

데이터 분석을 위한

Numpy

류영표

ryp1662@gmail.com

Copyright © "Youngpyo Ryu" All Rights Reserved.

This document was created for the exclusive use of "Youngpyo Ryu".

It must not be passed on to third parties except with the explicit prior consent of "Youngpyo Ryu".



류영표

Youngpyo Ryu

동국대학교 수학과/응용수학 석사수료

現 Upstage AI X 네이버 부스트 캠프 AI tech 1~5기 멘토

前 Innovation on Quantum & CT(IQCT) 이사

前 한국파스퇴르연구소 Image Mining 인턴(Deep learning)

前 (주)셈웨어(수학컨텐츠, 데이터 분석 개발 및 연구인턴)

강의 경력

- 현대자동차 연구원 강의 (인공지능/머신러닝/딥러닝/강화학습)
- (주) 모두의연구소 Aiffel 1기 퍼실리테이터(인공지능 교육)
- 인공지능 자연어처리(NLP) 기업데이터 분석 전문가 양성과정 멘토
- 공공데이터 청년 인턴 / SW공개개발자대회 멘토
- 고려대학교 선도대학 소속 30명 딥러닝 집중 강의
- 이젠 종로 아카데미(파이썬, ADSP 강사)
- 최적화된 도구(R/파이썬)을 활용한 애널리스트 양성과정(국비과정) 강사
- 한화, 하나금융사 교육
- 인공지능 신뢰성 확보를 위한 실무 전문가 자문위원
- 보건·바이오 AI활용 S/W개발 및 응용전문가 양성과정 강사
- Upstage AI X KT 융합기술원 기업교육 모델최적화 담당 조교

주요 프로젝트

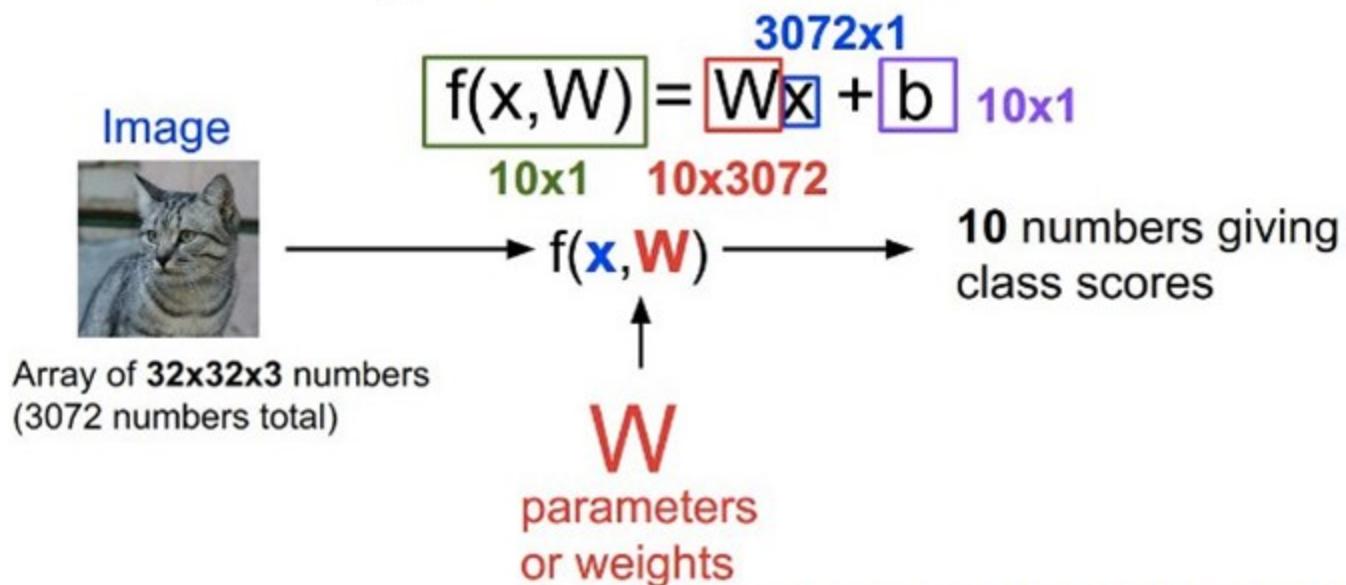
및 기타사항

- 개인 맞춤형 당뇨병 예방·관리 인공지능 시스템 개발 및 고도화(안정화)
- 폐플라스틱 이미지 객체 검출 경진대회 3위
- 인공지능(AI)기반 데이터 사이언티스트 전문가 양성과정 1기 수료
- 제 1회 산업 수학 스터디 그룹 (질병에 영향을 미치는 유전자 정보 분석)
- 제 4,5회 산업 수학 스터디 그룹 (피부암, 유방암 분류)
- 빅데이터 여름학교 참석 (혼잡도를 최소화하는 새로운 노선 건설 위치의 최적화 문제)

Numpy

- Numpy란
 - Numerical Python의 약자로 산술계산용 라이브러리

Parametric Approach: Linear Classifier



http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture02.pdf

Numpy

■ Numpy 특징

- ndarray(다차원 배열 객체) → 빠르고 효율적인 메모리 사용, 유연한 브로드캐스팅
- 디스크로부터 배열기반의 데이터를 읽거나 쓸 수 있는 도구
- C, C++, 포트란 등으로 쓰여진 코드를 통합하는 도구
- 선형대수 계산, 푸리에변환, 난수생성기 등

■ ndarray란

- numpy에서 제공하는 자료구조, N차원의 배열 객체
- 대규모의 데이터 집합을 담을 수 있는 자료구조

ndarray

- **ndarray vs. list**

```
[2] n = 1000000
    numpy_arr = np.arange(n)
    python_list = list(range(n))
```

```
[3] %%time
    python_list = [x**3+10 for x in python_list]
```

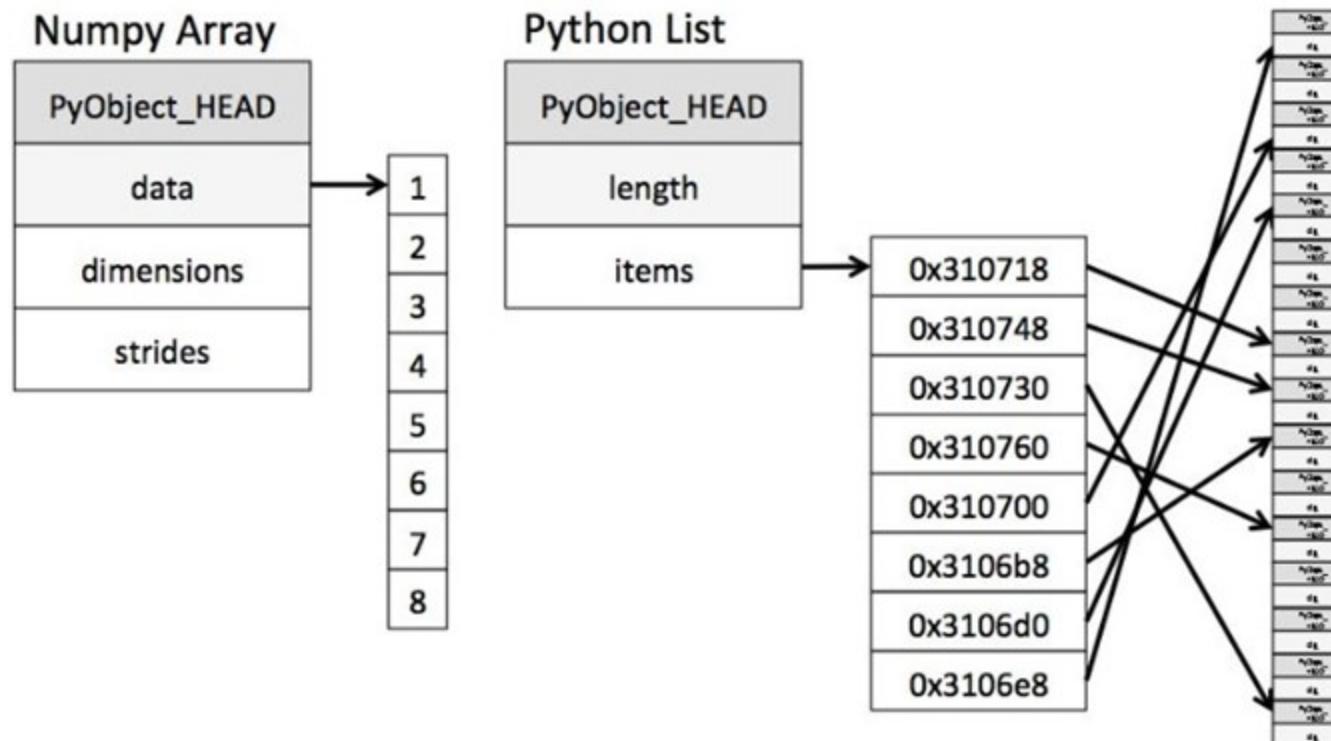
```
↳ CPU times: user 328 ms, sys: 52.9 ms, total: 381 ms
    Wall time: 382 ms
```

```
▶ %%time
    numpy_arr = numpy_arr**3+10
```

```
↳ CPU times: user 4.37 ms, sys: 850 µs, total: 5.22 ms
    Wall time: 9.3 ms
```

ndarray

- ndarray vs. list
 - 연속된 메모리 블록에 데이터를 저장
 - 같은 종류의 데이터를 담음



ndarray

- ndarray는 각 원소별로 동일한 데이터 타입으로 처리.

array([원소 , 원소 , 원소 , dtype=])

```
import numpy as np  
A = [1,2,3,4]  
a = np.array(A, np.int)  
print(type(a))  
print(a)  
  
<class 'numpy.ndarray'>  
[1 2 3 4]
```

```
b = np.array(A,np.str)  
print(type(b))  
print(b)
```

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

```
c = np.array(A,np.float)  
print(type(c))  
print(c)
```

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

```
c = np.array(A,np.complex)  
print(type(c))  
print(c)
```

```
<class 'numpy.ndarray'>  
[1.+0.j 2.+0.j 3.+0.j 4.+0.j]
```

ndarray

- Ndarray 타입을 검색하거나 슬라이싱은 참조만 할당하므로 변경을 방지하기 위해서는 새로운 ndarray로 만들어 사용 .copy 메소드가 필요.

```
import numpy as np
A = [1,2,3,4]

a = np.array(A)
s = a[:2]
print('A의 출력입니다 : %a' %a)
print('s의 출력입니다 : %s' %s)
```

A의 출력입니다 : array([1, 2, 3, 4])
s의 출력입니다 : [1 2]

```
ss = a[:2].copy()
print(ss.size)
ss[0] =99
print(a)
print(s)
print(ss)
```

2
[1 2 3 4]
[1 2]
[99 2]

ndarray

- 벡터화: 배열은 for 문을 작성하지 않고 데이터를 일괄 처리 하는 것
 - 배열은 for 문을 작성하지 않고 데이터를 일괄 처리 하는 것
 - 같은 크기의 배열 간의 산술 연산은 배열의 각 원소 단위로 적용됨

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

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

```
[3] arr + arr
```

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

```
[6] arr / arr
```

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

ndarray

브로드캐스팅

- 다른 모양의 배열 간의 산술 연산을 수행할 수 있도록 해주는 numpy의 기능
- 브로드 캐스팅이 가능하려면 연산을 수행하는 축을 제외한 나머지 축의 shape이 일치하거나 둘 중 하나의 길이가 1이여야 함
- 다른 배열들 간의 연산이 어떤 조건을 만족했을 때 가능해지도록 배열을 자동적으로 변환하는 것

산술 연산

- 브로드캐스팅

1) 스칼라 인자: 모든 원소에 각각 적용

```
[9] 10 - arr
```

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

```
[10] arr*3
```

```
[11] arr/3
```

```
↳ array([[0.33333333, 0.66666667, 1.  
         [1.33333333, 1.66666667, 2.
```

(2, 3)

1	2	3
4	5	6

(1, *)

3	3	3
3	3	3

산술 연산

- 브로드캐스팅

[15] arr

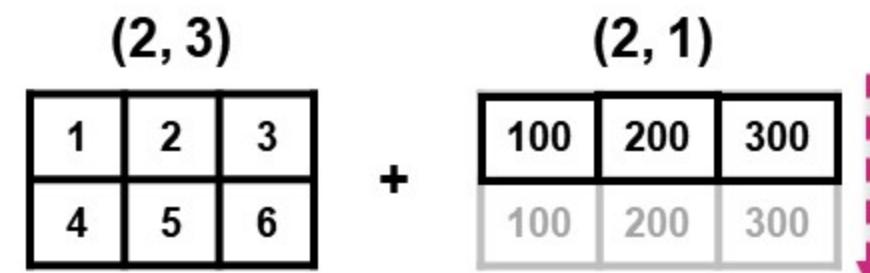
```
[> array([[1, 2, 3],  
          [4, 5, 6]]))
```

[36] arr2

```
[> array([100, 200, 300])
```

[37] arr + arr2

```
array([[101, 202, 303],  
       [104, 205, 306]])
```



산술 연산

- 브로드캐스팅

2) 배열

```
[15] arr
```

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

```
[21] arr3
```

```
↳ array([[100],  
         [200]])
```

```
[▶] arr + arr3
```

```
↳ array([[101, 102, 103],  
         [204, 205, 206]])
```

(2, 3)

1	2	3
4	5	6

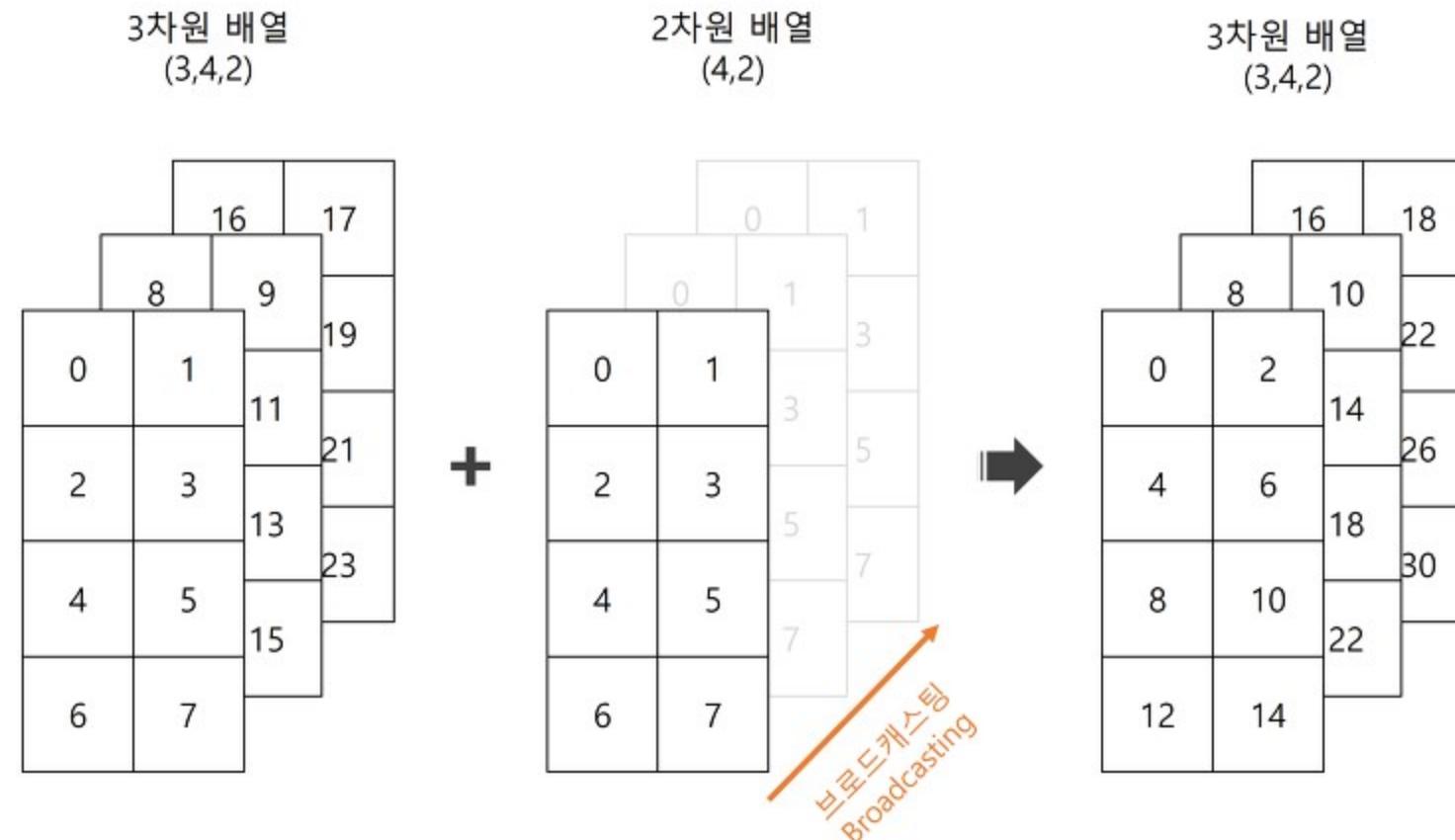
(2, 1)

100	100	100
200	200	200

+



broadcasting



ndarray_asarray()

```
] #np.asarray : Convert the input to an array  
a = [1,2,3,4,5]  
type(a)
```

list

```
] a = np.asarray(a)  
type(a)
```

numpy.ndarray

ndarray attributes

1. dtype

- 배열에 담긴 원소의 자료형(ndarray는 같은 자료형을 담음)

```
[7] arr = np.array([10, 20, 30, 40])  
arr
```

```
↳ array([10, 20, 30, 40])
```

```
[8] arr.dtype
```

```
↳ dtype('int64')
```

```
[10] arr2 = np.array([[0.1, 0.6], [-2, 6]])  
arr2
```

```
↳ array([[ 0.1,  0.6],  
         [-2. ,  6. ]])
```

```
▶ arr2.dtype
```

```
↳ dtype('float64')
```

ndarray attributes

1. dtype

- dtype으로 데이터 타입을 명시하지 않은 경우 자료형을 추론하여 저장
- dtype이 있어 C나 포트란 같은 저수준 언어로 작성된 코드와 쉽게 연동이 가능

```
[2] arr = np.array([10, 20, 30, 40])
arr
```

```
↳ array([10, 20, 30, 40])
```

```
[3] arr.dtype
```

```
↳ dtype('int64')
```

```
[8] float_arr = arr.astype(np.float64)
float_arr
```

```
↳ array([10., 20., 30., 40.])
```

```
[9] float_arr.dtype
```

```
↳ dtype('float64')
```

→ astype이라는 메소드를 이용하여
dtype을 명시적으로 변환할 수 있음

ndarray attributes

2. size

- 배열에 있는 원소의 전체 갯수

arr

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

arr2

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

[12] arr.size

```
↳ 8
```

[16] arr2.size

```
↳ 12
```

ndarray attributes

3. ndim

- 배열의 차원의 갯수

arr

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

arr2

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

[14] arr.ndim

```
↳ 1
```

[▶] arr2.ndim

```
↳ 2
```

ndarray attributes

3. 0차원

numpy.array 생성시 상수값(scalar value)를 넣으면 array 타입이 아니라 일반 타입을 만듦.

Column : 열

[0]

Row : 행

```
import numpy as np  
a = np.array(10)  
print(a)  
print(a.ndim)
```

10
0

ndarray attributes

3. 1차원

배열의 특징, 차원, 형태, 요소를 가지고 있음.

생성시 데이터와 타입을 넣으면 ndim(차원)으로 확인.

Column : 열

0 1 2

Row : 행

[0,0][0,0][0,0]

```
import numpy as np  
a = np.array([1,2,3])  
print(a)  
print(a.ndim)
```

[1 2 3]
1

ndarray attributes

3. 2차원

배열의 특징. 차원, 형태, 요소를 가지고 있음.

생성시 데이터와 타입을 넣으면 ndim(차원)으로 확인.

Column : 열

0 1 2

Row : 행

0 [0,0] [0,0] [0,2]

1 [0,1] [1,1] [1,2]

2 [0,2] [2,1] [2,2]

```
import numpy as np  
a = np.array([[1,2],[3,4]])  
print(a.ndim)
```

2

```
import numpy as np  
a = np.array([ [1,2],[3,4],[4,5],[6,7],[8,9] ])  
print(a.ndim)
```

2

ndarray attributes

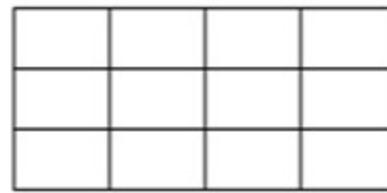
3. 3차원

numpy.array 생성시 sequence 각 요소에 대해 접근변수와 타입을 정할 수 있음.

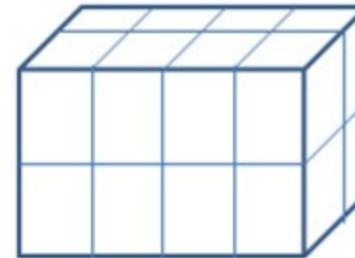
1-D array



2-D array



3-D array



1D array

7	2	9	10
---	---	---	----

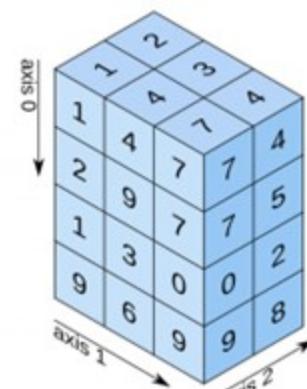
shape: (4,)

2D array

5.2	3.0	4.5
9.1	0.1	0.3

shape: (2, 3)

3D array



shape: (4, 3, 2)

출처 : <https://towardsdatascience.com/numpy-array-manipulation-5d2b42354688>

: <https://predictivehacks.com/tips-about-numpy-arrays/>

ndarray attributes

3. 3차원 -> 4차원

```
import numpy as np  
a = np.array([[ [1,2],[3,4]],[[4,5],[7,8]]])  
print(a)  
print(a.ndim)
```

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

```
[[4 5]  
 [7 8]]]
```

```
3
```

```
import numpy as np  
a = np.array([[[ [1,2],[3,4]],[[4,5],[7,8]]]])  
print(a)  
print(a.ndim)
```

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

```
[[[4 5]  
 [7 8]]]]
```

```
4
```

Ndarray attributes

4. shape

- 배열의 각 차원의 크기
- 튜플의 형태로 리턴

```
arr
```

```
In [14]: arr.array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
[14] arr.ndim
```

```
In [15]: arr.ndim
```

```
[13] arr.shape
```

```
In [16]: arr.shape
```

```
arr2
```

```
In [16]: arr2.array([[1, 2, 3, 4, 5, 6],  
[7, 8, 9, 10, 11, 12]])
```

```
[16] arr2.ndim
```

```
In [17]: arr2.ndim
```

```
[17] arr2.shape
```

```
In [18]: arr2.shape
```

ndarray 생성하기

1. array함수

- 기존에 있던 데이터(자료형)를 이용하여 새로운 배열을 생성

np.array(데이터, dtype=)

```
[2] arr = np.array([10, 20, 30])  
arr
```

→ 리스트를 이용하여 배열을 생성

```
↳ array([10, 20, 30])
```

```
[3] arr2 = np.array([10, 20, 30], dtype=np.float16)  
arr2
```

```
↳ array([10., 20., 30.], dtype=float16)
```

ndarray 생성하기

1. array함수

- 기존에 있던 데이터(자료형)를 이용하여 새로운 배열을 생성

```
[4] arr3 = np.array(((1, 0), (0, 1)), dtype=np.float32) → 튜플을 이용하여 배열을 생성  
arr3
```

```
↪ array([[1., 0.],  
          [0., 1.]], dtype=float32)
```

```
[5] arr4 = np.array(range(20)) → range 함수를 이용하여 배열을 생성  
arr4
```

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

ndarray 생성하기

1. array함수

- 기존에 있던 데이터(자료형)를 이용하여 새로운 배열을 생성

```
[6] arr5 = np.array([(10, 20), (40, 50)])  
arr5
```

```
↳ array([[10, 20],  
         [40, 50]])
```

```
[7] arr6 = np.array(((1, 2), (3)))  
arr6
```

```
↳ array([(1, 2), 3], dtype=object)
```

▶ arr6.size

```
↳ 2
```

ndarray attributes

2. 배열생성함수

- numpy의 표준배열함수([참고](#))
- 자료형을 명시하지 않으면 float64

함수이름	설명
zeros	모두 0으로 초기화
ones	모두 1로 초기화
full	어떠한 값으로 모두 채워 초기화
empty	초기화되지 않은 배열을 생성
identity, eye	NxN 크기의 단위행렬
_like	주어진 어떤 배열과 같은 shape의 배열을 생성
arrange	range 함수와 유사함, 범위와 stepsize
linspace	range 함수와 유사함, sample 갯수

ndarray attributes

2. 배열생성함수

1) zeros

```
[16] np.zeros((5))
```

→ shape을 명시

```
↳ array([0., 0., 0., 0., 0.])
```

```
[17] np.zeros((2, 4), dtype=np.int8)
```

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

2) ones

```
[18] np.ones((3,3))
```

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

ndarray attributes

2. 배열생성함수

3) full

```
[19] np.full((4), 5)
```

```
↳ array([5, 5, 5, 5])
```

```
[▶] np.full((2, 5), -1.0)
```

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

4) empty

```
[25] np.empty((2,3), dtype=np.float64)
```

```
↳ array([[1.8155276e-316, 0.0000000e+000, 0.0000000e+000],
          [0.0000000e+000, 0.0000000e+000, 0.0000000e+000]])
```

ndarray 생성하기

2. 배열 생성 함수

5) identity, eye

```
[22] np.identity(5, dtype=int)
```

→ N x N 정방행렬만 생성 가능

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

```
▶ np.eye(5, dtype=int)
```

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

```
▶ np.eye(5, 10, dtype=int, k=5)
```

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

→ N x M 행렬도 생성 가능

ndarray 생성하기

2. 배열생성함수

6) _like

- zeros_like, ones_like, full_like, empty_like

```
[29] arr1 = np.array([[1, 2, 3, 1], [2, 4, 5, 6]])  
arr1
```

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

→ shape을 명시하지 않고
기존에 존재하는 배열을
인자로 넘겨줌

```
▶ arr2 = np.ones_like(arr1)  
arr2
```

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

ndarray 생성하기

2. 배열생성함수

7) arange

- 파이썬내장함수range와 유사한 역할, ndarray를 반환

```
[43] np.arange(5)
```

```
↳ array([0, 1, 2, 3, 4])
```

```
[44] np.arange(-3, 3)
```

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

```
[45] np.arange(3, 50, 5)
```

```
↳ array([ 3, 8, 13, 18, 23, 28, 33, 38, 43, 48])
```

→ 세번째인자는 step size
(range와 동일)

ndarray attributes

2. 배열생성함수

8) linspace

- 범위내에서주어진sample의 갯수만큼 생성

Linspace(start, stop, num, endpoint ,retstep)

```
[▶] np.linspace(0, 1, 6)
```

```
↳ array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

ndarray attributes

3. 배열 결합함수

1) Hstack , concatenate(axis =0)

- 두 배열을 왼쪽에서 오른쪽으로 붙이기

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

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

```
)8] np.hstack([a,b])
```

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

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

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

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

```
b = np.array([[5,6],[7,8]])
```

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

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

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

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

ndarray attributes

3. 배열 결합함수

2) vstack, concatenate(axis=1)

- 두 배열을 위에서 아래로 붙이기.

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

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

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

```
-----  
AxisError
```

```
<ipython-input-111-5572ec4fb823> in <module>()
```

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

```
<__array_function__ internals> in concatenate(*args, **kwargs)
```

```
AxisError: axis 1 is out of bounds for array of dimension 1
```

1D 배열은 "AxisError: axis 1 is out of bounds for array of dimension 1"라는 AxisError가 나네요. 2D 이상 배열은 에러 없이 잘 됨.

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

```
d = np.array([[6, 7, 8], [9, 10, 11]])
```

```
np.concatenate((c,d),axis =1)
```

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

ndarray attributes

3. 배열 결합함수

3) column_stack, concatenate(C.T,D.T, axis=1)

- 두 배열을 위에서 아래로 붙이기.

```
[113] a,b
```

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

```
[115] np.column_stack([a,b])
```

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

1D 배열은 "AxisError: axis 1 is out of bounds for array of dimension 1"라는 AxisError가 나오네요. 2D 이상 배열은 에러 없이 잘 됨.

```
c = np.array([[0, 1, 2], [3, 4, 5]])  
d = np.array([[6, 7, 8], [9, 10, 11]])
```

```
np.concatenate((c.T, d.T), axis = 1)
```

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

Random Module

- Random는 난수를 발생시키는 모듈. 모듈에는 여러 함수가 존재

```
import random
```

```
random.random() # 0<= 리턴값 <1.0
```

```
0.4926223938498203
```

```
|: random.uniform(100, 200) # 괄호 안 두 수 사이의 실수 중에서 난수값을 리턴 a <= 실수 <= b
```

```
|: 174.57590071611548
```

```
data = [1,2,3,4,5,6,7]
```

```
random.choice(data)
```

```
6
```

```
data = ['apple', 'banana', 'grape', 'orange']
```

```
random.choice(data)
```

Choice 함수 : 매개변수로 시퀀스 타입(문자열, 튜플, 리스트)을 받음. 그중에서 무작위로 하나를 선택하여 리턴

Random Module

- Random는 난수를 발생시키는 모듈. 모듈에는 여러 함수가 존재
- Seed : 난수 생성 초기값 부여
- 난수 생성 할 때마다 값이 달라지는 것이 아니라, 누가 언제 하든지
간에 똑같은 난수 생성을 원한다면(즉, 재현가능성
(Reproducibility) seed 번호를 지정해줌.
- np.random.seed(100)

ndarray 생성하기

3. 난수 생성

- numpy.random 모듈을 이용하여 다양한 종류의 확률 분포로부터 표본값을 생성
(참고)

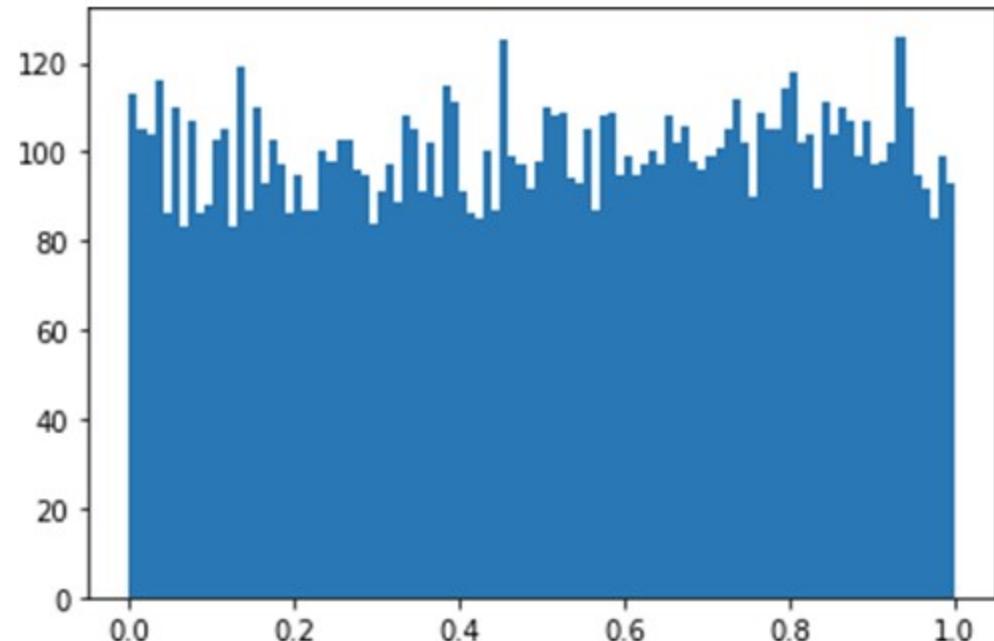
함수이름	설명
np.random.seed()	난수 생성기의 시드를 지정
np.random.rand()	[0, 1) 범위의 균등분포에서 표본을 추출
np.random.randn()	표준편차 1, 평균값 0인 정규분포에서 표본을 추출
np.random.randint()	주어진 범위 내에서 임의의 난수를 추출
np.random.permutation()	순서를 임의로 바꾸거나 임의의 순열을 반환
np.random.shuffle()	리스트나 배열의 순서를 뒤섞음

ndarray 생성하기

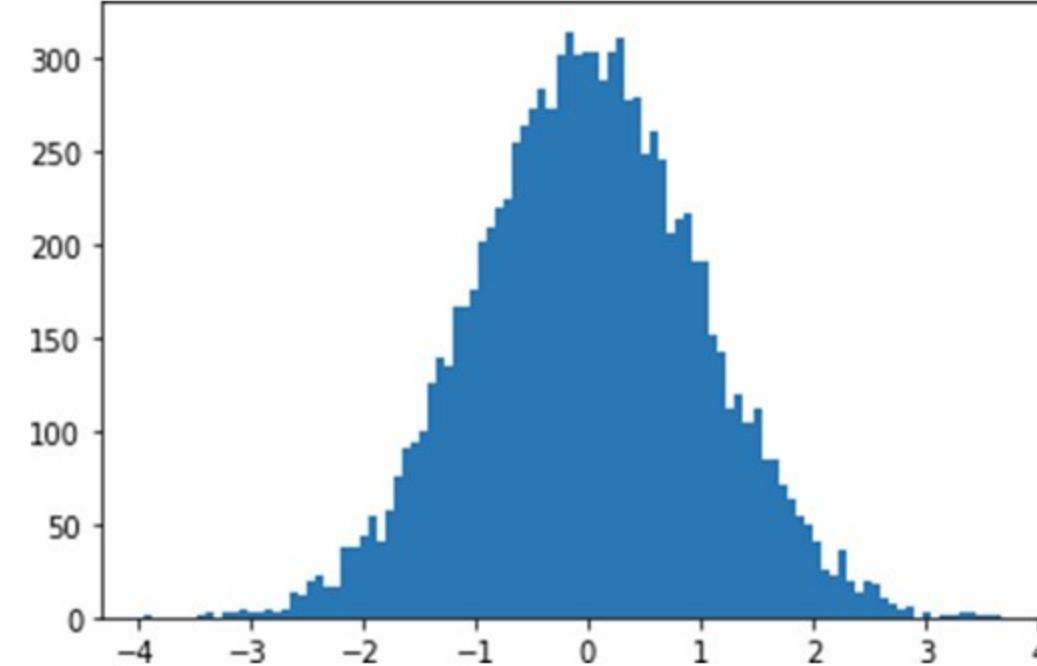
3. 난수 생성

- numpy.random 모듈을 이용하여 다양한 종류의 확률분포로부터 표본값을 생성

```
[▶] data = np.random.rand(10000)
```



```
[▶] data = np.random.randn(10000)
```



로또 번호 생성기를 만드세요.

로또 번호를 몇개 생성할까요? > 4

1. 로또번호: [13 16 20 22 27 43]
2. 로또번호: [21 27 31 36 37 45]
3. 로또번호: [4 15 16 26 30 39]
4. 로또번호: [7 15 26 29 40 41]

Flatten

N dim → 1 dim

1	2	3
4	5	6
7	8	9



1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

```
[60] arr = np.zeros((3,2))  
arr
```

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



```
[61] arr.flatten()
```

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

Reshape

- 이미 존재하는 배열을 내가 원하는대로 shape을 조정하는 함

np.reshape(arr, shape) **arr.reshape(shape)**

```
[76] arr = np.arange(12)  
arr
```

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

```
[77] arr.reshape(3, 4)
```

주의

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

원래주어진 shape의 약수로 이뤄진 shape만 가능

Reshape

이미 존재하는 배열을 내가 원하는대로 shape을 조정하는 함수

```
[80] arr = np.arange(20)  
arr
```

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

[81] arr.reshape(-1, 10) → -1을 사용하면 shape을 명시하지 않아도 자동으로 채워줌

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

▶ arr.reshape(5, -1)

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

Transpose

1	2	3
4	5	6
7	8	9

A



1	4	7
2	5	8
3	6	9

A^T

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

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Transpose

np.transpose(arr, axes)
arr.transpose(axes)

```
[83] arr = np.arange(20).reshape(4, 5)
      arr
```

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



```
[84] print(arr.transpose().shape)
      arr.transpose()
```

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

Transpose

- axes를 지정하지 않으면?

$a.shape = (i[0], i[1], \dots, i[n-1]) \rightarrow a^t.shape = (i[n-1], i[n-2], \dots, i[0])$

```
[2] arr = np.arange(30).reshape(3,2,5)  
arr.shape
```

↳ (3, 2, 5)

```
[3] print(arr.transpose().shape)
```

↳ (5, 2, 3)

Swapaxes

- np.swapaxes는 직관적으로 축을 선정 함.

```
import numpy as np

a = np.arange(3).reshape(1,3) # (1x3) 2D array

y = np.swapaxes(a,0,1) # 0은 가장 높은 차수의 축, 2차원 / 1은 그다음 높은 차수의 축 1차원 즉, 원소의 행과 열을 바꾸라는 것
y

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

Swapaxes_ 차원의 증가

```
a = np.arange(6).reshape(1,2,3)
print(a)
print(a.shape)
```

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

```
x = np.swapaxes(a, 1, 2) # 2차원 축과, 1차원 축을 바꿔라
x
```

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

```
y = np.swapaxes(a, 0, 1) # 3차원 축과, 2차원 축을 바꿔라.
print(y)
print(y.shape)
```

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

```
z = np.swapaxes(a, 0, 2) # 3차원 축과 1차원 축을 바꿔라.
print(z)
print(z.shape)
```

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

Transpose

- axes는 tuple이나 list로 지정해줄수 있음

```
[4] print(arr.shape)
    arr.transpose((1, 0, 2)).shape
```

```
↳ (3, 2, 5)
    (2, 3, 5)
```

```
▶ arr = np.arange(120).reshape(2,3,4,5)
print(arr.shape)
arr.transpose((1, 0, 2, 3)).shape
```

```
↳ (2, 3, 4, 5)
    (3, 2, 4, 5)
```

T operation

- transpose와 같은 역할을 하는 ndarray의 attribute
- 단, T의 경우 axes를 지정할 수 없음

```
[7] x = np.arange(6).reshape((-1,3))  
x
```

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

▶ x.T

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

연습문제 2)

0~ 20까지의 숫자를 배열을 만든 다음에 arr1에는 짝수,
arr2는 홀수가 들어간 배열을 출력해보자.

연습문제 3)

주어진 배열을 이용하여 아래와 같은 결과를 만들어보자

```
arr = np.arange(30).reshape(2,3,5)
arr
```

```
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]])
```

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

```
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])
```

Indexing & Slicing

- 1차원

기준의 리스트와 유사함

```
[9] arr1d = np.arange(8)  
arr1d
```

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

```
[10] arr1d[1]
```

```
↳ 1
```

```
[11] arr1d[-1]
```

```
↳ 7
```

```
[12] arr1d[:4]
```

```
↳ array([0, 1, 2, 3])
```

Indexing & Slicing

- 1차원

ndarray vs. list

1) 브로드캐스팅지원

- 브로드캐스팅이란?

다른 모양의 배열 간의 산술 연산을 수행할 수 있도록 해 주는 numpy의 기능

Indexing & Slicing

- 1차원

ndarray vs. list

```
[15] lst = list(range(6))
      lst
```

↳ [0, 1, 2, 3, 4, 5]

```
[16] lst[2:5] = -1
      lst
```

↳ -----
TypeError
<ipython-input-16-b4d2405268a4> in <modu
----> 1 lst[2:5] = -1
 2 lst

TypeError: can only assign an iterable

```
[17] lst[3] = -1
      lst
```

↳ [0, 1, 2, -1, 4, 5]

→ 브로드캐스팅을 지원하지 않음

Indexing & Slicing

- 1차원

ndarray vs. list

```
[9] arr1d = np.arange(8)  
arr1d
```

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

```
[13] arr1d[3:6] = 100
```

→ 브로드캐스팅을 지원

```
[14] arr1d
```

```
↳ array([ 0, 1, 2, 100, 100, 100, 6, 7])
```

Indexing & Slicing

- 1차원

ndarray vs. list

2) 뷰

- 넘파이의 슬라이싱은 원본데이터의 **뷰**를 제공:
데이터를 새롭게 복사해 오는 것이 아니라 기존의 데이터와 연결되어있음
- 리스트의 슬라이싱은 데이터를 복사하게됨

Indexing & Slicing

- 1차원

ndarray vs. list

```
[17] arr_part = arr1d[:3]  
arr_part
```

→ 슬라이싱

```
↳ array([0, 1, 2])
```

```
[18] arr_part[1:] = -1  
arr_part
```

→ 값을 변경

```
↳ array([ 0, -1, -1])
```

```
[19] arr1d
```

→ 원본데이터가 변경되어있음

```
↳ array([ 0, -1, -1, 100, 100, 100, 6, 7])
```

Indexing & Slicing

- 1차원

ndarray vs. list

```
[20] lst_part = lst[2:]  
     lst_part
```

→ 슬라이싱

```
↳ [2, -1, 4, 5]
```

```
[21] lst_part[3] = 100  
     lst_part
```

→ 값을 변경

```
↳ [2, -1, 4, 100]
```

```
▶ lst
```

→ 원본데이터는 그대로 유지

```
↳ [0, 1, 2, -1, 4, 5]
```

Indexing & Slicing

- 1차원

ndarray vs. list

2) 뷰

- 원본데이터를 훼손하지 않으려면?

arr.copy()

```
[27] new_arr = arr1d.copy()  
new_arr
```

```
↳ array([ 0, -1, -1, 100, 100, 100, 6, 7])
```

Indexing & Slicing

- 다차원

```
[3] arr2d = np.arange(20).reshape(4, -1)  
arr2d
```

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

```
[4] arr2d[0]
```

→ 다중리스트의 인덱싱과 유사함

```
↳ array([0, 1, 2, 3, 4])
```

```
[5] arr2d[1][2]
```

→ 재귀적으로 접근

```
↳ 7
```

```
[6] arr2d[1, 2]
```

→ 콤마(,)를 이용하여 쉽게 인덱싱을 할 수도 있음

```
↳ 7
```

Indexing & Slicing

- 다차원

```
arr2d = np.arange(20).reshape(4, -1)  
arr2d
```

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

```
arr2d[:3, :2]
```

```
arr2d[:3][:2]
```

→ 재귀적으로 슬라이싱을 함

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

```
array([[ 0,  1],  
       [ 5,  6],  
       [10, 11]])
```

Boolean indexing

- 불리언 배열: 불리언 값으로 이루어진 배열

```
[2] arr = np.array([True, False])
    arr.dtype
    ↪ dtype('bool')
```

- 불리언 인덱싱: 불리언 배열을 이용한 인덱싱
 - True에 해당되는 위치에 있는 값을 반환

```
[3] arr = np.array([0, 1, 2, 3, 4], int)
    arr[[True, False, True, False, True]]
    ↪ array([0, 2, 4])
```

boolean indexing

- 불리언배열을 이용한인덱싱

```
[4] arr = np.array([10, 20, 30, 40, 50, 60], int)  
arr
```

```
↳ array([10, 20, 30, 40, 50, 60])
```

```
[5] arr % 3 == 0
```

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

```
▶ arr[arr%3==0]
```

```
↳ array([30, 60])
```

- 불리언 인덱싱 후 해당 값을 다시 인덱싱/슬라이싱도 할 수 있음

→ 조건문을 통해 불리언 배열을 만들 수 있음

→ 이를 배열에 인덱싱으로 주면 해당 위치에 값을 반환

Fancy indexing

정수배열을 사용한 인덱싱

```
[5] arr = np.arange(10, 20)  
arr
```

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

```
[6] arr[[0, 2, 4, 6]]
```

```
↳ array([10, 12, 14, 16])
```

```
▶ arr[[3, 0, 1]]
```

```
↳ array([13, 10, 11])
```

→ 정수가 담긴 ndarray나 리스트로
특정 위치에 있는 값을 가져올 수 있음

→ 주어지는 순서대로 값을 가져오게 됨

fancy indexing

정수배열을 사용한 인덱싱

```
[7] arr2d = np.arange(20).reshape(4, 5)
    arr2d
```

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

```
[8] arr2d[[0, 2]]
```

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

```
[9] arr2d[[0, 1], [4]]
```

```
⇒ array([4, 9])
```

Fancy indexing

정수배열을 사용한 인덱싱

```
[7] arr2d = np.arange(20).reshape(4, 5)
arr2d
```

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

```
[10] arr2d[[0, 1], [4, 3]]
```

→ (0, 4), (1, 3)에 대응되는 원소들을 가져옴

```
↳ array([4, 8])
```

```
▶ arr2d[[0, 1, 2], [4, 3, 1]]
```

```
↳ array([ 4,  8, 11])
```

fancy indexing

정수배열을 사용한 인덱싱

```
[7] arr2d = np.arange(20).reshape(4, 5)  
arr2d
```

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

```
▶ arr2d[[0, 1, 2]] [:, [4, 3, 1]]
```

```
↳ array([[ 4,  3,  1],  
         [ 9,  8,  6],  
         [14, 13, 11]])
```

연습문제 3)

1번

주어진 배열을 이용하여 아래와 같은 결과를 만들어보자

```
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]]])
```

2번

주어진 배열과 불리언 인덱싱을 이용하여 아래와 같은 결과를 만들어보자

```
arr = np.arange(30).reshape(3,2,5)  
arr  
  
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]]])
```

```
cond = np.array(['a', 'b', 'c'])
```

```
array([[[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9]],  
  
      [[20, 21, 22, 23, 24],  
       [25, 26, 27, 28, 29]]])
```

유니버설 함수

- 유니버셜함수란
 - ndarray 안에 있는 데이터 원소별로 연산을 수행하는 함수
 - 하나 이상의 스칼라값을 받아서 하나 이상의 스칼라값을 반환하는 간단한 함수를
고속으로 수행할 수 있는 벡터화된 함수

유니버설 함수(ufunc)

- 단항유니버설 함수

Function	설명
abs, fabs	각 원소의 절대값. 복소수가 아닌 경우 fabs를 쓰면 빠른 연산이 가능
sqrt	각 원소의 제곱근 계산($arr^{**0.5}$)
square	각 원소의 제곱을 계산(arr^{**2})
exp	각 원소에 지수 e^x 계산
log, log10, log2, log1p	자연로그, 밀10인 로그, 밀2인 로그, $\log(1+x)$
sign	각 원소의 부호(양수 1, 영 0, 음수 -1)
ceil	각 원소의 값보다 같거나 큰 정수 중 가장 작은 값
floor	각 원소의 값보다 같거나 작은 정수 중 가장 큰 값
rint	각 원소의 소수자리를 반올림
modf	각 원소의 몫과 나머지를 각각의 배열로 반환
isnan	각 원소가 NaN인지 아닌지. 불리언 배열로 반환
isfinite, isinf	각 원소가 유한한지, 무한한지. 불리언 배열로 반환
cos, cosh, sin, sinh, tan, tanh	일반 삼각함수, 쌍곡삼각함수
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	역삼각함수

유니버설 함수(ufunc)

- 단항유니버설 함수

```
[2] arr = np.arange(-3, 3).reshape(3, -1)
    arr
```

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

```
[3] np.exp(arr)
```

```
↳ array([[0.04978707, 0.13533528],
          [0.36787944, 1.        ],
          [2.71828183, 7.3890561 ]])
```

```
[4] np.floor(arr)
```

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

```
▶ np.isinf([np.inf, -np.inf, 1, np.nan])
```

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

유니버설 함수(ufunc)

- 이항유니버설 함수

Function	설명
add	두 배열의 같은 위치의 원소끼리 합함
subtract	첫번째 배열의 원소에서 두번째 배열의 원소를 뺌
multiply	같은 위치의 원소끼리 곱함
divide, floor_divide	첫번째 배열의 원소를 두번째 배열의 원소를 나눔. floor는 몫만 취함
power	첫번째 매열의 원소를 두번째 배열의 원소만큼 제곱함
maximum, fmax	각 배열의 두 원소 중 큰 값을 반환. fmax는 NaN 무시
minimum, fmin	각 배열의 두 원소 중 작은 값을 반환. fmax는 NaN 무시
mod	첫번째 배열의 원소를 두번째 배열의 원소로 나눈 나머지
copysign	첫번째 배열의 원소의 기호를 두번째 배열의 원소의 기호로 바꿈
greater, greater_equal, less, less_equal, less, less_equal,	각 두 원소 간의 비교 연산(<, <=, >, >=, ==, !=)를 불리언 배열로 반환

유니버설 함수(ufunc)

- 이항유니버설 함수

```
[6] arr1 = np.arange(8).reshape(2, -1)
    arr2 = np.arange(-40, 40, 10).reshape(2, -1)
    print(arr1)
    print(arr2)
```

```
↳ [[0 1 2 3]
   [4 5 6 7]]
[[[-40 -30 -20 -10]
  [ 0  10  20  30]]]
```

```
[7] np.maximum(arr1, arr2)
```

```
↳ array([[ 0,  1,  2,  3],
   [ 4, 10, 20, 30]])
```

```
[9] np.subtract(arr2, arr1)
```

```
↳ array([[-40, -31, -22, -13],
   [-4,  5, 14, 23]])
```

```
[10] np.multiply(arr1, arr2)
```

```
↳ array([[ 0, -30, -40, -30],
   [ 0,  50, 120, 210]])
```

기본 통계 메소드

- 배열 전체 혹은 배열에서 한 축을 따르는 자료에 대한 통계를 계산하는 수학 함수

sum
mean
std, var
min, max
argmin, argmax
cumsum
cumprod

```
[11] arr = np.random.randn(200, 500)  
arr.shape
```

```
↳ (200, 500)
```

```
[12] arr.sum()      # ndarray의 메서드 이용
```

```
↳ -596.3978294521196
```

```
[13] np.sum(arr)    # numpy의 최상위 함수를 이용
```

```
↳ -596.3978294521196
```

기본 통계 메소드

- sum, mean과 같은 함수의 경우 axis를 인자로 받아서 해당 axis에 대한 통계를 계산할 수 있음
- 축을 지정하지 않을 경우 배열 전체에 대한 값을 연산

```
[21] arr.shape
```

```
↳ (200, 500)
```

```
[14] arr.mean()
```

```
↳ -0.005963978294521196
```

```
[22] arr.mean(axis=0).shape
```

```
↳ (500,)
```

```
[23] arr.mean(axis=1).shape
```

```
↳ (200,)
```

axis = 0

1	2	3
4	5	6

axis = 1

1	2	3
4	5	6

기본 통계 메소드

표본 분산/표준편차도 계산 가능

$$s^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2$$

$$s = \sqrt{s^2}$$

```
np.var(x) # 분산
```

```
115.23224852071006
```

```
np.std(x) # 표준 편차
```

```
10.734628476137871
```

Numpy의 rand/randn

```
import numpy as np
```

```
np.random.rand()
```

```
0.1454858837063564
```

```
np.random.rand(1)
```

```
array([ 0.36708798])
```

```
np.random.rand(2)
```

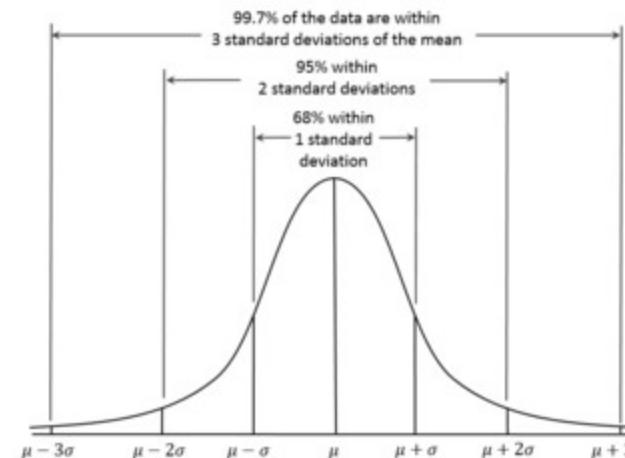
```
array([ 0.94584748,  0.10367624])
```

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

```
array([[ 0.52603453,  0.54570411,  0.99713388],
       [ 0.11226014,  0.64858858,  0.96787497],
       [ 0.76334219,  0.90901883,  0.48988668]])
```

```
np.random.rand(5, 2)
```

```
array([[ 0.78907522,  0.8219071 ],
       [ 0.84726884,  0.50501422],
       [ 0.26994445,  0.90803876],
       [ 0.35450963,  0.21855857],
       [ 0.43977184,  0.48358754]])
```



기타 메소드_any,all

- any
 - 하나 이상의 값이 True이면 True를 반환
- all
 - 모든값이 True 이면 True를 반환

```
[2] arr = np.array([True, False, True])
    arr.any()
```

↳ True

```
[3] arr = np.array([True, False, True])
    arr.all()
```

↳ False

```
[4] arr = np.array([0, 0, 3])
    arr.any()
```

↳ True

```
[5] arr = np.array([0, 0, 1])
    arr.all()
```

↳ False

기타 메소드_any,all

any, all

```
[6] arr = np.array([1, 1, 1])
arr.all()
```

↳ True

```
▶ arr = np.array([5, -1, np.inf])
arr.all()
```

↳ True

기타 메소드_where

where

- $x \text{ if } \text{조건} \text{ else } y$ 의 벡터화 버전
- numpy를 사용하여 큰배열을 빠르게 처리 할수있으며,
다차원도 간결하게 표현이 가능

np.where(조건, x, y)

```
[8] xarr = np.array([100, 200, 300, 400])
    yarr = np.array([1, 2, 3, 4])
    cond = np.array([True, False, True, False])
```

```
[9] result = np.where(cond, xarr, yarr)
    result
[5] array([100, 2, 300, 4])
```

→ True일때는 xarr에 있는값을,
False일때는 yarr에 있는값을대입하여반환

기타 메소드_where

where

- x와 y자리에 들어가는 인자는 배열이 아니여도 가능

```
[8] xarr = np.array([100, 200, 300, 400])
    yarr = np.array([1, 2, 3, 4])
    cond = np.array([True, False, True, False])
```

```
[10] np.where(xarr>200, max(xarr), 0)
⇒ array([ 0,  0, 400, 400])
```

```
[▶] np.where(xarr%3==0, 1, 0 )
⇒ array([0, 0, 1, 0])
```

기타 메소드_Sort

sort

- arr.sort()
 - 주어진 축에 따라 정렬하며, 다양한 정렬 방법들을 지원([참고](#))
 - arr 자체를 정렬함(*in-place*)

arr.sort(axis=-1)

```
[23] np.random.seed(10)
    arr = np.random.randint(1, 100, size=10)
    arr
    ↴ array([10, 16, 65, 29, 90, 94, 30, 9, 74, 1])
[4] arr.sort()
    arr
    ↴ array([ 1,  9, 10, 16, 29, 30, 65, 74, 90, 94])
```

기타 메소드_sort

sort

- np.sort()
 - np.sort는 배열을 직접 변경하지 않고 정렬된 결과를 가진 복사본을 반환([참고](#))

np.sort(arr, axis=-1)

```
[32] np.random.seed(20)
    arr = np.random.randint(1, 100, size=10)
    np.sort(arr)
```

```
↪ array([10, 16, 21, 23, 29, 72, 76, 91, 91, 96])
```

```
[33] arr
```

```
↪ array([91, 16, 96, 29, 91, 10, 21, 76, 23, 72])
```

```
[▶] -np.sort(-arr)
```

→ 부호를 이용하여
내림차순으로 정렬

```
↪ array([96, 91, 91, 76, 72, 29, 23, 21, 16, 10])
```

선형대수

- numpy는 행렬의 곱셈, 분할, 행렬식과 같은 선형대수에 관한 라이브러리를 제공함

Functions	설명
dot	내적
dialog	정사각 행렬의 대각/비대각 원소를 1차원 배열로 반환하거나, 1차원 배열을 대각선 원소로 하고 나머지는 0으로 채운 단위행렬 반환
trace	행렬의 대각선 원소의 합을 계산
linalg.det	행렬식을 계산($ad-bc$)
linalg.eig	정사각행렬의 고유값, 고유벡터를 계산
linalg.inv	정사각행렬의 역행렬을 계산
linalg.solve	A가 정사각 행렬일 때, $Ax=b$ 를 만족하는 x를 구함
linalg.svd	특이값 분해(SVD)를 계산

선형 대수_단위 행렬

- 단위 행렬(Identity Matrix)
 - 단위 행렬은 대각원소가 1이고, 나머지는 모두 0인 n차 정방행렬을 말하며, numpy의 eye() 함수를 사용하여서 만들 수 있습니다.

```
import numpy as np  
  
Identity = np.eye(4)  
print(Identity)  
  
[[1. 0. 0. 0.]  
 [0. 1. 0. 0.]  
 [0. 0. 1. 0.]  
 [0. 0. 0. 1.]]
```

[59] import numpy as np

```
Identity = np.eye(2,3)  
print(Identity)  
  
[[1. 0. 0.]  
 [0. 1. 0.]]
```

선형 대수_대각행렬

- 대각 행렬(Diagonal Matrix)

- 대각행렬은 대각성분 이외의 모든 성분이 모두 '0'인 n차 정방행렬을 말함.

```
import numpy as np  
  
x = np.arange(9).reshape(3,-1)  
  
print(x)  
  
[[0 1 2]  
 [3 4 5]  
 [6 7 8]]
```

```
np.diag(x)  
  
array([0, 4, 8])  
  
np.diag(np.diag(x))  
  
array([[0, 0, 0],  
       [0, 4, 0],  
       [0, 0, 8]])
```

선형 대수_dot

- dot
 - dot product : 벡터의 내적

np.dot(a, b)

a.sort(b)

$$\mathbf{a} = (a_1, a_2, \dots, a_n)$$

$$\mathbf{b} = (b_1, b_2, \dots, b_n)$$

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

선형 대수_dot

원소 간 곱(element-wise product) : 내적(dot product, inner product):
 a^*b np.dot(a,b)

```
import numpy as np  
  
a = np.arange(4).reshape(-1,2)  
print(a)  
a*a  
  
[[0 1]  
 [2 3]]  
array([[0, 1],  
       [4, 9]])
```

```
import numpy as np  
  
a = np.arange(4).reshape(-1,2)  
print(a)  
print(np.dot(a,a))  
  
[[0 1]  
 [2 3]]  
array([[ 2,  3],  
       [ 6, 11]])
```

동일한 작동

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

```
array([[ 2,  3],  
       [ 6, 11]])
```

선형 대수_matmul

- matmul
 - matrix multiplication: 행렬의 곱

$$\begin{array}{ccc} \begin{matrix} & m \\ | & \text{[light green]} & \text{[light green]} \\ | & \text{[light green]} & \text{[light green]} \end{matrix} & \cdot & \begin{matrix} & n \\ m & \text{[pink]} & \text{[pink]} \\ & \text{[pink]} & \text{[pink]} \\ & \text{[pink]} & \text{[pink]} \\ & \text{[pink]} & \text{[pink]} \end{matrix} & = & \begin{matrix} & n \\ | & \text{[light blue]} & \text{[light blue]} \\ | & \text{[light blue]} & \text{[light blue]} \end{matrix} \end{array}$$
$$A \quad \cdot \quad B \quad = \quad C$$
$$(l, m) \quad (m, n) \quad (l, n)$$

선형 대수_matmul

- Matmul : matrix multiplication : 행렬의 곱

- **np.matmul(a, b)**

```
[41] a = np.random.randint(-3, 3, 10).reshape(2, 5)
    b = np.random.randint(0, 5, 15).reshape(5, 3)
    a.shape, b.shape
```

```
⇒ ((2, 5), (5, 3))
```

```
[42] ab = np.matmul(a, b)
    print(ab.shape, '\n')
    print(ab)
```

```
⇒ (2, 3)
```

```
[[ 6   0   6]
 [-14  -6  -4]]
```

선형 대수_matmul

- Matmul : matrix multiplication : 행렬의 곱

- **np.matmul(a, b)**

```
[41] a = np.random.randint(-3, 3, 10).reshape(2, 5)
    b = np.random.randint(0, 5, 15).reshape(5, 3)
    a.shape, b.shape
```

```
↳ ((2, 5), (5, 3))
```

▶ **np.matmul(b, a)**

→ matmul시에는 shape에 유의할 것

```
↳ -----
ValueError                                Traceback (most recent call last)
<ipython-input-43-3c9936ad69e2> in <module>()
----> 1 np.matmul(b, a) # matrix곱할때는 shape0| 맞도록
```

```
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0,
with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 2 is different from 3)
```

선형 대수_matmul

- matmul
 - 3차원 이상의 경우에는 **마지막2개** 축으로 이루어진 행렬을 다른 축들에 따라쌓은 것으로 파악
 - 따라서, **마지막 2개의 차원이 행렬곱을 할 수 있으면 matmul 가능**

```
[44] c = np.arange(24).reshape(2, 3, 4)
    d = np.arange(2*4*5).reshape(2, 4, 5)
    c.shape, d.shape
```

```
⇒ ((2, 3, 4), (2, 4, 5))
```

```
[48] arr = np.matmul(c, d)
    arr.shape
```

```
(2, 3, 5)
```

선형 대수_대각합(Trace)

- Trace

- 정방행렬의 대각에 위치한 원소를 전부 더 해줌.

```
import numpy as np
```

```
b = np.arange(16).reshape(4,-1)
```

```
print(b)
```

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

```
np.trace(b)
```

```
c = np.arange(27).reshape(3, 3, 3)
print(c)
```

```
[[[ 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]]]
```

```
np.trace(c)
```

```
array([36, 39, 42])
```

선형 대수_행렬식(Matrix Determinant)

- Determinant

- 역행렬이 존재하는지 여부를 확인하는 방법으로 행렬식(determinant, 줄여서 det)이라는 지표를 사용함. 이 행렬식이 '0'이 아니면 역행렬이 존재하고, 이 행렬식이 '0'이면 역행렬이 존재하지 않습니다.

```
d = np.array([[1,2],[3,4]])
```

```
np.linalg.det(d)
```

```
-2.000000000000004
```

```
d = np.array([[1,2],[2,4]])
```

```
np.linalg.det(d)
```

```
0.0
```

선형 대수_행렬식(Inverse of a matrix)

- Inverse matrix

- 역행렬은 n차 정방행렬 A_{mn} 과의 곱이 항등행렬 또는 단위 행렬이 되는 n차 정방행렬을 말합니다.

A^*B 와 B^*A 모두 순서에 상관없이 곱했을 때 단위행렬이 나오는 n차 정방행렬이 있다면 역행렬이 존재함. 역행렬은 가우스 소거법(Gauss-Jordan elimination method) 혹은 여인수(Cofactor method)로 풀 수 있음.

```
a = np.array(range(4)).reshape(2, -1)  
print(a)
```

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

```
a_inv = np.linalg.inv(a)
```

```
a_inv
```

```
array([[-1.5,  0.5],  
       [ 1. ,  0. ]])
```

```
a.dot(a_inv)
```

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

선형 대수_고유 값(Eigenvalue), 고유벡터(Eigenvector)

정방 행렬 A에 대하여 $Ax = \lambda x$ (상수 λ)가 성립하는 0이 아닌 벡터 x가 존재할 때 상수 λ 를 행렬 A의 고유 값(eigenvalue), x를 이에 대응하는 고유 벡터(eigenvector)라고 함.

```
e = np.array([[4, 2], [3, 5]])
print(e)
```

```
[[4 2]
 [3 5]]
```

```
w,v = np.linalg.eig(e)
```

```
print(w)
```

```
[2. 7.]
```

```
print(v)
```

```
[[ -0.70710678 -0.5547002 ]
 [ 0.70710678 -0.83205029]]
```

선형 대수_특이값 분해(Singular Value Decomposition)

특이값 분해는 고유값 분해(eigen decomposition)처럼 행렬을 대각화하는 한 방법으로서, 정방 행렬뿐만 아니라 모든 $m \times n$ 행렬에 대해 적용 가능합니다. 특이값 분해는 차원축소, 데이터 압축 등에 사용할 수 있습니다.

```
A = np.array([[3,6], [2,3], [0,0], [0,0]])  
print(A)  
[[3 6]  
 [2 3]  
 [0 0]  
 [0 0]]  
  
u,svh = np.linalg.svd(A)  
  
print(u)  
[[-0.8816746 -0.47185793  0.          0.        ]  
 [-0.47185793  0.8816746   0.          0.        ]  
 [ 0.          0.          1.          0.        ]  
 [ 0.          0.          0.          1.        ]]  
  
print(s)  
[ 7.60555128  0.39444872]  
  
print(vh)  
[[-0.47185793 -0.8816746 ]  
 [ 0.8816746  -0.47185793]]
```

Thank you.

Numpy
ryp1662@gmail.com

Copyright © "Youngpyo Ryu" All Rights Reserved.
This document was created for the exclusive use of "Youngpyo Ryu".
It must not be passed on to third parties except with the explicit prior consent of "Youngpyo Ryu".