

파이썬 데이터분석

- Numpy

강사 : KAIST 김동훈

I . Numpy Arrays

- Numpy 특징
- Numpy arrays
- 다양한 생성방법
- Vectorization 개념
- Array mathematics
- Array data types

I . Numpy Arrays

▪ Numpy 특징

1. Matlab for Python

- Matlab 과 매우 유사한 문법 사용

2. Python ecosystem 의 핵심

- Scipy, Scikit-learn 등 다른 많은 라이브러리에서 Numpy 기능을 내부적으로 활용하고 있음.

3. 빠른 연산 지원

- 일반적으로 파이썬은 느리지만 cpython 으로 코딩 된 Numpy 는 연산 속도가 빠르다.
- array-like, matrix-like 고성능 연산 가능.

4. n차원 배열을 쉽게 조작할 수 있다.

- 벡터(1차원), 매트릭스(2차원), 텐서(3차원 이상)

I . Numpy Arrays

- **Numpy 설치**

- Anaconda 설치시 자동으로 호환되는 Numpy 설치 됨.
- 수동 설치시, pip installation
 - pip3 install numpy

I . Numpy Arrays

- **Numpy Arrays**

- ndarray : Numpy 의 토대가 되는 자료 구조.
- **nd** : **n-d**imensional, n 차원을 의미
- dimensions 은 "axes" 라고 사용하기도 함.

※ What is **array**?

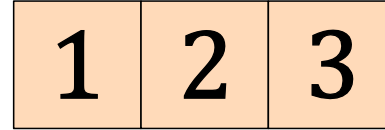
- An ordered collections of elements, like single numbers, lists, sets, vectors, matrices, or tensors
- array 의 원소들은 같은 데이터 type 을 갖고 있다.

I. Numpy Arrays

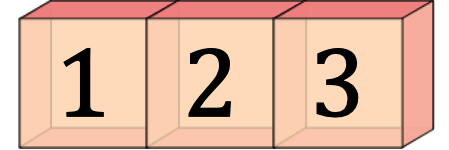
▪ np.array method

1 axis/dimensions array
one_dim= np.array([1, 2, 3])

Shape: (3,)
Axis/dim: 1
Elements: 3



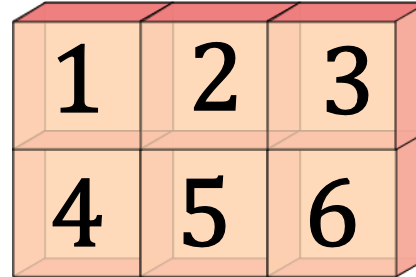
Shape: (1, 3)
Axes/dim: 2
Elements: 3



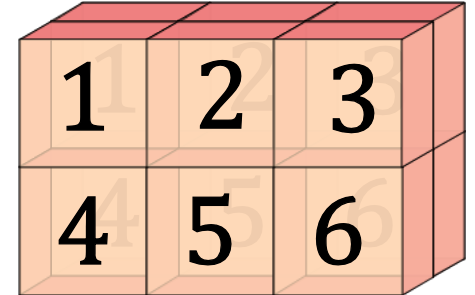
2 axis/dimensions array
two_dim_1= np.array([[1, 2, 3]])

2 axis/dimensions array
two_dim_2= np.array([[1, 2, 3],
[4, 5, 6]])

Shape: (2, 3)
Axes/dim: 2
Elements: 6



Shape: (2, 2, 3)
Axes/dim: 3
Elements: 12



3 axis/dimensions array
three_dim = np.array([[[1, 2, 3],
[4, 5, 6]],
[[1, 2, 3],
[4, 5, 6]]])

I. Numpy Arrays

▪ 다차원 array 원소개수 쉽게 파악하기

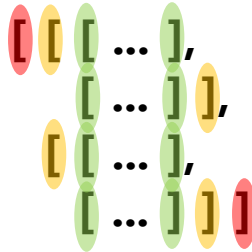
1 axis/dimensions array
one_dim= np.array([1, 2, 3])

2 axis/dimensions array
two_dim_1= np.array([[1, 2, 3]])

2 axis/dimensions array
two_dim_2= np.array([[1, 2, 3],
[4, 5, 6]])

3 axis/dimensions array
three_dim = np.array([[[1, 2, 3],
[4, 5, 6]],
[[1, 2, 3],
[4, 5, 6]]])

- 차원 개수 : 처음 나오는 원소까지 '[' 가 몇 개 나오는지 확인
- 개별 차원의 원소의 개수 (3차원 가정)



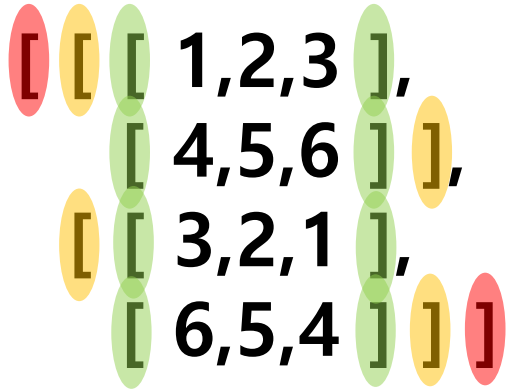
1차원의 원소의 개수 :
제일 바깥의 '[' , ']' 를 기준으로 콤마(,) 개수 확인

2차원의 원소의 개수 :
두번째 '[' , ']' 를 기준으로 콤마(,) 개수 확인

3차원의 원소의 개수 :
제일 안쪽 '[' , ']' 내부의 콤마(,) 개수 확인

I. Numpy Arrays

- 다차원 array 원소개수 쉽게 파악하기



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

shape => (2, 2, 3)

I . Numpy Arrays

- 차원, 원소의 개수, 사이즈
 - **ndim** : ndarray 의 차원의 수
 - **shape** : 각 차원별 원소의 개수
 - **size** : 전체 원소의 개수
 - **dtype** : 데이터 타입

I . Numpy Arrays

- 다양한 생성 방법
 - 다른 python 데이터 구조에서 생성
 - Numpy 고유의 array 생성 방식
 - 라이브러리 함수 사용

I . Numpy Arrays

- 다양한 생성 방법
 - 다른 python 데이터 구조에서 생성

```
array_list = np.array([1, 2, 3])  
array_tuple = np.array(((1, 2, 3), (4, 5, 6)))  
array_set = np.array({"pikachu", "snorlax", "charizard"})
```

I . Numpy Arrays

- 다양한 생성 방법
 - Numpy 고유의 array 생성 방식

```
zeros = np.zeros(5)
ones = np.ones((3, 3))
arange = np.arange(1, 10, 2)
empty = np.empty([2, 2])
linespace = np.linspace(-1.0, 1.0, num=10)
full = np.full((3,3), -2)
indices = np.indices((3,3))
```

I. Numpy Arrays

- 다양한 생성 방법
 - Numpy 고유의 array 생성 방식

`np.zeros(3)` →

0.	0.	0.
----	----	----

`np.ones(3)` →

1.	1.	1.
----	----	----

`np.empty(3)` →

5e-296	7e-297	1e-296
--------	--------	--------

`np.full(3, 7.)` →

7.	7.	7.
----	----	----

`np.array([1, 2, 3])` →

a		
1	2	3

`np.zeros_like(a)` →

0	0	0
---	---	---

`np.ones_like(a)` →

1	1	1
---	---	---

`np.empty_like(a)` →

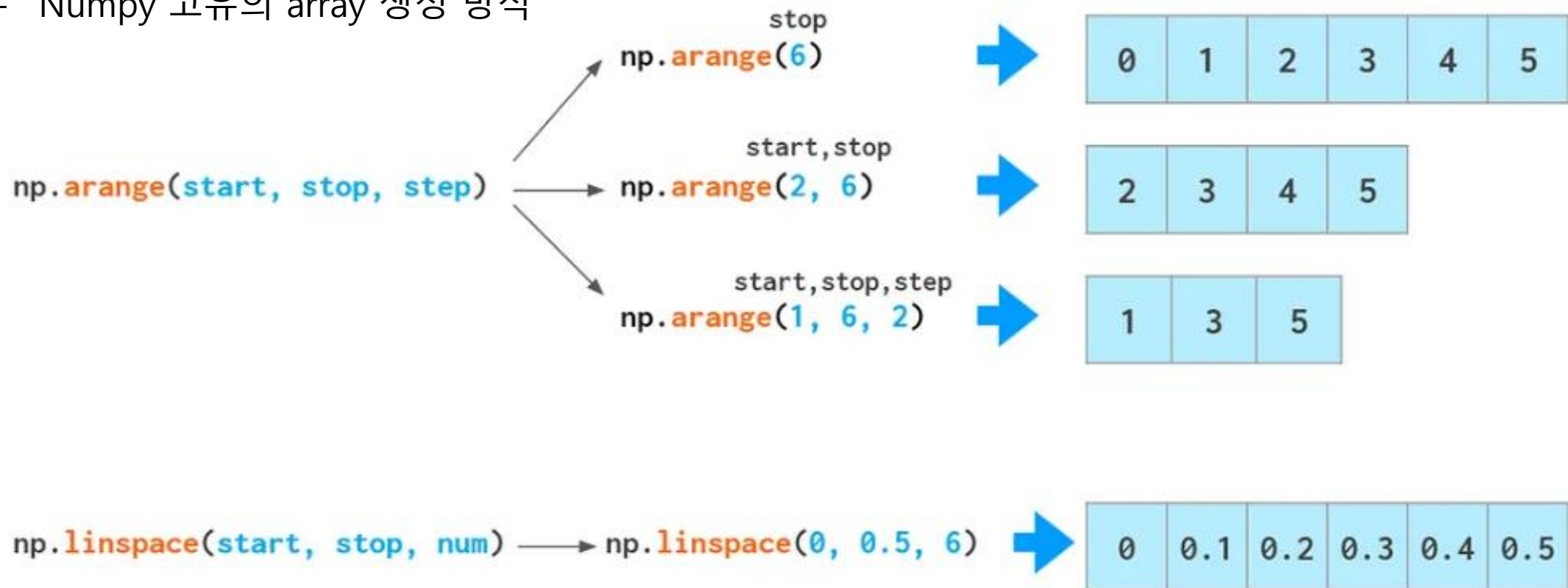
54087 6897	1630433 390	2036429 426
---------------	----------------	----------------

`np.full_like(a, 7)` →

7	7	7
---	---	---

I. Numpy Arrays

- 다양한 생성 방법
 - Numpy 고유의 array 생성 방식



I . Numpy Arrays

- 다양한 생성 방법
 - 라이브러리 함수 사용

```
diagonal = np.diag([1, 2, 3], k=0)
```

```
identity = np.identity(3)
```

```
eye = np.eye(4, k=1)
```

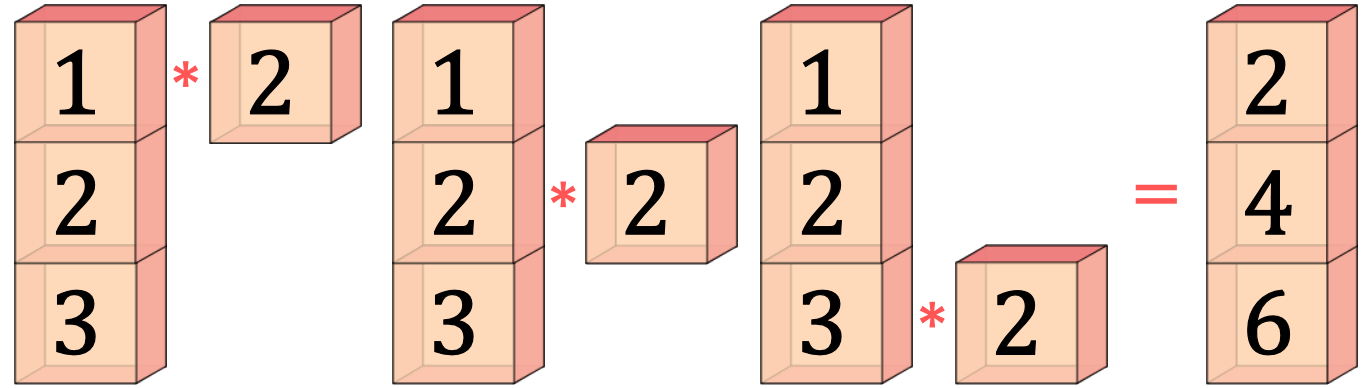
```
rand = np.random.rand(3,2)
```

I. Numpy Arrays

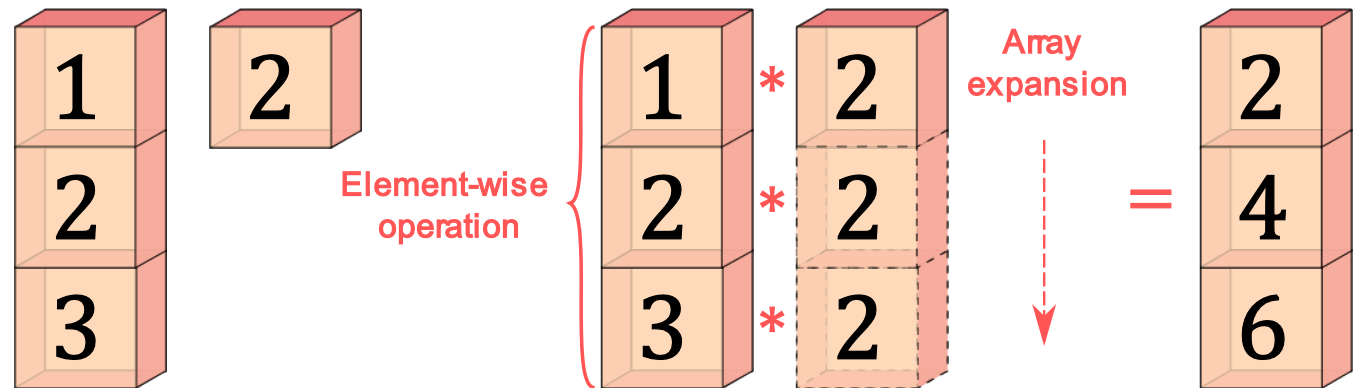
■ Vectorization

- python 자체는 느리다.
(loop 동작방식)
- Scientific Computing에선 속도가 중요하다.
- loop를 제거하고, vector로 만들어서 여러 원소들 간의 계산을 빠르게 하는 것

Looping logic



Vectorization logic



I. Numpy Arrays

- Array mathematics

- +, -, *, /

$$\begin{bmatrix} 1 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 10 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \end{bmatrix} - \begin{bmatrix} 4 & 8 \end{bmatrix} = \begin{bmatrix} -3 & -6 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} * \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 40 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} / \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 2.0 & 1.6 \end{bmatrix} \text{ np.float64}$$

$$\begin{bmatrix} 4 & 8 \end{bmatrix} // \begin{bmatrix} 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 1 \end{bmatrix} \text{ np.int32}$$

$$\begin{bmatrix} 3 & 4 \end{bmatrix} ** \begin{bmatrix} 2 & 3 \end{bmatrix} = \begin{bmatrix} 9 & 64 \end{bmatrix}$$

I. Numpy Arrays

- Array mathematics

- +, -, *, /

1	2
3	4

 +

1	0
0	1

 =

2	2
3	5

1	2
3	4

 -

1	0
0	1

 =

0	2
3	3

1	2
3	4

 *

2	0
0	2

 =

2	0
0	8

1	2
3	4

 @

2	0
0	2

 =

2	4
6	8

1	2
3	4

 /

2	1
1	2

 =

0.5	2.
3.	2.

1	2
3	4

 **

2	1
1	2

 =

1	2
3	16

I . Numpy Arrays

- 삼각함수
 - sin, cos, tan
- Rounding
 - round, floor, ceil
- Exponents and logarithms
 - exp, log, log10, log2
- others
 - sqrt, sign

I. Numpy Arrays

$$a^2 = \begin{bmatrix} 2 & 3 \end{bmatrix} ** 2 = \begin{bmatrix} 4 & 9 \end{bmatrix}$$

$$\sqrt{a} = \text{np.sqrt}(\begin{bmatrix} 4 & 9 \end{bmatrix}) = \begin{bmatrix} 2. & 3. \end{bmatrix}$$

$$e^a = \text{np.exp}(\begin{bmatrix} 1 & 2 \end{bmatrix}) = \begin{bmatrix} 2.72 & 7.39 \end{bmatrix}$$

$$\ln a = \text{np.log}(\begin{bmatrix} \text{np.e} & \text{np.e**2} \end{bmatrix}) = \begin{bmatrix} 1. & 2. \end{bmatrix}$$

I . Numpy Arrays

- Array data type
 - np.bool : True/False
 - np.int : int8, int16, int32, int64
 - np.uint : uint8, uint16, uint32, uint64
 - np.float : float16, float32, float64, float128
 - np.complex : complex64, complex128, complex256
 - np.str
 - np.bytes

I. Numpy Arrays

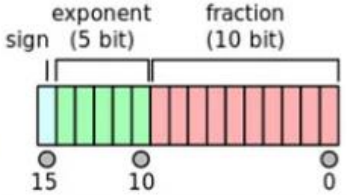
- Array data type

- Integer

dtype	range	dtype	range
<code>np.int8</code>	-128 .. 127	<code>np.uint8</code>	0 .. 255
<code>np.int16</code>	-32768 .. 32767	<code>np.uint16</code>	0 .. 65535
<code>np.int32</code>	$-2.1 \cdot 10^9 \dots 2.1 \cdot 10^9$	<code>np.uint32</code>	$0 \dots 4.2 \cdot 10^9$
<code>np.int64</code>	$-9.2 \cdot 10^{18} \dots 9.2 \cdot 10^{18}$	<code>np.uint64</code>	$0 \dots 1.8 \cdot 10^{19}$

I. Numpy Arrays

- Array data type
 - Float

type	range	signi- ficant digits*		type	composed of
float16	$\pm(6.0 \times 10^{-8} \dots 65504)$	3	1bit 5bit 10bit	—	—
float32	$\pm(1.4 \times 10^{-45} \dots 3.4 \times 10^{38})$	6	1bit 8bit 23bit	complex64	two float32's
float64	$\pm(4.9 \times 10^{-324} \dots 1.8 \times 10^{308})$	15	1bit 11bit 52bit	complex128	two float64's
float128**	$\pm(3.7 \times 10^{-4951} \dots 1.1 \times 10^{4932})$	18	1bit 15bit 64bit	complex256	two float128's

I . Numpy Arrays

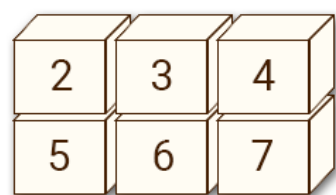
- Array data type and conversions
 - **dtype** 인자 : type 지정
 - **astype** 메서드 : type 변경
 - **isinstance** 메서드 : type check

II. **Array manipulation**

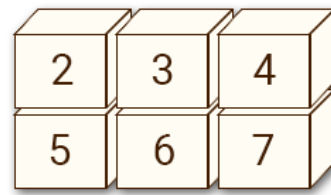
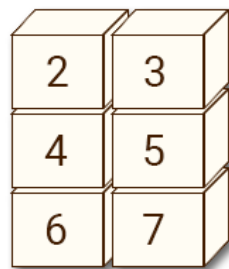
- Reshape
- Flattening
- Transpose-like
- Expanding & Squeezing
- Joining & Splitting
- Array repetition

II. Array manipulation

- Reshape 메서드
 - data 는 그대로 둔채 형태(차원, 원소의 개수) 를 변경
 - '-1' 를 사용하면 해당 차원은 Numpy 가 알아서 변형



`np.reshape (3, 2)`

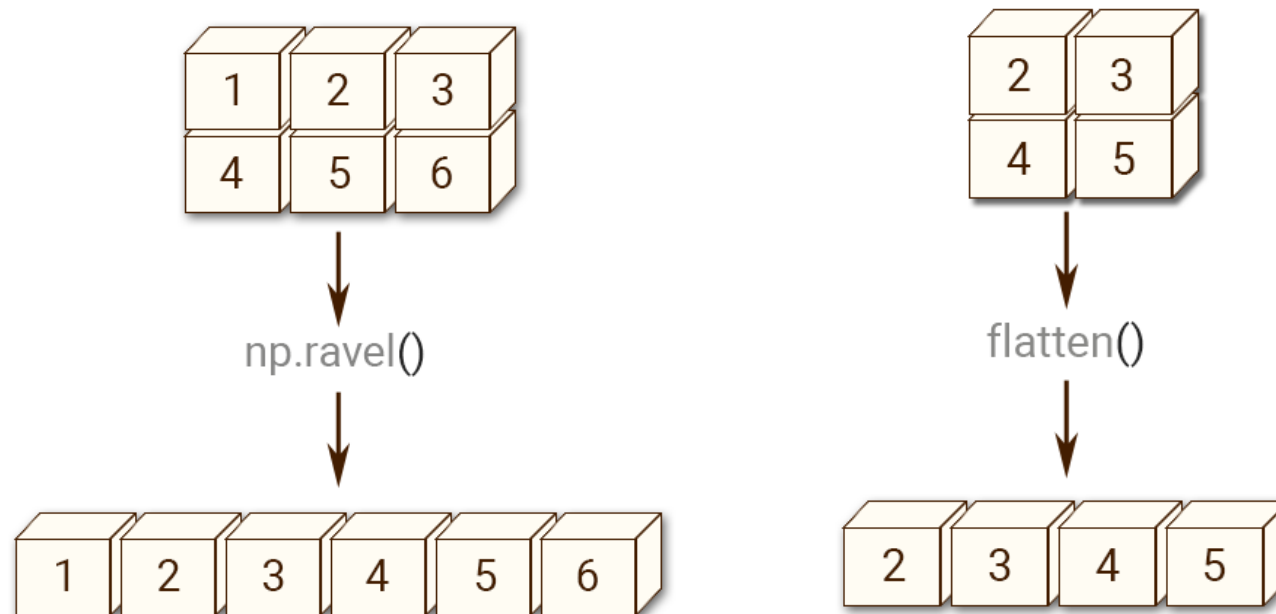


`np.reshape (3, -1)`



II. Array manipulation

- flatten and ravel 메서드
 - 두 메서드는 모두 다차원 array 를 1차원으로 평탄화하는 함수
 - ravel() : view 를 반환
 - flatten() : 독립된 copy 를 반환



II. Array manipulation

- (1,6) 과 (6,) 의 차이
 - 둘다 6개의 원소를 갖고 형태상으로 일차원 array
 - 하지만 (1,6) 은 2차원 array 이고 (6,) 은 1차원 array 이다.
 - 1을 다른 차원에 갖고 있으므로 얼마든지 고차원 array 가 될 수 있다. ex) (1,1,1,1,6) → 5차원

```
a = np.array([1, 2, 3])
print(f'Array a: {a}\n')
print(f'Array a shape: {a.shape}\n')
print(f'Array a dimensions: {a.ndim}\n')

a_row = a[np.newaxis, :]
print(f'Array a: {a_row}\n')
print(f'Array a shape: {a_row.shape}\n')
print(f'Array a dimensions: {a_row.ndim}\n')

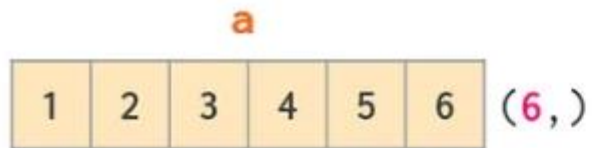
a_col = a[:, np.newaxis]
print(f'Array a:\n{a_col}\n')
print(f'Array a shape: {a_col.shape}\n')
print(f'Array a dimensions: {a_col.ndim}\n')
```

`a[np.newaxis, :] = a[None, :]` 동일한 문법

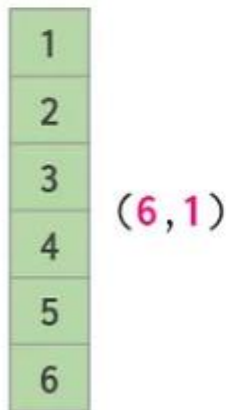
`a[:, np.newaxis] = a[:, None]` 동일한 문법

II. Array manipulation

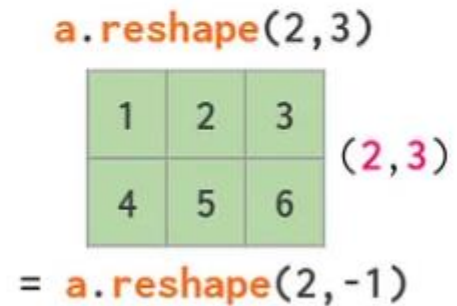
- (1,6) 과 (6,) 의 차이



a.reshape(-1, 1)

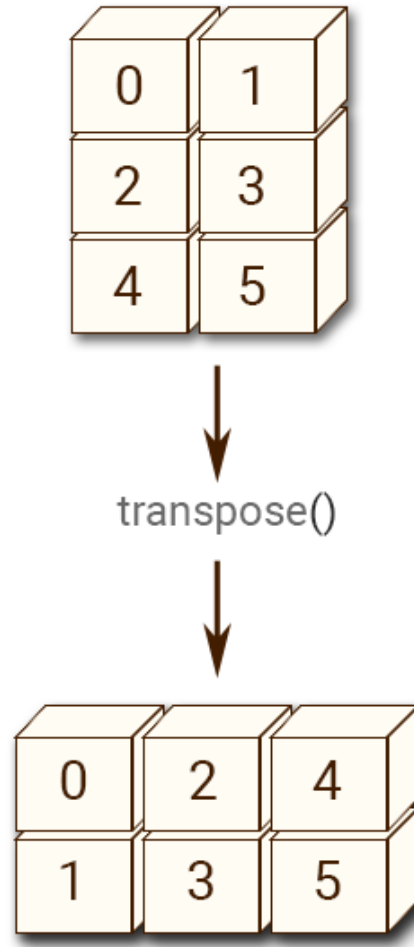


= **a[:, None]**



II. Array manipulation

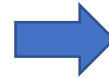
- transpose-like 메서드
 - .T : transpose, 행열 변환
 - moveaxis() : axes 이동



II. Array manipulation

- expand & squeezing
 - **expand_dims** : 원하는 위치에 axis 추가
 - squeeze : 원하는 axis 를 제거

```
array_one = np.array([1, 2, 3])  
array_two = np.array([[1, 2, 3], [4, 5, 6]])  
array_one_expand = np.expand_dims(array_one, axis=0)  
array_two_expand = np.expand_dims(array_two, axis=0)
```



One dimensional array:

[1 2 3]

shape: (3,)

One dimensional array expanded:

[[1 2 3]]

shape: (1, 3)

Two dimensional array:

[[1 2 3]

[4 5 6]]

shape: (2, 3)

Two dimensional array expanded:

[[[1 2 3]

[4 5 6]]]

shape: (1, 2, 3)

II. Array manipulation

- expand & squeezing
 - expand_dims : 원하는 위치에 axis 추가
 - **squeeze** : 원하는 axis 를 제거

```
array_one_squeez = np.squeeze(array_one_expand, axis=0)  
array_two_squeez = np.squeeze(array_two_expand, axis=0)
```



```
array_one_expand:  
[[1 2 3]]  
shape: (1, 3)
```

```
array_one_squeeze:  
[1 2 3]  
shape: (3,)
```

```
array_two_expand:  
[[[1 2 3]  
  [4 5 6]]]  
shape: (1, 2, 3)
```

```
array_two_squeez:  
[[1 2 3]  
 [4 5 6]]  
shape: (2, 3)
```

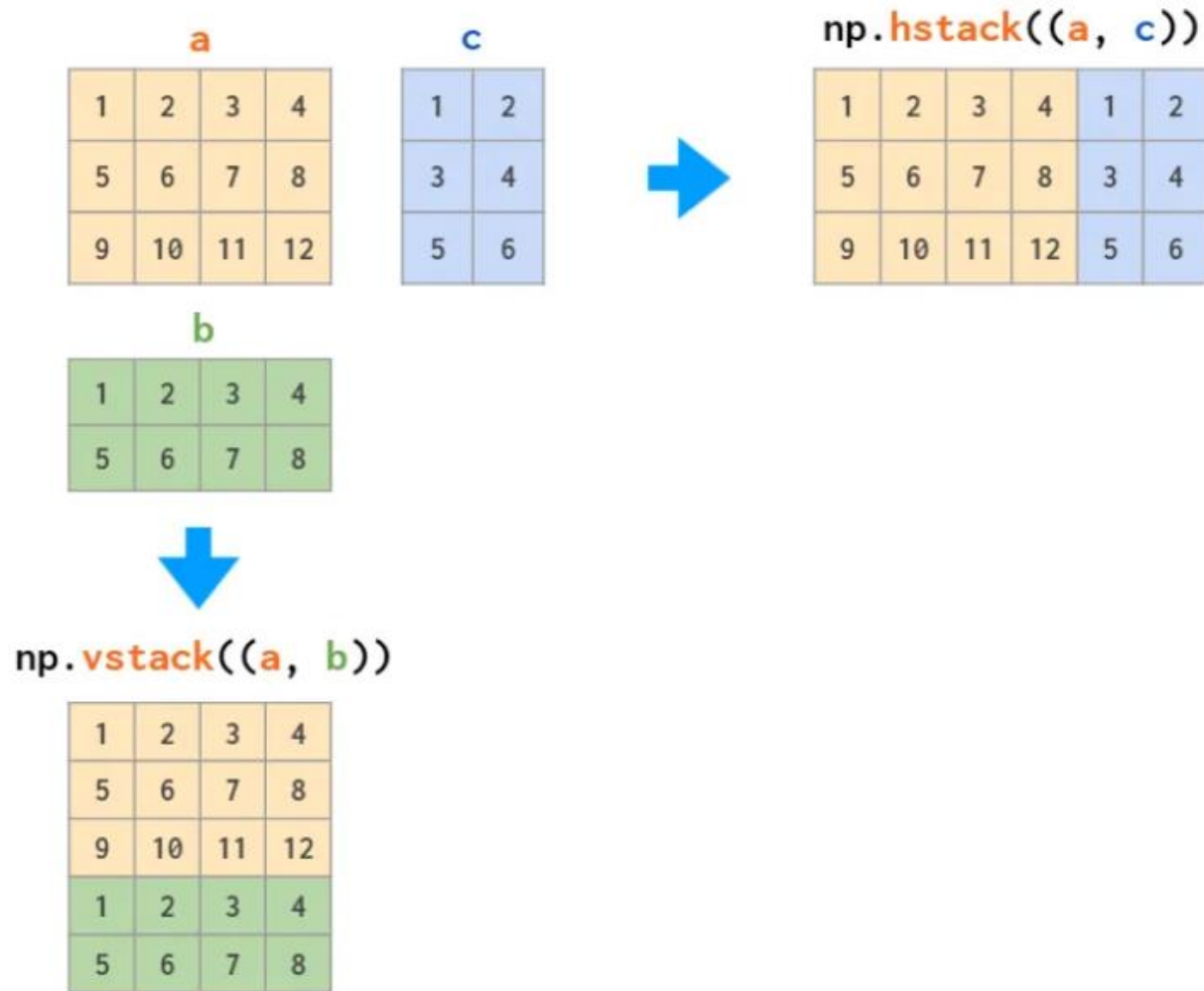

II. Array manipulation

- Joining

- `hstack()` : array 들을 가로 방향(column 방향) 으로 쌓음
- `vstack()` : array 들을 세로 방향(row 방향) 으로 쌓음
- `concatenate()` : array 들을 쌓을 때 축(axis)를 지정할 수 있음 (이미 존재하는 axis 여야 함)
- `stack()` : array 들을 쌓을 때 새로운 축(axis) 를 생성함.

II. Array manipulation

- Joining



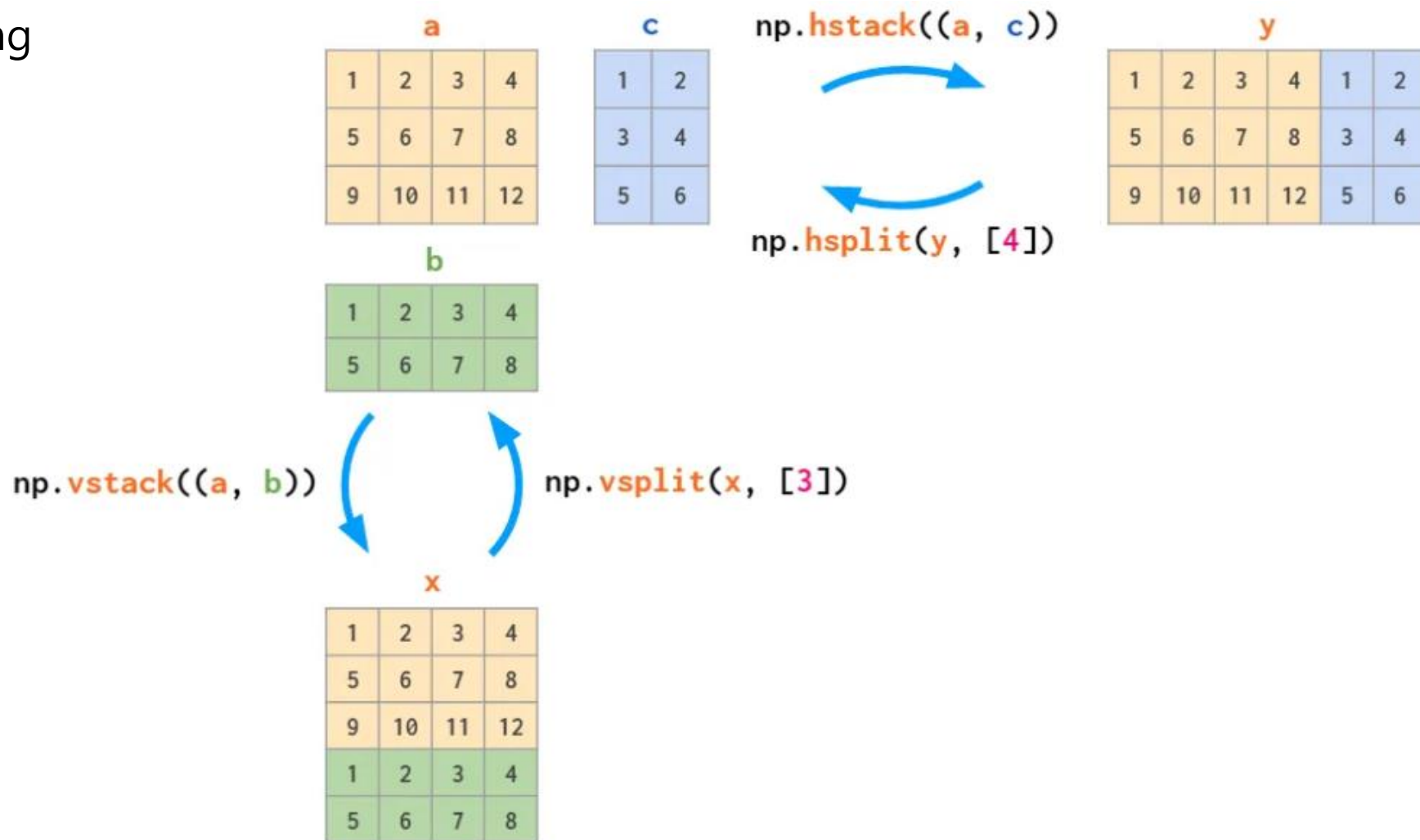
II. Array manipulation

- Splitting

- `hsplit()` : array 들을 가로 방향(column 방향) 으로 쪼갬
- `vsplit()` : array 들을 세로 방향(row 방향) 으로 쪼갬
- `split()` : array 들을 쪼갤 때 축(axis)를 지정할 수 있음 (이미 존재하는 axis 여야 함)

II. Array manipulation

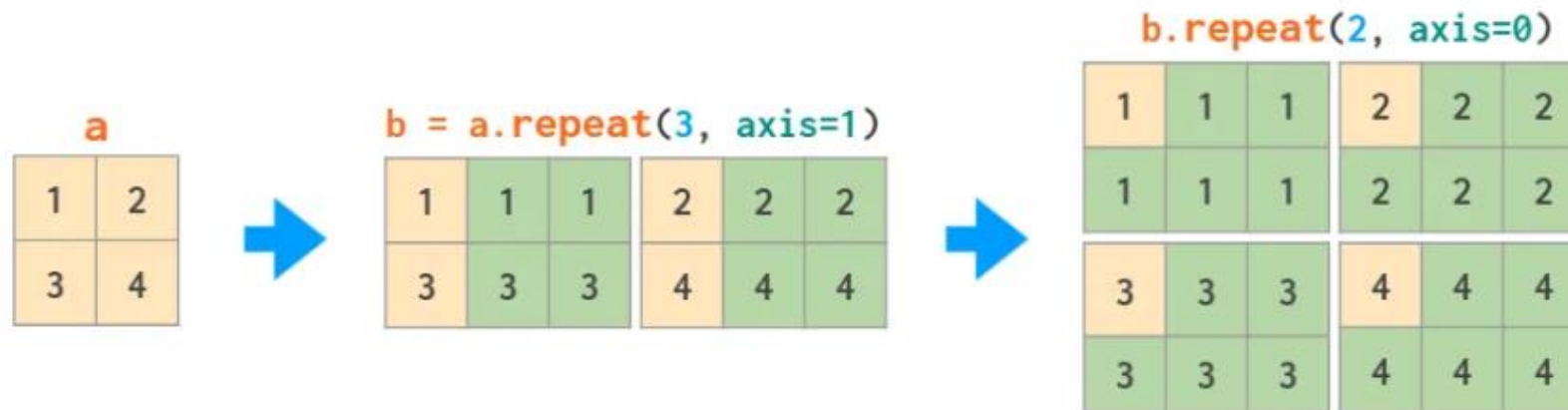
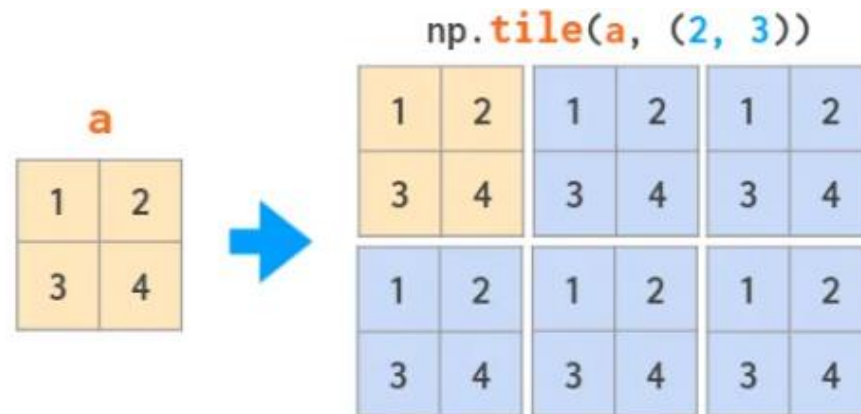
- Splitting



II. Array manipulation

■ Repetition

- `tile()` : 원하는 반복횟수 만큼 copy and paste
- `repeat()` : 원하는 축(axis) 방향으로 각 원소를 복제



III. Array Broadcasting

- Broadcasting 개념
- Broadcasting 방식
- Broadcasting 룰

Ⅲ. Array Broadcasting

- Broadcasting 개념
 - Numpy 고유의 자동화 계산 방식
 - 다른 shape 의 array 들을 적합(match) 시켜서 계산을 수행하는 것
 - 연산 속도 향상 목적(looping 회피)
 - C 로 compile 됨

Ⅲ. Array Broadcasting

- Broadcasting 개념

A 가 $(2, 2)$ matrix, x 가 스칼라 값일 때, 아래와 같이 x 가 확장된다.

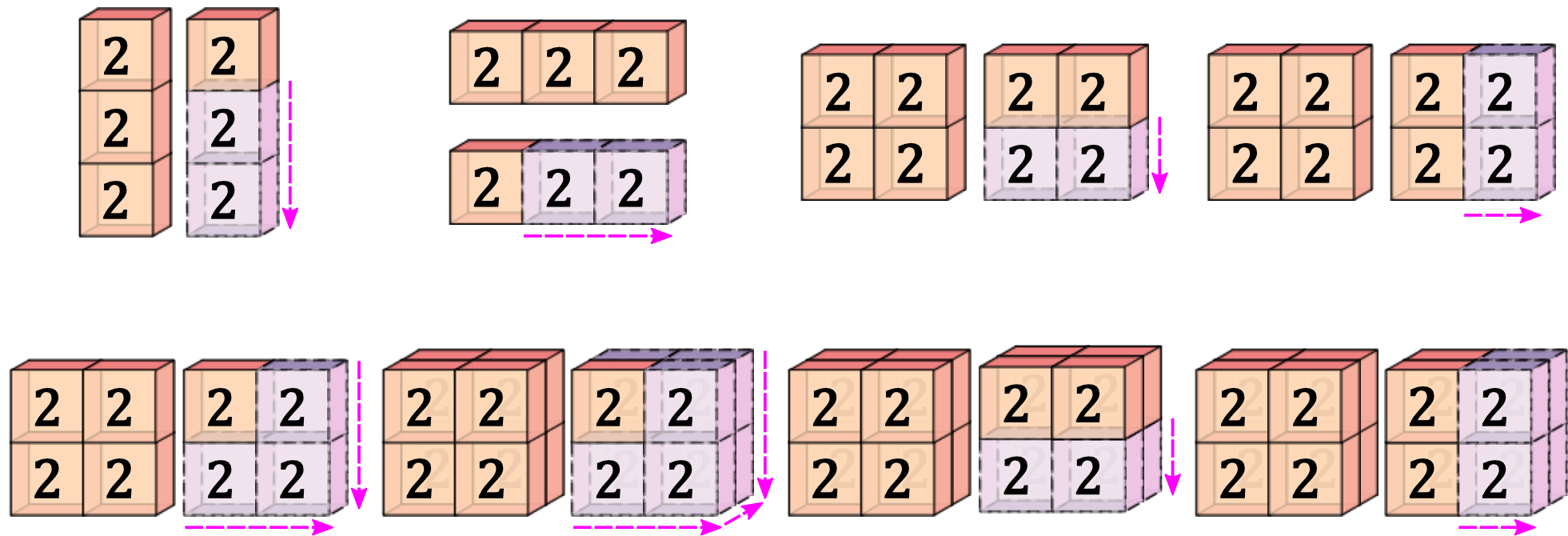
$$Ax = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} 2$$

$$Ax = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Ⅲ. Array Broadcasting

- Broadcasting 개념

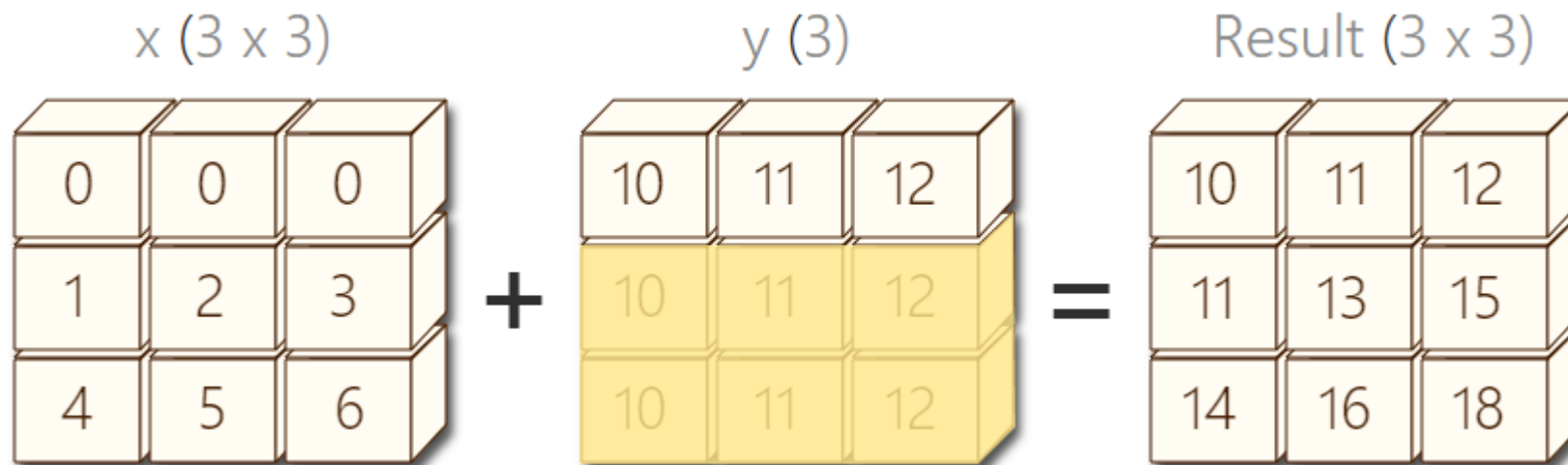
Valid broadcasting operations



Ⅲ. Array Broadcasting

- Broadcasting 개념

- 아래 그림과 같이 서로 다른 차원의 array 도 연산 가능 (Broadcasting rule 을 만족하면)



III. Array Broadcasting

- Broadcasting 여러 사용방식

$(3, 3)$

1	2	3
4	5	6
7	8	9

$*$

$(3,)$ or $(1, 3)$

-1	0	1
-1	0	1
-1	0	1

$=$

$(3, 3)$

-1	0	3
-4	0	6
-7	0	9

multiplying several
columns at once

$(3, 3)$

1	2	3
4	5	6
7	8	9

$/$

$(3, 1)$

3	3	3
6	6	6
9	9	9

$=$

$(3, 3)$

.3	.7	1.
.6	.8	1.
.8	.9	1.

row-wise
normalization

$(3,)$ or $(1, 3)$

1	2	3
1	2	3
1	2	3

$*$

$(3, 1)$

1	1	1
2	2	2
3	3	3

$=$

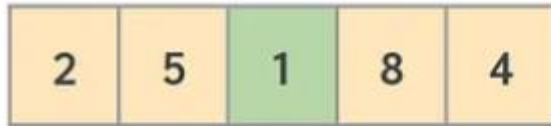
$(3, 3)$

1	2	3
2	4	6
3	6	9

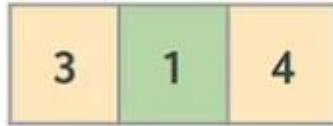
outer product

Ⅲ. Array Broadcasting

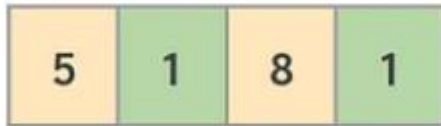
- Broadcasting 룰



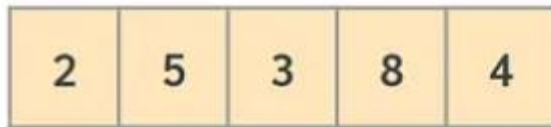
a.shape



b.shape



c.shape



result.shape

Rule1. 차원이 부족한 경우 1로 채운다.

Rule2. 동일 차원의 shape 이 다르면 shape 이 1인 것을 더 큰 수로 변경 (stretch)

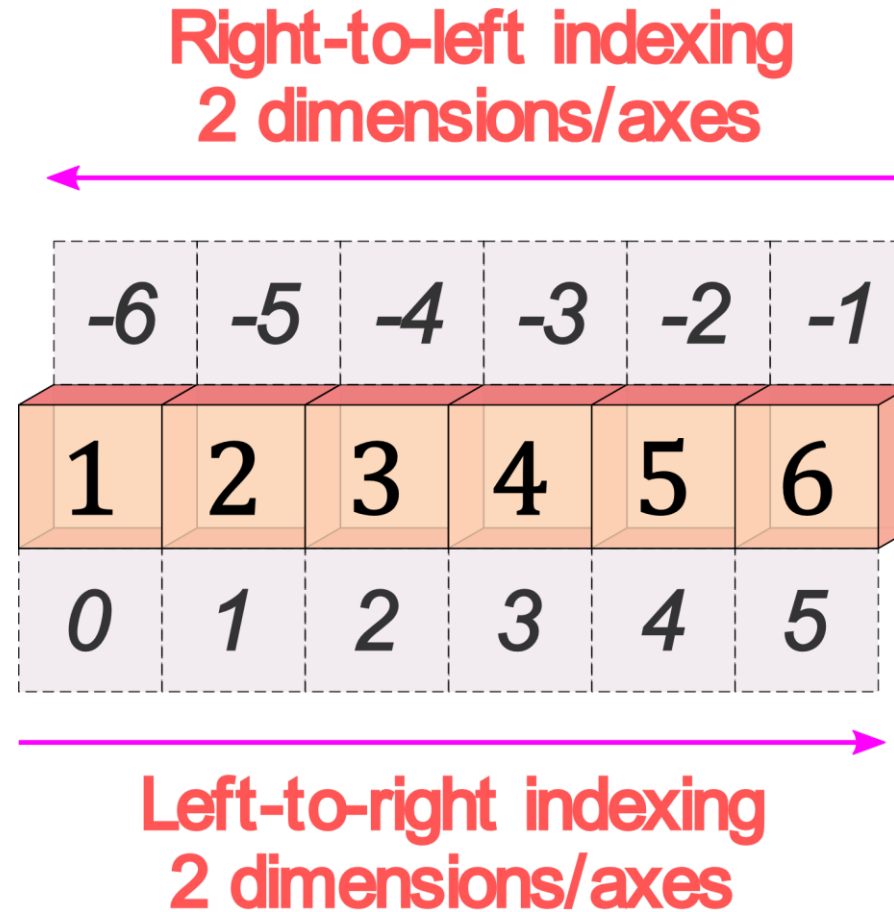
Rule3. shape 이 같지 않고 1인 경우도 아니면 **Error**

IV. **Array Indexing**

- 1차원 array indexing
- 2차원 array indexing
- Boolean(Masking) indexing

IV. Array Indexing

- 1차원 array indexing



IV. Array Indexing

- 1차원 array slicing

- $a[s:]$: s 인덱스부터 끝까지 Slicing
- $a[:s]$: 처음부터 $s-1$ 인덱스까지 Slicing
- $a[s:e:i]$: s 인덱스부터 $e-1$ 인덱스까지 i 간격으로 Slicing 해서 가져옴

$a[s:]$



$a[:s]$

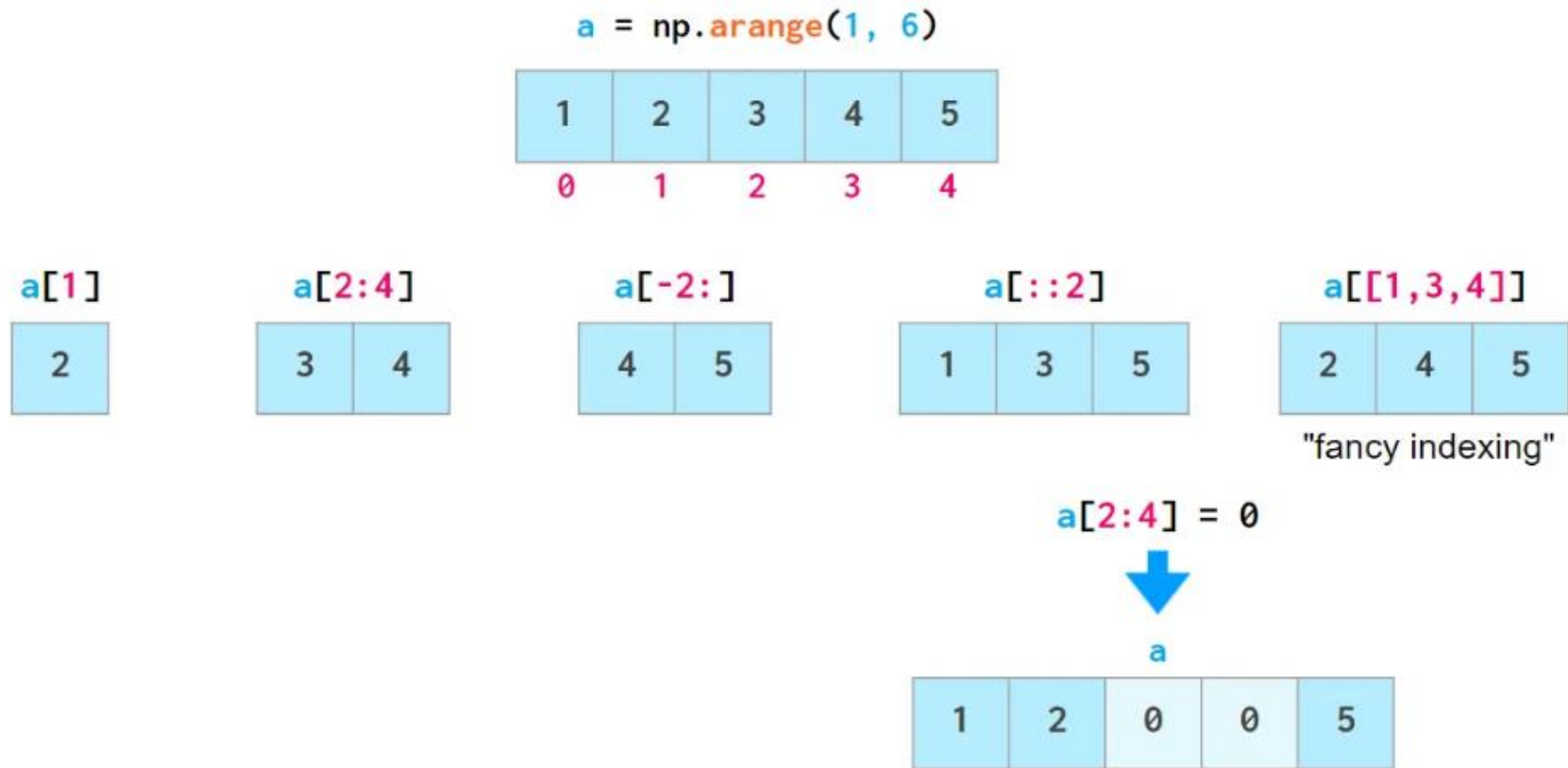


$a[s:e]$



IV. Array Indexing

- 1차원 array indexing & slicing



IV. Array Indexing

- 2차원 array indexing

0	(0,0) 1	(0,1) 2	(0,2) 3
1	(1,0) 4	(1,1) 5	(1,2) 6
2	(2,0) 7	(2,1) 8	(2,3) 9
	0	1	2

→
Left-to-right indexing
2 dimensions/axes

Right-to-left indexing
2 dimensions/axes

	-3	-2	-1
0	(0,-3) 1	(0,-2) 2	(0,-1) 3
1	(1,-3) 4	(1,-2) 5	(1,-1) 6
2	(2,-3) 7	(2,-2) 8	(2,-1) 9

IV. Array Indexing

- 2차원(이상) array indexing
 - [] 안에 콤마(,) 를 사용하는 방법 → view
 - [][]... 식으로 차원별로 '[' ']'를 사용하는 방법 → copy

```
array_two = np.arange(1,10).reshape((3,3))
```

```
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]
```

```
array_two[1,1]  
array_two[1][1]
```

IV. Array Indexing

- 2차원 array indexing & slicing

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

a[1,2]			
1	2	3	4
5	6	7	8
9	10	11	12

a[1,:]			
1	2	3	4
5	6	7	8
9	10	11	12

| **= a[1]** | | | |

a[:,2]			
1	2	3	4
5	6	7	8
9	10	11	12

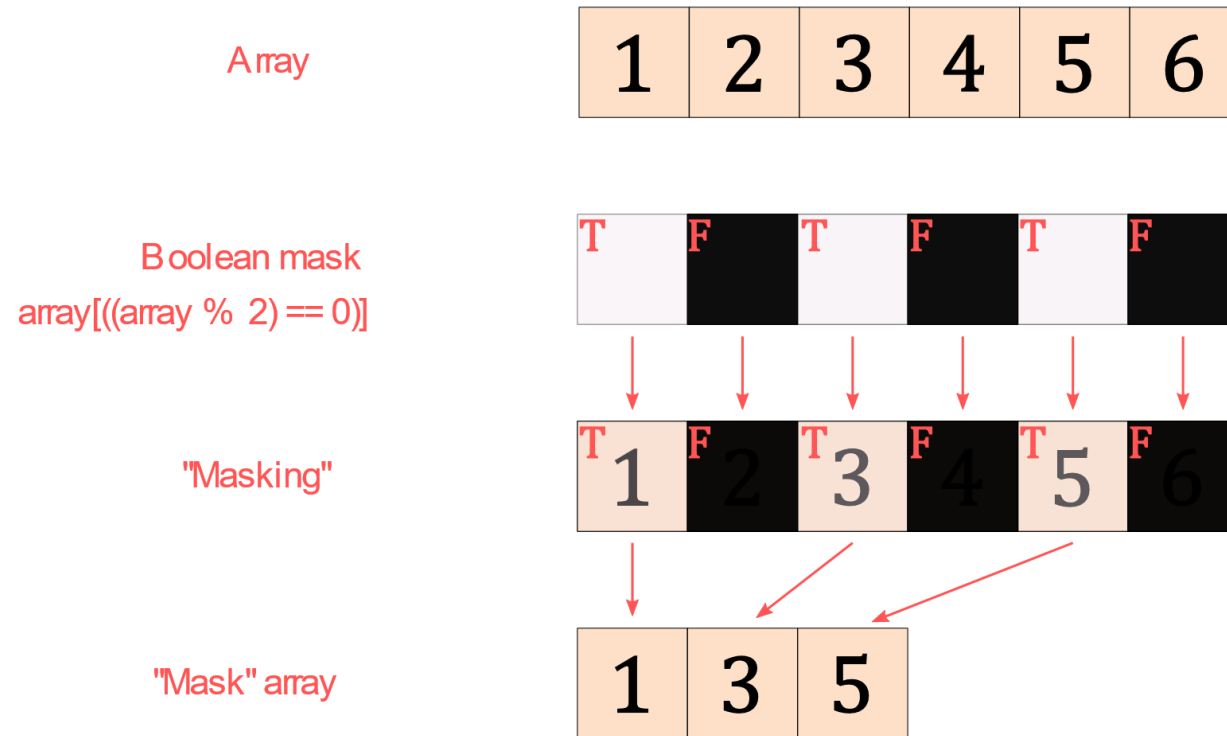
a[:,1:3]			
1	2	3	4
5	6	7	8
9	10	11	12

a[-2:,-2:]			
1	2	3	4
5	6	7	8
9	10	11	12

a[::2,1::2]			
1	2	3	4
5	6	7	8
9	10	11	12

IV. Array Indexing

- Boolean indexing or "masking"



IV. Array Indexing

- Boolean indexing or “masking”

- Boolean arrays : True / False 값으로 채워진 array. 다른 array 의 element 들을 선택하는데 사용됨.
- mask 라고도 칭함

```
array = np.arange(12)
mask = array > 6
subarray = array[mask]

print(f'Array: \n{array}\n')
print(f'Mask or Boolean array: \n{mask}\n')
print(f'sub-array: \n{}\n')
```

```
Array:
[ 0  1  2  3  4  5  6  7  8  9 10 11]

Mask or Boolean array:
[False False False False False False True True True True True]

sub-array:
[ 7  8  9 10 11]
```

IV. Array Indexing

- Boolean indexing or “masking”

- 비교 연산자와 and/or/not 을 결합하여 복잡한 조건문을 생성 가능
- and/or/not : &/|/!

```
print(f'array:\n{array}\n')
print(f'1st:\n{array[(2 == array) | (array > 9)]}\n')
print(f'2nd:\n{array[((array % 2) == 0) | (array > 9)]}\n')
print(f'3rd:\n{array[(2 != array) & (array != 7) & (array != 10)]}\n')
```

```
Array:
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

```
1st:
[ 2 10 11]
```

```
2nd:
[ 0  2  4  6  8 10 11]
```

```
3rd:
[ 0  1  3  4  5  6  8  9 11]
```

V. **Random number generation**

- Random sampling
- Setting a seed
- histogram

V. Random number generation

- Random Sampling (updated)

- version 1.17.0 부터 random generator 기능이 업데이트 됨
- `np.random.random` → `np.random.default_rng.random`
- 하지만, `np.random` 을 강의에서는 `np.random` 을 사용

```
from numpy import random  
sample = random.random((3,3))
```



```
from numpy.random import default_rng  
rng = default_rng()  
sample = rng.random((3,3))
```


V. Random number generation

- Basic Random Sampling

- **randint()** : 범위내의 정수 값을 무작위로 추출
- **random()** : 원하는 형태의 $[0, 1)$ 사이의 실수를 무작위로 추출
- **choice()** : 1차원 array 에서 무작위로 추출

```
a = np.random.randint(10)
b = np.random.randint(10, size=5)
print(a)
print(b)
```

[output]

```
8
[9 3 3 5 8]
```

V. Random number generation

■ Basic Random Sampling

- `integers()` : 범위내의 정수 값을 무작위로 추출
- **`random()` : 원하는 형태의 `[0, 1)` 사이의 실수를 무작위로 추출**
- `choice()` : 1차원 array 에서 무작위로 추출

```
a = np.random.random()
b = np.random.random((3,))
c = np.random.random((3,3))
print(f'a\n')
print(f'b\n')
print(f'c')
```

[output]

```
0.6419758158608625
[0.42937336 0.52477446 0.02976526]
[[0.3604225 0.88741889 0.07464158]
 [0.36458258 0.75476422 0.26216883]
 [0.69558381 0.49518423 0.77079096]]
```

¹V. Random number generation

■ Basic Random Sampling

- `integers()` : 범위내의 정수 값을 무작위로 추출
- `random()` : 원하는 형태의 $[0, 1)$ 사이의 실수를 무작위로 추출
- **`choice()` : 1차원 array 에서 무작위로 추출**

```
array = np.arange(10)
a = np.random.choice(array)
b = np.random.choice(array, 5)
c = np.random.choice(array, 5, replace=False)

print(f'a\n')
print(f'b\n')
print(f'c')
```

[output]

1

[2 7 2 1 6]

[2 8 3 6 9]

V. Random number generation

- Setting a Seed

- 무작위 추출의 재연을 위해서 Seed 값을 부여할 수 있다.
- seed 값은 임의의 정수를 택하면 된다.

```
np.random.seed(9320)
a = np.random.randint(10, size=5)
b = np.random.randint(10, size=5)
c = np.random.randint(10, size=5)
```

sample again!

```
np.random.seed(9320)
a = np.random.randint(10, size=5)
b = np.random.randint(10, size=5)
c = np.random.randint(10, size=5)
```

[output]

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

sample again!

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

V. Random number generation

- Histogram

- 주어진 값으로부터 도수분포표를 작성해 준다.
- 계급값, 범위, 계급 가중치, 확률밀도 등의 인자를 지정 가능

```
array = np.arange(10)
data = np.random.choice(10, size=100)
hist_1 = np.histogram(data)
hist_2 = np.histogram(data, bins=np.arange(10))
```

[output]

```
(array([ 7,  6, 11, 10, 11,  8, 14,  9, 10, 14], dtype=int64),
array([0. , 0.9, 1.8, 2.7, 3.6, 4.5, 5.4, 6.3, 7.2, 8.1, 9. ]))

(array([ 7,  6, 11, 10, 11,  8, 14,  9, 24], dtype=int64),
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

VI. Basic Linear Algebra

- Basic vector operations
- Basic matrix operations
- eigendecomposition
- svd

VI. Basic Linear Algebra

- Basic vector operations

```
x, y = np.arange(3), np.arange(4,7)
alpha, beta = 2, 3

print(f"Vector x: {x}, vector y: {y}\n")
print(f"Vector addition: {x + y}\n")
print(f"Vector scalar-multiplication: {x * alpha}\n")
print(f"Linear combinations of vectors: {x*alpha + y*beta}\n")
print(f"Vector-vector multiplication: dot product: {x @ y}\n")
```

Vector x: [0 1 2], vector y: [4 5 6]

Vector addition: [4 6 8]

Vector scalar-multiplication: [0 2 4]

Linear combinations of vectors: [12 17 22]

Vector-vector multiplication: dot product: 17

VI. Basic Linear Algebra

- Basic matrix operations

```
A, B, C = np.arange(1, 10).reshape(3,3), np.arange(11, 20).reshape(3,3), np.random.rand(3,3)
```

```
print(f"Matrix A:\n{A}\n")  
print(f"Matrix B:\n{B}\n")  
print(f"Matrix-matrix addition:\n{A+B}\n")  
print(f"Matrix-scalar multiplication:\n{A*alpha}\n")
```

[output]

Matrix A:

```
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]
```

Matrix B:

```
[[11 12 13]  
 [14 15 16]  
 [17 18 19]]
```

Matrix-matrix addition:

```
[[12 14 16]  
 [18 20 22]  
 [24 26 28]]
```

Matrix-scalar multiplication:

```
[[ 2  4  6]  
 [ 8 10 12]  
 [14 16 18]]
```


VI. Basic Linear Algebra

- Basic matrix operations

```
A, B, C = np.arange(1, 10).reshape(3,3), np.arange(11, 20).reshape(3,3), np.random.rand(3,3)

print(f"Matrix-vector multiplication: dot product:\n{A @ x}\n")
print(f"Matrix-matrix multiplication: dot product:\n{A @ B}\n")
print(f"Matrix inverse:\n{np.linalg.inv(C)}\n")
print(f"Matrix transpose:\n{A.T}\n")
print(f"Hadamard product: \n{A * B}")
```

Matrix-vector multiplication: dot product:
[8 17 26]

Matrix-matrix multiplication: dot product:
[[90 96 102]
 [216 231 246]
 [342 366 390]]

Matrix inverse:
[[-21.32169045 -3.8131569
 38.56457317]
 [4.50732439 3.31820901 -
 12.24388988]
 [7.14926296 -0.18735867 -
 8.81170042]]

Matrix transpose:
[[1 4 7]
 [2 5 8]
 [3 6 9]]

Hadamard product:
[[11 24 39]
 [56 75 96]
 [119 144 171]]

VI. Basic Linear Algebra

- eigendecomposition

```
eigen_values, eigen_vectors = np.linalg.eig(C)
print(f"Matrix eigenvalues:\n{eigen_values}\n\nMatrix
eigenvectors:\n{eigen_vectors}")
```

Matrix eigenvalues:

```
[ 1.68949164 -0.03099435  0.20589414]
```

Matrix eigenvectors:

```
[[-0.49129322 -0.93332275  0.55829177]
 [-0.79415823  0.21672628 -0.77249178]
 [-0.35769216  0.28624879  0.30259999]]
```

VI. Basic Linear Algebra

- singular value decomposition

```
U, S, T = np.linalg.svd(C)
```

```
print(f'Left orthogonal matrix C:\n{np.round(U, 2)}\n')  
print(f'Singular values diagonal matrix C:\n{np.round(S, 2)}\n')  
print(f'Right orthogonal matrix C:\n{np.round(T, 2)}')
```

Left orthogonal matrix C:

```
[[-0.54 -0.69  0.48]  
 [-0.75  0.65  0.09]  
 [-0.38 -0.31 -0.87]]
```

Singular values diagonal matrix C:

```
[1.89 0.27 0.02]
```

Right orthogonal matrix C:

```
[[-0.36 -0.57 -0.74]  
 [ 0.07  0.77 -0.63]  
 [-0.93  0.28  0.23]]
```