< Numpy란? >

- Numerical Python
- 파이썬의 고성능 과학 계산용 패키지
- Matrix와 Vector와 같은 Array 연산의 표준

< Numpy 특징 >

- 일반 List에 비해 빠르고 메모리를 효율적으로 사용
- 반복문 없이 데이터 배열에 대한 처리 지원
- 선형대수와 관련된 다양한 기능을 제공함
- C, C++, 포트란 등의 언어와 통합 가능

1. ndarray

numpy 호출

In [1]:

```
import numpy as np
```

array

- 하나의 데이터 type만 배열에 넣을 수 있음
- C의 Array를 사용하여 배열을 생성함
- List와 가장 큰 차이점 : Dynamic typing not supported

In [2]:

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

Out[2]:

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

In [3]:

```
type(test_array[3])      # Float Type
```

Out[3]:

```
numpy.float64
```

In [4]:

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

Out[4]:

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

In [5]:

```
type(test_array[3])      # Float Type
```

Out[5]:

```
numpy.float64
```

- dtype : numpy array의 데이터 type을 반환함
- shape : numpy array의 object의 dimension구성을 반환함

In [6]:

```
test_array.dtype      # Array Type
```

Out[6]:

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

In [7]:

```
test_array
```

Out[7]:

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

In [8]:

```
test_array.shape      # Array Type
```

Out[8]:

```
(4,)
```

array shape

In [9]:

```
vector = [1, 2, 3, 4]
np.array(vector, int).shape
```

Out[9]:

```
(4,)
```

In [10]:

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

Out[10]:

```
(3, 4)
```

In [11]:

```
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
```

Out[11]:

```
(4, 3, 4)
```

In [12]:

```
ten = np.array([[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]] * 4)
np.array(ten, int).shape
```

Out[12]:

```
(12, 4)
```

In [13]:

```
ten = ten.reshape(4, 3, 4)
np.array(ten, int).shape
```

Out[13]:

```
(4, 3, 4)
```

In [14]:

```
t = np.array([1, 2, 5, 8] * 12)
np.array(t, int).shape
```

Out[14]:

```
(48,)
```

In [15]:

```
t = t.reshape(4, 3, 4)
np.array(t, int).shape
```

Out[15]:

```
(4, 3, 4)
```

In [16]:

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

Out[16]:

```
3
```

In [17]:

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

Out[17]:

```
48
```

numpy dtype

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

In [18]:

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

Out[18]:

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

In [19]:

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

Out[19]:

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

In [20]:

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

Out[20]:

```
24
```

In [21]:

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

Out[21]:

```
6
```

In [22]:

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

Out[22]:

```
48
```

2. reshape

reshape

- Array의 size만 같다면 다차원으로 자유로이 변형 가능 (element의 갯수는 동일)

In [23]:

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

Out[23]:

```
(2, 4)
```

In [24]:

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

Out[24]:

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

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

In [25]:

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

Out[25]:

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

In [26]:

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

Out[26]:

```
(8,)
```

In [27]:

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

Out[27]:

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

In [28]:

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

Out[28]:

```
(2, 4)
```

In [29]:

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

Out[29]:

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

In [30]:

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

Out[30]:

```
(2, 4)
```

In [31]:

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

Out[31]:

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

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

In [32]:

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

Out[32]:

```
(2, 2, 2)
```

flat or flatten()

- 다차원 array를 1차원 array로 변환

In [33]:

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

```
np.array(test_matrix).flatten()
```

Out[33]:

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

3. indexing, slicing

indexing

In [34]:

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

Out[34]:

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

모두 0행 0열의 index를 가리킨다.

In [35]:

```
test_example[0][0]
```

Out[35]:

```
1
```

In [36]:

```
test_example[0, 0]
```

Out[36]:

```
1
```

In [37]:

```
test_example[0, 0] = 10
test_example
```

Out[37]:

```
array([[10,  2,  3],
       [ 4,  5,  6]])
```

In [38]:

```
test_example[0][0] = 5
test_example[0, 0]
```

Out[38]:

```
5
```

slicing

In [39]:

```
test_example = np.array([[1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8], [1, 2, 5, 8]],int)
test_example[:2, :]     # 처음부터 2미만 행까지 모든 열 (0 ~ 1행)
```

Out[39]:

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

In [40]:

```
print(test_example[:, 1:3])      # 모든 행 1열부터 3미만 열까지 (모든 행 1 ~ 2열)
print(test_example[1, :2])       # 1행 처음부터 3미만 열까지 (1행 0 ~ 2열)
```

```
[[2 5]
 [2 5]
 [2 5]
 [2 5]]
[1 2]
```

In [41]:

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

Out[41]:

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

In [42]:

```
test_example[1, 1:3]     # 1행 1 ~ 2열
```

Out[42]:

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

In [43]:

```
test_example[1:3]     # 1 ~ 2행  -> 1행까지밖에 없으므로 1행만 출력
```

Out[43]:

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

arange

- array의 범위를 지정하여 값의 list를 생성하는 명령어

In [44]:

```
np.arange(30)     # integer로 0부터 29까지 배열추출
```

Out[44]:

```
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])
```

In [45]:

```
np.arange(0, 5, 0.5)     # float로 0부터 4.5까지 0.5크기로 배열추출
```

Out[45]:

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

In [46]:

```
np.arange(30).reshape(5, 6)      # integer로 0부터 29까지 배열추출 후 5행 6열로 reshape
```

Out[46]:

```
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]])
```

In [47]:

```
a = np.arange(100).reshape(10, 10)      # integer로 0부터 99까지 배열추출 후 10행 10열로 reshape
a
```

Out[47]:

```
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, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

In [48]:

```
a[:, -1]      # -1열 즉, 가장 마지막 열인 9열을 의미함
```

Out[48]:

```
array([ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99])
```

In [49]:

```
a[:, -1].reshape(2, 5)      # 1차원 array를 같은 size인 2행 5열로 reshape
```

Out[49]:

```
array([[ 9, 19, 29, 39, 49],
       [59, 69, 79, 89, 99]])
```

4. creation function

zeros, ones & empty

- zeros : 0으로 가득찬 ndarray 생성

In [50]:

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

Out[50]:

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

In [51]:

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

Out[51]:

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

- ones : 1로 가득찬 ndarray 생성

In [52]:

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

Out[52]:

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

In [53]:

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

Out[53]:

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

- enpty : shape만 주어지고 비어있는 ndarray 생성

  (memory initialization이 되지 않음)

In [54]:

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

Out[54]:

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

In [55]:

```
np.empty((3, 5))
```

Out[55]:

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

- something_like : 기존 ndarray의 shape 크기만큼 1, 0 또는 empty array를 반환

In [56]:

```
test_matrix = np.arange(30).reshape(5, 6)
np.zeros_like(test_matrix)      # 0으로 30size의 5행 6열 array 생성
```

Out[56]:

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

identity, eye & digonal

- identity : 단위 행렬(i 행렬)을 생성함 -> i x i 행렬 생성

In [57]:

```
np.identity(n = 3, dtype = np.int8)      # 3행 3열의 array
```

Out[57]:

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

In [58]:

```
np.identity(5)      # 5행 5열의 array
```

Out[58]:

```
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.]])
```

- eye : 대각선이 1인 행렬 -> 기본 (i, i) = 1인 행렬

  N, M값 : array의 shape를 지정할 수 있음 (dsfault는 eye(n)에서 n행 n열의 array)
  k값 : 시작 index의 변경이 가능 -> (i+k, i) = 1

In [59]:

```
np.eye(N = 3, M = 5, dtype = np.int8)      # N은 행, M은 열  -> 3행 5열의 array
```

Out[59]:

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

In [60]:

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

Out[60]:

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

In [61]:

```
np.eye(3, 5, k = 2)      # 3행 5열의 array에서 0행 2열부터 대각선으로 1
```

Out[61]:

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

- diag : 행렬의 대각선 값을 추출함 -> (i, i)의 값 추출

  (eye와 마찬가지로 k값을 지정하여 시작 index 변경이 가능하다)

In [62]:

```
matrix = np.arange(9).reshape(3, 3)
np.diag(matrix)      # matrix의 0행 0열, 1행 1열, 2행 2열 값 array로 추출
```

Out[62]:

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

In [63]:

```
np.diag(matrix, k = 1)      # matrix의 0행 1열, 1행 2열 값 array로 추출
```

Out[63]:

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

- random sampling : 데이터 분포에 따른 sampling으로 array 생성

In [64]:

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

Out[64]:

```
array([[0.73848779, 0.20831259, 0.89779925, 0.79441573, 0.22393552],
       [0.05808157, 0.35508479, 0.19378695, 0.12023014, 0.41333885]])
```

In [65]:

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

Out[65]:

```
array([[-1.95435709, -0.36496446,  0.87751876, -0.05824508, -1.30337831],
       [ 0.7538363 , -0.43297659, -0.08523809,  0.35386668, -0.71366359]])
```

5. operation functions

operation in array

In [66]:

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

Out[66]:

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

- sum : ndarray의 element들간의 합을 구함 (list의 sum 기능과 동일)

In [67]:

```
test_array.sum(dtype = np.float)      # test_array의 모든 element의 합
```

Out[67]:

```
55.0
```

In [68]:

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

Out[68]:

```
78
```

In [69]:

```
test_array
```

Out[69]:

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

In [70]:

```
test_array.sum(axis = 1), test_array.sum(axis = 0)      # 2차원 matrix에서는 axis가 1이면 행, 0이면 열
# 결과는 [0행, 1행, 2행]의 합, [0열, 1열, 2열, 3열]의 합
```

Out[70]:

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

In [71]:

```
third_order_tensor = np.array([test_array, test_array, test_array])      # 3차원 matrix 생성
third_order_tensor
```

Out[71]:

```
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]]])
```

3차원 matrix에서는 axis가 0이면 depth, 2이면 행, 1이면 열

In [72]:

```
third_order_tensor.sum(axis = 2)      # 행의 합
# [0층[0행, 1행, 2행]의 합,
#  1층[0행, 1행, 2행]의 합,
#  2층[0행, 1행, 2행]의 합]
```

Out[72]:

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

In [73]:

```
third_order_tensor.sum(axis = 1)      # 열의 합
# [0층[0열, 1열, 2열]의 합,
#  1층[0열, 1열, 2열]의 합,
#  2층[0열, 1열, 2열]의 합]
```

Out[73]:

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

In [74]:

```
third_order_tensor.sum(axis = 0)      # 층(depth)의 합
# [0행[0열, 1열, 2열]의 합,
#  1행[0열, 1열, 2열]의 합,
#  2행[0열, 1열, 2열]의 합]
```

Out[74]:

```
array([[ 3,  6,  9, 12],
```

```
       [15, 18, 21, 24],
       [27, 30, 33, 36]])
```

In [75]:

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

Out[75]:

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

- mean : ndarray의 element들간의 평균

In [76]:

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

Out[76]:

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

- std : ndarray의 element들간의 표준편차

In [77]:

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

Out[77]:

```
(3.452052529534663, array([3.26598632, 3.26598632, 3.26598632, 3.26598632]))
```

- exp : ndarray의 각 element의 지수 e*x
- sqrt : ndarray의 각 element의 제곱근

In [78]:

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

Out[78]:

```
(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

- numpyarray를 합치는 함수

In [79]:

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

Out[79]:

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

In [80]:

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

Out[80]:

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

In [81]:

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

Out[81]:

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

In [82]:

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

Out[82]:

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

In [83]:

```
a.tolist()      # array를 list로
```

Out[83]:

```
[[1, 2], [3, 4]]
```

6. array operations

- matrix내 element들간 같은 위치에 있는 값들끼리 연산

In [84]:

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

Out[84]:

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

In [85]:

```
test_a + test_a      # matrix의 덧셈
```

Out[85]:

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

In [86]:

```
test_a - test_a      # matrix의 뺄셈
```

Out[86]:

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

In [87]:

```
test_a * test_a      # matrix의 같은 위치에 있는 element 값의 곱셈
```

Out[87]:

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

In [88]:

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

Out[88]:

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

dot product

- matrix의 기본 연산 (행렬곱)
- dot 함수 사용

In [89]:

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

In [90]:

```
test_a
```

Out[90]:

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

In [91]:

```
test_b
```

Out[91]:

```
array([[ 7,  8],
       [ 9, 10],
       [11, 12]])
```

In [92]:

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

Out[92]:

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

In [93]:

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

Out[93]:

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

- transpose : 행렬의 행과 열의 위치를 반전시킴

  (transpose 또는 T attribute 사용)

  ex) 0행 1열의 값이 1행 0열의 위치로 바뀜 -> shape도 바뀜

  (i, j).transpose()가 (j, i)로 바뀐다

In [94]:

```
test_a.transpose()
```

Out[94]:

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

In [95]:

```
test_a.T
```

Out[95]:

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

In [96]:

```
test_a.T.dot(test_a)      # 3행 2열 matrix와 2행 3열 matrix의 행렬곱
```

Out[96]:

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

In [97]:

```
test_a.dot(test_a.T)      # 2행 3열 matrix와 3행 2열 matrix의 행렬곱
```

Out[97]:

```
array([[14, 32],
       [32, 77]])
```

broadcasting

- shape가 다른 배열간 연산을 지원하는 기능

In [98]:

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

In [99]:

```
test_matrix + scalar      # matrix의 모든 element에 scalar 넛셈
```

Out[99]:

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

In [100]:

```
test_matrix - scalar      # matrix의 모든 element에 scalar 빼셈
```

Out[100]:

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

In [101]:

```
test_matrix * 5      # matrix의 모든 element에 5 곱셈
```

Out[101]:

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

In [102]:

```
test_matrix / 5      # matrix의 모든 element에 5 나눗셈
```

Out[102]:

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

In [103]:

```
test_matrix // 0.2      # matrix의 모든 element에 0.2 나눈 몫
```

Out[103]:

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

In [104]:

```
test_matrix ** 2      # matrix의 모든 element에 2 제곱
```

Out[104]:

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

In [105]:

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

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]] [10 20 30]
```

Out[105]:

```
array([[11, 22, 33],
       [14, 25, 36],
       [17, 28, 39],
```

```
       [20, 31, 42]])
```

numpy performance

%timeit : 코드의 소요시간을 여러 번 측정해 좀 더 정확히 파악할 때 쓰임

In [106]:

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

iternation_max = 100
vector = list(range(iternation_max))
scalar = 2

%timeit scalar_vector_product(scalar, vector)
```

```
24.4 µs ± 5.43 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [107]:

```
%timeit [scalar * value for value in range(iternation_max)]
```

```
17.7 µs ± 1.33 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [108]:

```
%timeit np.arange(iternation_max) * scalar
```

```
3.59 µs ± 180 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

7. comparison

all & any

array의 데이터 전부(and) 또는 일부(or)가 조건에 만족하는지 결과 반환

- all : 모두가 조건에 만족한다면 true
- any : 하나라도 조건에 만족한다면 true

In [109]:

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

Out[109]:

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

In [110]:

```
a < 0
```

Out[110]:

```
array([False, False, False, False, False, False, False, False, False,
       False])
```

In [111]:

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

Out[111]:

```
(True, False)
```

In [112]:

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

Out[112]:

```
(False, True)
```

comparison operation

- numpy는 배열의 크기가 동일할 때 element간 비교의 결과를 Boolean type으로 반환함

In [113]:

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

Out[113]:

```
array([False,  True, False])
```

In [114]:

```
test_a == test_b
```

Out[114]:

```
array([False, False, False])
```

In [115]:

```
(test_a > test_b).any()      # 하나라도 true라면 true
```

Out[115]:

```
True
```

In [116]:

```
a = np.array([1, 3, 0], float)
np.logical_and(a > 0, a < 3)      # and condition
```

Out[116]:

```
array([ True, False, False])
```

In [117]:

```
b = np.array([True, False, True], bool)
np.logical_not(b)      # not condition
```

Out[117]:

```
array([False,  True, False])
```

In [118]:

```
c = np.array([False, True, False], bool)
np.logical_or(b, c)      # or condition
```

Out[118]:

```
array([ True,  True,  True])
```

In [119]:

```
np.where(a > 0, 3, 2)      # where(condition, True, False)
# -> a>0이면 true이므로 3, a<=0이면 false이므로 2
```

Out[119]:

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

In [120]:

```
a = np.arange(5, 15)
a
```

Out[120]:

```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [121]:

```
np.where(a > 10)
```

Out[121]:

```
(array([6, 7, 8, 9], dtype=int64),)
```

In [122]:

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

Out[122]:

```
array([False,  True, False])
```

In [123]:

```
np.isfinite(a)
```

Out[123]:

```
array([ True, False, False])
```

argmax & argmin

- array 내 최댓값 또는 최솟값의 index를 반환함

In [124]:

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

Out[124]:

```
(5, 0)
```

In [125]:

```
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)
```

Out[125]:

```
(array([3, 1, 1], dtype=int64), array([0, 0, 2, 2], dtype=int64))
```

8. boolean fancy index

boolean index

- numpy는 특정 조건에 따른 값을 배열 형태로 추출할 수 있음
- comparison operation 함수들도 모두 사용가능

In [126]:

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

Out[126]:

```
array([False,  True, False, False, False,  True,  True,  True])
```

In [127]:

```
test_array[test_array > 3]     # 조건이 true인 element만 추출
```

Out[127]:

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

In [128]:

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

Out[128]:

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

In [129]:

```
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
```

Out[129]:

```
array([[ True,  True,  True,  True, False,  True,  True,  True,  True],
       [ True,  True,  True, False, False, False,  True,  True,  True],
       [ True,  True,  True, False,  True, False,  True,  True,  True],
       [ True,  True, False, False,  True, False, False,  True,  True],
       [ True,  True, False,  True,  True,  True, False,  True,  True],
       [ True, False, False,  True,  True,  True, False, False,  True],
       [ True, False,  True,  True,  True,  True,  True, False,  True],
       [ True, False,  True,  True,  True,  True,  True, False,  True],
       [ True, False,  True,  True,  True,  True,  True, False,  True],
       [ True, False, False, False, False, False, False, False,  True],
       [ True, False,  True,  True,  True,  True,  True, False,  True],
```

```
       [ True, False,  True,  True,  True,  True,  True, False,  True],
       [ True, False,  True,  True,  True,  True,  True, False,  True],
       [ True, False,  True,  True,  True,  True,  True, False,  True],
       [ True, False,  True,  True,  True,  True,  True, False,  True]])
```

In [130]:

```
B.astype(np.int)
```

Out[130]:

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

fancy index

- numpy는 array를 index value로 사용해서 값을 추출하는 방법

In [131]:

```
a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int)      # 반드시 integer로 선언
```

In [132]:

```
a[a > 4]
```

Out[132]:

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

In [133]:

```
a[b]      # bracket index, b배열의 값을 index로 하여 a의 값들을 추출함
```

Out[133]:

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

In [134]:

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

Out[134]:

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

In [135]:

```
a = np.array([[1, 4], [9, 16]], float)      # matrix 형태의 데이터도 가능
b = np.array([0, 0, 1, 1, 0], int)
c = np.array([0, 1, 1, 1, 1], int)
a[b, c]      # b를 행 index, c를 열 index로 변환하여 표시함
```

Out[135]:

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

In [136]:

```
a = np.array([[1, 4], [9, 16]], float)
a[b]
```

Out[136]:

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

9. numpy data

loadtxt & savetxt

- text type의 데이터를 읽고 저장하는 기능

In [137]:

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

Out[137]:

```
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.]])
```

In [138]:

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

Out[138]:

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

In [139]:

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

numpy object - npy

- numpy object(pickle)형태로 데이터를 저장하고 불러옴
- binary 파일 형태로 저장함

In [140]:

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

In [141]:

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

Out[141]:

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

In [142]:

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

In [143]:

```
np.savetxt('a_npy.txt', a)
```

In [144]:

```
a = np.loadtxt('a_npy.txt', dtype = int)
a
```

Out[144]:

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

In [145]:

```
a = np.loadtxt('a_npy.txt', dtype = float)
a
```

Out[145]:

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

In [146]:

```
np.savetxt('a_npy.csv', a)
```

In [147]:

```
a = np.loadtxt('a_npy.csv', dtype = int)
a
```

Out[147]:

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

In [148]:

```
b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], int)
b
```

Out[148]:

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

In [149]:

```
np.save('bnpy', b)
```

In [150]:

```
b = np.load('bnpy.npy')
```

In [151]:

```
b
```

Out[151]:

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