Numpy

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)
test_array
```

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

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

```
numpy.float64
```

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')
```

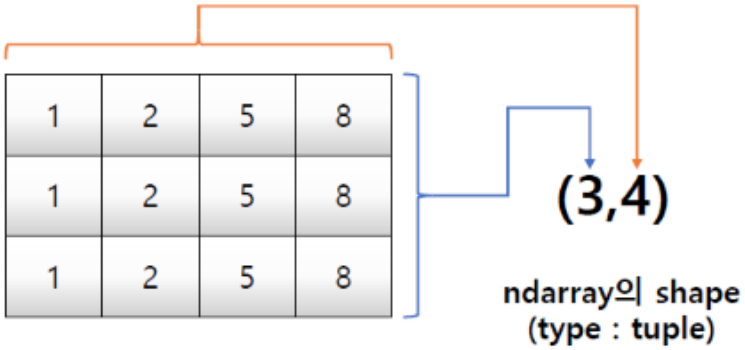
```
test_array.shape   # Array(배열) 의 shape을 반환함
```

```
(4,)
```

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)
```

Array shape – ndim & size

- ndim – number of dimension
- size – data의 개수

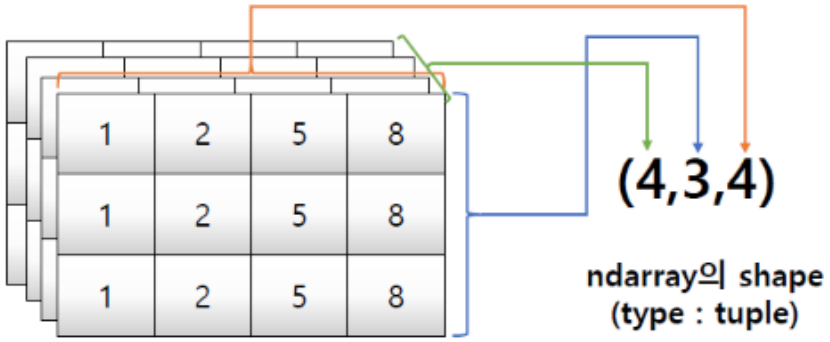
```
tensor  = [[[1,2,5,8],[1,2,5,8],[1,2,5,8]],
           [[1,2,5,8],[1,2,5,8],[1,2,5,8]],
           [[1,2,5,8],[1,2,5,8],[1,2,5,8]],
           [[1,2,5,8],[1,2,5,8],[1,2,5,8]]]
```

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

```
3
```

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

```
48
```



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)
```

```
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)
```

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

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
```

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

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]])
```

arange

- 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])
```

```
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]])
```

sum

- ndarray의 element들 간의 합을 구함, list의 sum 기능과 동일

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

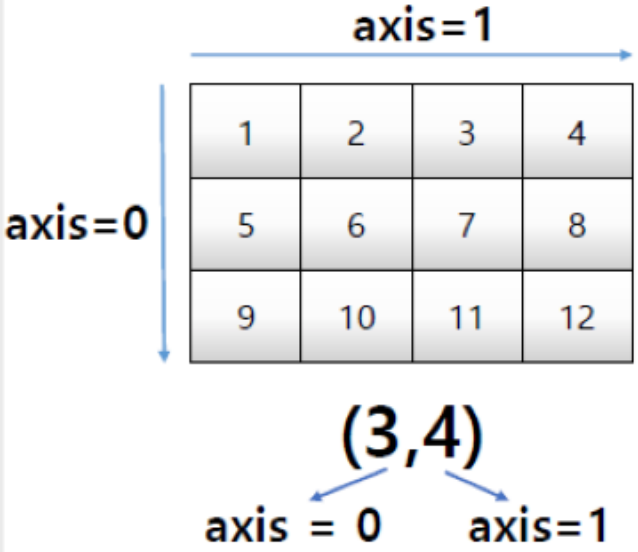
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

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

```
55.0
```

axis

- 모든 operation function을 실행할 때, 기준이 되는 dimension 축




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

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

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

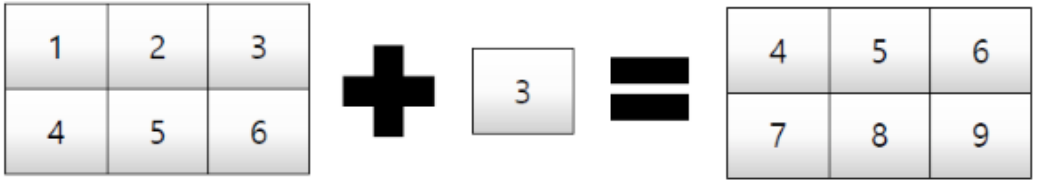
- 그 외에도 다양한 수학 연산자를 제공함 (np.something 호출)

```
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]]))
```

broadcasting

- Shape이 다른 배열 간 연산을 지원하는 기능



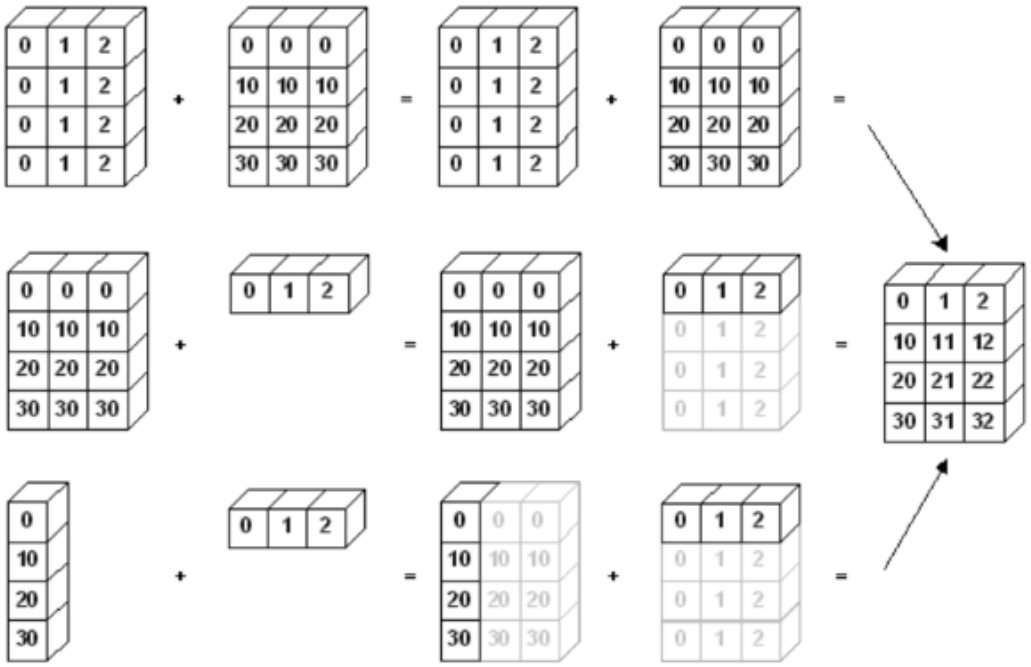

```
test_matrix = np.array([[1,2,3],[4,5,6]], float)
scalar = 3
```

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

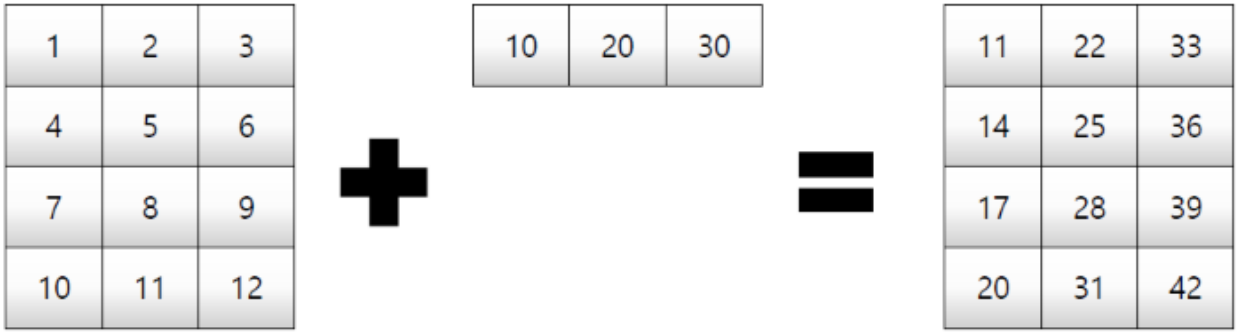
```
array([[ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
```

broadcasting

- Scalar – vector 외에도 vector – matrix 간의 연산도 지원



broadcasting




```
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]])
```

All & Any

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

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

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

```
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)
```

- 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)
```

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]
```

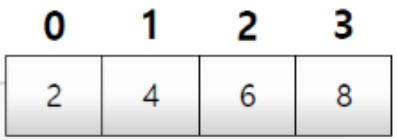
```
array([ 1.,  0.,  2.])
```

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 값들을 추출함
```

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




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

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