

# 단원 04

인공지능소프트웨어학과

## 다차원 배열 결합과 분리

강환수 교수

DONGYANG MIRAE UNIVERSITY  
Dept. of Artificial Intelligence



## 4.1 배열 결합

-



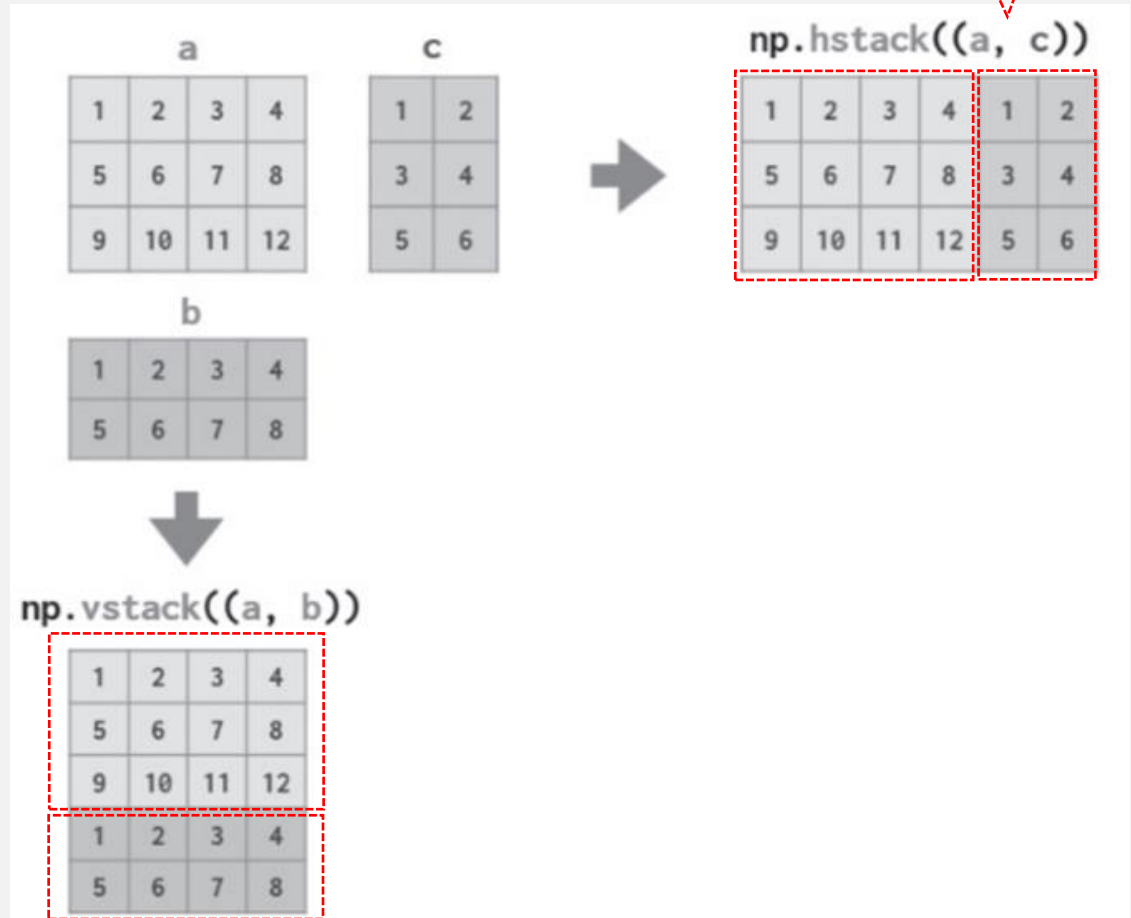
## 배열 결합

- **numpy.vstack()**
  - 배열을 수직(vertically)으로 결합
  - 세로로 결합
- **numpy.hstack()**
  - 배열을 수평(horizontally)으로 결합
  - 가로로 결합

vertical: 수직의, 세로의

horizontal: 수평의, 가로의

인자는 튜플



[그림 4-1] 배열 결합을 위한 np.vstack()과 np.hstack()

## numpy.vstack(tup)

첫 인자인 tup은 튜플 자료형으로, tup은 합칠 여러 배열이며, 이 배열들은 0축을 제외한 모양이 동일

```
import numpy as np
```

```
a = np.arange(6).reshape(3, 2)
```

```
a
```

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

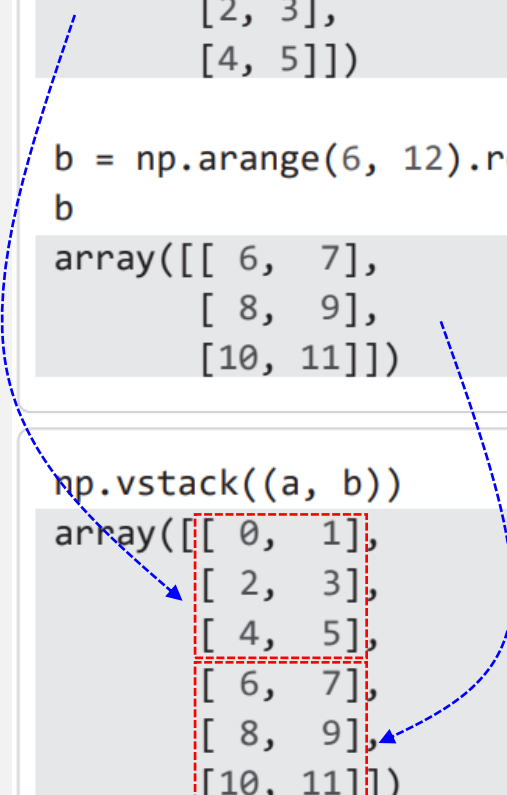
```
b = np.arange(6, 12).reshape(3, 2)
```

```
b
```

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

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

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



## numpy.vstack(tup)

인자인 배열이 1차원이면 길이가 같아야 수직으로 합칠 수 있음

```
import numpy as np

x = np.array([1, 2, 3])
y = np.array([5, 6, 7])
np.vstack((x, y))
array([[1, 2, 3],
       [5, 6, 7]])
```

## numpy.hstack()

수평으로 결합

- 2차원 배열
  - 행이 같아야 함
- 1차원 배열
  - 무조건 수평으로 결합 가능
- 2차원 이상의 배열
  - 두 번째 축을 제외하고 동일한 모양을 가져야 함

```
import numpy as np

a = np.arange(6).reshape(3, 2)
a
```

Python

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

```
b = np.arange(10, 19).reshape(3, 3)
b
```

Python

```
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

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

Python

```
array([[ 0,  1, 10, 11, 12],
       [ 2,  3, 13, 14, 15],
       [ 4,  5, 16, 17, 18]])
```

```
np.hstack((np.arange(3), np.arange(10, 15)))
array([ 0,  1,  2, 10, 11, 12, 13, 14])
```

## numpy.column\_stack()

2차원 배열 a, b를 수평으로 열(columns wise)로 결합

- 2차원 배열에서 함수 호출 `np.column_stack((a, b))`
  - 바로 `np.hstack((a, b))` 동일
- 1차원 배열 a, b로 함수 호출 `numpy.column_stack((a, b))`
  - 배열 a와 b를 먼저 축 1로 확장한 이후, 열로 쌓아 만든 2차원 배열을 반환

```
x = np.array([1, 2, 3])  
y = np.array([10, 20, 30])  
np.column_stack((x, y))
```

```
array([[ 1, 10],  
       [ 2, 20],  
       [ 3, 30]])
```

결과가 동일

```
x = np.array([1, 2, 3])  
y = np.array([10, 20, 30])  
np.column_stack((x[:, np.newaxis], y[:, np.newaxis]))
```

```
array([[ 1, 10],  
       [ 2, 20],  
       [ 3, 30]])
```

Python

```
💡 x[:, np.newaxis]
```

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

(3, )에서  
(3, 1)로 변환

```
np.hstack((x, y))
```

```
array([ 1,  2,  3, 10, 20, 30])
```

일반적으로 `hstack()`과  
`column_stack()`과는 다름

Python

```
np.column_stack == np.hstack
```

False

Python

# numpy.row\_stack()

np.vstack()의 결과와 동일

- 세로로 합치기

```
import numpy as np

a = np.arange(6).reshape(2, 3)
a
```

Python

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

```
b = np.arange(10, 22).reshape(4, 3)
b
```

Python

```
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18],
       [19, 20, 21]])
```

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

Python

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [10, 11, 12],
       [13, 14, 15],
       [16, 17, 18],
       [19, 20, 21]])
```



## numpy.concatenate()

기본적으로 축 0을 중심으로 합치는 기능을 수행, 결합하는 축을 수정 가능

- 1차원 배열 결합

- 인자 axis는 쓸 필요 없음
  - axis = 1
    - 1차원에서 축은 0만 존재
      - 오류 발생
- 가로로 결합

```
import numpy as np

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

✓ 0.9s

```
np.concatenate((a, b, c))
```

✓ 0.0s

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

```
np.concatenate((a, b, c), axis=0)
```

✓ 0.0s

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

```
# 1차 배열인 경우 axis=0과 동일
np.concatenate((a, b, c), axis=None)
```

✓ 0.0s

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

## numpy.concatenate() 2차원 배열 결합

- 기본은 **axis=0**
  - 첫 축인 행으로 결합
    - 세로로 합하기
  - 열 수가 동일해야 함

```
import numpy as np

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

y = np.arange(10, 16).reshape(2, 3)
y
array([[10, 11, 12],
       [13, 14, 15]])

z = np.arange(20, 24).reshape(2, 2)
z
array([[20, 21],
       [22, 23]])
```

```
np.concatenate((x, y))
```

Python

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [10, 11, 12],
       [13, 14, 15]])
```

vstack()과 row\_stack()과  
같은 기능 수행

## numpy.concatenate() 2차원 배열 결합, axis=1

축 1로 합치므로 수평으로 결합, 즉 수평으로 합치려면 행 수가 같아야 함

```
np.concatenate((x, y), axis=1)
```

Python

```
array([[ 0,  1,  2, 10, 11, 12],  
       [ 3,  4,  5, 13, 14, 15]])
```

hstack()과 column\_stack()과  
같은 기능 수행

```
np.concatenate((x, y, z), axis=1)
```

Python

```
array([[ 0,  1,  2, 10, 11, 12, 20, 21],  
       [ 3,  4,  5, 13, 14, 15, 22, 23]])
```

# 평면화 작업 후 연결

```
np.concatenate((x, y), axis=None)
```

axis= 0 | 1 과 다름

Python

```
array([ 0,  1,  2,  3,  4,  5, 10, 11, 12, 13, 14, 15])
```

## np.stack() 함수

주어진 둘 이상의 배열을 합쳐(쌓아) 하나의 배열로 생성

- 새 축을 따라 배열 시퀀스를 결합

- 인자 axis

- 결과 차원에서 새 축의 인덱스를 지정

- axis=0

- 첫 번째 차원

- axis=-1

- 마지막 차원

- 차수가 하나 증가

- 인자인 1차원 배열을 결합

- 하나의 2차원 배열 생성

- 인자인 2차원 배열을 결합

- 하나의 3차원 배열 생성

- concatenate() 함수와 다름

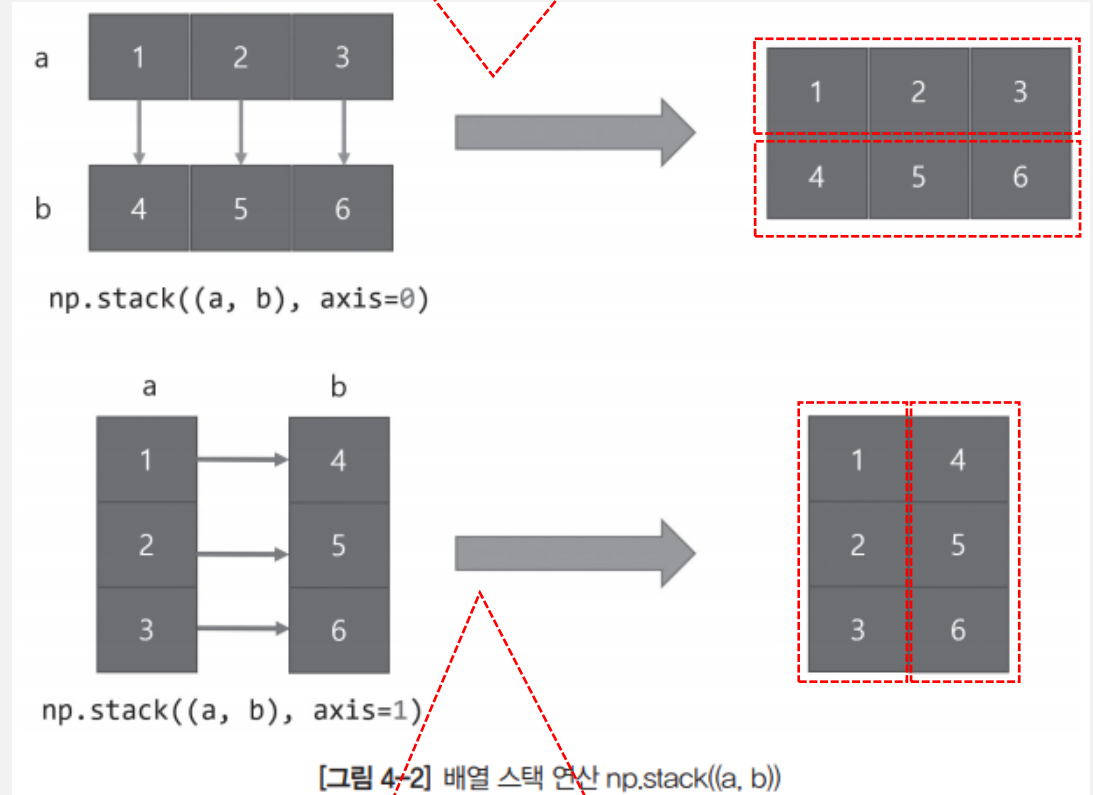
- np.stack()에 참여하는 배열

- 모두 모양이 같아야 하며

- np.stack()의 결과

- axis에 지정한 축으로 차수가 하나 증가시켜 결합

1차원 배열이 첫 축(axis=0)에 따라 차원이 증가되고 결합되어 2차원 배열이 결과



1차원 배열이 두 번째 축(axis=1)에 따라 차원이 증가되고 결합되어 2차원 배열이 결과

## np.stack((arr\_1, arr\_2), axis= 0 | 1)

```
import numpy as np
```

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

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

```
b = np.array([4, 5, 6])
```

```
b  
array([4, 5, 6])
```

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

Python

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

np.row\_stack()과 동일

```
np.stack((a, b), axis=1)
```

Python

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

np.column\_stack()과 동일

(3, )을 (3, 1)의  
2차원으로 바꾸어  
가로로 합치기

axis=1은 -1과 동일

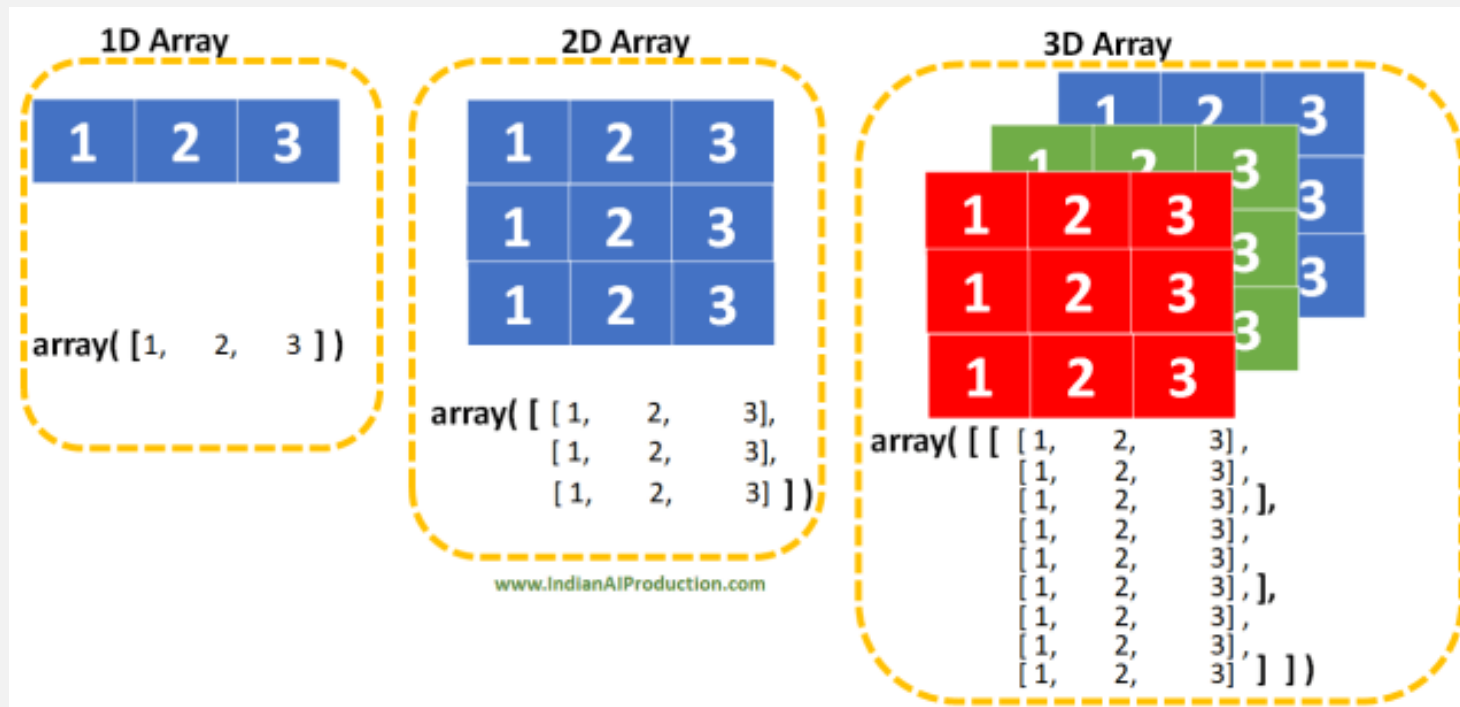
```
np.stack((a, b), axis=-1)
```

Python

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

## 3차원 배열 이해 1/3

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



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

## 3차원 배열 이해 2/3

모양 (n0, n1, n2)

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

```
np.arange(1, 19).reshape(3, 3, 2)
```

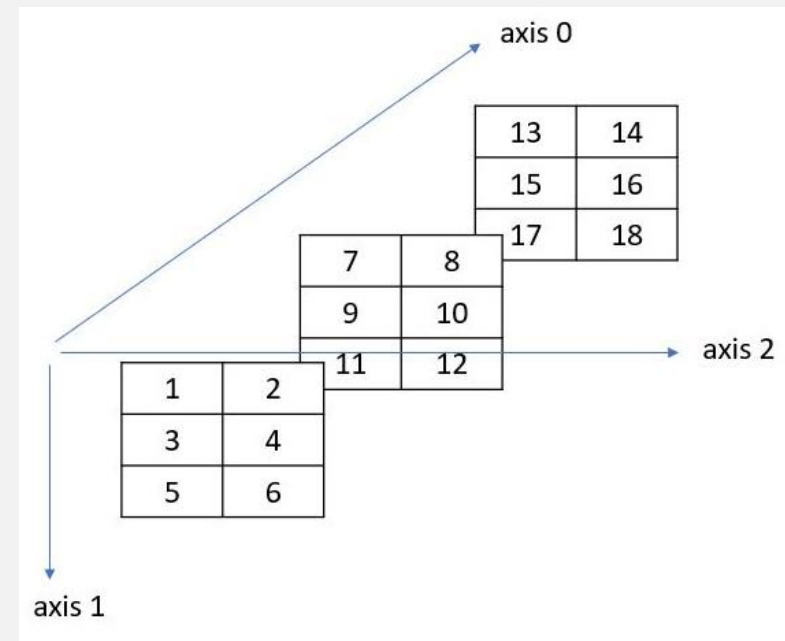
✓ 0.0s

axis = 2

axis = 1

axis = 0

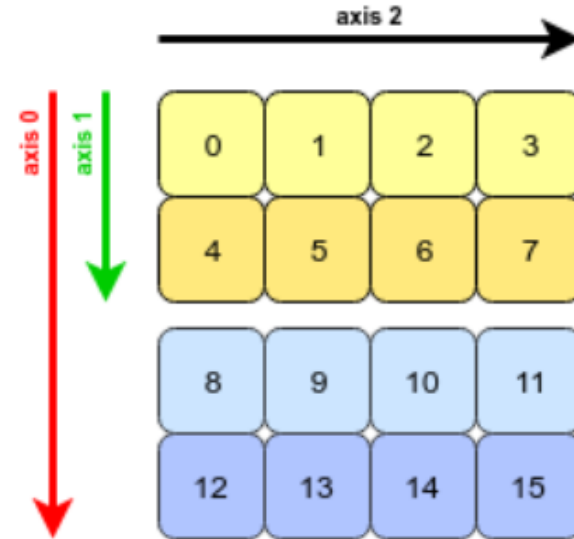
```
array([[[ 1,  2],  
       [ 3,  4],  
       [ 5,  6]],  
      [[ 7,  8],  
       [ 9, 10],  
       [11, 12]],  
      [[13, 14],  
       [15, 16],  
       [17, 18]]])
```



## 3차원 배열 이해 3/3

모양 (n0, n1, n2)

- **shape: (2, 2, 4)**
  - 2행 4열의 2차원 배열이 2개 모임



```
np.arange(16).reshape(2, 2, 4)
```

✓ 0.0s

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

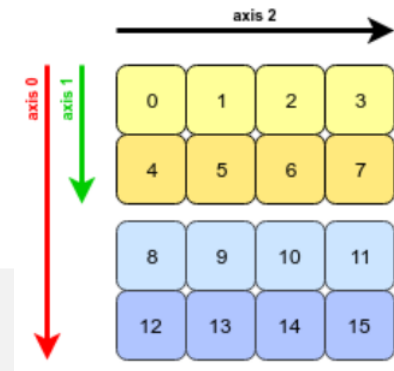


## np.stack((arr\_1, arr\_2), axis = 0)

모양 (2, 3)의 2차원 행렬 x, y, z

- 3차원 생성에서 axis = 0

늘어날 축이 0이며  
축 0로 결합



```
x = np.arange(6).reshape(2, 3)
```

x      모양이 (2, 3)에서 (1, 2, 3)으로 변환

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

```
y = np.arange(10, 16).reshape(2, 3)
```

y      모양이 (2, 3)에서 (1, 2, 3)으로 변환

```
array([[10, 11, 12],  
       [13, 14, 15]])
```

```
z = np.arange(20, 26).reshape(2, 3)
```

z      모양이 (2, 3)에서 (1, 2, 3)으로 변환

```
array([[20, 21, 22],  
       [23, 24, 25]])
```

```
ax0 = np.stack((x, y, z), axis=0)  
ax0
```

Python

```
array([[ [ 0, 1, 2],  
        [ 3, 4, 5]],  
       [[10, 11, 12],  
        [13, 14, 15]],  
       [[20, 21, 22],  
        [23, 24, 25]])
```

3차원 배열이 결과

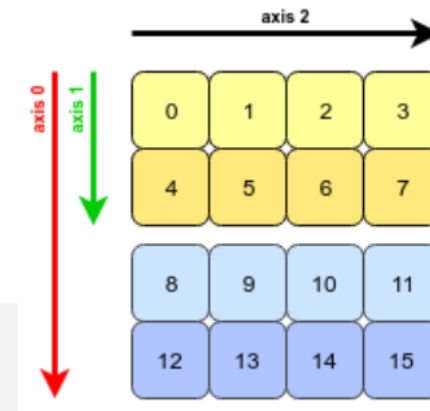
```
ax0.shape
```

Python

```
(3, 2, 3)
```

## `np.stack((arr_1, arr_2), axis = 1)`

모양 (2, 3)의 2차원 행렬 x, y, z



### • 3차원 생성에서 `axis = 1`

```
x = np.arange(6).reshape(2, 3)
```

x      모양이 (2, 3)에서 (2, 1, 3)으로 변환

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

```
y = np.arange(10, 16).reshape(2, 3)
```

y      모양이 (2, 3)에서 (2, 1, 3)으로 변환

```
array([[10, 11, 12],  
       [13, 14, 15]])
```

```
z = np.arange(20, 26).reshape(2, 3)
```

z      모양이 (2, 3)에서 (2, 1, 3)으로 변환

```
array([[20, 21, 22],  
       [23, 24, 25]])
```

```
ax1 = np.stack((x, y, z), axis=1)  
ax1
```

Python

```
array([[ [0, 1, 2],  
        [10, 11, 12],  
        [20, 21, 22]],  
       [[ [3, 4, 5],  
        [13, 14, 15],  
        [23, 24, 25]]])
```

```
ax1.shape
```

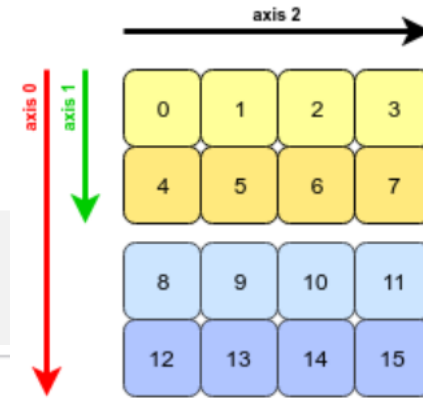
Python

```
(2, 3, 3)
```

## `np.stack((arr_1, arr_2), axis = 2)`

모양 (2, 3)의 2차원 행렬 x, y, z

### • 3차원 생성에서 axis = 2



```
x = np.arange(6).reshape(2, 3)
```

x  
모양이 (2, 3)에서 (2, 3, 1)으로 변환  
`array([[0, 1, 2],  
 [3, 4, 5]])`

```
y = np.arange(10, 16).reshape(2, 3)
```

y  
모양이 (2, 3)에서 (2, 3, 1)으로 변환  
`array([[10, 11, 12],  
 [13, 14, 15]])`

```
z = np.arange(20, 26).reshape(2, 3)
```

z  
모양이 (2, 3)에서 (2, 3, 1)으로 변환  
`array([[20, 21, 22],  
 [23, 24, 25]])`

```
ax2 = np.stack((x, y, z), axis=2)  
ax2
```

```
array([[[ 0, 10, 20],  
       [ 1, 11, 21],  
       [ 2, 12, 22]],  
      [[ 3, 13, 23],  
       [ 4, 14, 24],  
       [ 5, 15, 25]])
```

```
ax2.shape
```

```
(2, 3, 3)
```

Python

Python

## np.stack((arr\_1, arr\_2), axis= 0 | 1 | 2)

- x, y, z 모양: (2, 3)

- stack 결과 모양

- 결합하는 배열 수

- n=3

- axis = 0

- (n, 2, 3)

- axis = 1

- (2, n, 3)

- axis = 0

- (2, 3, n)

```
print(x, '\n')  
print(y, '\n')  
print(z, '\n')
```

✓ 0.0s

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

```
[[10 11 12]  
 [13 14 15]]
```

```
[[20 21 22]  
 [23 24 25]]
```

```
print('np.stack((x, y, z), axis=0)')  
print(np.stack((x, y, z), axis=0))  
print('np.stack((x, y, z), axis=1)')  
print(np.stack((x, y, z), axis=1))  
print('np.stack((x, y, z), axis=2)')  
print(np.stack((x, y, z), axis=2))
```

Python

```
np.stack((x, y, z), axis=0)
```

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

```
[[10 11 12]  
 [13 14 15]]
```

axis=0 stack 결과 모양: (3, 2, 3)

```
[[20 21 22]  
 [23 24 25]]]
```

```
np.stack((x, y, z), axis=1)
```

```
[[[ 0  1  2]  
   [10 11 12]  
   [20 21 22]]
```

axis=1 stack 결과 모양: (2, 3, 3)

```
[[ 3  4  5]  
 [13 14 15]  
 [23 24 25]]]
```

```
np.stack((x, y, z), axis=2)
```

```
[[[ 0 10 20]  
   [ 1 11 21]  
   [ 2 12 22]]
```

axis=2 stack 결과 모양: (2, 3, 3)

```
[[ 3 13 23]  
 [ 4 14 24]  
 [ 5 15 25]]]
```

## 연산 `np.stack((x, y, z), axis=2)` 과정

축 2로 차수를 증가시켜 3차원 생성해 축 2로 결합

```
np.stack((x, y, z), axis=2)
```

x

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

모양: (2, 3)

y

```
array([[10, 11, 12],  
       [13, 14, 15]])
```

모양: (2, 3)

z

```
array([[20, 21, 22],  
       [23, 24, 25]])
```

모양: (2, 3)

```
array([[[ 0, 10, 20],  
        [ 1, 11, 21],  
        [ 2, 12, 22]],  
       [[ 3, 13, 23],  
        [ 4, 14, 24],  
        [ 5, 15, 25]]])
```

모양: (2, 3, 3)

인자 `axis=2`에 따라 축 2로  
3개의 배열을 결합

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

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

모양: (2, 3, 1)

```
y[:, :, np.newaxis]
```

```
array([[[10],  
        [11],  
        [12]],  
       [[13],  
        [14],  
        [15]]])
```

모양: (2, 3, 1)

```
z[:, :, np.newaxis]
```

```
array([[[20],  
        [21],  
        [22]],  
       [[23],  
        [24],  
        [25]]])
```

모양: (2, 3, 1)

인자 `axis=2`에 따라  
축 2로 차원 증가

```
np.expand_dims(x, axis=2)
```

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

모양: (2, 3, 1)

```
np.expand_dims(y, axis=2)
```

```
array([[[10],  
        [11],  
        [12]],  
       [[13],  
        [14],  
        [15]]])
```

모양: (2, 3, 1)

```
np.expand_dims(z, axis=2)
```

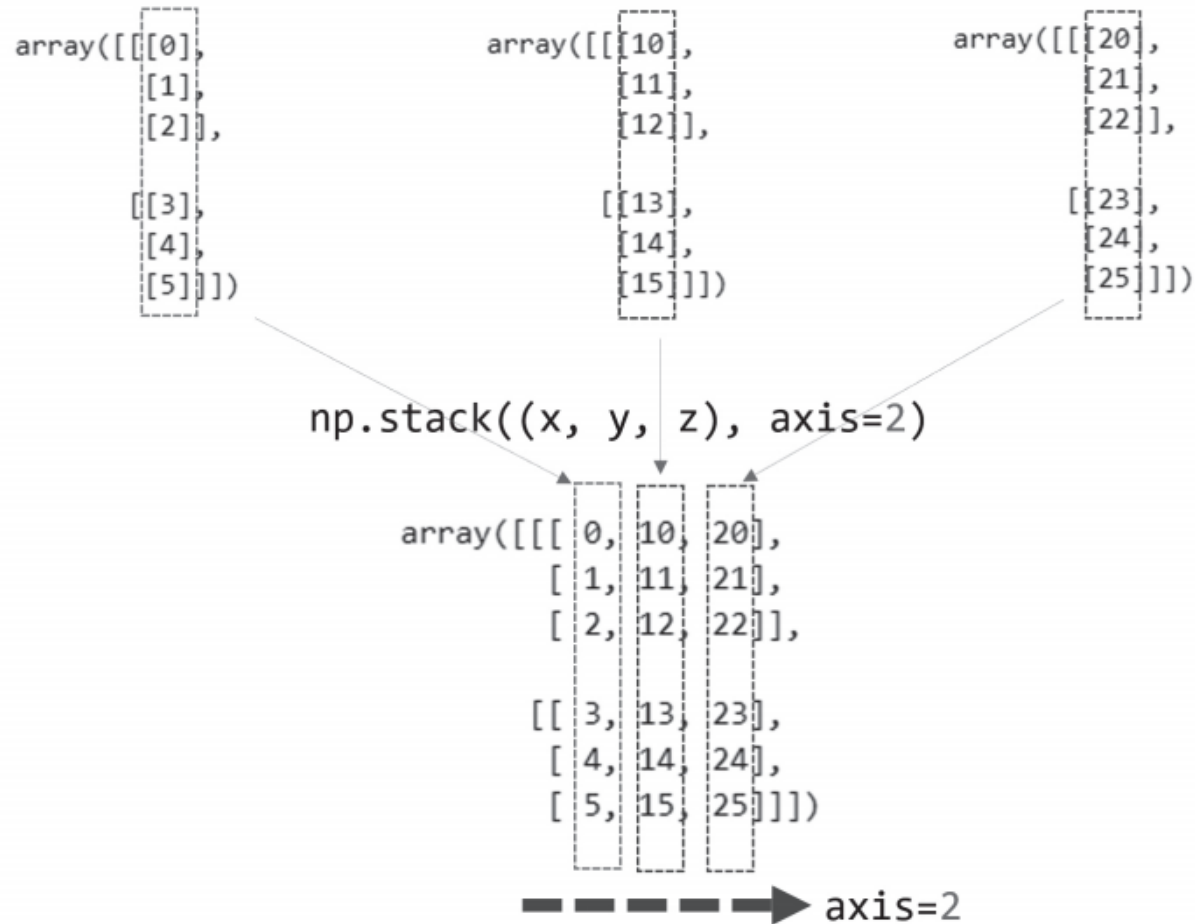
```
array([[[20],  
        [21],  
        [22]],  
       [[23],  
        [24],  
        [25]]])
```

모양: (2, 3, 1)

## 연산 `np.stack((x, y, z), axis=2)` 과정

`np.stack()`: 각 배열을 지정한 축으로 확장한 후 지정한 축으로 결합

`np.expand_dims(x, axis=2)`    `np.expand_dims(y, axis=2)`    `np.expand_dims(z, axis=2)`

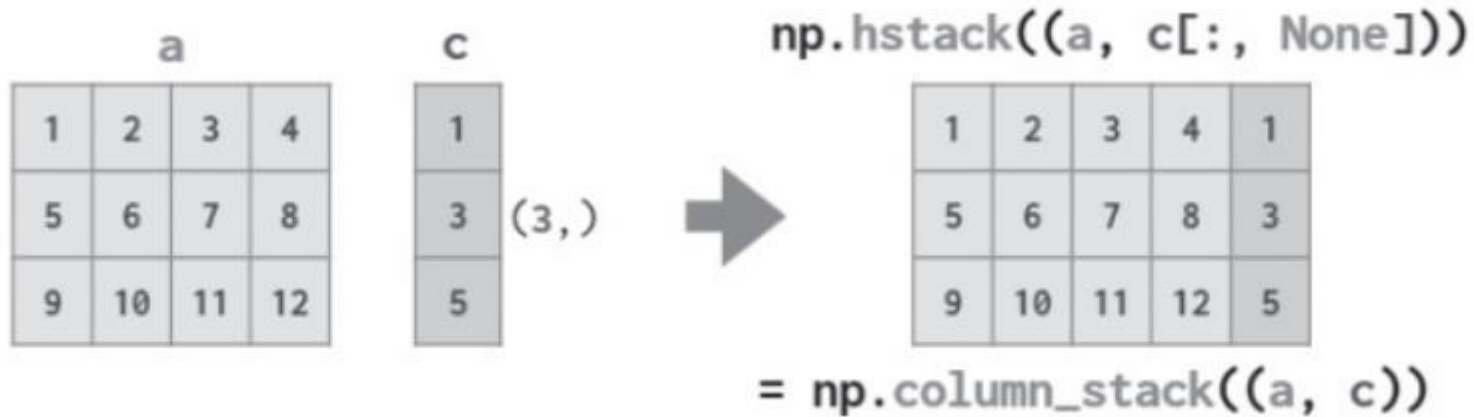


[그림 4-3] `np.stack((x, y, z), axis=2)` 이해를 위한 계산 과정

## column\_stack()과 hstack()

1차원은 다름

- 2차원은 동일



[그림 4-4] column\_stack()과 hstack() 비교

```
np.hstack((a, np.expand_dims(c, axis=1)))
```

Python

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

```
np.column_stack((a, c))
```

Python

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

## row\_stack()과 vstack()

row\_stack()과 vstack()은 완전히 같은 함수

다음은 옆 그림의 배열 a이다.

```
import numpy as np

a = np.arange(1, 13).reshape(3, 4)
a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

a

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

다음은 옆 그림의 배열 b이다.

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

b

1	2	3	4
---	---	---	---

(4,)



np.vstack((a, b))

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

다음 함수 호출 np.row\_stack((a, b))로 수직으로 두 배열을 합칠 수 있다.

```
np.row_stack((a, b))
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [ 1,  2,  3,  4]])
```

물론, 다음 함수 호출 np.vstack((a, b))로도 수직으로 두 배열을 합칠 수 있다.

```
np.vstack((a, b))
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [ 1,  2,  3,  4]])
```



## np.c\_[]

- 배열을 열 방향(2번째 축)으로(concatenation along the second axis, 즉 column-wise) 연결
  - 슬라이싱 문법을 지원하여, 인덱싱이나 슬라이싱과 함께 사용 가능
- 주요 특징
  - 배열 결합
    - 여러 배열을 수평(가로) 방향으로 쉽게 결합
  - 1차원 배열 확장
    - 1차원 배열을 자동으로 2차원으로 확장한 뒤 열로 결합

예시:

python

코드 복사

```
import numpy as np

# 두 개의 1D 배열을 열 방향으로 결합
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

result = np.c_[a, b]
print(result)
```

이 코드는 두 1차원 배열 [1, 2, 3] 과 [4, 5, 6] 을 세로로 결합하여 다음과 같은 결과를 출력합니다:

lua

코드 복사

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

주요 용도:

- 데이터를 컬럼 단위로 결합할 때 편리하게 사용됩니다.
- 1차원 배열들을 2차원 배열로 변환하는 기능을 지원하므로, 머신러닝이나 데이터 분석에서 종종 사용됩니다.

## np.c\_[]와 np.column\_stack()

np.c\_[]은 함수가 아니며 참조 연산자 [] 사용

- 1차원 -> 2차원, 열로 결합
  - 동일 기능

```
import numpy as np

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

✓ 0.0s

Python

```
np.c_[a, b]
```

Python

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

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

Python

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

## np.c\_[]와 np.column\_stack()

1행의 2차원 배열

- **x.shape**
  - (1, 3)
- **y.shape**
  - (1, 3)
- **열 결합 결과 모양**
  - (1, 6)

스칼라(scalar) 값도  
삽입 가능

스칼라(scalar) 값도  
삽입 가능

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

Python

```
np.c_[x, y]
```

Python

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

```
np.column_stack((x, y))
```

Python

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

```
np.c_[x, 10, 20, y]
```

Python

```
array([[ 1,  2,  3, 10, 20,  4,  5,  6]])
```

```
np.column_stack((x, 10, 20, y))
```

Python

```
array([[ 1,  2,  3, 10, 20,  4,  5,  6]])
```

## np.r\_[ ] 개요

수평으로 또는 수직으로도 합칠 수 있는 다양한 결합 방법을 제공

- 다양한 결합이 가능
- 1차원 벡터들을 나열
  - 그대로 1차원의 나열로 결합
- 배열이 n차원
  - 기본적으로 인자인 슬라이스 객체를 첫 번째 축 (axis=0)을 따라 연결해 변환
  - Translates slice objects to concatenation along the first axis

```
import numpy as np
```

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

```
np.r_[a, b]
```

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

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

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

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

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

## np.r\_[ ]

시퀀스, linspace 형태의 나열

- start:stop:step
- start:stop:numj
  - 코드 np.linspace(-1, 1, 6)
  - [-1, 1]
    - -1에서 1사이, 끝수 1(포함) 사이에서, 실수 6개로 등분하는 실수 시퀀스가 반환

```
import numpy as np
```

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

```
b = np.array([4,5,6])
```

```
np.r_[a, 0, 0, b]
```

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

```
np.r_[3:10:3, [0]*2, 5, 6]
```

```
array([3, 6, 9, 0, 0, 5, 6])
```

```
np.r_[ -1:1:6j, [0]*2, 5, 6]
```

```
array([-1. , -0.6, -0.2,  0.2,  0.6,  1. ,  0. ,  0. ,  5. ,  6. ])
```

```
np.linspace(-1, 1, 6)
```

```
array([-1. , -0.6, -0.2,  0.2,  0.6,  1. ])
```

## np.r\_['i', a, b]

i: 결합(쌓는)하는 축(axis) 지정

- np.r\_['0', a, a]
  - 배열 a 2개를 첫 축인 축 0으로 결합

```
a = np.array([[0, 1, 2], [3, 4, 5]])  
a
```

2차원 배열

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

```
np.r_[a, a, a] # concatenate along first axis
```

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

```
np.r_['0', a, a] # concatenate along first axis
```

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

```
np.r_['-1', a, a] # concatenate along last axis
```

np.r\_['-1', a, a]: 배열 a 2개를 마지막 축인 축 1로 결합

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

```
np.r_['1', a, a] # concatenate along last axis
```

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

## np.r\_['i, j', a, b]

j: 각 항목을 강제할 수 있는 최소 차원 수(the minimum number of dimensions to force each entry)

- 차원이 낮은 것은 최소 j차원이 되도록

- a, b는 모두 동일 차원이 되어야 함

- 0, 2

- 최소 2차원의 배열
- 0 축으로 결합

- 0, 3

- 최소 3차원의 배열
- 0 축으로 결합

```
np.r_['0, 2', a, a]
```

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

```
a = np.array([[0, 1, 2], [3, 4, 5]])  
a
```

Python

2차원 배열

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

```
np.r_['0, 2', [1,2,3], [4,5,6]] # concatenate along first axis, dim>=2
```

Python

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

```
np.r_['0, 3', [1,2,3], [4,5,6]] # concatenate along first axis, dim>=3
```

Python

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

# 배열 차수의 증가

1D -> 2D

- **shape**

- 새로 증가되는 축이 1이 됨
- 만일 (3, )에서 0축으로 증가
  - (1, 3)이 결과
- 만일 (3, )에서 1축으로 증가
  - (3, 1)이 결과

- **0 축을 증가**

- arr[**np.newaxis**, :]
- arr[**None**, :]
- np.expand\_dims(arr, **0**)

- **1 축을 증가**

- arr[:, **np.newaxis**]
- arr[:, **None**]
- np.expand\_dims(arr, **1**)

```
arr = np.array([1, 2, 3])  
print(arr[np.newaxis, :]) # 0 축으로 차원 증가  
print(arr[np.newaxis, :].shape, '\n')  
print(arr[:, np.newaxis]) # 1 축으로 차원 증가  
print(arr[:, np.newaxis].shape)
```

✓ 0.0s

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

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

(3, ) --- > (1, 3)

(3, ) --- > (3, 1)

```
np.expand_dims(arr, 0)
```

✓ 0.0s

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

```
np.expand_dims(arr, 1)
```

✓ 0.0s

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



## np.r\_['i, j, k', a, b]

a, b: 1차원 배열인 경우

- i

- 결합할(쌓는) 축 지정

- j

- 각 입력 배열인 a, b의 강제할 최소 차원
- j=2

- 모두 2차원이 되도록 차원 확장

- k

- 지정된 차원 수보다 작은 원 배열의 모양 시작을 포함해야 하는 축을 지정
  - 인자 값이 차원을 확장하는 축이 아님
- 1차원 모양 (3, )에서 j=2 k=0 이면
  - 2차원 배열 (3, 1) 모양이 됨
- 1차원 모양 (3, )에서 j=2 k=1 | -1 이면
  - 2차원 배열 (1, 3) 모양이 됨

- 축 1로 결합
- 최소 2차원으로 확장
- 모양 (1, 3)으로 차원 확장

```
np.r_[[1, 2, 3], [4, 5, 6]]
```

✓ 0.0s

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

```
np.r_['0, 2, 0', [1, 2, 3], [4, 5, 6]]
```

✓ 0.0s

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

- 축 0으로 결합
- 최소 2차원으로 확장
- 모양 (3, 1)로 차원 확장

```
np.r_['1, 2, 0', [1, 2, 3], [4, 5, 6]]
```

✓ 0.0s

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

- 축 1로 결합
- 최소 2차원으로 확장
- 모양 (3, 1)로 차원 확장

```
np.r_['0, 2, 1', [1, 2, 3], [4, 5, 6]]
```

✓ 0.0s

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

- 축 0로 결합
- 최소 2차원으로 확장
- 모양 (1, 3)으로 차원 확장

```
np.r_['1, 2, 1', [1, 2, 3], [4, 5, 6]]
```

✓ 0.0s

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

# 1차원 배열에서 3차원 배열로 차원 확장 방법

## • 3가지 방법

### - 축 0과 1로 차원 확장

#### • 모양 (3, )

- 모양 (1, 1, 3)

`np.r['i, 3, 2', ...]`

### - 축 0과 2로 차원 확장

#### • 모양 (3, )

- 모양 (1, 3, 1)

`np.r['i, 3, 1', ...]`

### - 축 1과 2로 차원 확장

#### • 모양 (3, )

- 모양 (3, 1, 1)

`np.r['i, 3, 0', ...]`

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

✓ 0.0s

(3,)

# 축 0과 1로 차원 확장

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

✓ 0.0s

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

# 축 0과 2로 차원 확장

```
print(np.expand_dims(a, axis=(0, 2)))  
print(np.expand_dims(a, axis=(0, 2)).shape)
```

✓ 0.0s

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

# 축 1과 2로 차원 확장

```
print(np.expand_dims(a, axis=(1, 2)))  
print(np.expand_dims(a, axis=(1, 2)).shape)
```

✓ 0.0s

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

## np.r\_['i, 3, k', a, b]

k: 차원을 확장하는 방법을 정수로 기술

### • k

- 지정된 차원 수보다 작은 원 배열의 시작을 포함해야 하는 축을 지정

### • 기술하지 않으면(k=-1과 동일)

- 기존 배열의 첨자가 배치되는 위치 지정
- 1차원 -> 3차원으로 늘리는 방법은 3가지

• (3, ) -> (1, 1, 3), (1, 3, 1), (3, 1, 1)  
k=1 | -2  
k=2 | -1k=0 | -3

### • 매개변수 '0, 3, 0'

- 세 번째 정수가 0
  - 확장 방법이 (3, 1, 1)로 확장

### • 매개변수 '0, 3, 1'

- 세 번째 정수가 1
  - 확장 방법이 (1, 3, 1)로 확장

### • 매개변수 '0, 3, 2'

- 세 번째 정수가 2 또는 -1(기본값)
  - 확장 방법이 (1, 1, 3)로 확장

```
np.expand_dims(np.array([1,2,3]), axis=(1, 2))
```

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

k=0로 차수를 증가시키는 것과 일치  
 (3, ) ---> (3, 1, 1)

```
np.expand_dims(np.array([4,5,6]), axis=(1, 2))
```

```
array([[4],  

    [[5]],  

    [[6]])
```

k=0로 차수를 증가시키는 것과 일치

```
np.r_['0, 3, 0', [1,2,3], [4,5,6]]
```

배열을 각각 (3, 1, 1), (3, 1, 1)로  
 차원 확장해서 축 0으로 결합

```
array([[1],  

    [[2]],  

    [[3]],  

    [[4]],  

    [[5]],  

    [[6]])
```

결과 (6, 1, 1)

```
np.r_['0, 3, 0', [1,2,3], [4,5,6]].shape
```

(6, 1, 1)

# numpy.r\_['i, j, k', ...] reference manual

- 세 개의 쉼표로 구분된 정수가 있는 문자열을 사용
  - 연결할 축
  - 항목을 강제할 최소 차원 수
  - 지정된 차원 수보다 작은 배열의 시작을 포함해야 하는 축을 지정
- 즉, 세 번째 정수를 사용
  - 모양이 업그레이드된 배열의 모양에 1을 배치할 위치를 지정
  - 기본적으로 모양 튜플의 앞에 1을 배치
    - 기본값은 '-1'
  - 따라서 '0'의 세 번째 인수를 사용
    - 배열 모양의 끝에 1을 배치

## numpy.r\_

`numpy.r_ = <numpy.lib._index_tricks_impl.RClass object>`

Translates slice objects to concatenation along the first axis.

This is a simple way to build up arrays quickly. There are two use cases.

1. If the index expression contains comma separated arrays, then stack them along their first axis.
2. If the index expression contains slice notation or scalars then create a 1-D array with a range indicated by the slice notation.

If slice notation is used, the syntax `start:stop:step` is equivalent to `np.arange(start, stop, step)` inside of the brackets. However, if `step` is an imaginary number (i.e. 100j) then its integer portion is interpreted as a number-of-points desired and the start and stop are inclusive. In other words `start:stop:stepj` is interpreted as `np.linspace(start, stop, step, endpoint=1)` inside of the brackets. After expansion of slice notation, all comma separated sequences are concatenated together.

Optional character strings placed as the first element of the index expression can be used to change the output. The strings 'r' or 'c' result in matrix output. If the result is 1-D and 'r' is specified a 1 x N (row) matrix is produced. If the result is 1-D and 'c' is specified, then a N x 1 (column) matrix is produced. If the result is 2-D then both provide the same matrix result.

A string integer specifies which axis to stack multiple comma separated arrays along. A string of two comma-separated integers allows indication of the minimum number of dimensions to force each entry into as the second integer (the axis to concatenate along is still the first integer).

A string with three comma-separated integers allows specification of the axis to concatenate along, the minimum number of dimensions to force the entries to, and which axis should contain the start of the arrays which are less than the specified number of dimensions. In other words the third integer allows you to specify where the 1's should be placed in the shape of the arrays that have their shapes upgraded. By default, they are placed in the front of the shape tuple. The third argument allows you to specify where the start of the array should be instead. Thus, a third argument of '0' would place the 1's at the end of the array shape. Negative integers specify where in the new shape tuple the last dimension of upgraded arrays should be placed, so the default is '-1'.

## np.r\_['i, j, k', a, b]

- np.r\_['0, 3, 1', ...]
  - 세 번째 정수가 1
    - (3, ) -> (1, 3, 1)로 3차원으로 증가

```
np.r_['0, 3, 1', [1,2,3], [4,5,6]]
```

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

(1, 3, 1), (1, 3, 1)을 축 0로 결합

```
print(np.expand_dims(np.array([1,2,3]), axis=(0, 2)))  
print(np.expand_dims(np.array([4,5,6]), axis=(0, 2)))
```

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

결과 (2, 3, 1)

```
np.r_['0, 3, 1', [1,2,3], [4,5,6]].shape
```

```
(2, 3, 1)
```

## np.r\_['i, j, k', a, b]

- np.r\_['0, 3, 2', ...]
  - 세 번째 정수가 2
    - (3, ) -> (1, 1, 3) 로 3차원으로 증가

```
np.r_['0, 3, 2', [1,2,3], [4,5,6]]
```

(1, 1, 3), (1, 1, 3)을 축 0로 결합

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

```
print(np.expand_dims(np.array([1,2,3]), axis=(0, 1)))  
print(np.expand_dims(np.array([4,5,6]), axis=(0, 1)))
```

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

결과 (2, 1, 3)

```
np.r_['0, 3, 2', [1,2,3], [4,5,6]].shape
```

```
(2, 1, 3)
```

# numpy.matrix 객체

특수한 2차원 배열

- `np.matrix(문자열_데이터)`

```
import numpy as np  
a = np.matrix('1 2; 3 4')  
a
```

✓ 9.2s

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

```
type(a)
```

✓ 0.0s

```
numpy.matrix
```

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

✓ 0.0s

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

행렬의 곱

```
a * b
```

✓ 0.0s

```
matrix([[ 7,  7],  
        [15, 17]])
```

```
a @ b
```

✓ 0.0s

```
matrix([[ 7,  7],  
        [15, 17]])
```

## 문자열 'r' | 'c': 출력을 변경하는 데 사용

- 문자열 'r' 또는 'c'는 행렬 출력을 생성
  - 결과가 1차원이고 'r'이 지정되면 1 x N matrix(행렬)가 생성
  - 결과가 1차원이고 'c'가 지정되면 N x 1 matrix(행렬)가 생성
- 결과가 2차원이면 둘 다 동일한 행렬 결과를 제공

```
print(np.r_[[1,2,3], [4,5,6]])  
  
# 2차원 매트릭스 생성  
np.r_['r', [1,2,3], [4,5,6]]
```

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

```
np.r_['c', [1,2,3], [4,5,6]]
```

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

```
print(np.r_[[1,2,3], [4,5,6]])  
  
# 2차원 매트릭스 생성  
np.r_['r', [1,2,3], [4,5,6]]
```

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

```
print(np.r_[[1,2,3], [4,5,6]])  
np.r_['c', [1,2,3], [4,5,6]]
```

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



## 1차원 배열 수평 방향으로 쌓기

- `numpy.hstack((a, b))`
- `numpy.r_[a, b]`
- `numpy.concatenate((a, b))`

```
import numpy as np

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

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

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

```
np.r_[a, b]
```

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

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

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

```
np.c_[a, b]
```

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

## 1차원 배열 수직 방향으로 쌓기

결과는 2차원 배열

- `numpy.vstack((a, b))`
- `numpy.row_stack((a, b))`
- `numpy.r_['0, 2', a, b]`

`np.r_['0, 2', a, b]` → `np.r_[a[np.newaxis, :], b[np.newaxis, :]]`

```
import numpy as np

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

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

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

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

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

```
np.r_[a[np.newaxis, :], b[np.newaxis, :]]
```

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

```
np.concatenate((a[np.newaxis, :], b[np.newaxis, :]))
```

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

## 1차원 배열을 옆로 바꿔 수평 방향으로 합치는 방법

- `np.r_['-1,2,0', a, b]`
  - `np.c_[a, b]`와 동일
  - `np.r_['1,2,0', a, b]`와 동일

`np.r_['1, 2, 0', a, b]` → `np.c_[a, b]`

```
import numpy as np

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

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

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

```
np.hstack((a[:, np.newaxis], b[:, np.newaxis]))
```

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

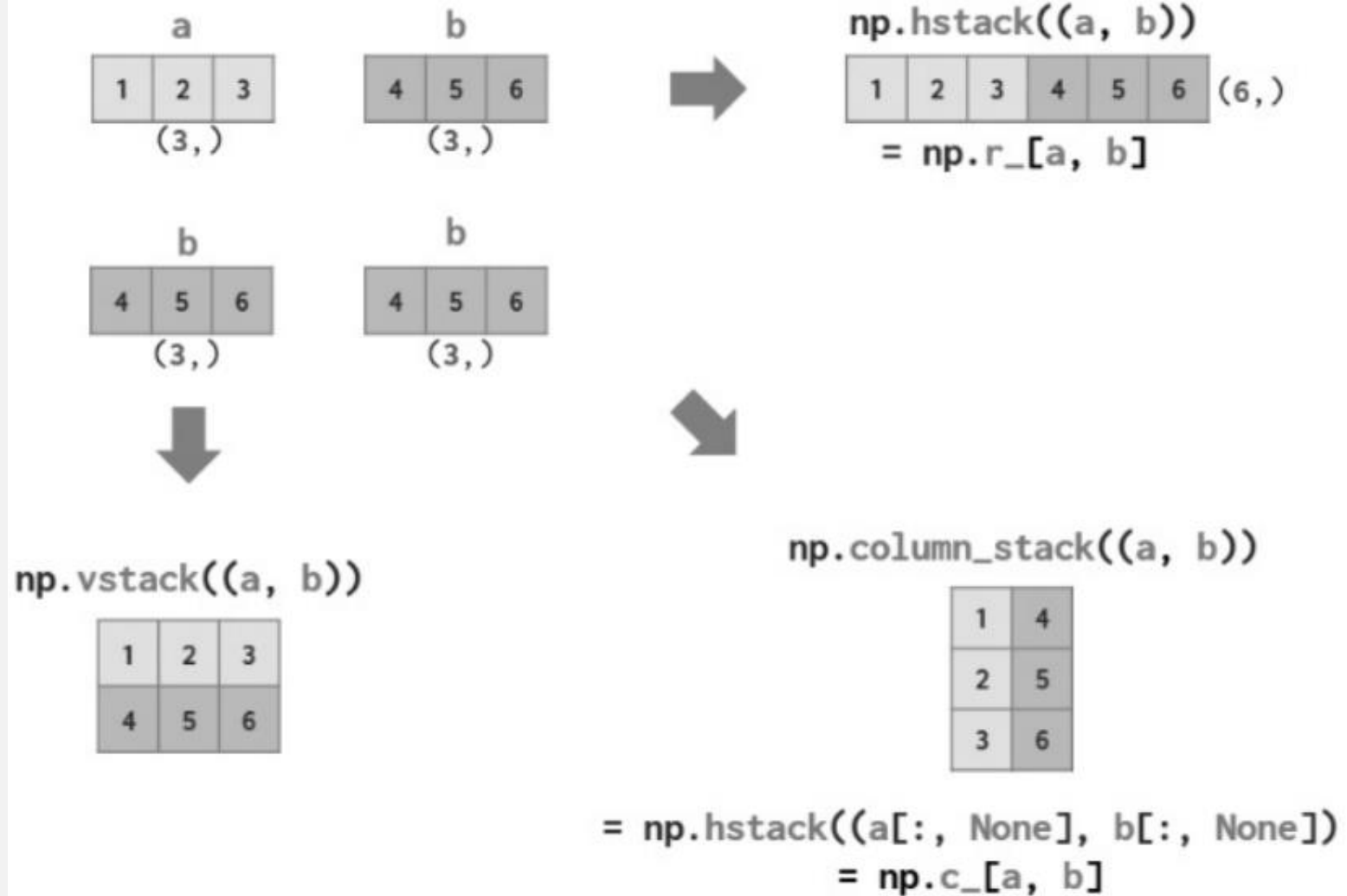
```
np.c_[a, b]
```

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

```
np.concatenate((a[:, np.newaxis], b[:, np.newaxis]), axis=1)
```

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

## 1차원 배열의 다양한 결합 방법 정리



[그림 4-8] 1차원 배열 벡터의 다양한 결합 방법

## 4.2 배열 분리

-



## np.split(ary) 개요

인자 ary를 여러 개의 부분 배열로 분리, 반환 값은 분리한 배열의 목록(list)

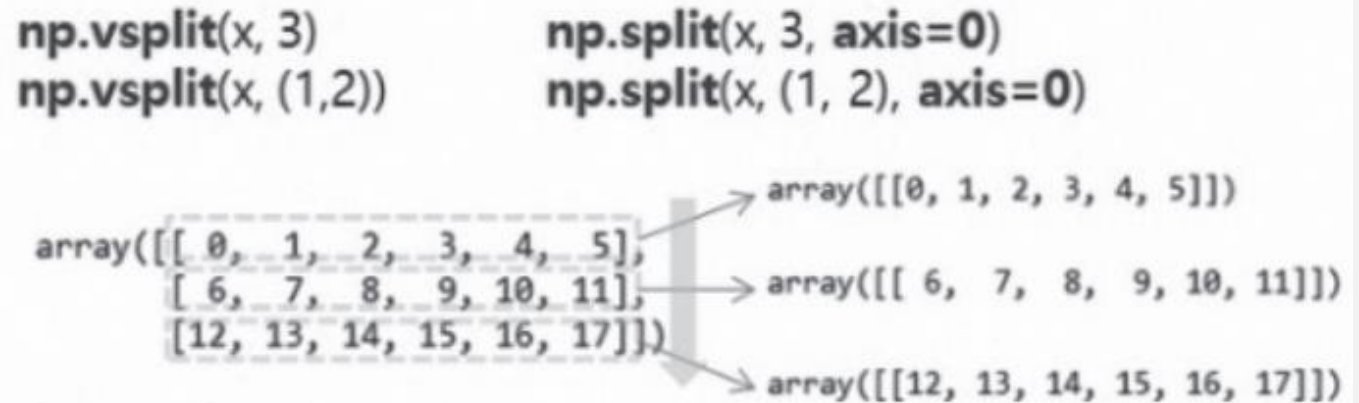
```
numpy.split(ary, indices_or_sections, axis=0)
```

배열 ary를 축 axis에 따라 기준 indices\_or\_sections로 분할해 분할된 배열의 목록을 반환

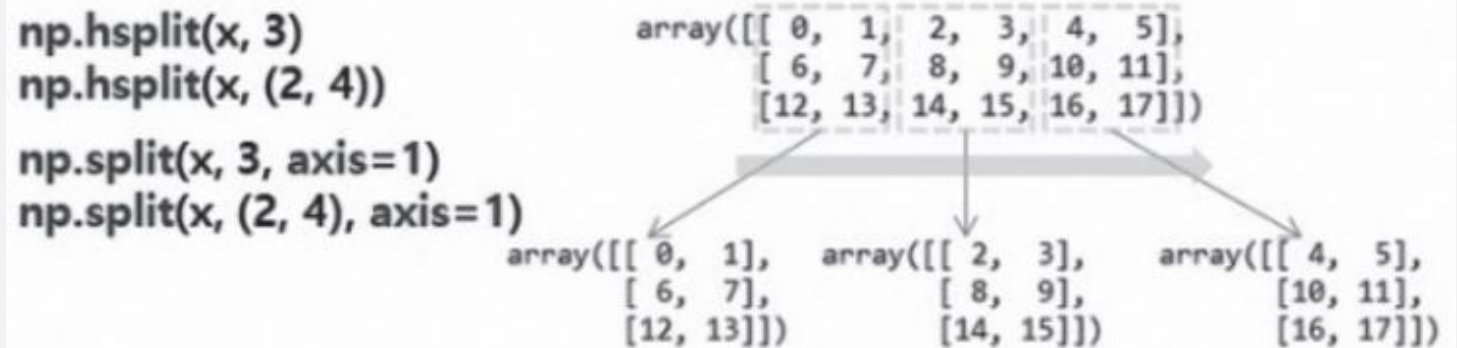
- ary: 하위 배열로 분할할 배열
- indices\_or\_sections: 정수 n인 경우, 배열은 축을 따라 n개의 동일한 배열로 분할되는데, 이러한 분할이 불가능하면 오류가 발생한다. indices\_or\_sections가 정렬된 정수의 1차원 배열인 경우, 항목은 축을 따라 배열이 분할되는 위치를 나타낸다. 예를 들어, 축=0인 경우 [2, 3]은 다음과 같으며, 첨자가 축을 따른 배열의 차원을 초과하는 경우 그에 따라 빈 하위 배열이 반환된다.
  - ary[:2]
  - ary[2:3]
  - ary[3:]
- axis=0: 기본은 0이며, 분할하는 축

## np.vsplit(a) np.hsplit(a)

- 함수 **np.split(a, axis=0)**
  - np.vsplit(a)와 동일
- 함수 **np.split(a, axis=1)**
  - np.hsplit(a)와 동일



[그림 4-9] np.split(a, axis=0)와 np.vsplit(a)



[그림 4-10] np.split(a, axis=1)과 np.hsplit(a)

## np.split(ary, n)

Split an array into multiple sub-arrays as views into ary.

- 배열을 뷰(view)인 여러 개의 하위 배열로 분할

```
import numpy as np
```

```
x = np.arange(9.0)
```

```
x
```

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

```
s = np.split(x, 3)
```

```
s
```

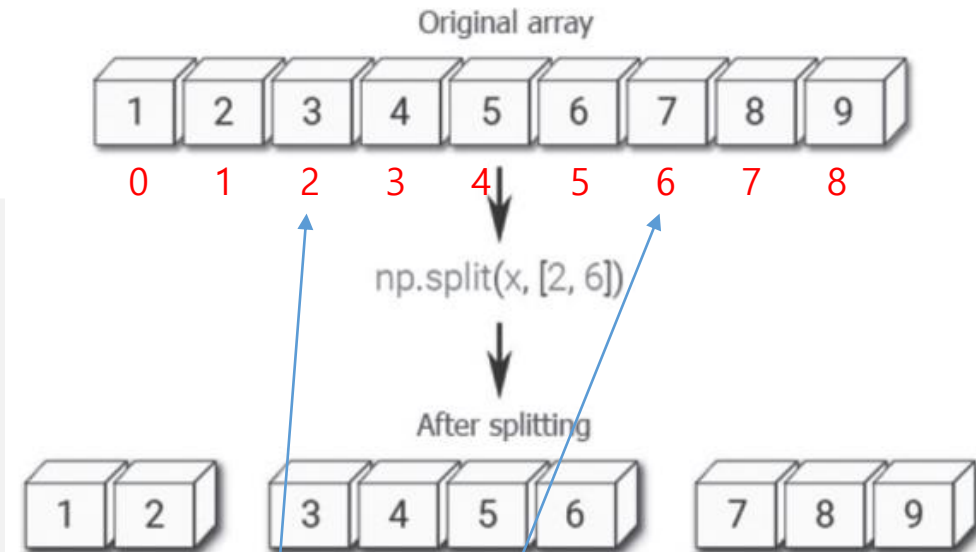
```
[array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7., 8.])]
```



## np.split(ary, [i, j, k, ...])

- **indices\_or\_sections**

- 정렬된 정수의 1차원 배열인 경우
- 항목은 배열이 축을 따라 분할되는 위치를 나타냄



[그림 4-11] np.split(x, [2, 6])의 이해

```
np.split(y, [2, 3, 4, 6])
```

```
[array([10, 20]),  
 array([30]),  
 array([40]),  
 array([50, 60]),  
 array([], dtype=int32)]
```

```
x = np.arange(1, 10)  
x
```

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

```
np.split(x, [2, 6])
```

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

## 다차원 배열 np.split(ary, n, axis=1)

numpy.hsplit(a, 3)와 동일

- np.hsplit(ary, n)

```
np.hsplit(a, 3)
✓ 0.0s
[array([[ 0,  1,  2,  3],
        [12, 13, 14, 15]]),
 array([[ 4,  5,  6,  7],
        [16, 17, 18, 19]]),
 array([[ 8,  9, 10, 11],
        [20, 21, 22, 23]])]
```

- np.split(a, 3, axis=1)

- 배열 a를 축 1 방향(수평으로 이동하며)으로 3개로 분리하면  $12/3 = 4$ 이므로,
- 4개의 열로 구성된 분할 배열 목록

```
a = np.arange(24).reshape(2, 12)
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]])
```

```
np.split(a, 3, axis=1)
```

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

```
np.split(a, 4, axis=1)
```

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

## 다차원 배열 np.split(ary, n, axis=0)

- **np.split(a, 3, axis=0)**

- 배열 a를 축 0 방향(수직으로 이동하며)으로 3개로 분리하면  $6/3 = 2$ 이므로,  
- 2개의 행으로 구성된 분할 배열 목록

```
b = np.arange(24).reshape(6, 4)  
b
```

```
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]])
```

```
np.split(b, 3, axis=0)
```

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

```
np.split(b, [2, 4, 6], axis=0)
```

```
[array([[0, 1, 2, 3],  
       [4, 5, 6, 7]]),  
 array([[ 8,  9, 10, 11],  
       [12, 13, 14, 15]]),  
 array([[16, 17, 18, 19],  
       [20, 21, 22, 23]]),  
 array([], shape=(0, 4), dtype=int32)]
```

## 여기까지 시험 범위

## 다차원 배열 `np.split(ary, [i, j, k, ...], axis= 0 | 1)`

[0, i) [i, j) [j, k) [k, ...) 분리된 배열 리스트

```
a = np.arange(24).reshape(2, 12)
a
```

✓ 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]])
```

```
np.split(a, [3, 5, 7], axis=1)
```

✓ 0.0s

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

```
b = np.arange(24).reshape(6, 4)
b
```

```
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]])
```

```
np.split(b, 3, axis=0)
```

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

## np.hsplit(ary, n)

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

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

```
np.hsplit(a, 2)
```

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

```
np.hsplit(a, [2, 5])
```

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

## np.hsplit(a, (3, 5, 8))

- 축 1 방향의 슬라이싱 인자 (3, 5, 8)
  - a[:, :3], a[:, 3:5], a[:, 5:8], a[:, 8:]의 슬라이싱 결과의 2차원 배열로 분할한 목록을 반환

```
a = np.arange(24).reshape(2, 12)
```

```
a
```

✓ 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]])
```

```
np.hsplit(a, (3, 5, 8))
```

✓ 0.0s

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

```
a[:, :3]
```

✓ 0.0s

```
array([[ 0,  1,  2],  
       [12, 13, 14]])
```

```
a[:, 8:]
```

✓ 0.0s

```
array([[ 8,  9, 10, 11],  
       [20, 21, 22, 23]])
```

## np.vsplit(a, n)

- **ValueError: array split does not result in an equal division**

```
a = np.arange(18).reshape(6, 3)
```

```
a
```

✓ 0.0s

```
array([[ 0,  1,  2],  
       [ 3,  4,  5],  
       [ 6,  7,  8],  
       [ 9, 10, 11],  
       [12, 13, 14],  
       [15, 16, 17]])
```

```
np.vsplit(a, 3)
```

✓ 0.0s

```
[array([[0, 1, 2],  
       [3, 4, 5]]),  
 array([[ 6,  7,  8],  
       [ 9, 10, 11]]),  
 array([[12, 13, 14],  
       [15, 16, 17]])]
```

```
np.vsplit(a, 4)
```

✗ 0.0s

-----  
**ValueError**

Traceback (most recent call last)



## np.vsplit(a, [i, j, k])

```
np.vsplit(a, (1, 4, 6))
```

```
[array([[0, 1, 2]]),  
 array([[ 3,  4,  5],  
        [ 6,  7,  8],  
        [ 9, 10, 11]]),  
 array([[12, 13, 14],  
        [15, 16, 17]]),  
 array([], shape=(0, 3), dtype=int32)]
```

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

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

```
a[4:6, :]
```

```
array([[12, 13, 14],  
       [15, 16, 17]])
```

```
a[6:, :]
```

```
array([], shape=(0, 3), dtype=int32)
```

## 3차원 배열의 hsplit(a, n)

3차원 이상에서의 hsplit()은 두 번째 축인 axis=1로 n 등분 분할

- np.split(a, axis=1)와 동일
- 2차원 배열 (4, 4, 2)를 축 1로 분할
  - (4, 2, 2)와 (4, 2, 2)로 분할

```
x = np.arange(32).reshape(4, 4, 2)
x
```

```
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]]])
```

```
np.hsplit(x, 2)
```

```
[array([[[ 0,  1],
         [ 2,  3]],
       [[ 8,  9],
        [10, 11]],
       [[16, 17],
        [18, 19]],
       [[24, 25],
        [26, 27]]]),
 array([[[ 4,  5],
         [ 6,  7]],
       [[12, 13],
        [14, 15]],
       [[20, 21],
        [22, 23]],
       [[28, 29],
        [30, 31]])])
```

```
np.split(x, 2, axis=1)
```

```
[array([[[ 0,  1],
         [ 2,  3]],
       [[ 8,  9],
        [10, 11]],
```

## 3차원 배열의 `hsplit(a, [i, j, ...])`

```
x = np.arange(32).reshape(4, 4, 2)
x
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]]])
```

```
np.hsplit(x, [1, 3])
```

```
[array([[ 0,  1],
        [ 8,  9],
        [16, 17],
        [24, 25]]),
 array([[ 2,  3],
        [ 4,  5],
        [10, 11],
        [12, 13],
        [18, 19],
        [20, 21],
        [26, 27],
        [28, 29]]),
 array([[ 6,  7],
        [14, 15],
        [22, 23],
        [30, 31]])]
```

## 3차원 배열의 vsplit(a, n)

수직으로 분할하는 np.vsplit()은 무조건 첫 번째 축인 axis=0로 n 등분 분할

- np.split(a, n, axis=0)과 동일
- 2차원 배열 (4, 4, 2)를 축 0로 분할
  - (2, 4, 2)와 (2, 4, 2)로 분할

```
np.split(x, 2, axis=0)
[array([[[ 0,  1],
          [ 2,  3],
          [ 4,  5],
          [ 6,  7]],

        [[ 8,  9],
          [10, 11],
          [12, 13],
          [14, 15]]]),
 array([[[16, 17],
          [18, 19],
          [20, 21],
          [22, 23]],

        [[24, 25],
          [26, 27],
          [28, 29],
          [30, 31]]])]
```

```
x = np.arange(32).reshape(4, 4, 2)
x
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]]])
```

```
np.vsplit(x, 2)
```

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

        [[ 8,  9],
          [10, 11],
          [12, 13],
          [14, 15]]]),
 array([[[16, 17],
          [18, 19],
          [20, 21],
          [22, 23]],

        [[24, 25],
          [26, 27],
          [28, 29],
          [30, 31]]])]
```

## 3차원 배열의 vsplit(a, [i, j, ...])

축 0으로 슬라이싱 구간 [i, j, ...]으로 분할

- `np.vsplit(x, [1, 3])`

- `x[:1, :, :]`
- `x[1:3, :, :]`
- `x[3:, :, :]`

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

```
x[1:3]  
✓ 0.0s  
array([[[ 8, 9],  
        [10, 11],  
        [12, 13],  
        [14, 15]],  
       [[16, 17],  
        [18, 19],  
        [20, 21],  
        [22, 23]]])
```

```
x[3:]  
✓ 0.0s  
array([[[24, 25],  
        [26, 27],  
        [28, 29],  
        [30, 31]]])
```

```
x = np.arange(32).reshape(4, 4, 2)  
x
```

```
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]])
```

```
np.vsplit(x, [1, 3])
```

✓ 0.0s

```
[array([[[0, 1],  
        [2, 3],  
        [4, 5],  
        [6, 7]]]),  
 array([[[ 8, 9],  
        [10, 11],  
        [12, 13],  
        [14, 15]],  
       [[16, 17],  
        [18, 19],  
        [20, 21],  
        [22, 23]]]),  
 array([[[24, 25],  
        [26, 27],  
        [28, 29],  
        [30, 31]]])]
```

## np.array\_split() 개요

- 함수 `np.array_split(ary)` 인자인 `ary`를 여러 개의 부분 배열로 분리
  - 반환 값은 `axis=0`로 분리한 배열의 목록(list)
  - `np.split()`와의 차이
    - 두 번째 인자가 정수인 경우, 등분이 안 되더라도 오류가 발생하지 않고 분할이 가능
    - $l // n + 1$  크기의 하위 배열  $l \% n$ 과 나머지 크기  $l // n$ 을 반환

```
numpy.array_split(ary, indices_or_sections, axis=0)
```

`indices_or_sections`가 축을 동일하게 나누지 않는 정수가 될 경우, 오류가 발생하지 않고 마지막 남은 배열을 부분 배열로 반환한다.  $n$ 개의 섹션으로 분할되어야 하는 길이가  $l$ 인 배열의 경우,  $l // n + 1$  크기의 하위 배열  $l \% n$ 과 나머지 크기  $l // n$ 을 반환함

- `ary`: 분할 대상 배열
- `indices_or_sections`: 분할할 수나 첨자 구간
- `axis=0`: 기본은 0이며, 분할하는 축

## np.array\_split(x, 5)

l=9, n=5

- $l // n + 1$  크기의 하위 배열  $l \% n$ 과 나머지 크기  $l // n$ 을 반환
  - $9 // 5 + 1$  결과 2
    - 분리된 배열 원소가 2개
  - $9 \% 5$  결과 4
    - 위 배열 수가 4개
  - $9 // 5$  결과 1
    - 나머지는 배열 원소가 1개

```
import numpy as np
```

```
x = np.arange(9.0)  
x
```

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

```
np.array_split(x, 2)
```

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

```
import numpy as np
```

```
np.array_split(x, 3)
```

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

```
np.array_split(x, 4)
```

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

```
np.array_split(x, 5)
```

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

```
np.array_split(x, [2, 4, 8])
```

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

## 2차원 np.array\_split() 분할

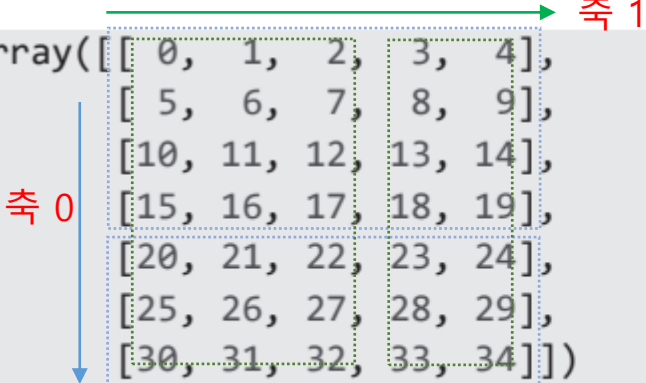
- axis=0
- axis=1

```
import numpy as np
```

```
a = np.arange(35).reshape(7, 5)
```

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]])
```



```
np.array_split(a, 2, axis=0)
```

```
[array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]]),  
 array([[20, 21, 22, 23, 24],  
       [25, 26, 27, 28, 29],  
       [30, 31, 32, 33, 34]])]
```

```
np.array_split(a, 2, axis=1)
```

```
[array([[ 0,  1,  2],  
       [ 5,  6,  7],  
       [10, 11, 12],  
       [15, 16, 17],  
       [20, 21, 22],  
       [25, 26, 27],  
       [30, 31, 32]]),  
 array([[ 3,  4],  
       [ 8,  9],  
       [13, 14],  
       [18, 19],  
       [23, 24],  
       [28, 29],  
       [33, 34]])]
```



## 2차원 np.array\_split() 분할

### 슬라이스 분할

- `[2, 4], axis=0`
  - `x[:2], x[2:4], x[4:]`
- `[2, 4], axis=1`
  - `x[:, :2], x[:, 2:4], x[:, 4:]`

```
import numpy as np
```

```
a = np.arange(35).reshape(7, 5)
```

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]])
```

축 0 (행)

축 1 (열)

```
np.array_split(a, [2, 4], axis=0)
```

```
[array([[0, 1, 2, 3, 4],  
       [5, 6, 7, 8, 9]]),  
 array([[10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]]),  
 array([[20, 21, 22, 23, 24],  
       [25, 26, 27, 28, 29],  
       [30, 31, 32, 33, 34]])]
```

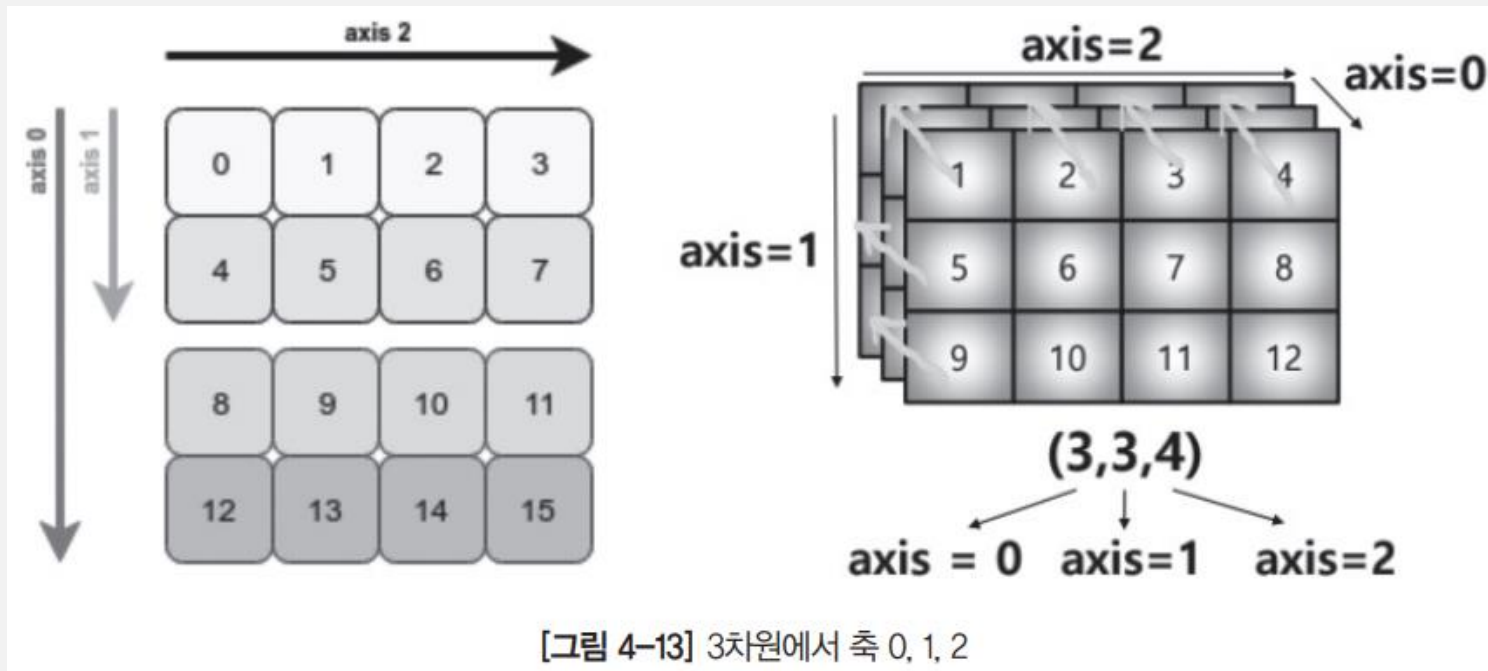
```
np.array_split(a, [2, 4], axis=1)
```

```
[array([[ 0,  1],  
       [ 5,  6],  
       [10, 11],  
       [15, 16],  
       [20, 21],  
       [25, 26],  
       [30, 31]]),  
 array([[ 2,  3],  
       [ 7,  8],  
       [12, 13],  
       [17, 18],  
       [22, 23],  
       [27, 28],  
       [32, 33]]),  
 array([[ 4],  
       [ 9],  
       [14],  
       [19],  
       [24],  
       [29],  
       [34]])]
```

## 3차원 이상에서 axis=2로 배열 분할

`np.array_split(a, axis=2)`

- 3차원에서 축 axis은 0, 1, 2를 사용
  - 왼쪽 그림
    - 모양 (2, 2, 4)인 3차원 배열
  - 오른쪽 그림
    - 모양 (3, 3, 4)인 3차원 배열



[그림 4-13] 3차원에서 축 0, 1, 2

## np.array\_split(x, 4, axis=2)

3차원 배열에서 마지막 축으로 4개 분할

- 모양 (2, 2, 6)

```
import numpy as np
```

```
x = np.arange(24).reshape(2, 2, 6)
```

```
x
```

```
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]]])
```

```
np.array_split(x, 4, axis=2)
```

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

## numpy.dsplitt()

3차원 이상의 배열에만 사용이 가능

- 함수 **np.dsplitt(a)**
  - 배열 a를 무조건 axis=2로만 분리
- 함수 **np.hsplitt(a)**
  - 배열 a를 무조건 axis=1로만 분리
- 함수 **np.vsplitt(a)**
  - 배열 a를 무조건 axis=0으로만 분리

```
x = np.arange(24).reshape(2, 2, 6)  
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]]])
```

```
np.dsplitt(x, 2)
```

✓ 0.0s

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

```
np.split(x, 2, axis=2)
```

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

## np.dsplit(x, [2, 4, 5])

- 슬라이싱 분할 가능

```
x = np.arange(24).reshape(2, 2, 6)
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]])
```

- n: 등분이 안되면 오류 발생

```
np.dsplit(x, 4)
```

```
-----
ValueError                                Traceback (most recent call last)
...
----> 1 np.dsplit(x, 4)
...
ValueError: array split does not result in an equal division
```

```
np.dsplit(x, [2, 4, 5])
```

```
array([[[ 0,  1],
        [ 6,  7]],

       [[12, 13],
        [18, 19]]]),
array([[[ 2,  3],
        [ 8,  9]],

       [[14, 15],
        [20, 21]]]),
array([[[ 4],
        [10]],

       [[16],
        [22]]]),
array([[[ 5],
        [11]],

       [[17],
        [23]]])
```

## 4.3 numpy의 난수와 다양한 함수

-



# 난수 기초

- 난수 생성기(제너레이터)
  - numpy.Random.default\_rng
    - 난수를 만드는 생성기(generator)의 생성자(constructor)
    - rng
      - 난수 생성기 RNG(Random Number Generator)
- 다음 코드로 실수 난수를 만드는 제너레이터를 하나 생성
  - 인자 12345는 비트제너레이터(BitGenerator)를 초기화하는 시드(seed) 값
    - 시드 값이 같으면 호출 순서에 따라 난수가 발생하는 값이 동일

```
import numpy as np

rg = np.random.default_rng(12345)
rg
Generator(PCG64) at 0x1612D98AC00
```

## 난수 제너레이터의 함수 random()

- `rg.random()` : [0, 1) 난수

```
# Generate one random float uniformly distributed over the range [0, 1)
r = rg.random()
r
```

Python

```
0.31675833970975287
```

```
rg.random(5)
```

Python

```
array([0.79736546, 0.67625467, 0.39110955, 0.33281393, 0.59830875])
```

```
rg.random((4, 3))
```

Python

```
array([[0.18673419, 0.67275604, 0.94180287],
       [0.24824571, 0.94888115, 0.66723745],
       [0.09589794, 0.44183967, 0.88647992],
       [0.6974535 , 0.32647286, 0.73392816]])
```



## 난수 제너레이터의 함수 integers()

- `rg.integers(low=0, high=10, size=3)`
  - `[low, high)`에서 난수 3개
    - `low, low + 1, low + 2, ... < high` 시퀀스 정수 중에서 난수 3개

```
import numpy as np

rg = np.random.default_rng(12345)
r = rg.integers(low=0, high=10, size=3)
r
```

Python

```
array([6, 2, 7], dtype=int64)
```

```
rg.integers(1, 7, (3, 4))
```

Python

```
array([[2, 2, 5, 4],
       [5, 6, 3, 6],
       [2, 4, 4, 2]], dtype=int64)
```

## 난수 생성자 함수 standard\_normal()

### 정규 분포 난수 발생

- `rg.standard_normal(10)`
  - 정규 분포 난수 10개

```
import numpy as np

rg = np.random.default_rng(seed=42)
rg.standard_normal(10)
```

Python

```
array([ 0.30471708, -1.03998411,  0.7504512 ,  0.94056472, -1.95103519,
        -1.30217951,  0.1278404 , -0.31624259, -0.01680116, -0.85304393])
```

```
rg.standard_normal((4, 3))
```

Python

```
array([[ 0.87939797,  0.77779194,  0.0660307 ],
       [ 1.12724121,  0.46750934, -0.85929246],
       [ 0.36875078, -0.9588826 ,  0.8784503 ],
       [-0.04992591, -0.18486236, -0.68092954]])
```

