



- 제작 : 나



Anaconda 설치

- 파이썬 통합 개발환경
- jupyter notebook 지원
- 라이브러리 패키지 관리 및 환경 설정을 쉽게 해줌
- <https://www.anaconda.com/distribution>

Anaconda Installers

Windows

Python 3.9

64-Bit Graphical Installer (594 MB)

32-Bit Graphical Installer (488 MB)

MacOS

Python 3.9

64-Bit Graphical Installer (591 MB)

64-Bit Command Line Installer (584 MB)

64-Bit (M1) Graphical Installer (316 MB)

64-Bit (M1) Command Line Installer (305 MB)

Linux

Python 3.9

64-Bit (x86) Installer (659 MB)

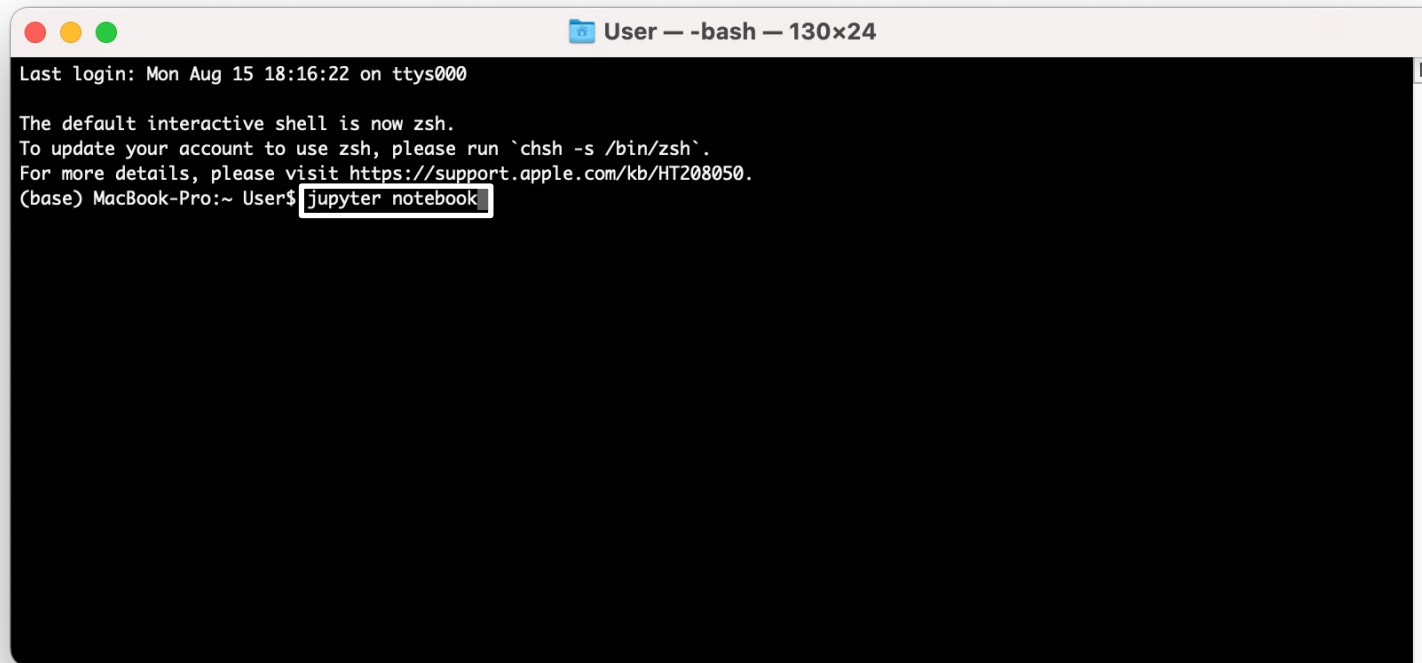
64-Bit (Power8 and Power9) Installer (367 MB)

64-Bit (AWS Graviton2 / ARM64) Installer (568 MB)

64-bit (Linux on IBM Z & LinuxONE) Installer (280 MB)

Jupyter notebook 실행

- Window : 명령 프롬프트 ➡ jupyter notebook
- Mac : 터미널 ➡ jupyter notebook



```
User — -bash — 130x24
Last login: Mon Aug 15 18:16:22 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) MacBook-Pro:~ User$ jupyter notebook
```

Jupyter notebook 커널 실행



Quit

Logout

Files

Running

Clusters

Select items to perform actions on them.

Upload

New



Notebook:

Python 3

Python 3.8.8 64-bit ('base': conda)

Other:

Text File


Folder

Terminal












Python 3 클릭!




코드 실행

 jupyter Untitled Last Checkpoint: 1분 전 (unsaved changes)

FileEditViewInsertCellKernelWidgetsHelp

 Run

Code



```
In [1]: print("Hello World!")  
Hello World!
```

Shift + Enter를 눌러서 실행!

데이터 처리 및 분석을 위한 파이썬 모듈

- Numpy : 다차원 배열 생성 및 조작
- Pandas : Dataframe 생성 및 조작
(계산을 매우 편하게 함 = 시간 단축)
- Scipy : 과학 및 공학 연산 → 통계 분석
- Matplotlib : 시각화

모듈 설치

- 각자 사용하는 환경에 따라 코드를 입력
- Jupyter notebook의 경우 3번째 코드

```
# 모듈설치
```

```
### Windows 명령 프롬프트(cmd), Powershell
```

```
pip install numpy scipy pandas matplotlib --user
```

```
### Mac OS 터미널
```

```
pip3 install numpy scipy pandas matplotlib
```

```
### Jupyter notebook
```

```
!pip install numpy scipy pandas matplotlib
```


모듈 실행 방법

- `import` : 모듈을 불러오는 코드
ex) `import matplotlib` → 시각화 모듈 `matplotlib`을 import
- `import A as B` : `A` 모듈을 `B` 라는 이름으로 import
ex) `import numpy as np` → 이제부터 `numpy`는 `np`로 쓰임
→ `np.array([1,1],[2,2])`

Numpy

- Numerical python의 약자
- 다차원 배열을 쉽고 빠르게 처리
- 숫자로 이루어진 Table형 데이터를 다룰 때 주로 사용
- 같은 종류의 data type을 원소로 함
- 차원에 따라 데이터의 이름을 다르게 부름
 - 1차원 : vector
 - 2차원 : matrix
 - 3차원 : tensor ...

numpy를 배워야 하는 이유는?

- 계산을 간단히 하기 위해
특히 반복문을 피하기 위함

```
tmp = [j for j in range(10)]  
tmp1 = []  
for i in range(len(tmp)):  
    if i>3:  
        tmp1.append(i)  
tmp1
```

[4, 5, 6, 7, 8, 9]

```
xx = np.array(tmp)  
list(xx[xx>3])
```

[4, 5, 6, 7, 8, 9]

ndarray

- ndarray
= numeric + dimension + array
- 우리가 흔히 아는 행렬,
python에서는 다차원배열로 부른다.
- numpy.array() 함수를 통해 생성
- 대괄호는 dimension(차원) 을 의미함
→ b는 1차원의 array
- tolist() 함수를 통해 list 변환 가능

```
# numpy 호출
import numpy as np
a = [1,2,3,4,5]
print(a, type(a))
b = np.array(a)
print(b, type(b))
```

```
[1, 2, 3, 4, 5] <class 'list'>
[1 2 3 4 5] <class 'numpy.ndarray'>
```

```
b_list=b.tolist(); print(type(b_list))
```

```
<class 'list'>
```

ndarray의 활용

- c는 list, d는 ndarray
c 안의 첫번째 list 는 첫번째 행,
두번째 list는 두번째 행에 들어간다

```
c = [[1,2,3,4,5],[6,7,8,9,10]]  
print(c, type(c))  
d = np.array(c)  
print(d, type(d))
```

```
[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]] <class 'list'>  
[[ 1  2  3  4  5]  
 [ 6  7  8  9 10]] <class 'numpy.ndarray'>  
↳ 대괄호가 2개니 애는 2차원
```

- d[m,n]를 통하여 선 ndarray 안의 원소를
선택할 수 있다. (단, m,n는 정수)

ex) d[1,2] → 1행, 2열의 8을 선택
python에서는 0부터 순서를 센다

```
print(d)  
print(d[1,2])
```

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10

8

ndarray

- # dtype = data type, 'U' = 유니코드 약자 = 문자열
- # 대괄호 [] = 차원
- # object = 객체
- # size: 총 원소의 개수
- # shape: 차원구조
- # ndim : 차원 수
- # dtype: 원소의 data type

```
a = "abcd"
b = list(a)
c = tuple(a)
d = dict((( 'a', 1), ( 'b', 2)))
e = set(a)

print(a, np.array(a).size, np.array(a).shape, np.array(a).ndim, np.array(a).dtype)
print(b, np.array(b).size, np.array(b).shape, np.array(b).ndim, np.array(b).dtype)
print(c, np.array(c).size, np.array(c).shape, np.array(c).ndim, np.array(c).dtype)
print(d, np.array(d).size, np.array(d).shape, np.array(d).ndim, np.array(d).dtype)
print(e, np.array(e).size, np.array(e).shape, np.array(e).ndim, np.array(e).dtype)
```

```
abcd 1 () 0 <U4
['a', 'b', 'c', 'd'] 4 (4,) 1 <U1
('a', 'b', 'c', 'd') 4 (4,) 1 <U1
{'a': 1, 'b': 2} 1 () 0 object
{'a', 'b', 'c', 'd'} 1 () 0 object
```

arange

- arange
= array + range
- np.arange([start], stop, [step], dtype)
- start: 시작
stop: 끝 (미포함)
step: 다음 숫자와의 차이
dtype: data type

```
a = np.arange(10)
b = np.arange(1,10.)
c = np.arange(1,10,0.5)
d = np.arange(1,10,3,dtype='float')
print(a)
print(b)
print(c)
print(d)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

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

```
[1. 1.5 2. 2.5 3. 3.5 4. 4.5 5. 5.5 6. 6.5 7. 7.5 8. 8.5 9. 9.5]
```

```
[1. 4. 7.]
```

linspace

일차함수 위의 규칙적인 점들

- linspace = linear + space
- np.linspace(start, stop, [num, endpoint, retstep, dtype])
- start: 시작; stop: 끝
num: 숫자 개수 [50]
endpoint: stop의 표시 여부 [True]
✓ retstep: step의 표시 여부 [False]
dtype: data type

retstep=True로 할 경우에만 자료형이 ndarray가 아니라 tuple이 되는데 왜그런거지?

```
a = np.linspace(1,10)
b = np.linspace(1,np.pi,num=5)
c = np.linspace(1,10, num=5, endpoint = False)
d = np.linspace(1, 10., num=5, retstep=True) # step 명시
print(a)
print(b)
print(c)
print(d)
```

```
[ 1.          1.18367347  1.36734694  1.55102041  1.73469388  1.91836735
 2.10204082  2.28571429  2.46938776  2.65306122  2.83673469  3.02040816
 3.20408163  3.3877551   3.57142857  3.75510204  3.93877551  4.12244898
 4.30612245  4.48979592  4.67346939  4.85714286  5.04081633  5.2244898
 5.40816327  5.59183673  5.7755102   5.95918367  6.14285714  6.32653061
 6.51020408  6.69387755  6.87755102  7.06122449  7.24489796  7.42857143
 7.6122449   7.79591837  7.97959184  8.16326531  8.34693878  8.53061224
 8.71428571  8.89795918  9.08163265  9.26530612  9.44897959  9.63265306
 9.81632653 10.         ]
[ 1.          1.53539816  2.07079633  2.60619449  3.14159265]
[ 1.   2.8  4.6  6.4  8.2]
(array([ 1.   ,  3.25,  5.5  ,  7.75, 10.   ]), 2.25) → step
```


logspace, log10

- `logspace` = `linspace`의 결과들이 나오는 `log`값들

- $10^2=100$; $10^3=1000$; $10^4=10000$
`np.linspace(2, 4, 3) = 2, 3, 4`;
2, 3, 4가 나오는 100, 1000, 10000

- `log10(100) = 2`
 $\log_{10} 100 = 2$

```
print(np.linspace(2, 4, 3))  
print(np.logspace(2, 4, 3))  
print(np.log10(100), np.log10(1000), np.log10(10000))
```

```
[2.  3.  4.]  
[ 100.  1000. 10000.]  
2.0  3.0  4.0
```

이 외의 다른 ndarray 생성 함수

```
a = np.ones((2,5))    # default dtype = float
b = np.zeros((2,5),dtype='int') # default dtype = float
c = np.full((2,3), 0.5)
d = np.eye(3, 5) # 3 by 3 단위행렬(identity)
e = np.identity(3)
```

```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

`np.ones((m, n))` → [m, n] 크기의 array를 1로 채움

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

`np.zeros((m, n))` → [m, n] 크기의 array를 0로 채움

```
[[0.5 0.5 0.5]
 [0.5 0.5 0.5]]
```

`np.full((m, n), k)` → [m, n] 크기의 array를 k로 채움

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
```

`np.eye(m, n)` → [m, n] 크기의 0행렬 + [m, m] 크기의 단위행렬

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

`np.identity(m)` → [m, m] 크기의 단위행렬

n차 정사각행렬에서 주대각선의 원소가 모두 1이고, 다른 원소는 모두 0인 행렬이다.
단위행렬 E를 임의의 행렬 A와 곱하면 행렬 A가 얻어진다.

ndarray 연산

- T: transpose (전치) 행과 열을 바꾼 행렬
- reshape: 차원 및 행렬 변환
- astype: ndarray의 data type 변환

```
a = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])  
print(a)  
b = a.T  
print(b)  
c = a.reshape(2,6)  
print(c)  
d = a.astype(float)  
print(d)
```

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

3x4

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

4x3

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

2x6

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

3x4

reshape 차원변환

- `a = np.arange(9) → [0 1 2 3 4 5 6 7 8]`
- `a.reshape(1, -1) → [[0 1 2 3 4 5 6 7 8]]`

`a.reshape(1,9(원소의개수))` -1의 값을 넣으면 알아서 원소의 개수를 계산하고 출력

→ `reshape(차원의 수, 행, 열)`

- `b = a.reshape((1, 1, 9))`
(1,9)의 모양을 가진 벡터가 1개
- `c = a.reshape((1, 9, 1))`
(9,1)의 모양을 가진 벡터가 1개
- `d = a.reshape((9, 1, 1))`
(1,1)의 모양을 가진 행렬이 9개의 차원으로 구성

<code>a:(9,)</code> [0 1 2 3 4 5 6 7 8]	<code>d:(9, 1, 1)</code> [[[0]]]
	[[1]]
<code>b:(1, 1, 9)</code> [[[0 1 2 3 4 5 6 7 8]]]	[[2]]
<code>c:(1, 9, 1)</code> [[[0]	[[3]]
[1]	[[4]]
[2]	[[5]]
[3]	[[6]]
[4]	[[7]]
[5]	[[8]]
[6]	
[7]	
[8]]]	

ndarray 연산

- `a = np.full((2, 4),5)`
- `a + [2]`
- `a - [1, 2, 3, 4]`
- `a * ([1],[2])`
- `a / [a+1]`
- `a % [1, 2 ,3, 4]`
- `a.tolist() + [1, 2, 3]`

- `aa = np.arange(10).reshape(2,5)`
`bb = np.arange(10).reshape(5,2)`
- `aa @ bb`
`= np.dot(aa,bb)` # 내적

함 해보세여~

indexing, slicing, step (1차원)

- `a=np.arange(1, 8)`
- `a[x]` → `a`에서 `x`번째에 있는 수
- `a[x:y]` → `a`에서 `x`번째 부터 `y`번째 까지의 수 (`y` 미포함)
- `a[x:y:n]` → `a`에서 `x`번째 부터 `n`간격을 갖는 `y`번째 까지의 수 (`y`미포함)

```
a=np.arange(1, 8)  
print(a)
```

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

```
print(a[1:6])  
print(a[1:6:2])  
print(a[6:1:-1])
```

```
[2 3 4 5 6]  
[2 4 6]  
[7 6 5 4 3]
```

indexing, slicing, step (2차원)

- `a=np.arange(1, 13).reshape(3,4)`
- `a[m][n]` → a에서 m행 n열

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

	0	1	2	3	
0	[1	2	3	4]
1	[5	6	7	8]
2	[9	10	11	12]

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

```
[[2 3]  
 [6 7]]
```

함 해보세여~

numpy.random

- seed : 저장소...?
- shuffle : 조합
- choice : 선택

- rand
- randn
- randint

numpy.random.seed()

```
np.random.seed(0); np.random.shuffle(x)  
x
```

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

```
np.random.seed(0); np.random.shuffle(x)  
x
```

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

```
np.random.seed(0); np.random.shuffle(x)  
x
```

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

```
np.random.seed(0); np.random.shuffle(x)  
x
```

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

```
np.random.seed(0); np.random.shuffle(x)  
x
```

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

```
np.random.seed(0); np.random.shuffle(x)  
x
```

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

```
np.random.seed(0); np.random.rand(5)
```

```
array([0.5488135, 0.71518937, 0.60276338, 0.54488318, 0.4236548 ])
```

```
np.random.seed(0); np.random.rand(5)
```

```
array([0.5488135, 0.71518937, 0.60276338, 0.54488318, 0.4236548 ])
```

```
np.random.seed(5); np.random.rand(5)
```

```
array([0.22199317, 0.87073231, 0.20671916, 0.91861091, 0.48841119])
```

seed의 이해

random 함수를 활용하여 무작위의 경우의 수를 저장
같은 seed에서는 항상 같은 패턴을 가지게 됨

seed

어떤 세트의 랜덤한 숫자들을 생성해두고,
seed가 선언된 후의 rand method에서
미리 생성한 세트의 난수를 리턴한다

numpy.random.랜덤

- `numpy.random.rand(차원, [size, dtype])`
: $[0, 1)$ 에서 무작위 숫자 size개
- `numpy.random.randn(차원, [size, dtype])`
: 표준정규분포에서 무작위 숫자 size개
- `numpy.random.randint(low, [high, size, dtype])`
: $[low, high)$ 에서 무작위 정수 size개

numpy 연산

- axis = 0: column 끼리의 연산, axis = 1: row끼리의 연산

- np.cumsum() : 누적합
- np.cumprod() : 누적곱
- np.diff() : 옆 숫자와 차이
등등...

엄청 많으니
궁금하면 스스로 더 찾아보기!

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

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

```
np.cumsum(x)
```

```
array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])
```

```
np.cumprod(x)
```

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

```
np.diff(np.cumsum(x))
```

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

numpy 1차원 변환

- ravel()
- flatten()
- reshape(-1)

```
a1=np.arange(1,5).reshape(2,2)
a2=a1.ravel()
a3=a1.flatten()
a4=a1.reshape(-1)
print(a1, '\n')
print(a2, '\n')
print(a3, '\n')
print(a4)
```

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

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

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

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

- 어떤 차이가 있나요??
→ 원본값을 바꿔보자!

```
a1[0][0] = 0
print(a1, '\n')
print(a2, '\n')
print(a3, '\n')
print(a4)
```

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

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

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

→ # flatten()은 원본값이 바뀌어도 안 바뀐다!
.copy()를 해줬다고 생각하면 편하다!

array 합치기

- axis = 0: column 끼리의 연산, axis = 1: row끼리의 연산
- np.concatenate(x1, x2, axis = 0 or 1)
사슬같이 잇다

```
a1 = np.array([[1,2],[3,4]])  
a2 = np.array([[5,6],[7,8]])  
print(np.concatenate((a1, a2), axis=0), '\n')  
print(np.concatenate((a1, a2), axis=1))
```

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

행이 2개에서 4개가 되었음

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

열이 2개에서 4개가 되었음

array 반복

- axis = 0: column 끼리의 연산, axis = 1: row끼리의 연산
- x1.repeat(n, axis = 0 or 1) : x1의 원소들을 n번 repeat

```
a1 = np.array([[1,2],[3,4]])  
print(a1)  
print(a1.repeat(2))
```

이렇게 axis가 없으면 axis=0이 아니라 그냥 일차원 배열로 풀어서 계산되는듯하다

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

array 반복 및 추가

repeat이랑 조금 다르게 작동하는듯
repeat은 각 원소별로 반복해주는거고

- `np.tile(x1, (m, n))` tile은 배열 자체를 반복해주는듯
- `np.append(x1, x2)`

```
a1 = np.array([[1,2],[3,4]])  
print(np.tile(a1,2), '\n')  
print(np.tile(a1,(2,3)))
```

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

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

a1

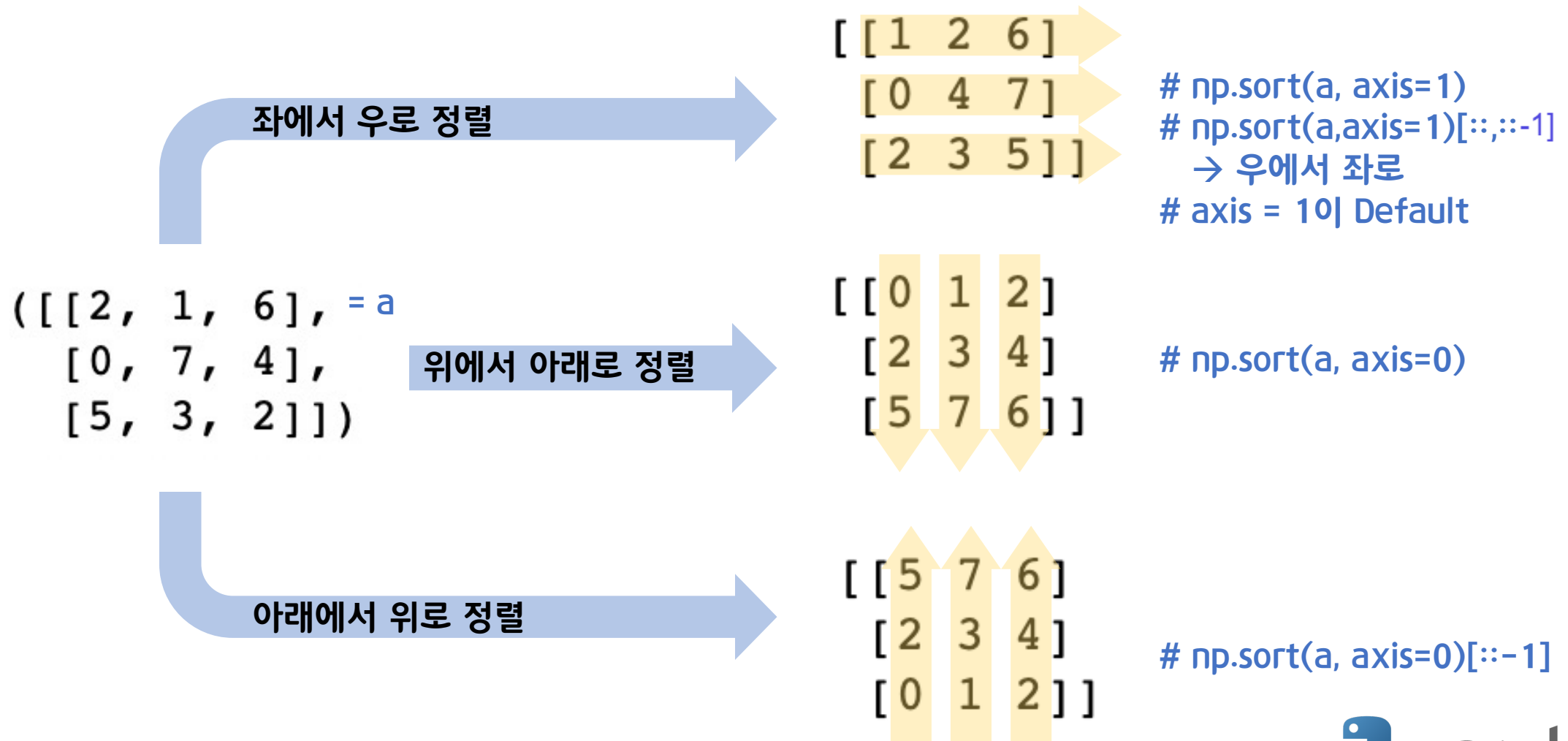
```
[[1 2 1 2 1 2]  
 [3 4 3 4 3 4]  
 [1 2 1 2 1 2]  
 [3 4 3 4 3 4]]
```

```
a1 = np.array([[1,2],[3,4]])  
a2 = np.array([[5,6],[7,8]])  
print(np.append(a1, a2))
```

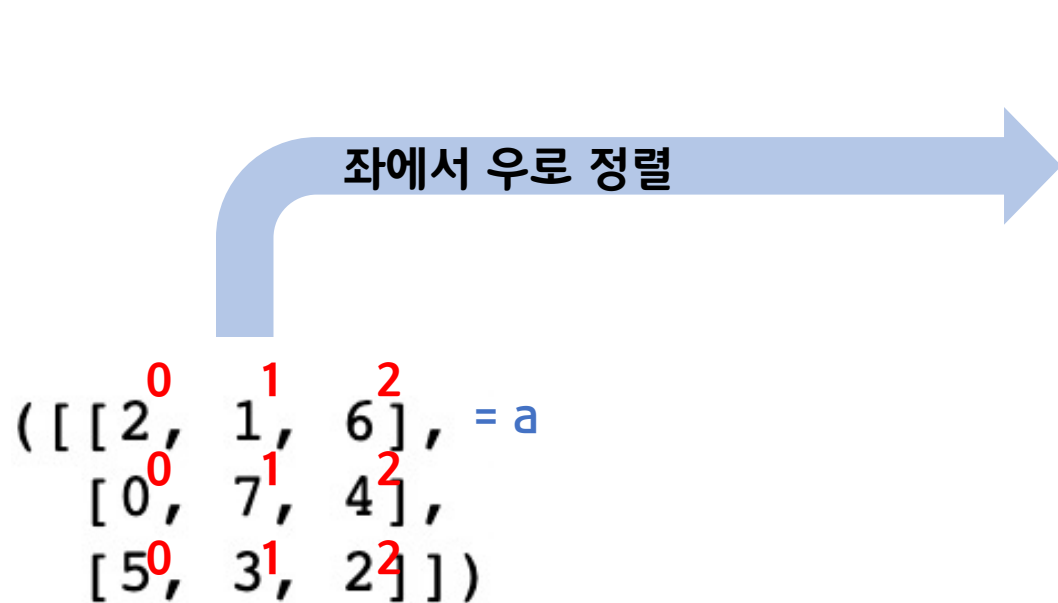
```
[1 2 3 4 5 6 7 8]
```

append는 모든 원소를 합쳐서 1차원으로 출력한다!

오름차순 내림차순



오름차순 내림차순



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

```
# np.sort(a)  
# np.sort(a)[::-,-1]  
→ 우에서 좌로  
# axis = 1이 Default
```

```
np.argsort(a)
```

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

```
# np.sort(a)로 정렬된  
출력값의 index
```

```
a.argsort()
```

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

Numpy 탐색

([[⁰2, ¹1, ²6],
 ³0, ⁴7, ⁵4],
 ⁶5, ⁷3, ⁸2]])

```
print(np.amin(a), a.min())    # 최솟값
print(np.amax(a), a.max())    # 최댓값
print(np.argmin(a), a.argmin()) # 최솟값 index
print(np.argmax(a), a.argmax()) # 최댓값 index
```

```
0 0
7 7
3 3
4 4
```

- 최댓값, 최솟값을 다음 함수를 통해 탐색할 수 있음
- min()대신 다른 통계 값을 넣는 것도 가능함
ex) mean, std 등등...
단, amin()은 안됨 (존재하지 않은 함수)

연립방정식 해 구하기

$$\text{연립일차방정식} \begin{cases} x + 2y = 5 \\ 2x + 3y = 8 \end{cases}$$

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ 8 \end{pmatrix}$$

$$\text{행렬} \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} \text{의 역행렬} \rightarrow \begin{pmatrix} -3 & 2 \\ 2 & -1 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -3 & 2 \\ 2 & -1 \end{pmatrix} \begin{pmatrix} 5 \\ 8 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

```
# x+2y=5 2x+3y=8
A = np.linalg.inv(np.array([[1,2],[2,3]]))
B = np.array([5,8])
C = np.dot(A,B); C

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

- $x+2y=5$
- $2x+3y=8$
- $x=1 \ y=2$

np.linalg ← numpy의 선형대수함수

ndarray 파일 입출력

- `np.savetxt(name, ...)`
해당 디렉토리에 'name'이라는 파일로 저장
- `np.loadtxt(name, ...)`
해당 디렉토리에 'name'이라는 파일을 출력

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

np.savetxt('out.txt', a, fmt = '%.4f', delimiter='\t')
np.loadtxt('out.txt', dtype='float', delimiter = '\t')

array([[0.6699, 0.7852],
       [0.2817, 0.5864],
       [0.064 , 0.4856]])
```



- 제작 : 나

