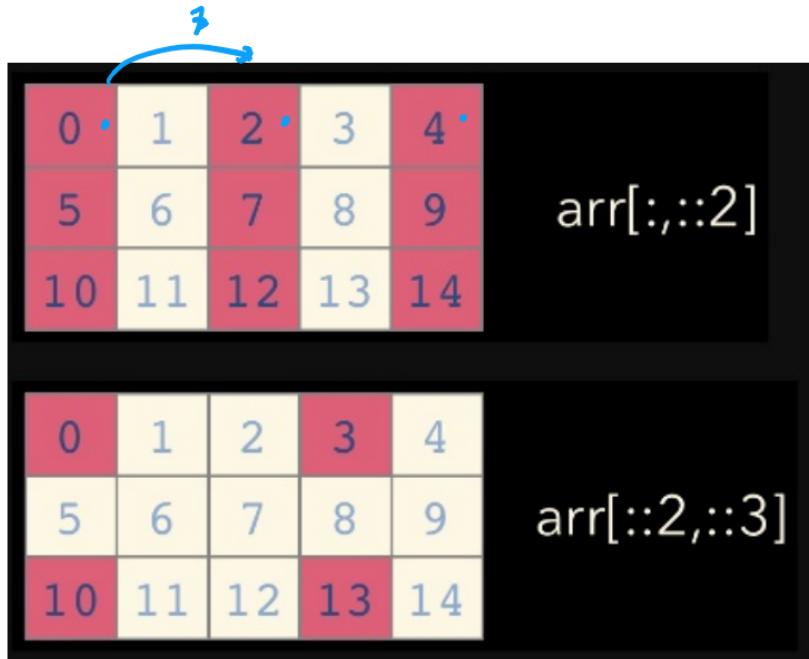
```
test_exmaple[1,1:3] # 1 Row의 1열 ~ 2열
```

```
array([7, 8])
```

```
test_exmaple[1:3] # 1 Row ~ 2Row의 전체
```

```
array([[ 6,  7,  8,  9, 10]])
```

slicing



$\frac{x}{4} = \frac{y}{2} = \frac{z}{step}$
몇 칸 건너

creation function

arange

.tolist() : 리스트로 바꿔줌

리스트에는 step floating point 런치 불허

- array의 범위를 지정하여, 값의 list를 생성하는 명령어

```
np.arange(30) # range: List의 range와 같은 효과, integer로 0부터 29까지 배열추출
```

```
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])
```

(시작, 끝, step)

```
np.arange(0, 5, 0.5) # floating point도 표시가능함
```

```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
```

+ 5 ← 몇 개는 자동으로 빼고 몇개는

```
np.arange(30).reshape(5,6)
```

```
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]])
```

ones, zeros and empty

- zeros – 0으로 가득찬 ndarray 생성

값을 0으로 지정

0으로 초기화

`np.zeros(shape, dtype, order)`

```
np.zeros(shape=(10,), dtype=np.int8) # 10 - zero vector 생성
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int8)
      0 1 2 3 4 5 6 7 8 9
```

```
np.zeros((2,5)) # 2 by 5 - zero matrix 생성
```

```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

ones, zeros and empty

- ones – 1로 가득찬 ndarray 생성

1337~

`np.ones(shape, dtype, order)`

```
np.ones(shape=(10,), dtype=np.int8)
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int8)
```

```
np.ones((2,5))
```

```
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

ones, zeros and empty

- empty – shape만 주어지고 비어있는 ndarray 생성
(memory initialization 이 되지 않음)

```
np.empty(shape=(10,), dtype=np.int8)  
array([ 0,  0,  0,  0,  0,  0,  0,  64, -74, 105], dtype=int8)
```

```
np.empty((3,5))  
array([[ 2.00000000e+000,   2.00000000e+000,   6.42285340e-323,  
        0.00000000e+000,   0.00000000e+000],  
       [ 0.00000000e+000,   0.00000000e+000,   0.00000000e+000,  
        0.00000000e+000,   0.00000000e+000],  
       [ 0.00000000e+000,   2.12199579e-314,   2.00000000e+000,  
        2.00000000e+000,   3.45845952e-323]])
```

0으로 채워져 있고
이상은 쓰레기 값이 들어감.
그냥 페인트 같은 걸로 잘라 기능

something_like

- 기존 ndarray의 shape 크기 만큼 1, 0 또는 empty array를 반환

```
test_matrix = np.arange(30).reshape(5,6)
np.ones_like(test_matrix) shape 만큼
```

```
array([[1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1],    13 가로줄 matrix
       [1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1]])
```

zeros_like.
emptys_like.

identity

- 단위 행렬(i 행렬)을 생성함

$n \rightarrow$ number of rows

```
np.identity(n=3, dtype=np.int8)
```

```
array([[1, 0, 0],  
       [0, 1, 0],  
       [0, 0, 1]], dtype=int8)
```

```
np.identity(5)
```

```
array([[ 1.,  0.,  0.,  0.,  0.],  
       [ 0.,  1.,  0.,  0.,  0.],  
       [ 0.,  0.,  1.,  0.,  0.],  
       [ 0.,  0.,  0.,  1.,  0.],  
       [ 0.,  0.,  0.,  0.,  1.]])
```

단위 행렬

eye

- 대각선인 1인 행렬, k값의 시작 index의 변경이 가능

row col

```
np.eye(N=3, M=5, dtype=np.int8)
```

```
array([[1, 0, 0, 0, 0],  
       [0, 1, 0, 0, 0],  
       [0, 0, 1, 0, 0]], dtype=int8)
```

```
np.eye(3)
```

```
array([[ 1.,  0.,  0.],  
       [ 0.,  1.,  0.],  
       [ 0.,  0.,  1.]])
```

np.eye(3, 5, k=2) k → start index

0 / 2
array([[0., 0., 1., 0., 0.],
 [0., 0., 0., 1., 0.],
 [0., 0., 0., 0., 1.]])

diag

- 대각 행렬의 값을 추출함

0	1	2
3	4	5
6	7	8

```
matrix = np.arange(9).reshape(3,3)
np.diag(matrix)

array([0, 4, 8])
```

0	1	2
3	4	5
6	7	8

```
np.diag(matrix, k=1)  k → start index

array([1, 5])
```

random sampling

- 데이터 분포에 따른 sampling으로 array를 생성

```
np.random.uniform(0,1,10).reshape(2,5) 균등분포
```

```
array([[ 0.67406593,  0.71072857,  0.06963986,  0.09194939,  0.47293574],  
       [ 0.13840676,  0.97410297,  0.60703044,  0.04002073,  0.08057727]])
```

```
np.random.normal(0,1,10).reshape(2,5) 정규분포
```

```
array([[ 1.02694847,  0.39354215,  0.63411928, -1.03639086, -1.76669162],  
       [ 0.50628853, -1.42496802,  1.23288754,  1.26424168,  0.53718751]])
```

operation functions

sum

- ndarray의 element들 간의 합을 구함, list의 sum 기능과 동일

```
test_array = np.arange(1,11)  
test_array
```

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

```
test_array.sum(dtype=np.float)
```

```
55.0
```

axis

- 모든 operation function을 실행할 때, 기준이 되는 dimension 축

axis=1

axis=1			
1	2	3	4
5	6	7	8
9	10	11	12
15 18 21 24			10 26 42
axis = 0		axis=1	

```
test_array = np.arange(1,13).reshape(3,4)
```

test_array *모든 sum () 를 찾음*

```
array([[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 11, 12]])
```

```
test_array.sum(axis=1), test_array.sum(axis=0)
```

10 26 42
(array([10, 26, 42]), array([15, 18, 21, 24]))

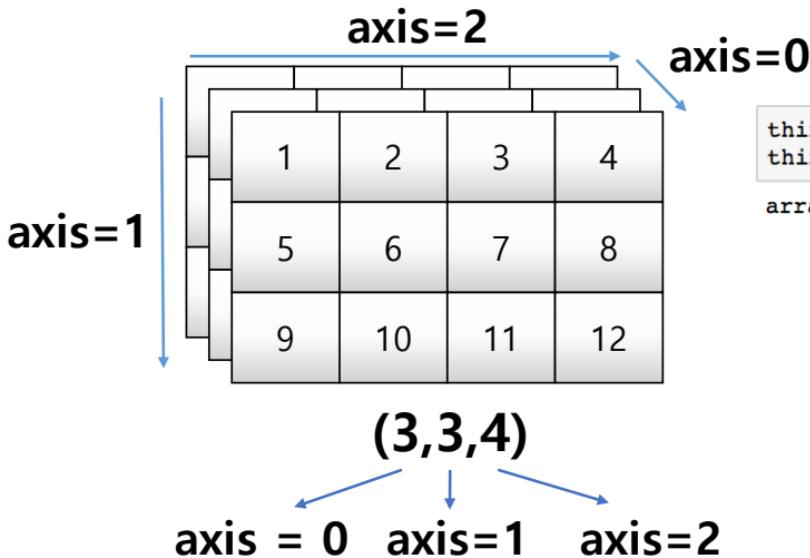
(3,)

(1, 3)

원점.

axis

- 모든 operation function을 실행할 때, 기준이 되는 dimension 축

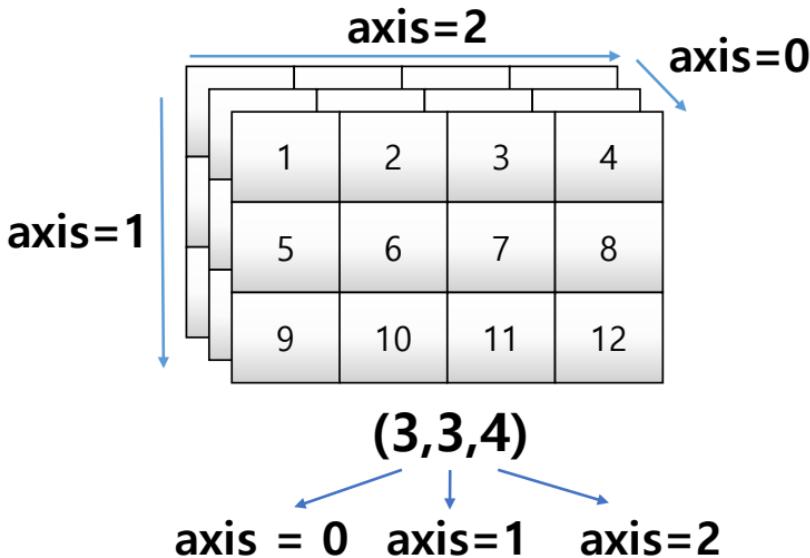


```
third_order_tensor = np.array([test_array,test_array,test_array])  
third_order_tensor  
  
array([[[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 11, 12]],  
  
      [[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 11, 12]],  
  
      [[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 11, 12]]])
```

새로생기는 축이 0
기준의 축은 위로 한석일직

axis

- 모든 operation function을 실행할 때, 기준이 되는 dimension 축



```
third_order_tensor.sum(axis=2)
```

```
array([[10, 26, 42],  
       [10, 26, 42],  
       [10, 26, 42]])
```

```
third_order_tensor.sum(axis=1)
```

```
array([[15, 18, 21, 24],  
       [15, 18, 21, 24],  
       [15, 18, 21, 24]])
```

```
third_order_tensor.sum(axis=0)
```

```
array([[ 3,  6,  9, 12],  
       [15, 18, 21, 24],  
       [27, 30, 33, 36]])
```

mean & std

- ndarray의 element들 간의 평균 또는 표준 편차를 반환

```
test_array = np.arange(1,13).reshape(3,4)  
test_array
```

```
array([[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 11, 12]])
```

```
test_array.mean(), test_array.mean(axis=0)  
(6.5, array([ 5.,  6.,  7.,  8.]))
```

```
test_array.std(), test_array.std(axis=0)
```

```
(3.4520525295346629,  
 array([ 3.26598632,  3.26598632,  3.26598632,  3.26598632]))
```

Mathematical functions

- 그 외에도 다양한 수학 연산자를 제공함

exponential: `exp`, `expm1`, `exp2`, `log`, `log10`, `log1p`, `log2`,
`power`, `sqrt`

trigonometric: `sin`, `cos`, `tan`, `acsin`, `arccos`, `atctan`

hyperbolic: `sinh`, `cosh`, `tanh`, `acsinh`, `arccosh`, `atctanh`

Mathematical functions

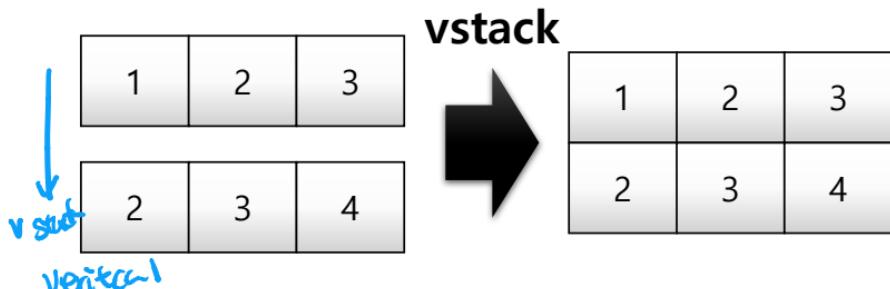
- 그 외에도 다양한 수학 연산자를 제공함 (np.something 호출)

```
np.exp(test_array), np.sqrt(test_array)
```

```
(array([[ 2.71828183e+00,    7.38905610e+00,    2.00855369e+01,
          5.45981500e+01],
         [ 1.48413159e+02,    4.03428793e+02,    1.09663316e+03,
          2.98095799e+03],
         [ 8.10308393e+03,    2.20264658e+04,    5.98741417e+04,
          1.62754791e+05]]),
 array([[ 1.          ,   1.41421356,   1.73205081,   2.          ],
         [ 2.23606798,   2.44948974,   2.64575131,   2.82842712],
         [ 3.          ,   3.16227766,   3.31662479,   3.46410162]]))
```

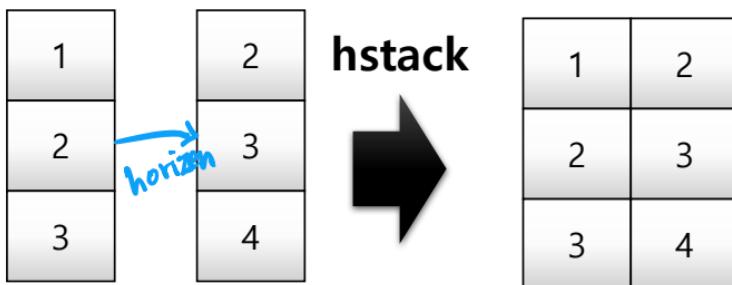
concatenate 풀이

- Numpy array를 합치는 함수



```
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])
np.vstack((a,b))
```

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



```
a = np.array([[1, 2, 3]])
b = np.array([[2, 3, 4]])
np.hstack((a,b))
```

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

concatenate

concatenate / axis=0

1	2	3
2	3	4



1	2	3
2	3	4

concatenate / axis=1

1	2
3	4

5
6



1	2	5
3	4	6

한줄

열.인수

한줄

행.인수

or

axis=0

axis=0

axis=1

```
a = np.array([[1, 2, 3]])  
b = np.array([[2, 3, 4]])  
np.concatenate( (a,b) ,axis=0)
```

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

수행을 위한 예제입니다.

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

```
np.concatenate( (a,b.T) ,axis=1)
```

```
array([[1, 2, 5],  
       [3, 4, 6]])
```

array operations

Operations b/t arrays

- Numpy는 array간의 기본적인 사칙 연산을 지원함

```
test_a = np.array([[1,2,3],[4,5,6]], float)
```

```
test_a + test_a # Matrix + Matrix 연산 행렬덧셈
```

```
array([[ 2.,  4.,  6.],  
       [ 8., 10., 12.]])
```

```
test_a - test_a # Matrix - Matrix 연산 빼셈
```

```
array([[ 0.,  0.,  0.],  
       [ 0.,  0.,  0.]])
```

x. dot(Y) 행렬곱셈

```
test_a * test_a # Matrix와 element들 간 같은 위치에 있는 값들끼리 연산
```

```
array([[ 1.,  4.,  9.],  
       [16., 25., 36.]])
```

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix} \times \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix} = \begin{pmatrix} 1 & 4 & 9 \end{pmatrix}$$

Element-wise operations

- Array간 shape이 같을 때 일어나는 연산

1	2	3	4
5	6	7	8
9	10	11	12



1	2	3	4
5	6	7	8
9	10	11	12



1	4	9	16
25	36	49	64
81	100	121	144

```
matrix_a = np.arange(1,13).reshape(3,4)  
matrix_a * matrix_a
```

```
array([[ 1,   4,   9,  16],  
       [ 25,  36,  49,  64],  
       [ 81, 100, 121, 144]])
```

Dot product

- Matrix의 기본 연산
- dot 함수 사용

"Dot Product"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \end{bmatrix}$$

```
test_a = np.arange(1,7).reshape(2,3)
test_b = np.arange(7,13).reshape(3,2)
```

```
test_a.dot(test_b)
```

```
array([[ 58,  64],
       [139, 154]])
```

transpose

전치행렬.

- transpose 또는 T attribute 사용

```
test_a = np.arange(1,7).reshape(2,3)  
test_a
```

```
array([[1, 2, 3],  
       [4, 5, 6]])
```

```
test_a.T.dot(test_a) # Matrix 간 곱셈
```

```
array([[ 17.,  22.,  27.],  
       [ 22.,  29.,  36.],  
       [ 27.,  36.,  45.]])
```

```
test_a.transpose()
```

```
array([[1, 4],  
       [2, 5],  
       [3, 6]])
```

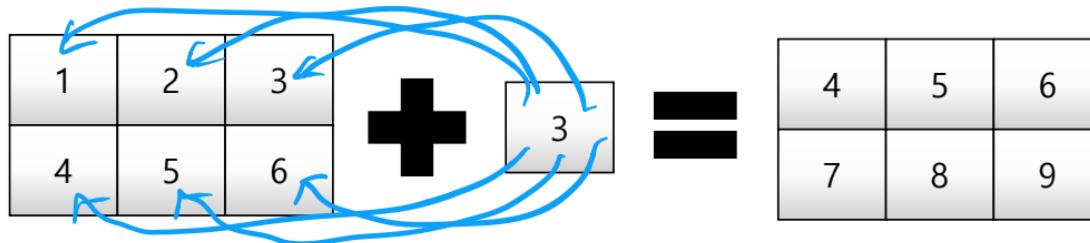
```
test_a.T
```

```
array([[1, 4],  
       [2, 5],  
       [3, 6]])
```

broadcasting

- Shape이 다른 배열 간 연산을 지원하는 기능

matrix or scalar \Rightarrow
계산



```
test_matrix = np.array([[1,2,3],[4,5,6]], float)  
scalar = 3
```

```
test_matrix + scalar # Matrix - Scalar 덧셈
```

```
array([[ 4.,  5.,  6.],  
       [ 7.,  8.,  9.]])
```

broadcasting

```
test_matrix - scalar # Matrix - Scalar 뺄셈
```

```
array([[ -4., -3., -2.],  
       [-1.,  0.,  1.]])
```

```
test_matrix * 5 # Matrix - Scalar 곱셈
```

```
array([[ 5., 10., 15.],  
       [20., 25., 30.]])
```

```
test_matrix / 5 # Matrix - Scalar 나눗셈
```

```
array([[ 0.2,  0.4,  0.6],  
       [ 0.8,  1. ,  1.2]])
```

```
test_matrix // 0.2 # Matrix - Scalar 몫
```

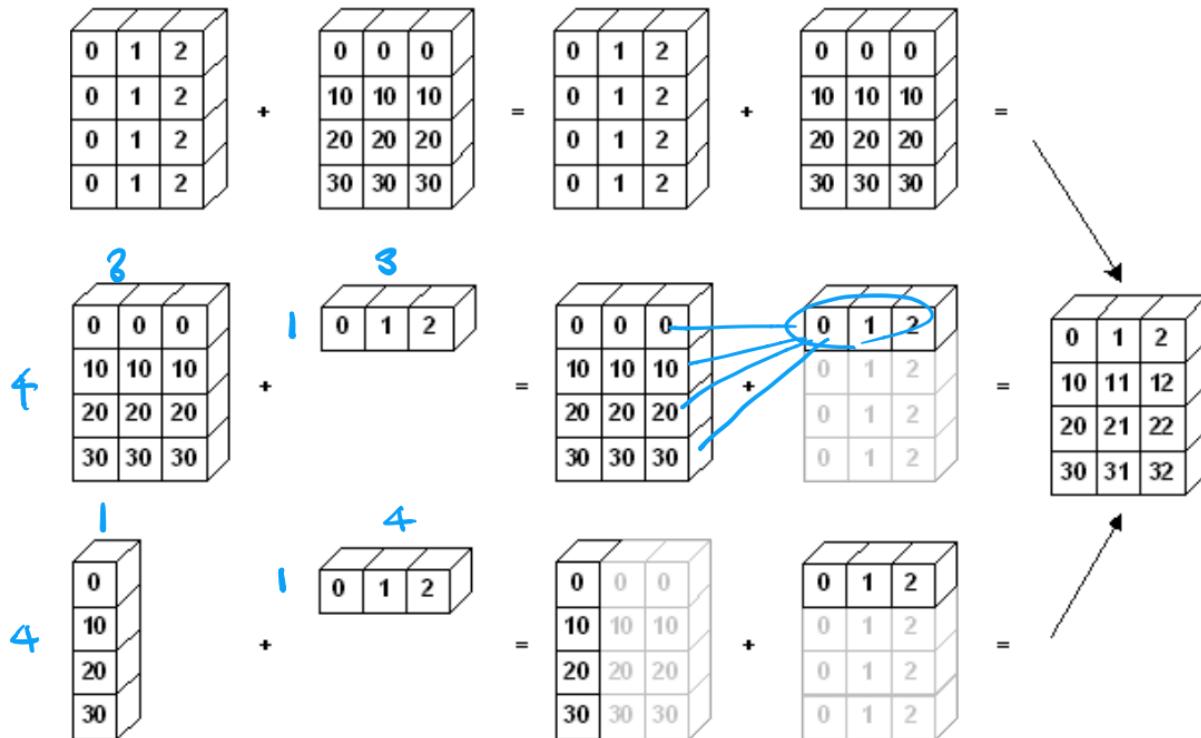
```
array([[ 4.,  9., 14.],  
       [19., 24., 29.]])
```

```
test_matrix ** 2 # Matrix - Scalar 제곱
```

```
array([[ 1.,  4.,  9.],  
       [16., 25., 36.]])
```

broadcasting

- Scalar – vector 외에도 vector – matrix 간의 연산도 지원



broadcasting

1	2	3
4	5	6
7	8	9
10	11	12



10	20	30
----	----	----



11	22	33
14	25	36
17	28	39
20	31	42

```
test_matrix = np.arange(1,13).reshape(4,3)
test_vector = np.arange(10,40,10)
test_matrix + test_vector
```

```
array([[11, 22, 33],
       [14, 25, 36],
       [17, 28, 39],
       [20, 31, 42]])
```

Numpy performance #1

```
def scalar_vector_product(scalar, vector):
    result = []
    for value in vector:
        result.append(scalar * value)
    return result

iteration_max = 100000000

vector = list(range(iteration_max))
scalar = 2

%timeit scalar_vector_product(scalar, vector) # for loop을 이용한 성능
%timeit [scalar * value for value in range(iteration_max)] # list comprehension을 이용한 성능
%timeit np.arange(iteration_max) * scalar # numpy를 이용한 성능
```

- **timeit:** jupyter 환경에서 코드의 퍼포먼스를 체크하는 함수

Numpy performance #2

- 일반적으로 속도는 아래 순
for loop < list comprehension < numpy
- 100,000,000 번의 loop이 돌 때 약 4배 이상의 성능 차이를 보임
- Numpy는 C로 구현되어 있어, 성능을 확보하는 대신
- 파이썬의 가장 큰 특징인 **dynamic typing**을 포기함
- 대용량 계산에서는 가장 흔히 사용됨
- **Concatenate** 처럼 계산이 아닌, 할당에서는 연산 속도의 이점이 없음

comparisons*

All & Any

- Array의 데이터 전부(and) 또는 일부(or)가 조건에 만족 여부 반환

```
a = np.arange(10)
```

```
a
```

$a > 5$

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

False False False False

True True True True

Broadcasting.

```
np.any(a>5), np.any(a<0)
```

any → 하나라도 조건에 만족한다면 true

```
(True, False)
```

```
np.all(a>5), np.all(a < 10)
```

all → 모두가 조건에 만족한다면 true

```
(False, True)
```

Comparison operation #1

- Numpy는 배열의 크기가 동일 할 때
element간 비교의 결과를 Boolean type으로 반환하여 돌려줌

```
test_a = np.array([1, 3, 0], float)
test_b = np.array([5, 2, 1], float)
test_a > test_b
```

```
array([False, True, False], dtype=bool)
```

```
test_a == test_b
```

```
array([False, False, False], dtype=bool)
```

```
(test_a > test_b).any()
```

True

any → 하나라도 true라면 true

Comparison operation #2

```
a = np.array([1, 3, 0], float)
np.logical_and(a > 0, a < 3) # and 조건의 condition
&&
array([ True, False, False], dtype=bool)
```

```
b = np.array([True, False, True], bool)
np.logical_not(b) # NOT 조건의 condition
^
array([False,  True, False], dtype=bool)
```

```
c = np.array([False, True, False], bool)
np.logical_or(b, c) # OR 조건의 condition
||
array([ True,  True,  True], dtype=bool)
```

np.where

조건을 만족하는 index를 반환

```
np.where(a > 0, 3, 2) # where(condition, TRUE, FALSE)
```

True case
array([3, 3, 2])
만족하는
값 반환

```
a = np.arange(10)      Index 값 반환
```

```
(array([6, 7, 8, 9]),)
```

```
a = np.array([1, np.NaN, np.Inf], float)
```

np.isnan(a) Not a Number null값, 무한값

```
array([False, True, False], dtype=bool)
```

```
np.isfinite(a)       is finite number
```

```
array([ True, False, False], dtype=bool)
```

argmax & argmin

- array내 최대값 또는 최소값의 index를 반환함

```
a = np.array([1,2,4,5,8,78,23,3])  
np.argmax(a) , np.argmin(a)
```

(5, 0) 최대값의 index, 최소값의 index.

- axis 기반의 반환

axis=1

1	2	4	7
9	88	6	45
9	76	3	4

axis=0

```
a=np.array([[1,2,4,7],[9,88,6,45],[9,76,3,4]])  
np.argmax(a, axis=1) , np.argmin(a, axis=0)
```

(array([3, 1, 1]), array([0, 0, 2, 2]))

최대값으로

최소.최대

boolean & fancy index

numpy의高级 쓰는 법 학습 중.

boolean index *

앞이쓰임.

(*index를 뽑는건 where.
값을 뽑는건 boolean index*)

- numpy는 배열은 특정 조건에 따른 값을 배열 형태로 추출 할 수 있음
- Comparison operation 함수들도 모두 사용가능

```
test_array = np.array([1, 4, 0, 2, 3, 8, 9, 7], float)  
test_array > 3
```

```
array([False, True, False, False, False, True, True, True], dtype=bool)
```

```
test_array[test_array > 3]
```

조건이 True인 index의 element만 추출

```
array([ 4., 8., 9., 7.])
```

```
condition = test_array < 3  
test_array[condition]
```

```
array([ 1., 0., 2.])
```

boolean index

```
A = np.array([
[12, 13, 14, 12, 16, 14, 11, 10, 9],
[11, 14, 12, 15, 15, 16, 10, 12, 11],
[10, 12, 12, 15, 14, 16, 10, 12, 12],
[ 9, 11, 16, 15, 14, 16, 15, 12, 10],
[12, 11, 16, 14, 10, 12, 16, 12, 13],
[10, 15, 16, 14, 14, 14, 16, 15, 12],
[13, 17, 14, 10, 14, 11, 14, 15, 10],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 19, 12, 14, 11, 12, 14, 18, 10],
[14, 22, 17, 19, 16, 17, 18, 17, 13],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 19, 12, 14, 11, 12, 14, 18, 10],
[14, 22, 12, 14, 11, 12, 14, 17, 13],
[10, 16, 12, 14, 11, 12, 14, 18, 11]]))

B = A < 15
B.astype(np.int) True → 1
                           False → 0
```

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

fancy index

- numpy는 array를 index value로 사용해서 값을 추출하는 방법

```
a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int) # 반드시 integer로 선언
a[b] #bracket index, b 배열의 값을 index로 하여 a의 값을 추출함
```

array([2., 2., 4., 8., 6., 4.])

참조하는 방식

a.take(b) #take 함수: bracket index와 같은 효과

0 1 2 3

2	4	6	8
---	---	---	---

array([2., 2., 4., 8., 6., 4.])

fancy index

- Matrix 형태의 데이터도 가능

```
a = np.array([[1, 4], [9, 16]], float)
b = np.array([0, 0, 1, 1, 0], int)
c = np.array([0, 1, 1, 1, 1], int)
a[b,c] # b는 row index, c는 column index로 변환하여 표시함
```

	0	1
0	1	4
1	9	16

```
array([ 1., 4., 16., 16., 4.])
```

numpy data i/o

loadtxt & savetxt

- Text type의 데이터를 읽고, 저장하는 기능

```
a = np.loadtxt("./populations.txt")  
a[:10]
```

파일 호출

```
array([[ 1900., 30000., 4000., 48300.],  
       [ 1901., 47200., 6100., 48200.],  
       [ 1902., 70200., 9800., 41500.],  
       [ 1903., 77400., 35200., 38200.],  
       [ 1904., 36300., 59400., 40600.],  
       [ 1905., 20600., 41700., 39800.],  
       [ 1906., 18100., 19000., 38600.],  
       [ 1907., 21400., 13000., 42300.],  
       [ 1908., 22000., 8300., 44500.],  
       [ 1909., 25400., 9100., 42100.]])
```

Int type 변환

```
a_int = a.astype(int)  
a_int[:3]
```

```
array([[ 1900, 30000, 4000, 48300],  
       [ 1901, 47200, 6100, 48200],  
       [ 1902, 70200, 9800, 41500]])
```

```
np.savetxt('int_data.csv',a_int, delimiter=",")
```

int_data.csv로 저장

1.900000000000000e+03,3.000000000000000e+04,4.000000000000000e+03,4.830000000000000e+04
1.901000000000000e+03,4.720000000000000e+04,6.100000000000000e+03,4.820000000000000e+04
1.902000000000000e+03,7.020000000000000e+04,9.800000000000000e+03,4.150000000000000e+04
1.903000000000000e+03,7.740000000000000e+04,3.520000000000000e+04,3.820000000000000e+04
1.904000000000000e+03,3.630000000000000e+04,5.940000000000000e+04,4.060000000000000e+04
1.905000000000000e+03,2.060000000000000e+04,4.170000000000000e+04,3.980000000000000e+04
1.906000000000000e+03,1.810000000000000e+04,1.900000000000000e+04,3.860000000000000e+04
1.907000000000000e+03,2.140000000000000e+04,1.300000000000000e+04,4.230000000000000e+04
1.908000000000000e+03,2.200000000000000e+04,8.300000000000000e+03,4.450000000000000e+04
1.909000000000000e+03,2.540000000000000e+04,9.100000000000000e+03,4.210000000000000e+04
1.910000000000000e+03,2.710000000000000e+04,7.400000000000000e+03,4.600000000000000e+04
1.911000000000000e+03,4.030000000000000e+04,8.000000000000000e+03,4.680000000000000e+04
1.912000000000000e+03,5.700000000000000e+04,1.230000000000000e+04,4.380000000000000e+04
1.913000000000000e+03,7.660000000000000e+04,1.950000000000000e+04,4.090000000000000e+04
1.914000000000000e+03,5.230000000000000e+04,4.570000000000000e+04,3.940000000000000e+04
1.915000000000000e+03,1.950000000000000e+04,5.110000000000000e+04,3.900000000000000e+04
1.916000000000000e+03,1.120000000000000e+04,2.970000000000000e+04,3.670000000000000e+04
1.917000000000000e+03,7.600000000000000e+03,1.580000000000000e+04,4.180000000000000e+04
1.918000000000000e+03,1.460000000000000e+04,9.700000000000000e+03,4.330000000000000e+04
1.919000000000000e+03,1.620000000000000e+04,1.010000000000000e+04,4.130000000000000e+04
1.920000000000000e+03,2.470000000000000e+04,8.600000000000000e+03,4.730000000000000e+04

numpy object - npy

- Numpy object (pickle) 형태로 데이터를 저장하고 불러옴
- Binary 파일 형태로 저장함

```
np.save("npy_test", arr=a_int)
```

Last executed 2017-09-26 11:36:56 in 4ms

```
npy_array = np.load(file="npy_test.npy")
npy_array[:3]
```

Last executed 2017-09-26 11:37:07 in 5ms

```
array([[ 1900, 30000,   4000, 48300],
       [ 1901, 47200,   6100, 48200],
       [ 1902, 70200,   9800, 41500]])
```

TEAMLAB

Human knowledge belongs to the world.