단원 03
과학용 컴퓨팅 패키지
numpy

DONGYANG MIRAE UNIVERSITY
Dept. of Artificial Intelligence

인공지능소프트웨어학과

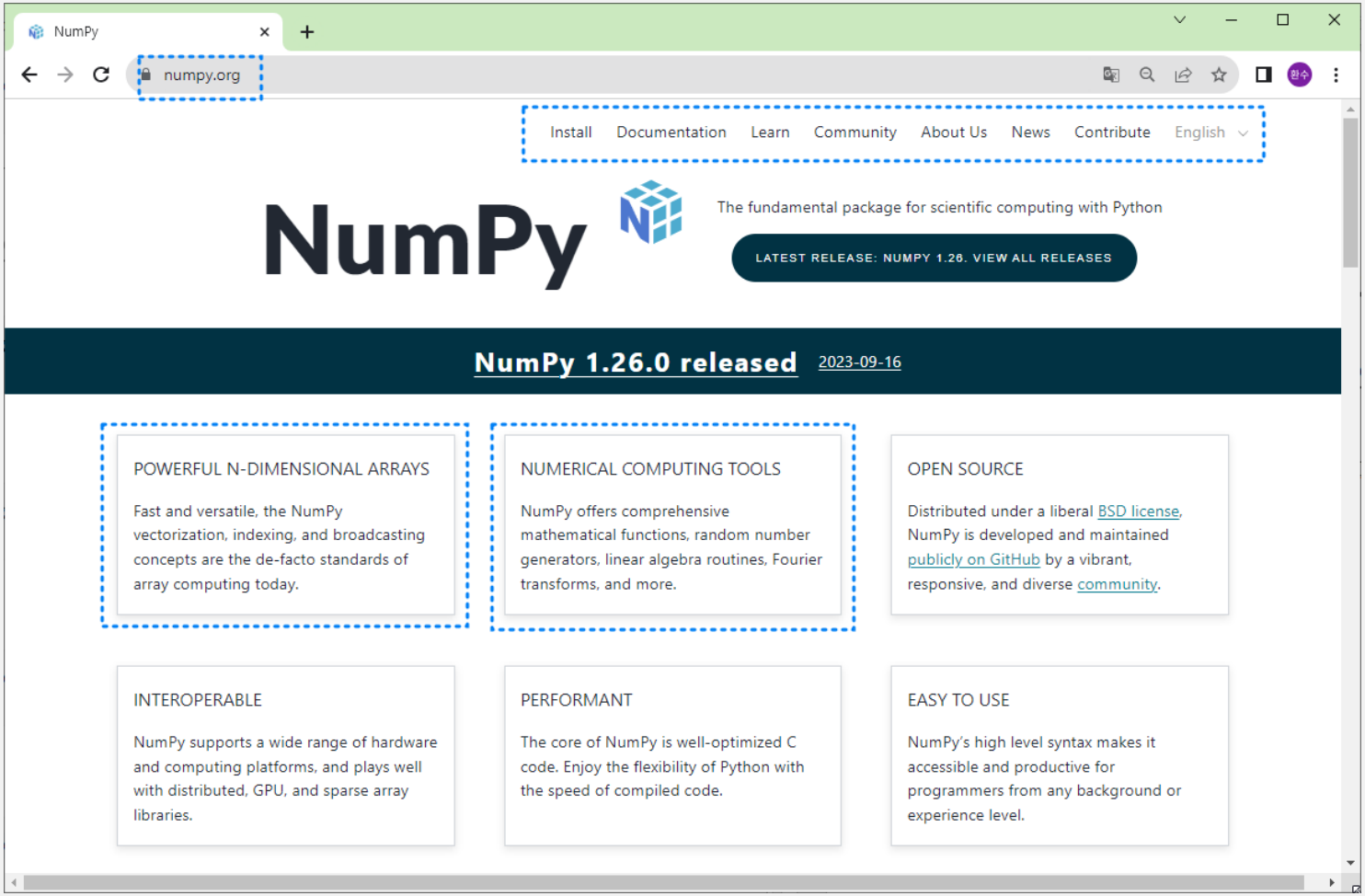
강환수 교수

3.1 패키지 numpy 개요와 자료형 ndarray

-

과학용 컴퓨팅 패키지 numpy 라이브러리

- 수치 계산을 빠르게 수행하는 고성능 배열과 행렬 연산을 위한 과학용 컴퓨팅 라이브러리



Numpy 개요

- 과학 및 엔지니어링 응용 프로그램에서 매우 널리 사용
  - 데이터 분석, 기계 학습, 신호 처리 등 다양한 분야에서 핵심 라이브러리
  - 데이터 과학 및 공학 분야에서 많은 다른 라이브러리와 도구와의 통합을 촉진
  - 수치 계산 및 데이터 분석의 강력한 플랫폼으로 만들어주는 핵심 라이브러리 중 하나
- numpy의 주요 특징과 기능
  - 다차원 배열(N-dimensional arrays)
    - numpy의 핵심 기능은 다차원 배열 또는 ndarray
    - 동일한 데이터 타입의 요소들로 이루어진 다차원 배열
  - 브로드캐스팅(Broadcasting)
    - 서로 다른 모양이면서 확장이 가능한 배열 간에도 연산이 가능하도록 하는 기능을 제공
    - 배열 간의 크기나 차원이 다를 때, 자동으로 배열을 확장하여 연산이 가능
  - 유니버설 함수(Universal Functions)
  - 선형 대수(Linear Algebra):
    - 행렬의 곱셈, 행렬 분해, 역행렬 등 다양한 선형 대수 연산이 가능
  - 난수 생성(Random)
  - 인덱싱과 슬라이싱

다차원 배열 ndarray

- **ndarray**
  - numpy 다루는 대표적인 자료형
  - nd
    - 다차원(multi dimension)을 의미
  - 고정된 크기(fixed-size)의 동일한 (homogeneous) 자료형의 항목들로 구성
- **함수 numpy.ndarray(size, dtype)**
  - 값은 임의의 값

```
import numpy as np

a = np.ndarray((2, 3))
a
```

✓ 0.0s

```
array([[6.23042070e-307, 4.67296746e-307, 1.69121096e-306],
       [6.23043429e-307, 2.22526399e-307, 2.05837348e-312]])
```

```
import numpy as np

a = np.ndarray((3, 2), dtype=int)
a
```

✓ 0.0s

```
array([[763586208,       708],
       [        0,         0],
       [   131074,       708]])
```

```
a = np.ndarray((3, 2), int)
a
```

✓ 0.0s

```
array([[     1113,         0],
       [785929112,       708],
       [      123,         0]])
```

numpy의 다양한 기본 자료형

- **numpy는 표준 파이썬보다 다양한 기본 자료형을 제공**

[표 3-1] numpy 자료형

자료형	설명
bool	불리언 값 (참(True) 또는 거짓(False))
int	기본 정수형 (일반적으로 int32 또는 int64)
float	부동 소수점 수 (일반적으로 float32 또는 float64)
complex	복소수 (일반적으로 complex64 또는 complex128)
int8, int16, int32, int64	8비트, 16비트, 32비트, 64비트 정수
uint8, uint16, uint32, uint64	부호 없는 8비트, 16비트, 32비트, 64비트 정수
float16, float32, float64	16비트, 32비트, 64비트 부동 소수점 수
complex64, complex128	64비트, 128비트 복소수

ndarray(다차원 배열)의 주요 속성

속성	설명
ndarray.shape	배열의 차원을 나타내는 튜플. 각 차원의 크기를 표현
ndarray.ndim	배열의 차원 수
ndarray.size	배열의 총 요소 수
ndarray.dtype	배열 요소의 데이터 형식
ndarray.itemsize	배열의 각 요소의 크기(바이트)
ndarray.nbytes	배열의 전체 크기(바이트)
ndarray.data	배열 요소를 실제로 포함하고 있는 버퍼의 시작 위치

내장함수 array()

- 다양한 속성 확인

```python
import numpy as np

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

✓ 0.0s Python

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

```python
type(a)
```

✓ 0.0s Python

```
numpy.ndarray
```

```python
print(a.shape)
print(a.ndim)
print(a.dtype)
```

✓ 0.0s Python

```
(3,)
1
int32
```

```python
print(a.size)
print(a.itemsize)
print(a.nbytes)
```

✓ 0.0s Python

```
3
4
12
```

ndarray 특징

모든 요소(elements)가 단일한 자료형으로 구성

- **하나라도 항목이 문자열이면 모든 항목이 동일하게 문자열 자료형인 '<U32'으로 지정**
  - numpy의 자료형은 데이터의 저장 방식과 크기를 지정하는 데 사용
  - 배열의 요소는 이러한 자료형 중 하나만을 가짐
    - **numpy에서 배열의 빠른 계산 처리를 위한 방법**

```
np.array([1, 'py', 3.14])
array(['1', 'py', '3.14'], dtype='<U32')
```

문자열 자료형

```python
np.array([1, 'py', 3.14])
```

✓ 0.0s Python

```
array(['1', 'py', '3.14'], dtype='<U32')
```

```python
import numpy as np

unicode_array = np.array(['가나다', '라마바', '사아자'], dtype='<U32')
unicode_array
```

✓ 0.0s Python

```
array(['가나다', '라마바', '사아자'], dtype='<U32')
```

```python
unicode_array = np.array(['가나다라마'*7, '라마바', '사아자'], dtype='<U32')
unicode_array
```

✓ 0.0s Python

```
array(['가나다라마가나다라마가나다라마가나다라마가나다라마가나다라마가나', '라마바', '사아자'], dtype='<U32')
```

3.2 내장함수 arange()와 linspace()

-

numpy.arange()

실수도 가능

```
numpy.arange([start=0,]stop,[step=1,]dtype=None,*,like=None)
```

주어진 간격 내에서 균등한 간격의 값을 반환하며, 다양한 수의 위치 인수를 사용하여 호출하고, 정수 인수의 경우 함수는 파이썬 내장함수 range()와 거의 동일하지만 range 인스턴스가 아닌 ndarray를 반환

- arange(stop): 반 개방 구간 [0, stop) (즉, 시작은 포함하고 정지는 제외한 구간) 내에서 값이 생성되며, step은 1
- arange(start, stop): 반 개방 구간 [start, stop) 내에서 값이 생성되며, step은 1
- arange(start, stop, step): 반 개방 간격 [start, stop) 내에서 생성되며 값 사이의 간격은 step으로 지정

```python
import numpy as np

print(np.arange(3))
print(np.arange(3, 7))
print(np.arange(3, 7, 2))
```

Python

```
[0 1 2]
[3 4 5 6]
[3 5]
```

```python
import numpy as np

print(np.arange(2.0))
print(np.arange(2.0, 6.5))
print(np.arange(2.0, 6.5, 0.8))
```

Python

```
[0. 1.]
[2. 3. 4. 5. 6.]
[2.  2.8 3.6 4.4 5.2 6. ]
```

배열의 모양을 바꾸는 reshape()

- 모양은 원소의 수가 같으나 차수가 다른 모양은 모두 허용
  - 차수에 따라 원소의 수가 맞지 않으면 오류가 발생
  - 모양이 (3, 5)이면 원소가 15개인데 원래 배열 원소는 12개이므로 오류가 발생

```python
import numpy as np

a = np.arange(12).reshape(3, 4)
a
```

Python

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```python
np.reshape(a, (2, 6))
```

Python

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

```python
np.reshape(a, (3, 5))
```

Python

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
d:\(1 Drive)\m365\OneDrive - 동양미래대학\[학교 교과목 강의]\2023 2학기 데이터분석입문\[교재개발 작성] 1023~1223\c
----> 1 np.reshape(a, (3, 5))
```

함수 linspace(start, stop, num)

일정한 간격의 값으로 배열 생성

- 동일한 길이의 하위 간격으로 분할하는 수가 나열된 1차원 배열을 반환
  - arrays with regularly-spaced
  - 전체 구간은 [start, stop]으로 기본적으로 양쪽을 모두 포함
  - 끝점인 stop은 인자 endpoint=False로 제외 가능
  - 일정한 간격을 구성하는 값인 배열의 수를 매개변수 num에 지정하며 기본값은 50
- numpy.linspace(start,stop,num=50, endpoint=True, retstep=False, dtype=None, axis=0)
  - 지정된 간격에 걸쳐 균등한 간격의 숫자를 반환
    - 간격 [start, stop]에 걸쳐 계산된 균일한 간격의 샘플 수를 반환
    - 간격의 끝점은 endpoint로 선택적으로 제외될 수 있음
  - • start: array_like
    - 시퀀스의 시작 값
  - • stop: array_like
    - 시퀀스의 마지막 값으로 endpoint=False이면 제외되며. 구간의 크기도 변경됨
  - • num: int, optional
    - 생성되는 샘플 수로 기본(default)은 50
  - • endpoint: bool, optional
    - True이면 stop이 마지막 값이 되며 False이면 제외됨
  - • retstep: bool, optional
    - True이면 (samples, step)가 반환되며 아니면 samples만 반환되고 step은 샘플 수 사이의 간격을 말함

함수 linspace()

기본은 50개

```python
np.linspace(2.0, 3.0)
```

```
array([2.        , 2.02040816, 2.04081633, 2.06122449, 2.08163265,
       2.10204082, 2.12244898, 2.14285714, 2.16326531, 2.18367347,
       2.20408163, 2.2244898 , 2.24489796, 2.26530612, 2.28571429,
       2.30612245, 2.32653061, 2.34693878, 2.36734694, 2.3877551 ,
       2.40816327, 2.42857143, 2.44897959, 2.46938776, 2.48979592,
       2.51020408, 2.53061224, 2.55102041, 2.57142857, 2.59183673,
       2.6122449 , 2.63265306, 2.65306122, 2.67346939, 2.69387755,
       2.71428571, 2.73469388, 2.75510204, 2.7755102 , 2.79591837,
       2.81632653, 2.83673469, 2.85714286, 2.87755102, 2.89795918,
       2.91836735, 2.93877551, 2.95918367, 2.97959184, 3.        ])
```

```python
np.linspace(2.0, 3.0, retstep=True)
```

```
(array([2.        , 2.02040816, 2.04081633, 2.06122449, 2.08163265,
        2.10204082, 2.12244898, 2.14285714, 2.16326531, 2.18367347,
        2.20408163, 2.2244898 , 2.24489796, 2.26530612, 2.28571429,
        2.30612245, 2.32653061, 2.34693878, 2.36734694, 2.3877551 ,
        2.40816327, 2.42857143, 2.44897959, 2.46938776, 2.48979592,
        2.51020408, 2.53061224, 2.55102041, 2.57142857, 2.59183673,
        2.6122449 , 2.63265306, 2.65306122, 2.67346939, 2.69387755,
        2.71428571, 2.73469388, 2.75510204, 2.7755102 , 2.79591837,
        2.81632653, 2.83673469, 2.85714286, 2.87755102, 2.89795918,
        2.91836735, 2.93877551, 2.95918367, 2.97959184, 3.        ]),
 0.02040816326530612)
```

```python
print( (3-2) / (50-1) )
```

```
0.02040816326530612
```

함수 linspace()

```python
np.linspace(2.0, 3.0, endpoint=False, retstep=True)
```

```
(array([2.  , 2.02, 2.04, 2.06, 2.08, 2.1 , 2.12, 2.14, 2.16, 2.18, 2.2 ,
        2.22, 2.24, 2.26, 2.28, 2.3 , 2.32, 2.34, 2.36, 2.38, 2.4 , 2.42,
        2.44, 2.46, 2.48, 2.5 , 2.52, 2.54, 2.56, 2.58, 2.6 , 2.62, 2.64,
        2.66, 2.68, 2.7 , 2.72, 2.74, 2.76, 2.78, 2.8 , 2.82, 2.84, 2.86,
        2.88, 2.9 , 2.92, 2.94, 2.96, 2.98]),
 0.02)
```

```python
print( (3-2) / 50 )
```

```
0.02
```

```python
np.linspace(2.0, 3.0, num=5, retstep=True)
```

```
(array([2.  , 2.25, 2.5 , 2.75, 3.  ]), 0.25)
```

```python
np.linspace(2.0, 3.0, num=5, endpoint=False, retstep=True)
```

```
(array([2. , 2.2, 2.4, 2.6, 2.8]), 0.2)
```

3.3 내장함수 zeros()와 ones(), empty()

-

내장함수 np.zeros()

주어지는 shape 형태로 0.0이 채워진 ndarray를 반환하는 함수

- **주어지는 shape 형태로 0.0이 채워진 ndarray를 반환하는 함수**

```
import numpy as np

a = np.zeros(5)
a
array([0., 0., 0., 0., 0.])

a.dtype
dtype('float64')
```

```
np.zeros((5,), dtype=int)
```

Python

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

```
b = np.zeros((2, 1))
b
```

Python

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

원소가 모두 1.0인 배열을 생성하는 ones()

zeros()처럼 모두 실수 1.0인 배열을 반환하는 함수

```python
np.ones((2, 4))
```

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

```python
np.ones((2, 4), dtype=int)
```

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

numpy.eye()

대각선 원소가 모두 1인 2차원 배열을 반환하는 함수

- (수학에서) 단위행렬 생성 함수

$$I_1 = [1],\ I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},\ \cdots,\ I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

- 함수 eye(m, n, k) 호출
  - m행과 n열의 2차원 배열을 반환
  - k=+n으로 양수
    - 오른쪽, 위쪽 방향으로 n번 이동된 위치의 대각선이 1이 저장
  - k=-n으로 음수
    - 왼쪽, 아래쪽 방향으로 n번 이동된 위치의 대각선이 1이 저장

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

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

```python
np.eye(2, dtype=int)
```

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

```python
np.eye(3, 4)
```

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

```python
a = np.eye(3, 4, 1)
a
```

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

```python
b = np.eye(3, 4, -1)
b
```

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

numpy.full(shape, fill_value)

fill_value로 shape 모양의 배열을 반환하는 함수

- 값 fill_value가 없으면 오류 발생

```
np.full([2, 3], 10)
```
✓ 0.0s Python

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

```
np.full([2, 3], (1, 2, 3))
```
Python

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

```
np.full([2, 3], (1, 2))
```
Python

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[83], line 1
----> 1 np.full([2, 3], (1, 2))

File d:\ProgramData\anaconda3_240601\Lib\site-packages\numpy\core\numeric.py:330, i
    328     dtype = fill_value.dtype
    329 a = empty(shape, dtype, order)
--> 330 multiarray.copyto(a, fill_value, casting='unsafe')
    331 return a

ValueError: could not broadcast input array from shape (2,) into shape (2,3)
```

```
np.full([3, 4], range(4))
```
Python

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

```
np.full([4, 3], fill_value=np.arange(4).reshape(4, 1))
```
Python

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

지정한 모양의 배열을 생성하는 numpy.empty(shape)

값은 메모리 값 지정

- **원소 값은 예측할 수 없음**

```python
np.empty(3)
```

```
array([1.11257149e-311, 0.00000000e+000, 1.31544754e-259])
```

```python
np.empty([2, 2])
```

```
array([[2.12199579e-314, 1.11192579e-311],
       [3.99205042e-321, 3.79442416e-321]])
```

```python
np.empty([2, 3], dtype=int)
```

```
array([[1306894352,        524,          0],
       [         0,          1,  170929712]])
```

원소값이 0인 주어진 형태의 배열을 생성하는 zeros_like(a)

인자인 a 형태의 배열로 원소 값은 모두 0인 배열을 반환

```
import numpy as np

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

np.zeros_like(a)
array([[0, 0, 0, 0],
       [0, 0, 0, 0]])
```

원소값이 1인 주어진 형태의 배열을 생성하는 ones_like()

zeros()처럼 모두 1인 배열을 반환하는 함수

```
np.ones_like(a)
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

원소값이 주어진 값이며 주어진 형태의 배열을 생성하는 full_like()

주어진 값으로 배열을 반환하는 함수

```
np.full_like(a, 5)
array([[5, 5, 5, 5],
       [5, 5, 5, 5]])
```

3.4 배열 연산

-

배열과 스칼라와의 연산

- 모든 원소에 연산을 수행한 같은 모양의 배열을 반환

```python
import numpy as np

a = np.arange(3)
a
```

✓ 0.2s Python

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

```python
print(a + 2)
print(a - 2)
print(a * 2)
print(a / 2)
```

Python

```
[2 3 4]
[-2 -1  0]
[0 2 4]
[0.  0.5 1. ]
```

배열과 배열의 연산

기본적으로 모양이 동일해야 함

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

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

```python
b = np.full_like(a, [1, 2, 3])
b
```

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

```python
print(a + b)
print(a - b)
print(a * b)
print(a / b)
```

```
[[2 4 6]
 [5 7 9]]
[[0 0 0]
 [3 3 3]]
[[ 1  4  9]
 [ 4 10 18]]
[[1.  1.  1. ]
 [4.  2.5 2. ]]
```

복합 대입 연산자도 지원

- 배열 연산 +=, -=, *=, /=
- 변수 a
  - dtype이 int
- 변수 b
  - dtype이 float

```python
a = np.arange(4)
a
```

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

```python
b = np.arange(0.5, 4, 1)
b
```

```
array([0.5, 1.5, 2.5, 3.5])
```

```python
a += b
```

```
---------------------------------------------------------------------------
UFuncTypeError                            Traceback (most recent call last)
Cell In[107], line 1
----> 1 a += b

UFuncTypeError: Cannot cast ufunc 'add' output from dtype('float64') to dtype('int3
```

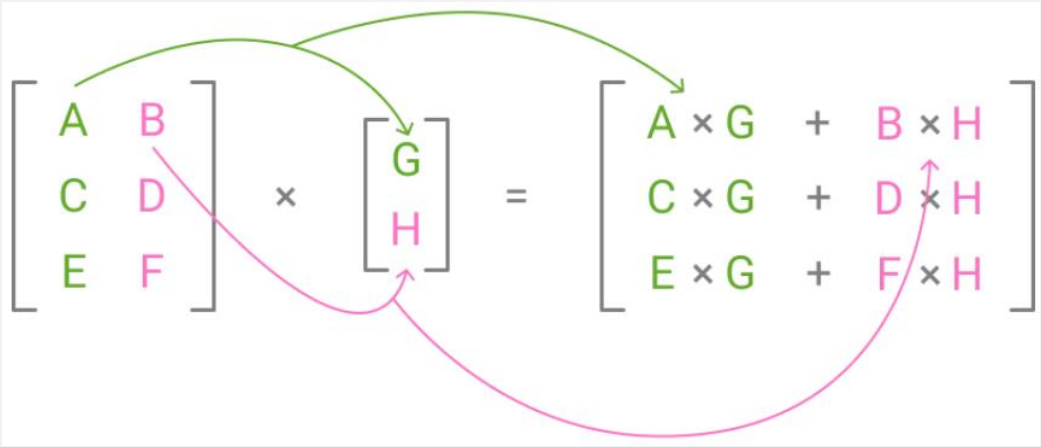
```python
b += a
b
```

```
array([0.5, 2.5, 4.5, 6.5])
```

2차원 배열의 곱 연산자 @

앞 배열의 열 수와 뒤 배열의 행 수가 같아야 수행 가능

- (3, 2) @ (2, 1) 결과
  - (3, 1)
    - 3행 1열의 2차원 배열



x

✓ 0.0s

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

y

✓ 0.0s

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

x @ y

✓ 0.0s

```
array([[3],
       [4],
       [5]])
```

다양한 배열의 곱 연산자

a @ b, a.shape[1] == b.shape[0] 이 성립해야 계산 가능(앞 배열의 열 수와 뒤 배열의 행 수가 같아야 수행 가능)

- 배열의 곱은 교환법칙이 성립 불가능
  - a
    - 모양 (2, 3)
  - B
    - 모양 (3, 2)
  - a와 b의 a.dot(b) 연산
    - (2, 3) @ (3, 2)이므로
    - 곱의 결과 (2, 2) 모양
  - a와 b의 b.dot(a) 연산
    - (3, 2) @ (2, 3)이므로
    - 곱의 결과 (3, 3) 모양
- numpy.matmul(a, b)
  - a @ b 동일

```
a = np.full((2, 3), [1, 2, 3])
a
```

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

```
b = np.full((3, 2), [2, 1])
b
```

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

```
a @ b
```

```
array([[12,  6],
       [12,  6]])
```

```
a.dot(b)
```

```
array([[12,  6],
       [12,  6]])
```

```
b.dot(a)
```

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

```
import numpy as np
print(np.matmul(a, b))
print(np.matmul(b, a))
```

```
[[12  6]
 [12  6]]
[[3 6 9]
 [3 6 9]
 [3 6 9]]
```

배열 함수

a.sum()

- **a.sum()**
  - 모든 원소의 합
- **A.sum(axis=0)**
  - 열 합 배열
- **a.sum(axis=1)**
  - 열 합 배열

```python
a = np.array([[10, 20], [5, 8], [1, 2]])
a
```

```
array([[10, 20],
       [ 5,  8],
       [ 1,  2]])
```

```python
a.sum()
```

```
46
```

```python
a.sum(axis=0)
```

```
array([16, 30])
```

```python
a.sum(axis=1)
```

```
array([30, 13,  3])
```

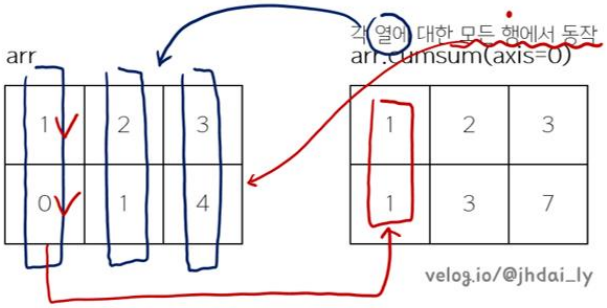
```python
import numpy as np

print(np.sum(a))
print(np.sum(a, axis=0))
print(np.sum(a, axis=1))
```

```
46
[16 30]
[30 13  3]
```

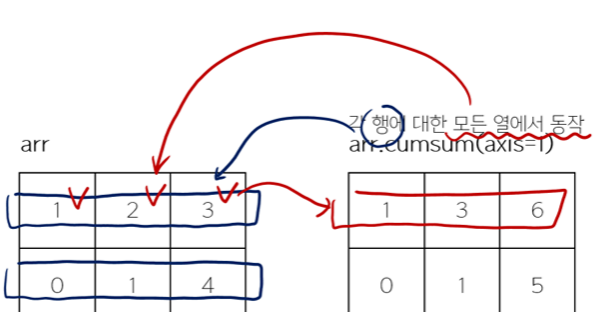
배열 함수

a.cumsum() cumulate : 쌓아 올리다, 누적하다.

- **a.cumsum()**
  - 원소를 누적한 1차원 배열
- **a.cumsum(axis=0)**



- **a.cumsum(axis=1)**




```
a = np.array([[3, 5, 8], [1, 2, 3]])
a
```
✓ 0.0s Python

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

```
# 일차원으로 펼쳐서 계산
a.cumsum()
```
✓ 0.2s Python

```
array([ 3,  8, 16, 17, 19, 22])
```

```
a.cumsum(axis=0) # 시험예상, 행을 따라 세로로 합산
```
✓ 0.0s Python

```
array([[ 3,  5,  8],
       [ 4,  7, 11]])
```

```
a.cumsum(axis=1) # 시험예상, 열을 따라 가로로 합산
```
✓ 0.0s Python

```
array([[ 3,  8, 16],
       [ 1,  3,  6]])
```

범용 함수

다양한 수학 및 통계적 연산을 빠르게 수행할 수 있도록 도와주는 핵심적인 기능 중 하나

- **범용 함수(universal functions, 간단히 ufunc)를 제공**
  - 배열에서 요소별로 작동하여 배열을 출력으로 생성

```
import numpy as np

a = np.array([1, 4, 9])
np.sqrt(a)
array([1., 2., 3.])
```

[표 3-1] numpy의 범용 함수

함수	설명	예제
np.add(x1, x2)	요소별 덧셈	np.add([1, 2], [3, 4]) → [4, 6]
np.subtract(x1, x2)	요소별 뺄셈	np.subtract([3, 4], [1, 2]) → [2, 2]
np.multiply(x1, x2)	요소별 곱셈	np.multiply([1, 2], [3, 4]) → [3, 8]
np.divide(x1, x2)	요소별 나눗셈	np.divide([3, 4], [1, 2]) → [3., 2.]
np.power(x1, x2)	요소별 거듭제곱	np.power([1, 2], [3, 4]) → [1, 16]
np.sqrt(x)	요소별 제곱근	np.sqrt([1, 4, 9]) → [1., 2., 3.]
np.sin(x)	요소별 삼각 함수(sin)	np.sin([0, np.pi/2, np.pi]) → [0., 1., 0.]
np.cos(x)	요소별 삼각 함수(cos)	np.cos([0, np.pi/2, np.pi]) → [1., 0., −1.]
np.exp(x)	요소별 지수 함수(exp)	np.exp([1, 2, 3]) → [2.71828183, 7.3890561, 20.08553692]
np.log(x)	요소별 자연 로그 함수(log)	np.log([1, np.e, np.e**2]) → [0., 1., 2.]

다양한 이항 연산 범용 함수

```
import numpy as np

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

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

```
b = np.full_like(a, 2)
b
```

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

```
np.add(a, b)
```

```
array([ 3,  6, 11])
```

```
np.divide(a, b)
```

```
array([0.5, 2. , 4.5])
```

```
np.power(a, b)
```

```
array([ 1, 16, 81])
```

3.5 배열 슬라이싱과 형태 수정

-

1차원 배열의 첨자와 슬라이싱

- a[i]
  - 첨자
- a[m:n]
- a[m:n:s]
  - 슬라이싱

```python
import numpy as np

a = np.power(np.arange(10), 2)
a
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81], dtype=int32)
```

```python
print(a[3])
print(a[-2])
```

```
9
64
```

```python
print(a[1:4])
print(a[-5:-1])
```

```
[1 4 9]
[25 36 49 64]
```

```python
print(a[8:2:-1])
print(a[-1:-7:-1])
```

```
[64 49 36 25 16  9]
[81 64 49 36 25 16]
```

역순과 대입

```python
a[::-1]
```

```
array([81, 64, 49, 36, 25, 16,  9,  4,  1,  0], dtype=int32)
```

```python
a[::-2]
```

```
array([81, 49, 25,  9,  1], dtype=int32)
```

```python
a[2:5] = -100
a
```

```
array([   0,    1, -100, -100, -100,   25,   36,   49,   64,   81],
      dtype=int32)
```

```python
a[2:5] = [-5, -5, -5]
a
```

```
array([ 0,  1, -5, -5, -5, 25, 36, 49, 64, 81], dtype=int32)
```

대입될 원소 수가
3개 이어야 함

```python
a[2:5] = [-5, -5]
a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
d:\(1 Drive)\m365\OneDrive - 동양미래대학\[학교 교과목 강의]\2023 2학기 데이터분석입문\
----> 1 a[2:5] = [-5, -5]
      2 a

ValueError: could not broadcast input array from shape (2,) into shape (3,)
```

```python
a[2:5] = -10
a
```

```
array([  0,   1, -10, -10, -10,  25,  36,  49,  64,  81], dtype=int32)
```

2차원 배열의 첨자와 슬라이싱

- **행 참조**
  - 다음 모두 동일 기능
    - x[row]
    - x[row,]
    - x[row, :]

```python
a = np.arange(30).reshape(5, 6)
a
```

```
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]])
```

```python
a[3]
```

```
array([18, 19, 20, 21, 22, 23])
```

```python
a[-1]
```

```
array([24, 25, 26, 27, 28, 29])
```

```python
a[-2, :]
```

```
array([18, 19, 20, 21, 22, 23])
```

```python
a[-2, ]
```

```
array([18, 19, 20, 21, 22, 23])
```

2차원 배열의 첨자와 슬라이싱

- **행 반환**
  - x[row], x[row,], x[row, :]
- **열 반환**
  - x[:, col]
  - x[, col]: 오류
- **원소 반환**
  - x[row, col]

비우면 오류

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

```
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]])
```

```
a[2:4]
```
Python

```
array([[12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

```
a[1:5:2]
```
Python

```
array([[ 6,  7,  8,  9, 10, 11],
       [18, 19, 20, 21, 22, 23]])
```

```
a[, 1]
```
Python

```
  Cell In[157],   line 1
    a[, 1]
      ^
SyntaxError: invalid syntax
```

```
a[:, 1]
```
Python

```
array([ 1,  7, 13, 19, 25])
```

```
a[:, 1:6]
```
Python

```
array([[ 1,  2,  3,  4,  5],
       [ 7,  8,  9, 10, 11],
       [13, 14, 15, 16, 17],
       [19, 20, 21, 22, 23],
       [25, 26, 27, 28, 29]])
```

2차원 배열의 첨자와 슬라이싱

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

```
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]])
```

```
a[:, ::2]
```

```
array([[ 0,  2,  4],
       [ 6,  8, 10],
       [12, 14, 16],
       [18, 20, 22],
       [24, 26, 28]])
```

```
a[2, 4]
```

```
16
```

```
a[2][4]
```

```
16
```

```
a[1:3, 2:5]
```

```
array([[ 8,  9, 10],
       [14, 15, 16]])
```

```
a[:, :]
```

```
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]])
```

```
a[:, ]
```

```
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]])
```

```
a[:]
```

```
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]])
```

```
a[::-1, ::-1]
```

```
array([[29, 28, 27, 26, 25, 24],
       [23, 22, 21, 20, 19, 18],
       [17, 16, 15, 14, 13, 12],
       [11, 10,  9,  8,  7,  6],
       [ 5,  4,  3,  2,  1,  0]])
```

```
# 마지막 행과 열을 제외한 배열 반환
a[:-1, :-1]
```

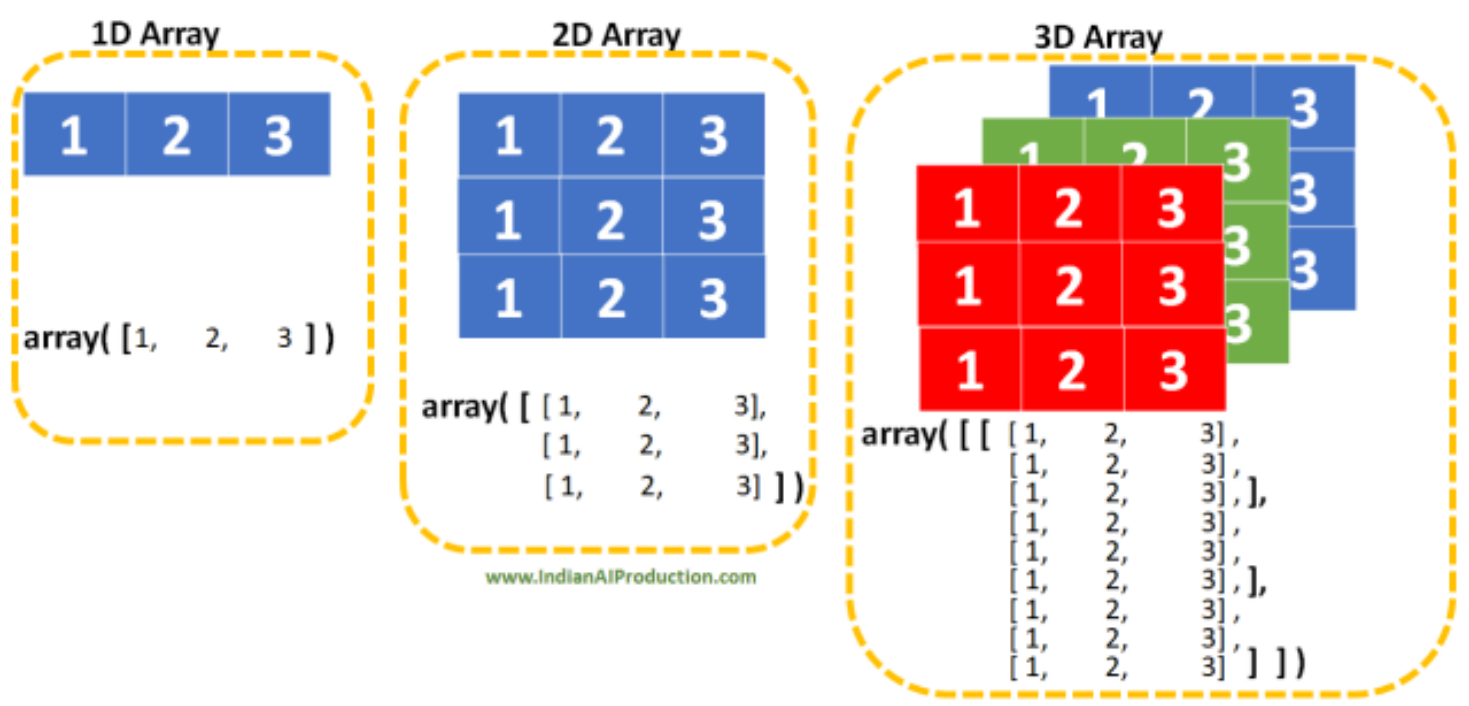
```
array([[ 0,  1,  2,  3,  4],
       [ 6,  7,  8,  9, 10],
       [12, 13, 14, 15, 16],
       [18, 19, 20, 21, 22]])
```

행과 열을 모두 역순으로 바꾼 배열

행과 열에서 마지막을 제거한 배열

3차원 배열의 이해

- **모양 (3, 3, 3)**
  - 3행 3열의 2차원 배열이 3개 모임



https://indianaiproduction.com/python-numpy-array/

3차원 배열의 첨자와 슬라이싱

3차원 배열 (2, 3, 4)는 3행 4열인 2차원 배열 2개로 구성된 배열로 이해

```
a = np.arange(24).reshape(2, 3, 4)
a
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]]])
```

- **축 axis = 0 | 1 | 2 의 이해**

Axis 2의 첨자 번호

Axis 0의 첨자 번호

Axis 1의 첨자 번호

[표 3-2] 3차원 배열의 이해

axis=0	3차원 배열 원소				
	axis=2	0	1	2	3
	axis=1				
0	0	0	1	2	3
	1	4	5	6	7
	2	8	9	10	11
	axis=2	0	1	2	3
	axis=1				
1	0	12	13	14	15
	1	16	17	18	19
	2	20	21	22	23

3차원 배열의 다양한 참조

```
a = np.arange(24).reshape(2, 3, 4)
a
```

```
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]]])
```

- a[1]
  - axis=0
  - 첨자 1
    - 두 번째 (3, 4) 배열
- a[:, 1]
  - axis=0인 모든 배열
    - axis=1의 첨자 1인 (2, 4) 모양 2차원 배열이 반환
- a[:, 1, 1]
  - axis=0인 모든 배열
  - 위치 (1, 1)
    - (1, 2) 모양 2차원 배열이 반환
- a[:, :, 1]
  - 마지막 축 axis=2의 첨자 1
  - (2, 3) 모양 2차원 배열이 반환

```
a[1]
```
Python

```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

```
a[:, 1]
```
Python

```
array([[ 4,  5,  6,  7],
       [16, 17, 18, 19]])
```

```
a[:, 1, 1]
```
Python

```
array([ 5, 17])
```

```
a[:, :, 1]
```
Python

```
array([[ 1,  5,  9],
       [13, 17, 21]])
```

3차원 배열의 다양한 참조

```
a = np.arange(24).reshape(2, 3, 4)
a
```

```
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]]])
```

- a[..., 1]
  - 배열에서 ...
    - 해당되는 축(axis)이 모두 : 인 것을 의미
  - 즉, a[..., 1]
    - a[:, :, 1] 와 동일
- a[1, ..., 1]
  - axis=0의 첨자 1
  - axis=2의 첨자 1인 (3,) 모양 1 차원 배열이 반환
- a[0, ...]
  - 즉, a[0, ...]은 a[0, :, :]와 동일
  - 첫 번째 2차원 배열
- a[0]
  - a[0, ...]와 a[0, :, :]와 동일
  - 위와 동일

```
a[..., 1]
```
Python

```
array([[ 1,  5,  9],
       [13, 17, 21]])
```

```
a[1, ..., 1]
```
Python

```
array([13, 17, 21])
```

```
a[0, ...]
```
Python

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
a[0]
```
Python

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

1차원 배열로 만드는 ndarray.ravel()

- 행렬인 a에서 a.ravel()의 반환 값은 (a.size,) 모양의 1차원 배열

```python
a = np.arange(12).reshape(2, 3, 2)
a
```

```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
```

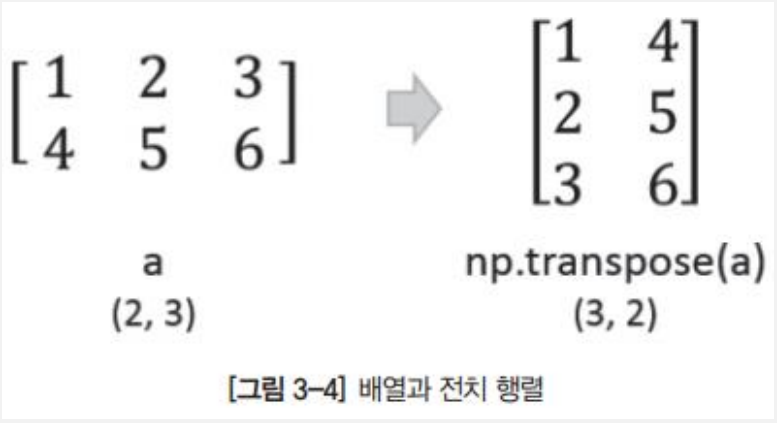
```python
b = a.ravel()
b
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

np.transpose(a)와 a.transpose(), a.T

전치행렬

- **행과 열을 바꾼 전치 배열을 반환**



[그림 3-4] 배열과 전치 행렬

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

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

```python
np.transpose(a)
```

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

```python
a.transpose()
```

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

```python
a.T
```

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

3차원 배열의 전치 행렬

3차원에서의 기본적인 전치 방법은 축 (i, j, k)가 순서가 바뀐 (k, j, i)

- **모양 (3, 2, 2) => 모양 (2, 2, 3)**
  - 위치 (0, 1, 1)인 원소 4는 전치 행렬에서 (1, 1, 0)인 위치의 원소가 됨

축 2에 따라 배치한 원소가
축 0에 따라 배치 됨

```
b = np.arange(1, 13).reshape(3, 2, 2)
b
```

Python

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

       [[ 5,  6],
        [ 7,  8]],

       [[ 9, 10],
        [11, 12]]])
```

```
b.T
```

Python

```
array([[[ 1,  5,  9],
        [ 3,  7, 11]],

       [[ 2,  6, 10],
        [ 4,  8, 12]]])
```

두 번째 인자로 조정하는 3차원의 전치 행렬

- **np.transpose(b, (0, 1, 2)) 호출**
  - 원래의 3차원 행렬 b가 그대로 출력
- **np.transpose(b, (2, 1, 0)) 호출**
  - b.T와 같은 전치 행렬
- **np.transpose(b, (0, 2, 1)) 호출**
  - 1축과 2축을 서로 바꾼 전치 행렬

[표 3-4] 3차원 행렬의 다양한 transpose 방법

axes 기술 종류	비고
(0, 1, 2)	원 행렬
(0, 2, 1)	
(1, 0, 2)	
(1, 2, 0)	
(2, 0, 1)	
(2, 1, 0)	기본 전치 행렬

```
b = np.arange(1, 13).reshape(3, 2, 2)
b
```

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

       [[ 5,  6],
        [ 7,  8]],

       [[ 9, 10],
        [11, 12]]])
```

```
np.transpose(b, (0, 1, 2))
```

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

       [[ 5,  6],
        [ 7,  8]],

       [[ 9, 10],
        [11, 12]]])
```

```
np.transpose(b, (2, 1, 0))
```

Python

```
array([[[ 1,  5,  9],
        [ 3,  7, 11]],

       [[ 2,  6, 10],
        [ 4,  8, 12]]])
```

```
np.transpose(b, (0, 2, 1))
```

Python

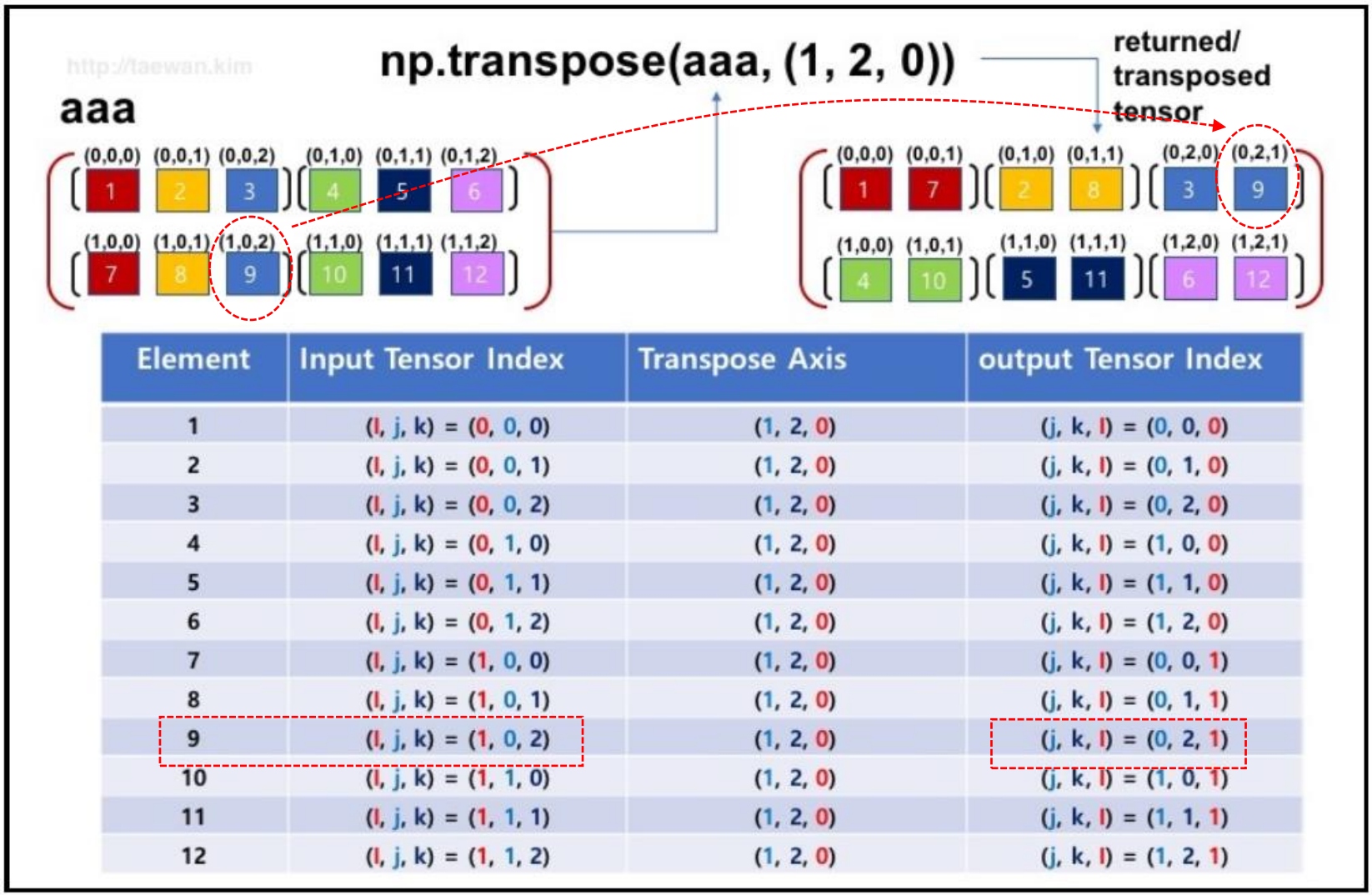
```
array([[[ 1,  3],
        [ 2,  4]],

       [[ 5,  7],
        [ 6,  8]],

       [[ 9, 11],
        [10, 12]]])
```

축 1, 2를 교환, 내부 2차원 배열의 전치

3차원 배열의 전치 행렬 이해

(2, 2, 3) ---> (2, 3, 2)

- (1, 2, 0)으로 전치
  - 원래: (0, 1, 2)
- 원소 이동 사례
  - 첨자 (1, 0, 2)
    - 원소 값 9
- 전치(이동) 후
  - 첨자 위치
    - (0, 2, 1)

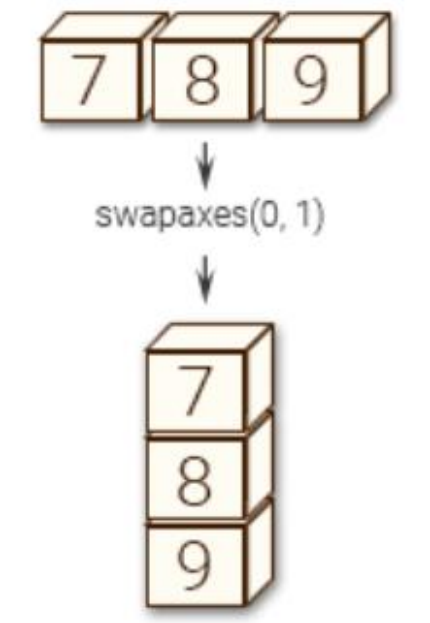


Element	Input Tensor Index	Transpose Axis	output Tensor Index
1	(I, j, k) = (0, 0, 0)	(1, 2, 0)	(j, k, I) = (0, 0, 0)
2	(I, j, k) = (0, 0, 1)	(1, 2, 0)	(j, k, I) = (0, 1, 0)
3	(I, j, k) = (0, 0, 2)	(1, 2, 0)	(j, k, I) = (0, 2, 0)
4	(I, j, k) = (0, 1, 0)	(1, 2, 0)	(j, k, I) = (1, 0, 0)
5	(I, j, k) = (0, 1, 1)	(1, 2, 0)	(j, k, I) = (1, 1, 0)
6	(I, j, k) = (0, 1, 2)	(1, 2, 0)	(j, k, I) = (1, 2, 0)
7	(I, j, k) = (1, 0, 0)	(1, 2, 0)	(j, k, I) = (0, 0, 1)
8	(I, j, k) = (1, 0, 1)	(1, 2, 0)	(j, k, I) = (0, 1, 1)
9	(I, j, k) = (1, 0, 2)	(1, 2, 0)	(j, k, I) = (0, 2, 1)
10	(I, j, k) = (1, 1, 0)	(1, 2, 0)	(j, k, I) = (1, 0, 1)
11	(I, j, k) = (1, 1, 1)	(1, 2, 0)	(j, k, I) = (1, 1, 1)
12	(I, j, k) = (1, 1, 2)	(1, 2, 0)	(j, k, I) = (1, 2, 1)

np.swapaxes(a, axis1, axis2)

배열 a에서 axis1과 axis2를 서로 교환한 배열의 뷰(view)를 반환

- 함수 np.swapaxes()
  - 두 축만 교환하므로 그 기능이 np.transpose()의 부분적
  - 또한, 새로운 배열을 생성하지 않고 뷰만 반환
    - 뷰를 수정해도 원본 배열에 반영된다는 점에 주의




```
x = np.array([[7, 8, 9]])
x
```
✓ 0.0s Python

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

0, 1이나 1, 0이나 동일한 기능으로 축 1, 2를 교환

```
y = np.swapaxes(x, 0, 1)
y
```
✓ 0.0s Python

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

```
y = np.swapaxes(x, 1, 0)
y
```
✓ 0.0s Python

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

```
y[0, 0] = 10
x
```
✓ 0.0s Python

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

3차원 배열에서 swapaxes()

- **np.swapaxes(x, 0, 2)**
  - 축 0와 축 2를 서로 바꾼 뷰를 반환

- **np.transpose(x, (2, 1, 0))**
  - 위와 같은 기능을 수행

```python
x = np.array(np.arange(8).reshape(2, 2, 2))
x
```

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

       [[4, 5],
        [6, 7]]])
```

```python
np.swapaxes(x, 0, 2)
```

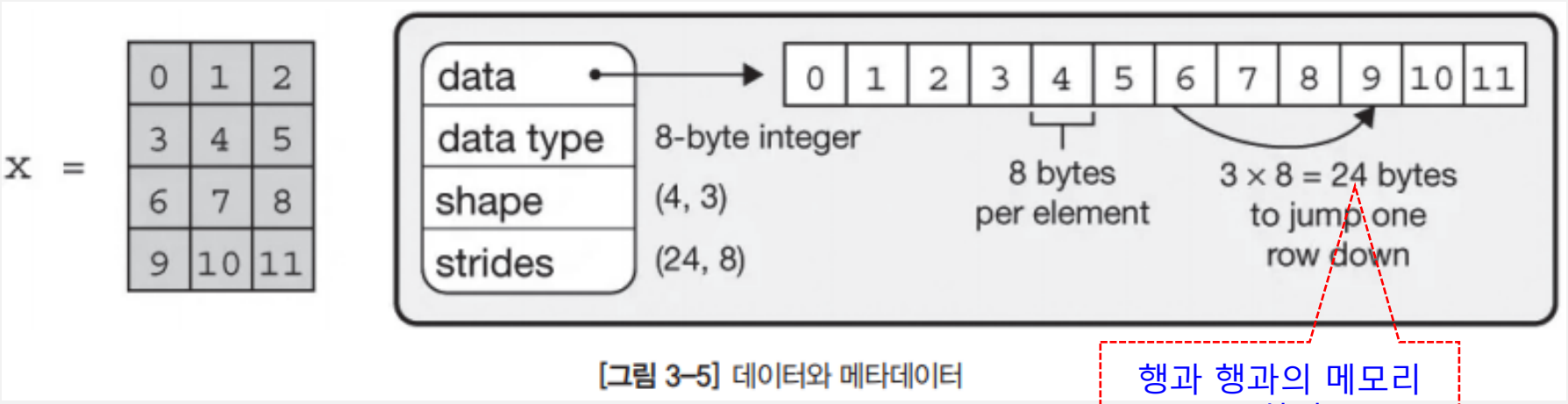
```
array([[[0, 4],
        [2, 6]],

       [[1, 5],
        [3, 7]]])
```

```python
np.transpose(x, (2, 1, 0))
```

```
array([[[0, 4],
        [2, 6]],

       [[1, 5],
        [3, 7]]])
```

ndarray의 데이터와 메타데이터

- 데이터의 복사(copy)와 보기(view)로 내부 데이터에 접근 가능
  - 보기를 사용하면 좋은 성능을 보장
    - 사용자가 원하지 않는 문제 발생 가능
      - 보기를 수정하면 원본에 반영
- NumPy 배열: 두 부분으로 구성된 데이터 구조
  - 실제 데이터 요소가 저장된 연속된 데이터 저장소
  - 데이터 저장소에 대한 정보를 갖는 메타데이터(metadata)
    - 데이터 유형, 보폭 및 ndarray 쉽게 조작할 수 있도록 도움이 되는 기타 중요한 정보



[그림 3-5] 데이터와 메타데이터

보기(view)

- **ndarray.view()**
  - 동일한 데이터를 갖는 배열의 새로운 보기
    - new view of array with the same data
- **배열을 다르게 참조하는 방법**
  - 데이터 자체를 변경하지 않고
  - Stride 및 dtype와 같은 메타데이터의 특정 항목을 변경
  - 새로운 배열을 보기(뷰, view)
  - 데이터 저장소는 동일하게 유지
    - 뷰에 대한 변경 사항은 원본에도 반영

```python
org = np.arange(5)
org
```

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

```python
v = org.view()
v
```

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

```python
v[2] = 200
v
```

```
array([  0,   1, 200,   3,   4])
```

```python
org
```

```
array([  0,   1, 200,   3,   4])
```

복사(copy)

- **복사(copy)**
  - 데이터 버퍼와 메타 데이터를 복제하여 새로운 배열을 생성
- **복사본에 대한 변경 사항**
  - 원래 배열에 반영되지 않음
- **복사는 속도도 느리고 새로운 메모리가 필요하지만 필요할 때가 있음**
  - 완전히 새로운 배열을 원할 경우

```python
org = np.arange(5)
org
```

Python

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

```python
c = org.copy()
c
```

Python

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

```python
c[2] = 200
c
```

Python

```
array([  0,   1, 200,   3,   4])
```

```python
org
```

Python

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

배열이 뷰인지 복사본인지 확인 가능

ndarray의 base 속성을 사용

- **복사이나 원본의 base 속성**
  - None을 반환

- **뷰의 base 속성**
  - 원본을 반환
  - 뷰 반환
    - 함수 호출 x.reshape(2, 3)
    - 모양이 수정된 보기를 반환
      - y = x.reshape(2, 3)
      - y.base의 결과
        - 원본 배열 x가 표시

```
x = np.arange(6)
x
```

```
array([0, 1, 2, 3, 4, 5])
```

```
y = x.reshape(2, 3) # view를 반환
y
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
print(x.base)
```

```
None
```

```
y.base
```

```
array([0, 1, 2, 3, 4, 5])
```

함수 z.copy() 복사본 반환

- 전치 행렬의 결과를 저장: z
  - 뷰(view)
    - 속성 base는 원본

- z.copy()로 새로운 복사본을 저장한 변수: c
  - 속성 base는 None

```
z = y.transpose()
z.base
```

✓ 0.0s

```
array([0, 1, 2, 3, 4, 5])
```

```
c = z.copy()
print(c.base)
```

✓ 0.0s

```
None
```

Ndarray: 같은 구조만 수정 가능

원래 원소가 2개인 리스트 부분에 3개의 원소를 삽입 가능: 이미 생성된 리스트의 구조 변경 가능, 속도 처리가 저하될 수 있는 구조

원래 원소가 2개인 배열 부분에 2개의 원소를 삽입 가능

원래 원소가 2개인 배열 부분에 3개의 원소를 삽입 불가능: 이미 생성된 배열의 구조 변경 불가능, 속도 처리를 빠르게 하는 구조

```python
lst = list(range(8))
lst
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

```python
lst[3:5] = [30, 40, 50]
lst
```

```
[0, 1, 2, 30, 40, 50, 5, 6, 7]
```

```python
x = np.arange(8)
x
```

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

```python
x[3:5] = [30, 40]
x
```

```
array([ 0,  1,  2, 30, 40,  5,  6,  7])
```

```python
x[3:5] = [30, 40, 50]
x
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[207], line 1
----> 1 x[3:5] = [30, 40, 50]
      2 x

ValueError: could not broadcast input array from shape (3,) into shape (2,)
```

보기(뷰, view)와 얕은 복사

변수 a, b는 같은 객체

```
a = np.array([[1, 2, 3], [4, 5, 6]])
a
```
✓ 0.0s
```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
b = a
b is a
```
✓ 0.0s
```
True
```

```
print(id(a))
print(id(b))
```
✓ 0.0s
```
3041647011344
3041647011344
```

```
print(a.base)
print(b.base)
```
✓ 0.0s
```
None
None
```

```
b[0, 0] = 100
a
```
```
array([[100,   2,   3],
       [  4,   5,   6]])
```

```
c = a.copy()
c
```
```
array([[100,   2,   3],
       [  4,   5,   6]])
```

```
print(id(a))
print(id(c))
```
```
2251917688144
2251917688432
```

```
print(c.base)
```
```
None
```

```
v = a.reshape(3, 2)
v
```
```
array([[100,   2],
       [  3,   4],
       [  5,   6]])
```

```
print(id(a))
print(id(v))
```
```
2251917688144
2251917682768
```

뷰인 v는 변수 a와 다른 객체이나 base가 a임

```
v[0, 1] = 200
a
```
```
array([[100, 200,   3],
       [  4,   5,   6]])
```

```
v.base
```
```
array([[100, 200,   3],
       [  4,   5,   6]])
```

3.6 브로드캐스팅

-

브로드캐스팅(broadcasting) 개요

산술 연산 시, 서로 다른 모양의 배열을 연산이 수행되도록 처리하는 방법

- 일반적, 두 배열에서 연산 가능 하려면, 두 배열의 모양은 정확히 동일
- 모양이 다른 배열 연산 시, 브로드캐스팅이 가능하면 수행 후 연산 가능
  - 작은 배열은 연산이 호환되는 모양이 되도록 전파(" 브로드캐스트 ")
    - 모두 변환이 되지는 않으며 특정한 제약조건에 따름
    - 현재의 작은 배열을 크게 전파(" 브로드캐스트 ")해 더 큰 배열로 만들어 연산을 수행
      - 즉, 배열에 대한 요소별 산술을 수행할 수 있도록 모양이 다른 배열의 크기를 일치시키는 작업

```python
a = np.array([3, 4, 5])
b = np.array([8, 7, 6])
a + b
```

Python

```
array([11, 11, 11])
```

원소 3개를 4개와 맞출 수 없어
브로드캐스팅이 수행 안됨

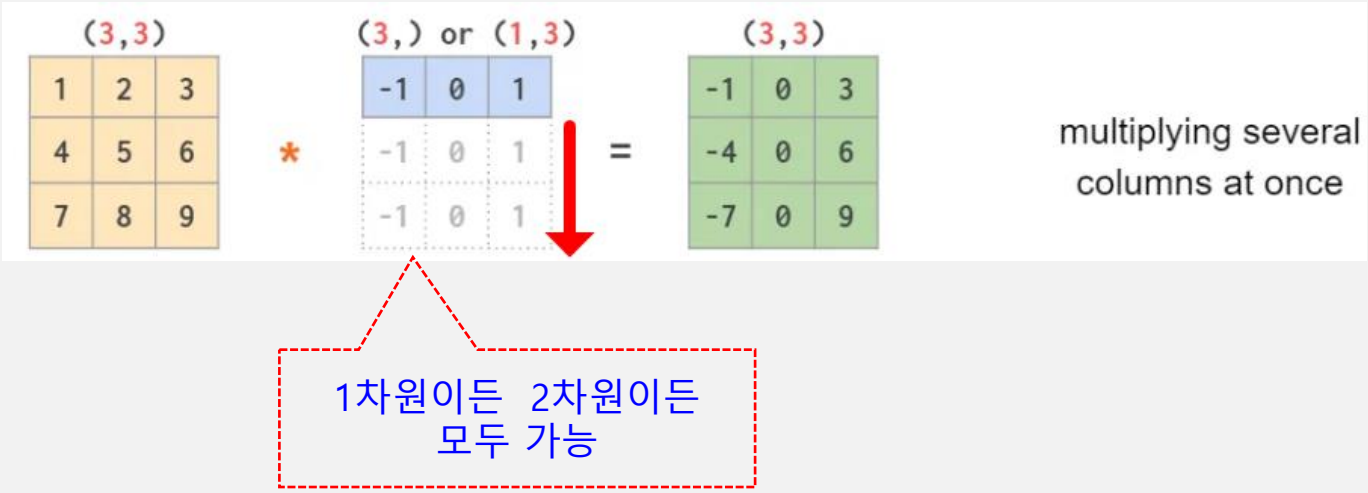
```python
a = np.array([3, 4, 5, 7])
b = np.array([8, 7, 6])
a * b
```

Python

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[223], line 3
      1 a = np.array([3, 4, 5, 7])
      2 b = np.array([8, 7, 6])
----> 3 a * b

ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

하나의 배열이 1차원으로 다른 배열 열수와 같은 구조의 브로드캐스팅

- (3, 3) * (3,) 또는 (1, 3)
  - 모양 (3,) 또는 (1, 3)
    - 세로로 전파, (3, 3)으로 전파(확장)




```
from numpy import arange
x = arange(1, 10).reshape(3, 3)
x
```

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

```
y = arange(-1, 2)
y
```

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

```
x * y
```

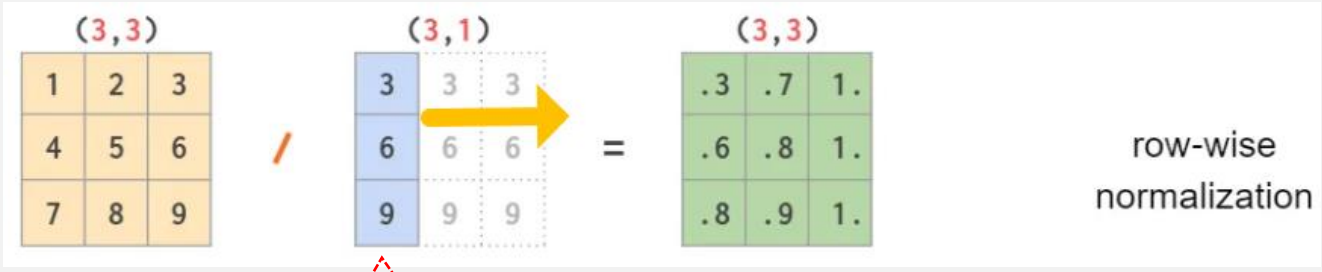
```
array([[-1,  0,  3],
       [-4,  0,  6],
       [-7,  0,  9]])
```

```
x + y
```

```
array([[ 0,  2,  4],
       [ 3,  5,  7],
       [ 6,  8, 10]])
```

하나의 배열이 2차원으로 다른 배열 행수와 같은 구조의 브로드캐스팅

- (3, 3) / (3, 1)
  - 모양 (3, 1)
    - 가로로 전파, (3, 3)으로 전파(확장)




```
from numpy import arange

x = arange(1, 10).reshape(3, 3)
x
```

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

```
y = arange(3, 10, 3).reshape(3, 1)
y
```

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

```
# 소수점 이하 자릿수를 2자리로 설정
np.set_printoptions(precision=2)
```

```
x / y
```

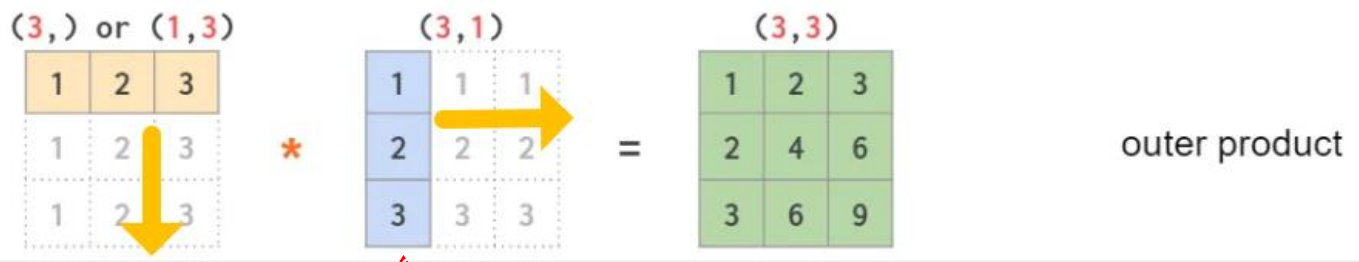
```
array([[0.33, 0.67, 1.  ],
       [0.67, 0.83, 1.  ],
       [0.78, 0.89, 1.  ]])
```

```
x - y
```

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

행 또는 열이 1인 두 배열의 브로드캐스팅

- (3,) 또는 (1, 3) * (3, 1)
  - 서로 전파해 동일한 모양으로 확장해 연산
    - 세로 전파
      - (3,) 또는 (1, 3)에서 (3, 3)으로 전파(확장)
    - 가로 전파
      - (3, 1)에서 (3, 3)으로 전파(확장)




```
from numpy import arange

x = arange(1, 4)
x
```

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

```
y = arange(1, 4).reshape(3, 1)
y
```

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

```
x * y
```

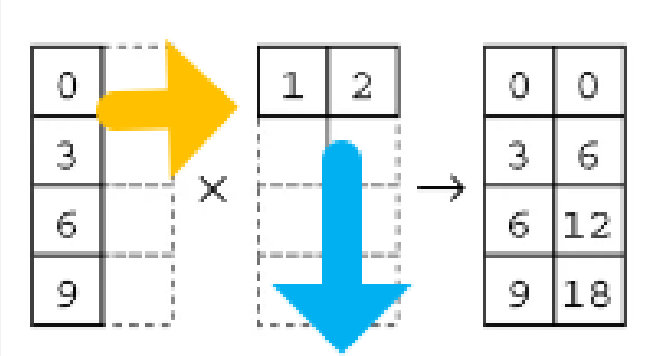
```
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```

```
x / y
```

```
array([[1.  , 2.  , 3.  ],
       [0.5 , 1.  , 1.5 ],
       [0.33, 0.67, 1.  ]])
```

행 또는 열이 1인 두 배열의 브로드캐스팅

- (4, 1) * (1, 2)
  - 서로 전파해 동일한 모양으로 확장해 연산
    - (4, 1)에서 (4, 2)으로 전파(확장)
    - (1, 2)에서 (4, 2)으로 전파(확장)




```
from numpy import arange

x = arange(0, 10, 3).reshape(4, 1)
x
```

```
array([[0],
       [3],
       [6],
       [9]])
```

```
y = arange(1, 3)
y
```

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

```
x * y
```

```
array([[ 0,  0],
       [ 3,  6],
       [ 6, 12],
       [ 9, 18]])
```

```
x + y
```

```
array([[ 1,  2],
       [ 4,  5],
       [ 7,  8],
       [10, 11]])
```

상수 np.newaxis 활용한 배열 차원 확장(증가)

배열 x: (4,)인 1차원 배열

- x[None, :] 또는 x[np.newaxis, :]
  - None이 있는 축인 행이 추가
    - 모양 (1, 4)의 2차원 배열의 보기(view)가 반환
  - 즉, 1차원에서 행이 추가되어 2차원으로 차원이 증가

```
import numpy as np
from numpy import array

x = array([0, 10, 20, 30])
x
```

```
array([ 0, 10, 20, 30])
```

```
np.newaxis is None
```

```
True
```

```
# 차수를 증강
a = x[None, :]
a
```

```
array([[ 0, 10, 20, 30]])
```

추가된 차원이 1이 됨:
모양이 (4,)에서
(1, 4)가 됨

```
a = x[np.newaxis, :]
a
```

```
array([[ 0, 10, 20, 30]])
```

```
a = x[:, np.newaxis]
a
```

```
array([[ 0],
       [10],
       [20],
       [30]])
```

```
a.shape
```

```
(4, 1)
```

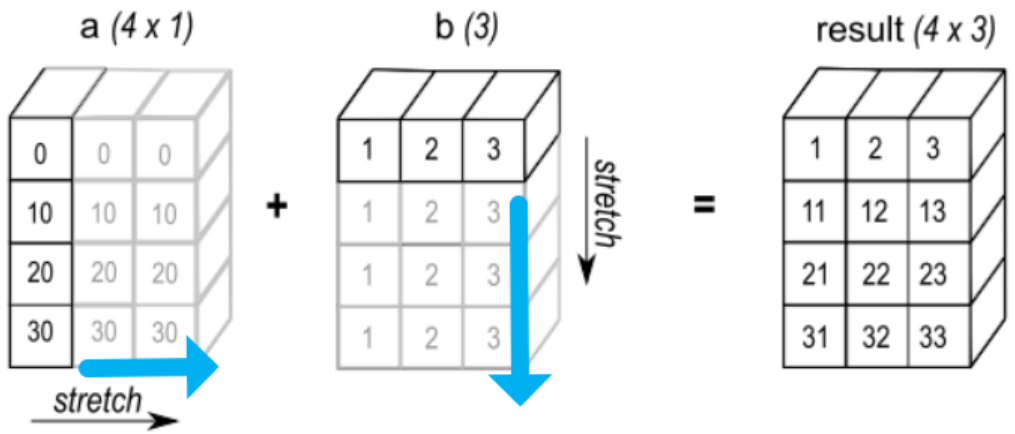
추가된 차원이 1이 됨:
모양이 (4,)에서
(4, 1)이 됨

연산 a + b를 수행

배열 a는 열이 1이며 배열 b는 행이 1

```
a = x[:, np.newaxis]
a
```

```
array([[ 0],
       [10],
       [20],
       [30]])
```




```
from numpy import array

b = array([1, 2, 3])
b
```

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

```
a + b
```

```
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

```
a * b
```

```
array([[ 0,  0,  0],
       [10, 20, 30],
       [20, 40, 60],
       [30, 60, 90]])
```

3.7 고급 색인

-

1차원 배열 색인

1차원 배열 x

- **x[i]**
  - 첨자 i인 원소 참조
  - 시퀀스인 첨자라면 여러 원소로 구성된 배열 반환

- **X[[i, j, k, ...]] : 정수 배열(또는 리스트) 인덱싱**
  - 첨자 i, j, k, ...인 행으로 구성된 배열 반환
  - 배열에서 여러 첨자 행 선택
    - 정수 배열 또는 리스트 [2]와 [2, 3]
    - 반환 값도 배열

결과는 1차원 배열

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

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

-4 -3 -2 -1

```
print(x[1])
print(x[:5])
```

```
1
[0 1 2 3 4]
```

```
print(x[[2]])
print(x[[2, 3]])
```

```
[2]
[2 3]
```

```
x[[2, 3, 5, 8]]
```

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

```
x[np.array([5, 3, -4, 8])]
```

```
array([5, 3, 6, 8])
```

2차원 배열의 색인

고급 색인: row, col이 배열이면 ...

- **행 참조**
  - x[row], x[row,], x[row, :]
- **열 참조**
  - x[:, col]
  - x[, col]: 오류
- **row, col 원소 참조**
  - x[row, col]

```
from numpy import arange

x = arange(56).reshape(7, 8)
x
```
✓ 0.0s

```
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]])
```

```
x[0]
```
✓ 0.0s
```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
x[0, 1]
```
✓ 0.0s
```
1
```

```
x[:, 2:8]
```
✓ 0.0s
```
array([[ 2,  3,  4,  5,  6,  7],
       [10, 11, 12, 13, 14, 15],
       [18, 19, 20, 21, 22, 23],
       [26, 27, 28, 29, 30, 31],
       [34, 35, 36, 37, 38, 39],
       [42, 43, 44, 45, 46, 47],
       [50, 51, 52, 53, 54, 55]])
```

축 0을 비우면 오류

```
x[, 2:8]
```
⊗ 0.0s
```
  Cell In[71], line 1
    x[, 2:8]
      ^
SyntaxError: invalid syntax
```

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

x[0, 1]
x[0][1]
위 참조는 원소 1을 반환하므로 다름

```
# 0, 1행
x[[0, 1]]
```
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])
```

행 목록의 2차원 배열 반환

```
x[:, [5, 4]]
```
```
array([[ 5,  4],
       [13, 12],
       [21, 20],
       [29, 28],
       [37, 36],
       [45, 44],
       [53, 52]])
```

열 목록의 2차원 배열 반환

```
x[[1, 3]]
```
```
array([[ 8,  9, 10, 11, 12, 13, 14, 15],
       [24, 25, 26, 27, 28, 29, 30, 31]])
```

```
x[[1, 3], :]
```
```
array([[ 8,  9, 10, 11, 12, 13, 14, 15],
       [24, 25, 26, 27, 28, 29, 30, 31]])
```

동일

행과 열 참조 매핑

- x[[1, 3], [2, 2]]
  - 첨자 [1, 2]와 [3, 2]
    - 원소 x[1, 2], x[3, 2]로 구성된 배열 반환

```
from numpy import arange

x = arange(56).reshape(7, 8)
x
```

✓ 0.0s

```
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]])
```

```
# 첨자 [1, 2]과 [3, 2]에 해당하는 10과 26
x[[1, 3], [2, 2]]
```

✓ 0.0s

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

참조하는 축의 배열 구조가 동일해야 하며 다르면 브로드캐스팅이 가능해야 함

```
# 위와 동일
x[[1, 3], [2]]
```

✓ 0.0s

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

[2]가 브로드캐스팅이 되어 [2, 2]가 되어 원소 x[1, 2], x[3, 2]로 구성된 배열 반환

```
# 1, 3행의 2열
x[[1, 3], 2]
```

✓ 0.0s

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

행과 열 참조 매핑

- **브로드캐스팅이 될 수 없는 형태**
  - 모양 (3,)과 모양 (2,)이면 오류 발생

```
x[np.array([0, 2, 4]), 1]
```

```
array([ 1, 17, 33])
```

```
print(x[0, 1])
print(x[2, 1])
print(x[4, 1])
```

```
1
17
33
```

1축의 1이 브로드캐스팅이 수행, [1, 1, 1]이 되어 각각 대응되는 순서대로 첨자 [0, 1]과 [2, 1], [4, 1]에 해당하는 [1, 17, 33]으로 구성된 모양 (3,)의 1차원 배열이 반환

- x[0, 1], x[2, 1], x[4, 1]의 참조 값으로 구성된 1차원 배열

```
x[array([0, 2, 4]), :2]
```

```
array([[ 0,  1],
       [16, 17],
       [32, 33]])
```

행 0, 2, 4에서 열 0, 1로 구성된 2차원 배열 반환

```
from numpy import arange

x = arange(56).reshape(7, 8)
x
```

✓ 0.0s

```
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]])
```

```
from numpy import array

x[array([0, 2, 4]), array([0, 1])]
```

Python

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[287], line 3
      1 from numpy import array
----> 3 x[array([0, 2, 4]), array([0, 1])]

IndexError: shape mismatch: indexing arrays could not be broadcast together with
```

기본 색인 방법

```python
from numpy import arange

a = arange(40).reshape(5, 8)
a
```

```
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]])
```

```python
a[2:4, 3:7]
```

```
array([[19, 20, 21, 22],
       [27, 28, 29, 30]])
```

```python
a[arange(2, 4)]
```

```
array([[16, 17, 18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29, 30, 31]])
```

첨자 2, 3행으로 구성된 2차원 배열

고급 색인 방법

색인 결과 배열을 다시 색인

```
from numpy import arange

a = arange(40).reshape(5, 8)
a
```

```
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]])
```

```
a[arange(2, 4)][:, [1, 0]]
```

```
array([[17, 16],
       [25, 24]])
```

결과를 다시 색인, 즉 앞 결과에서 1열과 0열을 선택

```
a[arange(2, 5)][:, [1, 0, 3, 5]]
```

```
array([[17, 16, 19, 21],
       [25, 24, 27, 29],
       [33, 32, 35, 37]])
```

```
a[arange(2, 5)]
```

```
array([[16, 17, 18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29, 30, 31],
       [32, 33, 34, 35, 36, 37, 38, 39]])
```

```
a[arange(2, 5)][:, arange(-1, -8, -1)]
```

```
array([[23, 22, 21, 20, 19, 18, 17],
       [31, 30, 29, 28, 27, 26, 25],
       [39, 38, 37, 36, 35, 34, 33]])
```

행 2에서 4열까지의 결과에서 열 -1에서 -7까지 선택