인공지능수학 개론

(저자 김종락)

(임시적인) 목록

Chapter 2. 선형대수 기초

선형대수는 대수학의 한 분야로 하나 이상의 변수로 이루어진 일차식의 해(solution)를 다루는 수학분야다. 예를 들어 $ax+by=c$ (a,b,c는 상수, x,y는 변수)는 변수가 두 개이고 식이 하나인 선형방정식이다. 이 해 집합을 그래프로 나타내면 직선이 나온다. 따라서 선형대수학은 기하학과 연관이 있다. 3개의 변수로 이루어진 선형 방정식 $ax+by+cz=d$의 경우 해 집합은 평면을 나타낸다. 일반적으로 n개의 변수로 이루어진 선형 방정식의 해 집합은 $(n-1)$-차원의 도형이 된다. 여러 개의 선형방정식을 연립선형방정식(a system of linear equations)라고 하고 선형대수는 연립선셩방정식의 해를 어떻게 효율적으로 구할지 연구하는 분야이다.

이제 반대로 어떤 점들(좌표들) 혹은 데이터들이 있다고 가정하자. 우선 아주 간단한 질문을 해보도록 하자. 이 주어진 점들이 어떤 선형 시스템의 해 혹은 부분적인 해로 표현될 수 있을까? 예를 들어 $xy-$평면에 주어진 $P_1=(0,0), P_2=(2,2)$가 주어졌을 때 이 두 점을 지나는 직선은 $y=x$ 혹은 $y-x=0$이 된다. 따라서 이 두 점은 선형방정식의 부분해로 이해할 수 있다. 실제로 임의의 서로 다른 두 점은 유일한 선형 방정식을 만들어 낸다.

질문을 좀 더 확장하여 위의 두 점 P_1, P_2외에 $P_3=(1, \frac{1}{2})$를 지나는 선형방정식은 존재할까? P_3는 $y-x=0$을 만족하지 않으므로 그런 방정식은 존재하지 않는다. 여기에 두 가지의 접근 방법이 존재한다. 수학자는 일차 방정식에서 탈피하여 2차 방정식 즉 $y=ax^2+bx+c$(단, a,b,c는 상수)을 생각하고 이 세 점이 이 2차 방정식의 해가 됨을 보일 것이다. 이제 데이터 사이언티스트의 관점에서 해결하도록 하자. 이 세 점들이 반드시 직선 위에 있지 않아도 된다는 가정을 하면 재미있는 현상이 벌어진다. 비록 직선 위에 있지 않더라도 **이 직선과 주어진 세 점의 거리를 최단으로 만드는 방법**이다. 이것이 더 현실적인 해법이다. 왜냐하면 이 방법은 주어진 점의 수가 증가한다고 하더라고 직선을 구하는 문제이기 때문에 고차원의 방정식의 해를 구하는 것보다 훨씬 쉽기 때문이다. 이 방법이 바로 **선형회귀 방법**이다. 선형회귀 방법은 머신러닝의 기초가 되고 딥러닝의 수식에도 등장한다. 이쯤 되면 왜 선형대수학을 공부해야 하는 것 충분이 설득이 되었으리나 믿는다.

이번 장에서는 선형대수의 기본적이 개념들과 이를 파이썬으로 구현해 보고자 한다.

■ 선형방정식

- **선형방정식(linear equation)**
  - 최고차항의 차수가 1인 방정식 (=**일차방정식**)

- **연립선형방정식(system of linear equations)**
  - 여러 선형방정식이 모여 있는 것

$$\begin{cases} 2x_1 + 3x_2 + 3x_3 = 9 \\ 3x_1 + 4x_2 + 2x_3 = 0 \\ -2x_1 + 2x_2 + 3x_3 = 2 \end{cases}$$

  - 연립선형방정식의 해(solution)
    : 연립선형방정식의 모든 선형방정식을 만족하는 미지수들의 값
  - 행렬과 벡터를 이용하여 표현 가능

$$\begin{bmatrix} 2 & 3 & 3 \\ 3 & 4 & 2 \\ -2 & 2 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 0 \\ 2 \end{bmatrix}$$

>> 선형방정식을 효율적으로 풀고 싶다면 어떻게 할까? 위의 변수들은 정말로 중요할까?

■ 행렬(Matrix)

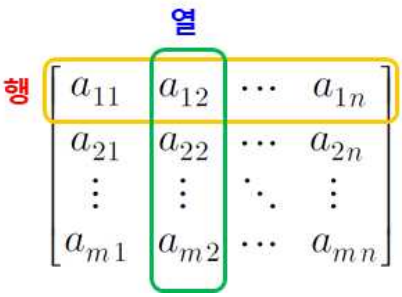
- 수나 식을 사각형 모양으로 배열하고 괄호로 묶어 놓은 것

$$\begin{bmatrix} 2 & 4 & 1 \\ 5 & 7 & 2 \end{bmatrix} \qquad \begin{bmatrix} a & a^2 & a^3 \\ a+b & a & 2b^2 \\ b^2 & ab & 3b \end{bmatrix}$$

- **성분**(원소, element)
- **행**(row)
- **열**(column)

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \end{bmatrix}$$

행벡터



$$\begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix}$$

열벡터

>

▪ 행렬의 종류

- **정방행렬(정사각행렬)**
  - 행과 열의 수가 같은 행렬

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

- **대각행렬**
  - 주대각 성분을 제외한 모든 성분이 0인 행렬

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **단위행렬(항등행렬)**
  - 주대각 성분이 모두 1이고, 나머지 성분은 모두 0인 정방행렬

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **전치행렬**
  - 어떤 행렬에서 모든 행을 각각 대응하는 열로 바꾼 행렬
  - A^{T} (A의 전치행렬)

$$A = \begin{bmatrix} 2 & 4 & 1 \\ 5 & 7 & 2 \end{bmatrix}, \quad A^{\top} = \begin{bmatrix} 2 & 5 \\ 4 & 7 \\ 1 & 2 \end{bmatrix}$$

- **대칭행렬**
  - 전치행렬이 자기 자신과 같은 행렬

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 2 \end{bmatrix}, \quad A^{\top} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 2 \end{bmatrix}$$

▶

▪ 행렬의 연산

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{bmatrix}$$

- **합(덧셈)**

$$A + B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+6 & 2+5 \\ 3+4 & 4+3 \\ 5+2 & 6+1 \end{bmatrix} = \begin{bmatrix} 7 & 7 \\ 7 & 7 \\ 7 & 7 \end{bmatrix}$$

- **차(뺄셈)**

$$A - B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} - \begin{bmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1-6 & 2-5 \\ 3-4 & 4-3 \\ 5-2 & 6-1 \end{bmatrix} = \begin{bmatrix} -5 & -3 \\ -1 & 1 \\ 3 & 5 \end{bmatrix}$$

- **스칼라배(스칼라곱)**

$$10A = 10\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 1\times 10 & 2\times 10 \\ 3\times 10 & 4\times 10 \\ 5\times 10 & 6\times 10 \end{bmatrix} = \begin{bmatrix} 10 & 20 \\ 30 & 40 \\ 50 & 60 \end{bmatrix}$$

▶

- **곱 연산**

  A의 열 개수와 B의 행 개수가 같을 때 AB의 곱

$$A=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix},\ B=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} \Rightarrow\ AB=C=\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij}=a_{i1}b_{1j}+\cdots+a_{in}b_{nj}=\sum_{k=1}^{n}a_{ik}b_{kj} \quad (\text{단},\ 1\le i,j\le n)$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}=\begin{bmatrix} 1\times5+2\times7 & 1\times6+2\times8 \\ 3\times5+4\times7 & 3\times6+4\times8 \end{bmatrix}=\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

▶ 두 행렬의 곱이 자연스러운가?

■ 행렬과 연립선형방정식의 관계

▪ 연립선형방정식의 행렬 표현

$$\begin{cases} x_1 + 2x_2 + x_3 = 3 \\ 3x_1 - 2x_2 - 3x_3 = -1 \\ 2x_1 + 3x_2 + x_3 = 4 \end{cases} \Rightarrow \begin{bmatrix} 1 & 2 & 1 \\ 3 & -2 & -3 \\ 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \\ 4 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 3 & -2 & -3 \\ 2 & 3 & 1 \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 3 \\ -1 \\ 4 \end{bmatrix}$$

$$\Rightarrow A\boldsymbol{x} = \boldsymbol{b}$$

계수행렬 — A

상수벡터 — $\boldsymbol{b}$

▶ 연립선형방정식을 행렬과 백터의 곱 $Ax = b$로 나타낼 수 있다. 이때 $Ax = b$을 행렬 식이라고 부른다.

▶ 연립선형방정식을 동치인 다른 연립선형방정식으로 변환할 수 있다. 마찬가지로 이에 대응하는 행렬방정식도 같이 변환된다.

(1) 두 선형방정식의 위치를 교환하는 것은 행렬방정식의 계수행렬과 상수벡터에서 대응하는 두 행을 교환하는 것과 같다.
(2) 선형방정식의 양변에 0이 아닌 상수를 곱하는 것은 행렬방정식의 계수행렬과 상수벡터에서 대응하는 행에 해당 상수를 곱하는 것과 같다.
(3) 특정 선형방정식의 0이 아닌 상수배를 다른 선형방정식에 더하는 것은 행렬방정식의 계수행렬과 상수벡터에서 대응하는 한 행의 상수배를 다른 행에 더하는 것과 같다.

▶

●예제

$$\begin{cases} x_1+2x_2+x_3=3 \\ 3x_1-2x_2-3x_3=-1 \\ 2x_1+3x_2+x_3=4 \end{cases} \Leftrightarrow \begin{bmatrix} 1 & 2 & 1 \\ 3 & -2 & -3 \\ 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \\ 4 \end{bmatrix}$$

- **첫 번째와 두 번째 방정식 교체**

$$\begin{cases} 3x_1-2x_2-3x_3=-1 \\ x_1+2x_2+x_3=3 \\ 2x_1+3x_2+x_3=4 \end{cases} \Leftrightarrow \begin{bmatrix} 3 & -2 & -3 \\ 1 & 2 & 1 \\ 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \\ 4 \end{bmatrix}$$

- **첫 번째 방정식의 3배**

$$\begin{cases} 3x_1+6x_2+3x_3=9 \\ 3x_1-2x_2-3x_3=-1 \\ 2x_1+3x_2+x_3=4 \end{cases} \Leftrightarrow \begin{bmatrix} 3 & 6 & 3 \\ 3 & -2 & -3 \\ 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ -1 \\ 4 \end{bmatrix}$$

- **첫 번째 방정식의 2배를 세 번째 방정식에 더하기**

$$\begin{cases} x_1+2x_2+x_3=3 \\ 3x_1-2x_2-3x_3=-1 \\ 4x_1+7x_2+3x_3=10 \end{cases} \Leftrightarrow \begin{bmatrix} 1 & 2 & 1 \\ 3 & -2 & -3 \\ 4 & 7 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \\ 10 \end{bmatrix}$$

▶ 이제 행렬식에 나타나는 변수 벡터는 중용하지 않음을 알 수 있다.

▶ 행렬에서 각 행의 맨 왼쪽에 0이 아닌 성분을 피벗(pivot) 또는 추축성분(pivot entry)라고 한다.

▶ 이중에서 row echelon form이 중요하다.

다음 조건 (1), (2), (3)을 만족하는 행렬을 **행 사다리꼴 행렬**row echelon form matrix이라 한다.

(1) 모든 성분이 0인 행은 맨 아래에 위치한다.
(2) 모든 행의 추축성분은 위쪽 행의 추축성분보다 오른쪽 열에 있다.
(3) 모든 추축성분은 1이고, 추축성분 아래쪽의 모든 성분은 0이다.

- **행 사다리꼴 행렬의 예**

$$\begin{bmatrix} 1 & 4 & 5 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- **행 사다리꼴 행렬이 아닌 예**

$$\begin{bmatrix} 2 & 4 & 5 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

▶ ref를 더 심화한 행렬 형태가 reduced row echelon form(rref)이다.

모든 추축성분이 해당 열에서 0이 아닌 유일한 성분인 행 사다리꼴 행렬을 **기약행 사다리꼴 행렬**reduced row echelon form matrix 또는 **축약행 사다리꼴 행렬**이라 한다.

- **기약행 사다리꼴 행렬의 예**

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

●예제

다음 각 행렬이 행 사다리꼴 행렬인지 기약행 사다리꼴 행렬인지 설명하라.

(a) $A = \begin{bmatrix} 1 & 3 & 0 \\ 0 & 2 & 3 \\ 0 & 0 & 1 \end{bmatrix}$ (b) $B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

(c) $C = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ (d) $D = \begin{bmatrix} 1 & 0 & 5 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

■ 벡터(Vector)

- 행이나 열이 하나 밖에 없는 행렬

- **행벡터**(row vector)

$$B = \begin{bmatrix} 4 & 6 & 7 & 9 \end{bmatrix}$$

- **열벡터**(column vector)

$$A = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

▶ 벡터 -> 행렬 -> ?

● 예제

$A=\begin{bmatrix} 3 & 0 \\ -1 & 2 \\ 1 & 1 \end{bmatrix}$, $B=\begin{bmatrix} -3 & -1 \\ 2 & 1 \\ 4 & 3 \end{bmatrix}$, $C=\begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 1 \end{bmatrix}$, $D=\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ 일 때, 다음 식을 계산하라.

(a) $A+2B$ (b) AC (c) $3D$

■ 실습(행렬과 벡터)

(Reference 파이썬과 NumPy로 배우는 선형대수, 이정주(BJ 출판사))

```
#이미 정행진 행렬들에 대하여 알아보기

import numpy as np
A = np.zeros((2,3))
print(A)
```

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

▶

```
A.dtype  #64비트 실수 타입
```

```
dtype('float64')
```

```
B = np.ones((2,3))
print(B)
B.dtype
```

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

dtype('float64')
```

```
print(A+B)
```

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

▶

```
I= np.eye(3)
print(I)
np.multiply(3, I) # scalar multiple of I by scalar 3
```

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

array([[3., 0., 0.],
       [0., 3., 0.],
       [0., 0., 3.]])
```

▶

```
J = np.ones((3,3))
print(J)
print(np.dot(J,J))
print(np.dot(I,J))
print(J.dot(J))
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
[[3. 3. 3.]
 [3. 3. 3.]
 [3. 3. 3.]]
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
[[3. 3. 3.]
 [3. 3. 3.]
 [3. 3. 3.]]
```

▶

```
# 0~1사이의 수에 대하여 random 함수를 이용하여 행렬 생성하기
D = np.random.random((2,3))
print(D)
```

```
[[0.3597899  0.16629705 0.12115443]
 [0.47939376 0.87808984 0.69995902]]
```

```
# using randint을 사용하여 임의의 자연수 만들기
E = np.random.randint(1, 10, (2,3)) #1~9
print("E=", E)
print()
F = np.random.randint(10, 20, (2,3)) # 10~19
print("F=", F)
```

```
E= [[2 8 3]
 [8 5 4]]

F= [[15 13 15]
 [11 19 10]]
```

```
# 일정한 간격의 수열로 1*n 행렬 만들기
# arange를 이용하여 0~30 범위에서 4씩 증가하여 행렬로 만들기
A = np.arange(0, 30, 4)
print("A =", A)

B = np. arange(2, 10)
print("B =", B)  # 세번째 인자가 없으면 1씩 증가

C = np.arange(10)
print("C=", C)  #인자가 하나면 0부터 시작, 1씩 증가
```

```
A = [ 0  4  8 12 16 20 24 28]
B = [2 3 4 5 6 7 8 9]
C= [0 1 2 3 4 5 6 7 8 9]
```

```
#  [a, b] 범위내에서 고정된 샘플수 생성. linspace 사용
A = np.linspace(0, 10, 5)
print("A =", A)
type(A)
```

```
A = [ 0.   2.5  5.   7.5 10. ]

numpy.ndarray
```

```
# reshape를 이용하여 1*n행렬을 m*n행렬로 변환하기
A = np.arange(12)
print("A = ", A)

B = A.reshape(3,4)
print("B= ", B)

print("B.shape=", B.shape)
```

```
A =  [ 0  1  2  3  4  5  6  7  8  9 10 11]
B=  [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
B.shape= (3, 4)
```

▶

```
print("B[0]=", B[0])# 행렬 B의 첫 번째 행, 0부터 인덱스 시작
print("B[0,1]=", B[0,0]) # 행과 열 모두 0부터 인덱스 시작
```

```
B[0]= [0 1 2 3 4 5]
B[0,1]= 0
```

```
# ravel 메소드이용하여 m*n 행렬을 1*n 행렬로 변환하기
A = np.array([[1,3], [5,7]])
print("A=", A)
A.shape
```

```
A= [[1 3]
 [5 7]]

(2, 2)
```

```
B = A.ravel()
print("B=", B)  #B는 뷰임
```

```
B= [1 3 5 7]
```

```
B.shape
```

```
(4,)
```

```
B.base # 실제적인 B
```

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

▶

```
# resize는 뷰는 생성하지 않고 바로 형태를 바꿈
A = np.arange(10)

print("A=", A)
```

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

```
A.resize(2,5)
```

```
print("A=", A) # 새로운 A로 변함
```

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

```
import numpy as np

A = np.array([[1,3], [5,7]])
B = np.array([[2,4], [6,8]])
print(A)
print(B)
```

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

```
 # 세로로 붙이기
C = np.vstack((A,B))
print(C)
C.shape
```

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

(4, 2)
```

▶

```
 # 가로로 붙이기
C = np.hstack((A,B))
print(C)
C.shape
```

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

(2, 4)
```

```
# concatenate 함수로 vstack과 hstack을 작성할 수 있음
import numpy as np

A = np.array([[1,3], [5,7]])
B = np.array([[2,4], [6,8]])

C = np.concatenate((A,B), axis =0) # 열방향으로
print(C)
print()
D =  np.concatenate((A,B), axis =1) # 행방향으로
print(D)
```

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

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

▶

```
import numpy as np

A = np.arange(0, 40, 2).reshape(4,5)
print(A)
```

```
[[ 0  2  4  6  8]
 [10 12 14 16 18]
 [20 22 24 26 28]
 [30 32 34 36 38]]
```

```
# A의 전치행렬 transpose
A.T
```

```
array([[ 0, 10, 20, 30],
       [ 2, 12, 22, 32],
       [ 4, 14, 24, 34],
       [ 6, 16, 26, 36],
       [ 8, 18, 28, 38]])
```

```
# 슬라이싱

A = np.arange(1, 10, 2)
print(A)
```

```
[1 3 5 7 9]
```

```
A[0:2]  # 0부터 1까지
```

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

```
A[:2] # 0생략 가능
```

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

```
A[:] # 전체 생성
```

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

```
A[::2]  # 두 칸씩 건너뛰기
```

```
array([1, 5, 9])
```

```
A[: -2]  # 뒤에서 2칸을 제외하기
```

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

```
A[-2 :] # 마지막 2개만 선택
```

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

▶

```
### array 합, 차, 곱, 나누기

import numpy as np
a = np.array([1,3,5,7])
b = np.array([2,4,6,8])
print(a)
print(b)
print(a+b)
print(a-b)
print(a*b)  # 성분별 곱하기
print(a/b)  # 성분별 나누기
```

```
[1 3 5 7]
[2 4 6 8]
[ 3  7 11 15]
[-1 -1 -1 -1]
[ 2 12 30 56]
[0.5        0.75       0.83333333 0.875     ]
```

▶

```python
###2차원 배열을 이용하여 행렬 표현하기 *****
import numpy as np

A = np.array([[1,3],[5,7]])  # 큰 []안에 벡터를 넣는다.
print(A)
print()
B = np.array([[2,4],[6,8]])
print(B)
print()
print(A+B)
print()     # 한 줄 띄우기
print(A-B)
print()
print(A*B)  #성분별 곱
print()
print(A/B)  #성분별 나누기
```

```
[[1 3]
 [5 7]]

[[2 4]
 [6 8]]

[[ 3  7]
 [11 15]]

[[-1 -1]
 [-1 -1]]

[[ 2 12]
 [30 56]]

[[0.5        0.75      ]
 [0.83333333 0.875     ]]
```

```
### 행렬의 곱 dot이용하기
import numpy as np

A = np.array([[1,3],[5,7]])  # 큰 []안에 벡터를 넣는다.
B = np.array([[2,4],[6,8]])
C = np.dot(A,B)   # dot 함수 이용하여 행렬의 곱을  정의함. *****
print(A)
print()
print(B)
print()
print(C)
```

```
[[1 3]
 [5 7]]

[[2 4]
 [6 8]]

[[20 28]
 [52 76]]
```

▶

```
### 브로드캐스트
import numpy as np

A = np.array([[1,3],[5,7]])  # 큰 []안에 벡터를 넣는다.
B = np.array([[2,4],[6,8]]) # 2*2 사이즈
C = np.array([[2,4]])       #1*2사이즈 copy this row to the second row
D = np.array([[2],  # column can be repeated
              [4]])
E = 3  # one number can be repeated(broadcast)
print(A.shape) #A의 사이즈는?

print(C.shape)
print(A+B)
print()
print(A+C)  # 행렬의 사이즈가 다르지만  [2,4]행을 반복한다.
print()
print(A+D)  # 행렬의 사이즈가 다르지만 [2,4]^T 열을 반복한다.
print()
print(A+E)  # 각 성분에 2를 더한다.
```

```
(2, 2)
(1, 2)
[[ 3  7]
 [11 15]]

[[ 3  7]
 [ 7 11]]

[[ 3  5]
 [ 9 11]]

[[ 4  6]
 [ 8 10]]
```

▶