

02_Numpy 활용

• Reshape

새로운 array를 생성한 것임
기존의 array는 수정, 변형된 것이 아님

- 배열의 shape을 변형해주는 함수
- 배열의 요소 개수를 유지하며 형태만 변경
- 변형 전 size와 변형 후 size가 동일해야 함 차원도 변경 가능

> `array.reshape(size)`

```
arr = np.array([[1, 2, 3], [4, 5, 6]])    # 2x3 형태의 2차원 배열  
print(arr.reshape(3,2))    # 3x2 형태의 배열로 변형
```

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

- Reshape

03

> 6 x 1 배열로 변형

```
print(arr.reshape(6, 1))
```

Out : `[[1]`

`[2]`

`[3]`

`[4]`

`[5]`

`[6]]`

> 1 x 6 배열로 변형

```
print(arr.reshape(1, 6))
```

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

• Reshape

04

> 3 x 1 x 2 배열로 변형

```
print(arr.reshape(3, 1, 2))
```

Out : [[[1 2]]

[[3 4]]

[[5 6]]]

3차원으로 변경



> ? x 2 배열로 변형

```
print(arr.reshape(-1, 2))
```

Out : [[1 2]

[3 4]

[5 6]]

미지수처럼 -1 사용됨

• Reshape

> [문제1]

`range()` 함수를 사용해 1부터 36까지의 숫자가 나열된 `tensor` 배열을 생성하고 (3, 3, 4) 형태의 3차원 배열로 변형해봅시다.

```
tensor=np.array(range(1,37))  
tensor.reshape(3,3,4)
```

> [문제2]

`tensor` 배열을 2차원 배열로 변형시켜봅시다. * 열의 개수는 9

```
tensor.reshape(-1,9)
```

- Reshape

- > [문제1]

range() 함수를 사용해 1부터 36까지의 숫자가 나열된 tensor 배열을 생성하고 (3, 3, 4) 형태의 3차원 배열로 변형해봅시다.

```
tensor = np.array(range(1, 37)).reshape(3, 3, 4)
```

Out :

```
[[[ 1  2  3  4]
    [ 5  6  7  8]
    [ 9 10 11 12]]
 ...
 [[25 26 27 28]
 [29 30 31 32]
 [33 34 35 36]]]
```

- Reshape

> [문제2]

tensor 배열을 2차원 배열로 변형시켜봅니다. * 열의 개수는 9

```
print(tensor.reshape(-1, 9))
```

Out : `[[1 2 3 4 5 6 7 8 9]
[10 11 12 13 14 15 16 17 18]
[19 20 21 22 23 24 25 26 27]
[28 29 30 31 32 33 34 35 36]]`

• Flatten

- 다차원 배열을 1차원 배열로 변형

> `shape(2, 3) -> shape (6,)`

```
print(arr.flatten())
```

Out : [1 2 3 4 5 6]

> `shape(1, 6) -> shape (6,)`

```
print(arr.reshape(1, 6).flatten())
```

Out : [1 2 3 4 5 6]

> `shape(3, 1, 2) -> shape (6,)`

```
print(arr.reshape(3, 1, 2).flatten())
```

Out : [1 2 3 4 5 6]

• Arange

- range 함수와 비슷한 기능
- 값의 범위를 지정하여 값이 채워져 있는 배열을 생성
- 특정한 규칙에 따라 증가하는 값을 넣는 것도 가능

> np.arange(size)

```
print(np.arange(30))
```

Out : [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]

Numpy 활용

- Arange 바로 배열을 만드는 함수

> 간격 지정 : np.arange(start, end, step)

```
print(np.arange(0, 10, 2))
```

Out : [0 2 4 6 8]

```
print(np.arange(0, 2, 0.2))
```

Out : [0. 0.2 0.4 0.6 0.8 1. 1.2 1.4 1.6 1.8]

010

|

- Arange + Reshape

- 배열 요소를 순서대로 가지는 다차원 배열 생성 가능

> arange + reshape

```
print(np.arange(1, 26).reshape(5,5))
```

Out : [[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]]

• Arange + Reshape

- 배열 요소를 순서대로 가지는 다차원 배열 생성 가능

> **arange + reshape**

```
print(np.arange(0, 6, 0.2).reshape(3, 2, 5))
```

Out :

```
[[[0.  0.2 0.4 0.6 0.8]
  [1.  1.2 1.4 1.6 1.8]]

 [[2.  2.2 2.4 2.6 2.8]
  [3.  3.2 3.4 3.6 3.8]]

 [[4.  4.2 4.4 4.6 4.8]
  [5.  5.2 5.4 5.6 5.8]]]
```

- Arange + Reshape

- > [문제3]

arange, reshape 함수를 사용해 100부터 190까지의 숫자가 10 간격으로 나열된 배열을 생성하고 (2, 5) 형태의 2차원 배열로 변형해봅시다.

- > [문제4]

arange, reshape 함수를 사용해 -2부터 2까지의 숫자가 0.5 간격으로 나열된 배열을 생성하고 (3, 3) 형태의 2차원 배열로 변형해봅시다.

- Arange + Reshape

> [문제3]

arange, reshape 함수를 사용해 100부터 190까지의 숫자가
10 간격으로 나열된 배열을 생성하고 (2, 5) 형태의 2차원 배열로
변형해봅시다.

```
print(np.arange(100,191,10).reshape(2,5))
```

Out : `[[100 110 120 130 140]
[150 160 170 180 190]]`

- Arange + Reshape

> [문제4]

arange, reshape 함수를 사용해 -2부터 2까지의 숫자가
0.5 간격으로 나열된 배열을 생성하고 (3, 3) 형태의 2차원 배열로
변형해봅시다.

```
print(np.arange(-2, 2.1, 0.5).reshape(3,3))
```

Out :
[[-2. -1.5 -1.]
 [-0.5 0. 0.5]
 [1. 1.5 2.]]

• Zeros

- 0으로 채워진 배열 생성
- shape 지정이 가능

> np.zeros(shape, dtype)

```
print(np.zeros(shape=(5, ), dtype=np.int8))
```

Out : [0 0 0 0 0]

```
print(np.zeros((5), int))
```

Out : [0 0 0 0 0]

```
print(np.zeros((2,3)))
```

Out : [[0. 0. 0.]
[0. 0. 0.]]

• Ones

- 1으로 채워진 배열 생성
- shape 지정이 가능

> **np.ones(shape, dtype)**

```
print(np.ones(shape=(5, ), dtype=np.int8))
```

Out : [1 1 1 1 1]

```
print(np.ones((5), int))
```

Out : [1 1 1 1 1]

```
print(np.ones((2,3)))
```

Out : [[1. 1. 1.]
[1. 1. 1.]]

• Empty

- shape만 주어지고 비어있는 배열 생성(초기화 하지 않음)
- 배열을 생성만 하고 값을 주지 않아 메모리에 저장되어 있던 기존 값이 저장될 수 있음

python에는 null 값이 없음
이미 메모리에 저장된 바로 앞에 실행한 값들을 가져옴

> **np.empty(shape, dtype)**

```
print(np.empty(shape = (10,), dtype = np.int32))
```

Out : [0 0 0 0 0 0 0 0 0 0]

```
print(np.empty((2,5), np.int8))
```

Out : [[0 0 0 0 0]
[0 0 0 0 0]]

- Zeros_like / Ones_like / Empty_like

- 입력 받은 배열과 같은 shape, dtype의 배열 생성

> np.zeros_like(ndarray)

```
np.ones_like(ndarray)
```

```
matrix = np.arange(24).reshape(4, 6)
print(np.zeros_like(matrix))
```

Out :

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

```
matrix = np.arange(-2, 2, 0.2).reshape(2, 10)
print(np.ones_like(matrix))
```

Out :

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

- Zeros_like / Ones_like / Empty_like

- 입력 받은 배열과 같은 shape, dtype의 배열 생성

> np.empty_like(ndarray) 원본의 값을 그대로 가지고 올 경우가 많음

```
matrix = np.arange(30).reshape(5, 6)
print(np.empty_like(matrix))
```

Out : [[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]]

• Identity

- 단위 행렬 – 주대각선의 값이 1이고 나머지는 0인 정사각 행렬

> `np.identity(size, dtype)` 곱했을 때 자신의 값이 나오는

```
print(np.identity(5, dtype=np.int8))
```

Out : `[[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]]`

```
print(np.identity(2))
```

Out : `[[1. 0.]
[0. 1.]]`

- Eye

- 대각선이 1로 채워지는 행렬
- 대각선의 시작 위치 지정 가능

> `np.eye(size, M, k, dtype)`

```
print(np.eye(5, dtype=np.int8))
```

Out : `[[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]]`

• Eye

- k : 대각선의 시작 위치를 입력 받은 수 만큼 오른쪽으로 이동
- M : 보여지는 열의 개수를 지정

> `np.eye(size, M, k, dtype)`

```
print(np.eye(5, k = 2))
```

Out : `[[0. 0. 1. 0. 0.]`
`[0. 0. 0. 1. 0.]`
`[0. 0. 0. 0. 1.]`
`[0. 0. 0. 0. 0.]`
`[0. 0. 0. 0. 0.]]`

• Eye

- k : 대각선의 시작 위치를 입력 받은 수 만큼 오른쪽으로 이동
- M : 보여지는 열의 개수를 지정

> `np.eye(size, M, k, dtype)`

```
print(np.eye(5, M = 10, k = -1))
```

Out : `[[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. 0. 0. 0. 0. 0. 0.]]`

• Full

- 입력 값으로 채워지는 행렬

자료형 넣지않아도 입력값에 따라 자동으로 정해짐

> `np.full((shape), value)`

```
print(np.full((2, 5), 3))
```

Out : `[[3 3 3 3 3]
[3 3 3 3 3]]`

```
print(np.full((5, 4), 'a'))
```

Out : `[['a' 'a' 'a' 'a']
['a' 'a' 'a' 'a']
['a' 'a' 'a' 'a']
['a' 'a' 'a' 'a']
['a' 'a' 'a' 'a']]`

• Random.randint

- 주어진 범위 안의 랜덤한 값을 뽑는 함수
정수

> `np.random.randint(start, end, shape)`

```
# 0~5 까지의 정수 중 하나 뽑기  
print(np.random.randint(6))
```

Out : 5

```
# 1~19 까지의 정수 중 하나 뽑기  
print(np.random.randint(1,20))
```

Out : 11

```
# 1~9 까지의 정수 (2, 5) 모양으로 뽑기  
print(np.random.randint(1, 10, (2,5)))
```

Out :
[[5 5 7 9 8]
 [5 5 3 4 4]]

• Random.rand

- 랜덤한 값으로 채워지는 배열
- 표준정규분포 난수를 shape 형태의 배열로 생성
- 0 ~ 1 사이의 값을 가짐 실수

> **np.random.rand(shape)**

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

Out : [0.57905605 0.00533085 0.12858647]

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

Out : [[0.68689508 0.32713029]
[0.6481698 0.35997786]
[0.57364588 0.79425818]]

• Random.randn

- 랜덤한 값으로 채워지는 배열
- 평균 0, 표준편차 1의 가우시안 표준정규분포 난수를 shape 형태의

배열로 생성 음수를 포함한 랜덤한 수 배열

거의 합치면(?) 0에 가까움

> **np.random.randn(shape)**

```
print(np.random.randn(3))
```

Out : [-0.01259062 -1.0451196 1.35525447]

```
print(np.random.randn(3,2))
```

Out : [[0.69510343 -2.16322795]

[-1.13541398 -1.63182969]

[-0.48371222 0.77480652]]

완전히 규격없이 random 값 구할때(0-1)
.random.random_sample()

Numpy 활용

• 연산함수

통계의 요약정보(합, 평균, 최대값 등

- 배열의 요소값들을 이용하여 연산을 할 수 있게 해주는 함수

sum (합)	mean (평균)	max (최대값)	min (최소값)
log (로그)	sqrt (제곱근)	std (표준편차)	exp (지수)
sin (삼각함수)	cos (삼각함수)	tan (삼각함수)	abs / fabs (절대값)
ceil (올림)	floor (버림)	round (반올림)	mod (나머지)
add (덧셈)	subtract (뺄셈)	multiply (곱셈)	divide (나눗셈)
power (제곱)	sort, median, var, ...		

• 합 (sum)

- 요소들의 합을 구해주는 함수 특이한 경우
- `ndarray.sum()` 모든 요소의 합

> 2차원 배열 합

```
matrix = np.arange(1, 7).reshape(2, 3)
print(matrix.sum())
```

Out : 21

```
print(matrix.sum(axis=0)) # 행끼리 더하기(위 아래)
print(matrix.sum(axis=1)) # 열끼리 더하기(좌 우)
```

Out : [5 7 9]

[6 15]

- 합 (sum)

> 3차원 배열 합

```
tensor = np.arange(1, 19).reshape(3, 2, 3)
print(tensor)
print(tensor.sum())
```

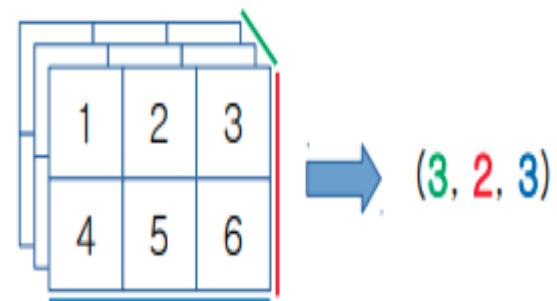
Out :
[[[1 2 3]
 [4 5 6]]
 [[7 8 9]
 [10 11 12]]
 [[13 14 15]
 [16 17 18]]]
171

- 합 (sum)

- > 3차원 배열 합

```
print(tensor.sum(axis=0))  
print(tensor.sum(axis=1))  
print(tensor.sum(axis=2))
```

Out : **[[21 24 27]**
 [30 33 36]]
 [[5 7 9]
 [17 19 21]
 [29 31 33]]
 [[6 15]
 [24 33]
 [42 51]]



- 평균 (mean)

- 합과 동일

> 2차원 배열 평균

기본이 실수로 표현

```
print(matrix.mean())          # 전체 평균  
print(matrix.mean(axis=0))    # 행끼리 평균(위 아래)  
print(matrix.mean(axis=1))    # 열끼리 평균(좌 우)
```

Out : 3.5

[2.5 3.5 4.5]

[2. 5.]

- 표준 편차 (std)

- 합, 평균과 동일

- > 2차원 배열 표준 편차

```
print(matrix.std())           # 전체 표준 편차
print(matrix.std(axis=0))     # 행끼리 표준 편차(위 아래)
print(matrix.std(axis=1))     # 열끼리 표준 편차(좌 우)
```

Out : 1.707825127659933

[1.5 1.5 1.5]

[0.81649658 0.81649658]

분산 .var()

- 그 외 수학 연산자

035

> 지수 (exp) 제공

```
print(np.exp(matrix))
```

Out : **[[2.71828183 7.3890561 20.08553692]**
[54.59815003 148.4131591 403.42879349]]

> 로그 (log)

```
print(np.log(matrix))
```

Out : **[[0. 0.69314718 1.09861229]**
[1.38629436 1.60943791 1.79175947]]

- 그 외 수학 연산자

- > 제곱근 (sqrt)

```
print(np.sqrt(matrix))
```

```
Out : [[1.         1.41421356 1.73205081]
       [2.         2.23606798 2.44948974]]
```

- > 삼각함수 (sin)

```
print(np.sin(matrix))
```

```
Out : [[ 0.84147098  0.90929743  0.14112001]
       [-0.7568025  -0.95892427 -0.2794155 ]]
```

- 그 외 수학 연산자

- > 삼각함수 (cos)

```
print(np.cos(matrix))
```

Out : `[[0.54030231 -0.41614684 -0.9899925]
[-0.65364362 0.28366219 0.96017029]]`

- > 삼각함수 (tan)

```
print(np.tan(matrix))
```

Out : `[[1.55740772 -2.18503986 -0.14254654]
[1.15782128 -3.38051501 -0.29100619]]`

- 산술연산 배열끼리의 산술연산

- 배열은 기본적으로 요소의 연산이 가능

- > 리스트의 덧셈

```
list1 = [1,2,3]
list2 = [4,5,6]
print(list1 + list2)
```

Out : [1, 2, 3, 4, 5, 6]

- > 배열의 덧셈 같은 자리의 요소끼리 계산
shape가 같아야 동작함

```
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
print(arr1 + arr2)
```

Out : [5 7 9]

• 산술 연산

- shape이 같은 배열은 연산 가능

> 모양이 같은 배열 연산

```
arr1 = np.arange(1, 7, dtype=np.float32).reshape(2, 3)
arr2 = np.arange(11,17, dtype=np.int32).reshape(2, 3)
print(arr1 + arr2) # 더하기
```

Out : **[[12. 14. 16.]**
[18. 20. 22.]]

```
print(arr2 - arr1) # 빼기
```

Out : **[[10. 10. 10.]**
[10. 10. 10.]]

• 산술 연산

- shape이 같은 배열은 연산 가능

> 모양이 같은 배열 연산

```
print(arr1 * arr2) # 곱하기
```

Out : **[[11. 24. 39.]**
[56. 75. 96.]]

```
print(arr2 / arr1) # 나누기
```

Out : **[[11. 6. 4.33333333]**
[3.5 3. 2.66666667]]

```
print(arr1 ** arr2) # 제곱
```

Out : **[[1.00000000e+00 4.09600000e+03 1.59432300e+06]**
[2.68435456e+08 3.05175781e+10 2.82110991e+12]]

• 산술 연산

- shape이 같은 배열은 연산 가능

> 모양이 같은 배열 연산

```
print(arr2 // arr1) # 몫
```

Out : `[[11. 6. 4.]`
`[3. 3. 2.]]`

```
print(arr2 % arr1) # 나머지
```

Out : `[[0. 0. 1.]`
`[2. 0. 4.]]`

- 산술연산

- Numpy의 연산 함수

```
> np.add(arr1, arr2)
```

```
print(arr1 + arr2)
```

```
Out : [[12. 14. 16.]  
       [18. 20. 22.]]
```

```
print(np.add(arr1, arr2))
```

```
Out : [[12. 14. 16.]  
       [18. 20. 22.]]
```

• 산술연산

- Numpy의 연산 함수

> `np.subtract(arr1, arr2)`

```
print(arr1 - arr2)
```

꼭 row 수가 같아야

Out : `[[-10. -10. -10.]`
`[-10. -10. -10.]]`

```
print(np.subtract(arr1, arr2))
```

Out : `[[-10. -10. -10.]`
`[-10. -10. -10.]]`

- 산술연산

- Numpy의 연산 함수

> `np.multiply(arr1, arr2)`

```
print(arr1 * arr2)
```

Out : `[[11. 24. 39.]`
`[56. 75. 96.]]`

```
print(np.multiply(arr1, arr2))
```

Out : `[[11. 24. 39.]`
`[56. 75. 96.]]`

• 산술연산

- Numpy의 연산 함수

> **np.divide(arr1, arr2)**

옆으로 이어붙이기: 꼭 row 수가 같아야

```
print(arr1 / arr2)
```

Out : `[[0.09090909 0.16666667 0.23076923]
[0.28571429 0.33333333 0.375]]`

```
print(np.divide(arr1, arr2))
```

Out : `[[0.09090909 0.16666667 0.23076923]
[0.28571429 0.33333333 0.375]]`

• 산술연산

▪ Broadcasting

Broadcasting

scalar연산일 경우 모두 허용됨

1차원과 2차원: 행과 열 중 하나가 같을때

2차원과 3차원의 경우: shape 중 2가지가 같을때

- 연산하고자 하는 배열의 모양이 다른 경우의 연산

> 일반적인 상황

```
arr3 = np.arange(10).reshape(5, 2)
```

```
arr4 = np.arange(10).reshape(2, 5)
```

```
print(arr3 + arr4)
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-35-1a8162fbbf08> in <module>  
      1 arr3 = np.arange(10).reshape(5, 2)  
      2 arr4 = np.arange(10).reshape(2, 5)  
----> 3 print(arr3 + arr4)
```

```
ValueError: operands could not be broadcast together with shapes (5,2) (2,5)
```

• 산술 연산

- **Broadcasting** : shape이 다른 배열 간 연산 지원

> **배열과 스칼라의 연산** scala의 값은 배열 연산 모두 가능

```
scalar = 10  
vector = np.array([1, 2, 3])  
print(scalar + vector)
```

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 10 & 10 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \end{bmatrix}$$

Out : [11 12 13]

```
scalar = 10  
matrix = np.array([[1, 2, 3], [4, 5, 6]])  
print(scalar + matrix)
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 14 & 15 & 16 \end{bmatrix}$$

**Out : [[11 12 13]
[14 15 16]]**

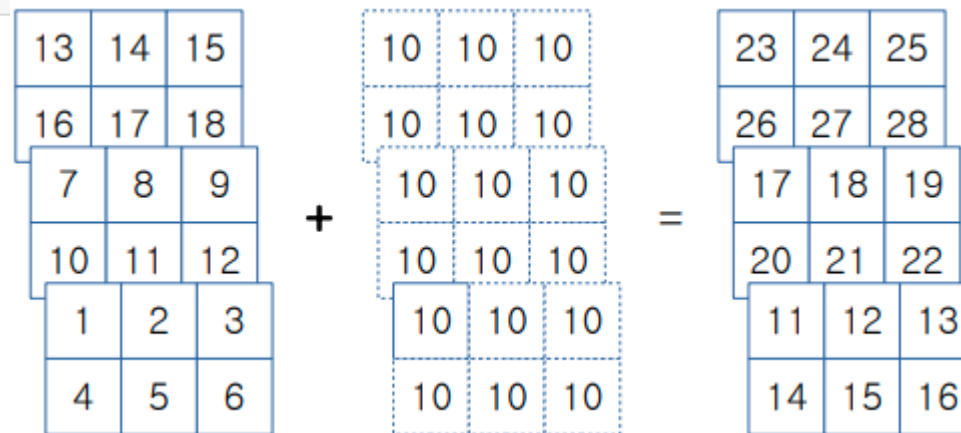
• 산술연산

- Broadcasting : shape이 다른 배열 간 연산 지원

> 배열과 스칼라의 연산

```
scalar = 10
tensor = np.array([[[1, 2, 3],[4, 5, 6]],
                  [[7, 8, 9], [10, 11, 12]],
                  [[13, 14, 15], [16, 17, 18]]])
print(scalar + tensor)
```

Out : **[[[11 12 13]**
 [14 15 16]]
 [[17 18 19]
 [20 21 22]]
 [[23 24 25]
 [26 27 28]]]



• 산술연산

미이여붙이기

049

> 벡터와 매트릭스의 연산

```
matrix = np.arange(1,10).reshape(3, 3)
vector = np.arange(11,14).reshape(1, 3)
print(matrix + vector)
```

Out : **[[12 14 16]**
[15 17 19]
[18 20 22]]

1	2	3		11	12	13		12	14	16
4	5	6	+	11	12	13	=	15	17	19
7	8	9		11	12	13		18	20	22

• 산술연산

050

> 벡터와 매트릭스의 연산

같은 배열의 모양일때
즉 행이나 열이 같을때

```
matrix = np.arange(1,10).reshape(3,3)
vector = np.arange(11,14).reshape(3,1)
print(matrix + vector)
```

옆으로 옆으로 이어붙이기: 꼭 row 수가 같아야

Out : [[12 13 14]

[16 17 18]

[20 21 22]]

1	2	3		11	11	11		12	13	14
4	5	6	+	12	12	12	=	16	17	18
7	8	9		13	13	13		20	21	22

같은 차원일때 숫자가 2개만 같아도 가능함

• 산술연산

051

> 벡터와 벡터의 브로드캐스팅 연산

```
arr1 = np.array([1,2,3]) (3,)  
arr2 = np.array([1,2,3]).reshape(3, 1)  
print(arr1)  
print(arr2)
```

Out : [1 2 3]

[1]

[2]

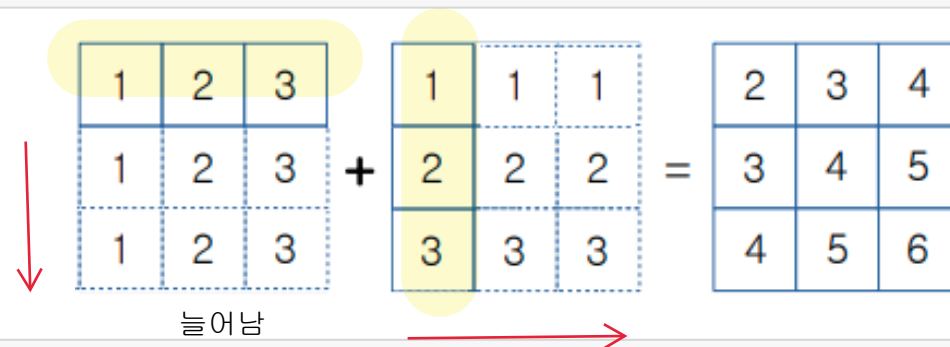
[3]

```
print(arr1 + arr2)
```

Out : [[2 3 4]

[3 4 5]

[4 5 6]]

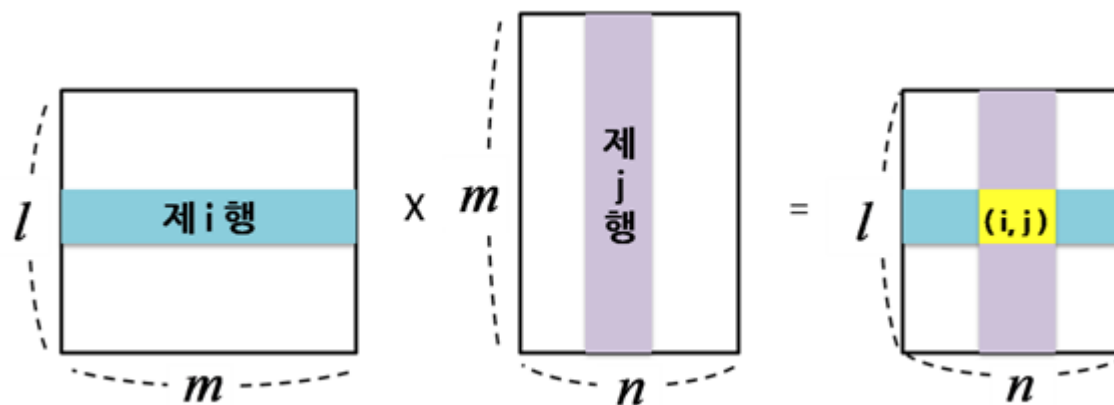


shape 값이 대각선이라고 같은 수라면 연산 가능
예) (2,5)와 (5,2)

• 행렬 연산

- 수학에서의 행렬의 곱셈

딥러닝에서 방정식을 하므로 알아둬야 함



A행렬

\times

B행렬

$=$

AB행렬

$l \times m$

$m \times n$

$l \times n$

일치할 때 행렬의 곱셈이 가능

• 행렬 연산

- `np.dot(arr1, arr2)` 함수 사용

> 일반적인 행렬곱

```
arr1 = np.array([1,2,3])  
arr2 = np.array([1,2,3])  
print(arr1 * arr2)
```

Out : [1 4 9]

> `np.dot` 사용 행렬곱

```
arr1 = np.array([1,2,3])  
arr2 = np.array([1,2,3]).T  
print(np.dot(arr1, arr2))
```

Out : 14

• 행렬 연산

scalar연산일 경우 모두 허용됨

- `np.dot(arr1, arr2)` 함수

> `np.dot(arr1, arr2)`

```
print(np.dot(arr1, arr2))
```

Out : 14

> `arr1.dot(arr2)`

```
print(arr1.dot(arr2))
```

Out : 14

> `arr1 @ arr2`

```
print(arr1 @ arr2)
```

Out : 14

• 행렬 연산

055

> 2차원 배열 행렬곱

```
arr3 = np.arange(1, 7).reshape(2, 3)
arr4 = np.arange(11, 17).reshape(3, 2)
print(np.dot(arr3, arr4))
```

Out : **[[82 88]**
[199 214]]

shape가 맞아야/ 대각선으로 같은 수라야 곱셈 가능

1	2	3
4	5	6

 \times

11	12
13	14
15	16

 $=$

82	88
199	214

• 배열 정렬 (Sort)

056

> 1차원 배열 정렬

```
# randint를 사용하여 1~100 숫자 중 5개를 뽑아 배열 생성
vector = np.array([np.random.randint(1, 100)
                    for n in range(5)])
print(vector)
```

Out : [23 91 34 76 61]

```
print(np.sort(vector))          # 오름차순으로 정렬
```

Out : [23 34 61 76 91]

```
print(np.sort(vector)[:,-1])    # 내림차순으로 정렬
```

Out : [91 76 61 34 23]

[end:start:-1] ==> 결국 끝에서부터 가져옴 / 역순으로 슬라이싱
벡터에서는 모두 가능
matrix는 axis=0에서는 가능하나
axis=1에서는 역순 어려움 [:, ::-1]

• 배열 정렬 (Sort)

> 2차원 배열 정렬

```
# randint를 사용하여 1~100 숫자 중 20개를 뽑아 배열 생성
matrix = np.array([np.random.randint(1, 100)
                    for n in range(20)]).reshape(4, 5)
print(matrix)
```

Out : **[[82 5 18 67 91]**
 [20 53 46 25 66]
 [59 46 27 92 54]
 [76 62 15 24 44]]

• 배열 정렬 (Sort)

> 2차원 배열 정렬

```
print(np.sort(matrix)) # 열 기준으로 정렬
```

```
Out : [[ 5 18 67 82 91]
        [20 25 46 53 66]
        [27 46 54 59 92]
        [15 24 44 62 76]]
```

```
print(np.sort(matrix, axis = 0)) # 행 기준으로 정렬
```

```
Out : [[20  5 15 24 44]
        [59 46 18 25 54]
        [76 53 27 67 66]
        [82 62 46 92 91]]
```

• 인덱스 반환 함수

> 최대값, 최소값의 인덱스

argmax, agrmin

최소값(min)과 최대값(max)의 인덱스

059

```
a = np.arange(1, 11).reshape(2, 5)
print(a)
```

```
Out : [[ 1  2  3  4  5]
       [ 6  7  8  9 10]]
```

```
print(np.argmax(a), np.argmin(a))
```

```
Out : 9 0
```

```
print(np.argmax(a, axis=0))
```

```
Out : [1 1 1 1 1]
```

가장 큰 수들의 자리값을 출력

• 인덱스 반환 함수

> 최대값, 최소값의 인덱스

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

```
Out : [[1 2 3 4]  
       [2 3 4 1]  
       [3 4 1 2]]
```

```
print(np.argmax(b, axis = 1))
```

```
Out : [3 2 1]
```

```
print(np.argmax(b, axis=0))
```

```
Out : [2 2 1 0]
```

• 인덱스 반환 함수

061

> 정렬 인덱스 반환 함수

```
c = np.random.randn(2,3)
print(c)
```

```
Out : [[-0.58623016  0.07263361 -0.34018553]
       [ 0.31310596  0.76271241  0.79869664]]
```

```
print(np.argsort(c, axis = 0))
```

```
Out : [[0 0 0]
       [1 1 1]]
```

위아래 비교

```
print(np.argsort(c, axis = 1))
```

```
Out : [[0 2 1]
       [0 1 2]]
```

좌우 비교 정렬된 인덱스값을 출력

• 배열 인덱싱

> **인덱스 반환 함수 + fancy index**

```
a = np.arange(10, 20)
b = np.argmax(a)
print(a)
print(b)
```

Out : [10 11 12 13 14 15 16 17 18 19]
9

```
print(a[b])
```

Out : 19

• 배열 합치기 (Vstack)

- 2개 이상의 배열을 수직으로 합체

> `np.vstack([arr1, arr2])`

밑으로 이어붙이기: col값이 같아야

```
vector1 = np.array([1, 2, 3])  
vector2 = np.array([4, 5, 6])  
  
print(np.vstack([vector1, vector2]))
```

Out : `[[1 2 3]`

`[4 5 6]]`

2차원 배열로 됨

모든 합치기 작업은 shape 정보/ size갯수가 같아야

• 배열 합치기 (Hstack)

vstack과 함께 1차원 합치기

- 2개 이상의 배열을 수평으로 합체

> `np.hstack([arr1, arr2])`

옆으로 이어붙이기: 꼭 row 수가 같아야

```
vector1 = np.array([1, 2, 3]).reshape(3, 1)
vector2 = np.array([4, 5, 6]).reshape(3, 1)

print(np.hstack([vector1, vector2]))
```

Out : `[[1 4]`

`[2 5]`

`[3 6]]`

수평으로 합쳐져서 2차원으로 변함

vstack은 axis=0 의 경우와 같고

hstack은 axis=1의 경우와 같음

=> 1차원이 합친후에는 2차원으로 변경됨

• 배열 합치기 (Concatenate)

- 2개 이상의 배열을 수직, 수평으로 합체

> `np.concatenate(arr1, arr2, axis)`

```
vector1 = np.array([1, 2, 3])  
vector2 = np.array([4, 5, 6])  
print(np.concatenate([vector1, vector2], axis = 0))
```

Out : [1 2 3 4 5 6]

- 배열 합치기 (Concatenate)

- 2개 이상의 배열을 수직, 수평으로 합체

> `np.concatenate(arr1, arr2, axis)`

```
vector1 = np.array([1, 2, 3]).reshape(3, 1)
vector2 = np.array([4, 5, 6]).reshape(3, 1)
print(np.concatenate([vector1, vector2], axis = 1))
```

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

- 배열 합치기 (Concatenate)

- 2개 이상의 배열을 수직, 수평으로 합체

> `np.concatenate(arr1, arr2, axis)`

```
matrix1 = np.arange(1,5).reshape(2,2)
matrix2 = np.arange(5,9).reshape(2,2)
print(np.concatenate([matrix1, matrix2], axis = 0))
```

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

- 배열 합치기 (Concatenate)

- 2개 이상의 배열을 수직, 수평으로 합체

> `np.concatenate(arr1, arr2, axis)`

```
matrix1 = np.arange(1,5).reshape(2,2)
matrix2 = np.arange(5,9).reshape(2,2)
print(np.concatenate([matrix1, matrix2], axis = 1))
```

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

• 행 / 열 바꾸기 (transpose)

- 행과 열의 뒤집어 주는 함수

> `ndarray.transpose()`

2차원 이상에서만 작동

> `ndarray.T`

```
matrix3 = np.array([[5, 6]])  
print(matrix3)
```

Out : `[[5 6]]`

```
print(matrix3.transpose())  
print(matrix3.T)
```

Out : `[[5]`

`[6]]`

`[[5]`

`[6]]`

• Transpose + Concatenate

> 형태를 바꿔 서로 결합할 수 없는 배열을 합칠 수 있다.

```
print(np.concatenate([matrix1, matrix3]))
```

Out :
[[1 2]
 [3 4]
 [5 6]]

```
print(np.concatenate([matrix1, matrix3], axis = 1))
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-652-2419e8fc4633> in <module>  
----> 1 print(np.concatenate([matrix1, matrix3], axis = 1))  
  
<__array_function__ internals> in concatenate(*args, **kwargs)
```

ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 0, the array at index 0 has size 2 and the array at index 1 has size 1

• Transpose + Concatenate

> 형태를 바꿔 서로 결합할 수 없는 배열을 합칠 수 있다.

```
print(np.concatenate([matrix1, matrix3.T], axis = 1))
```

Out : $\begin{bmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \end{bmatrix}$

전치 후 shape 맞춰서 합치기

