넘파이의 함수

- np.prod: 배열 원소들의 곱을 계산합니다.
- np.cumsum: 배열 원소들의 누적 합을 계산합니다.
- np.cumprod: 배열 원소들의 누적 곱을 계산합니다.
- np.mean: 배열의 평균을 계산합니다.
- np.median: 배열의 중앙값을 계산합니다.
- np.std: 배열의 표준편차를 계산합니다.
- np.var: 배열의 분산을 계산합니다.
- np.min / np.max: 배열의 최솟값과 최댓값을 계산합니다.
- np.any: 배열 내 값 중 하나라도 참이면 True 반환.
- np.all: 배열 내 모든 값이 참이면 True 반환.
- 등등

```
import numpy as np

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

arr.shape
# axis = 0 부터 바깥 쪽 축
```

```
(2, 3)
```

```
#  shape 이  (2,3) 튜플을 리턴. ** 튜플의 인덱스와 axis(축) 값이 같다. **
# axis=0 기준 합 계산
result = np.sum(arr, axis=0)   # 축 axis=0 을 기준으로 더하기 : 같은 축(행) 더하기

print(result)    # [5,7,9]
print(type(result))
```

```
[5 7 9]
<class 'numpy.ndarray'>
```

```
# axis=1 기준 합 계산
result = np.sum(arr, axis=1)  # 축 axis=1 을 기준으로 더하기 : 열을 더하기

print(result)
```

```
[ 6 15]
```

```
# axis 속성이 없다면?
result = np.sum(arr)
print(result)
print(type(result))
```

```
21
<class 'numpy.int64'>
```

행렬 연산

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

```
3 b = np.array([[3, 4, 5],
4               [1, 2, 3]])
5 print(a.shape, b.shape)
```

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

- ndarray 의 shape 이 같은 두 행렬의 사칙연산

```
1 # 행렬 사칙 연산 : 같은 행,열 위치의 요소끼리 연산
2 # 행렬 덧셈
3 print(a + b)
```

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

```
1 # 행렬 뺄셈
2 print(a - b)
```

```
[[-2 -2 -2]
 [ 1  1  1]]
```

```
1 # 행렬 원소별 곱셈
2 print(a * b)
```

```
[[ 3  8 15]
 [ 2  6 12]]
```

```
1 # 행렬 나눗셈
2 print(a / b)
```

```
[[0.33333333 0.5        0.6        ]
 [2.         1.5        1.33333333]]
```

행렬 곱

`np.dot(a, b)`

- 내적(dot product)
- 벡터, 스칼라, 행렬 모두 처리 가능.
- 1차원 배열의 내적(product) => 스칼라 값
- 2차원 배열은 행렬 곱
- 다차원일 경우 규칙이 복잡.

```
1 # 1차원 배열의 내적(product) => 스칼라 값
2 #    ㄴ dot 메소드 , 기호 @
3 a = np.array([1, 2, 3])
4 b = np.array([4, 5, 6])
5 # 내적 연산 : 같은 위치의 값을 각각 곱해서 더하기. 수학적 의미는 벡터 간의 곱셈.
6 print( np.dot(a, b))   # 1x4 + 2x5 +3x6
7
8
```

```
32
```

```
1 # 파이썬 3.5 이상 버전. 행렬 곱
2 print(a @ b)
```

```
32
```

```
# 2차원 배열 행렬곱은 A행과 B열의 내적으로 구성된다.
ndarr3 = np.array([[1, 2, 3],
                   [2, 3, 4]])
ndarr4 = np.array([[1, 2],
                   [3, 4],
                   [5, 6]])
print(ndarr3)
print(ndarr4)
print(ndarr3.shape)
print(ndarr4.shape)

# 2차원 배열에서 내적 (다차원은 더 복잡)
print((1*1 + 2*3 + 3*5), (1*2 + 2*4 + 3*6))   # ndarr3 0행 @ ndarr4 0열   , ndarr3 0행 @ ndarr4 1열
print((2*1 + 3*3 + 4*5), (2*2 + 3*4 + 4*6))   # ndarr3 1행 @ ndarr4 0열   , ndarr3 1행 @ ndarr4 1열
```

```
[[1 2 3]
 [2 3 4]]
[[1 2]
 [3 4]
 [5 6]]
(2, 3)
(3, 2)
22 28
31 40
```

```
# 곱(product, 내적) 연산 정리
r1 = ndarr3 @ ndarr4    #
r2 = np.dot(ndarr3, ndarr4)  #
r3 = np.matmul(ndarr3, ndarr4)  # 다차원 배열에 특화한 행렬 곱(딥러닝, 텐서 연산에 적합)
# matmul : matrix multuply

print (r1)
print(type(r1))
print('-------------------------')
print (r2)
print(type(r2))
print('-------------------------')
print (r3)
print(type(r3))
```

```
[[22 28]
 [31 40]]
<class 'numpy.ndarray'>
-------------------------
[[22 28]
 [31 40]]
<class 'numpy.ndarray'>
-------------------------
[[22 28]
 [31 40]]
<class 'numpy.ndarray'>
```

요약

```
- np.dot(a,b)                 1D: 내적 / 2D: 행렬곱
- np.matmul(A,B) 또는 A @ B   다차원 행렬곱만 수행
```

순차적인 값 생성

```
arr1 = range(1, 11)
print(arr1)
print(type(arr1))
for i in arr1:
    print(i, end=' ')

```

```
range(1, 11)
<class 'range'>
1 2 3 4 5 6 7 8 9 10
```

- `np.arange(start, stop, step)` 는 start부터 stop-1까지의 정수(또는 실수)를 일정한 간격(step) 으로 넘파이 배열 생성
- start, stop, step 는 실수값도 가능

```

ndarr2 = np.arange(1, 11)  # array range
print(ndarr2)
print(type(ndarr2))
print(ndarr2.shape)  # 1차원 배열

```

```
[ 1  2  3  4  5  6  7  8  9 10]
<class 'numpy.ndarray'>
(10,)
```

- `np.linspace(start,end, 개수)` 는 시작과 종료값을 포함하여 개수만큼 정확한 간격으로 배열 생성함. 정밀한 실수 간격에 사용하는 메소드(lineary spaced)

```
np.linspace(0, 1, 5)
# 결과: [0. , 0.25, 0.5 , 0.75, 1. ]
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
np.linspace(0, 1.2, 6)
```

```
array([0.  , 0.24, 0.48, 0.72, 0.96, 1.2 ])
```

정렬

- sort() 함수

```
ndarr1 = np.array([1, 10, 5, 7, 2, 4, 3, 6, 8, 9])
print(ndarr1)
result = np.sort(ndarr1)
# 오름차순 정렬하여 새로운 넘파이 배열 리턴.
print(type(result))
print(result)          # ndarr1 은 변경안됨.

```

```
[ 1 10  5  7  2  4  3  6  8  9]
<class 'numpy.ndarray'>
[ 1  2  3  4  5  6  7  8  9 10]
```

```
print(result[::-1]) # 내림차순 정렬
print(np.sort(ndarr1)[::-1]) # 위와 동일

```

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

```
import numpy as np

ndarr2d = np.array([[11, 10, 2, 9],
                    [3, 6, 4, 2],
                    [5, 1, 7, 8]])
print(ndarr2d.shape)  # 3행 4열
```

```
(3, 4)
```

```
# 같은 열끼리 정렬. 각 열의 값들을 개별적으로 오름차순으로 정렬하며,
# 행(row)의 요소가 달라짐
np.sort(ndarr2d, axis=0)
```

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

```
# 같은 행끼리 정렬. 각 행의 원소들이 개별적으로 오름차순으로 정렬
np.sort(ndarr2d, axis=1)
```

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

```
np.sort(ndarr2d, axis=1)[:, ::-1]
```

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

```
print(ndarr2d)
np.sort(ndarr2d, axis=-1)  # 행렬에서 axis=-1 은 axis=1 과 동일
```

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

1차원 배열을 2차원 배열로 재구성

`ndarr1.reshape(2, 3)`

```
ndarr1 = np.array([1, 2, 3, 4, 5, 6])
ndarr2 = ndarr1.reshape(2, 3)  # (2행, 3열) 형태로 재구성
print(type(ndarr2))
print(ndarr2.shape)
ndarr2
```

```
<class 'numpy.ndarray'>
(2, 3)
```

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

```
1 ndarr2.reshape(6,)    # 1차원 배열의 shape
```

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

2차원 배열을 다른 행태로 재구성

```
1 ndarr3 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
2 print(ndarr3.shape)
3 # len(arr)
```

```
(2, 4)
```

```
1 # reshape 차원은 전체 요소의 갯수는 변함 없게 합니다.
2 reshaped_arr =ndarr3.reshape(4,2)
3 print(reshaped_arr)
```

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

```
1 import numpy as np
2 tensor = np.array([[[1,2,3,4],[4,5,6,5]],
3                   [[7,8,9,1],[10,11,12,4]],
4                   [[13,14,15,5],[16,17,18,6]]])
5 tensor.shape   # 갯수 : 24
```

```
(3, 2, 4)
```

```
1 result = tensor.reshape(2,4,3)
2 result
```

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

       [[10, 11, 12],
        [ 4, 13, 14],
        [15,  5, 16],
        [17, 18,  6]]])
```

- `tensor.reshape(-1,4)` 에서 -1 은 자동 계산되는 차원을 의미. 열을 4의 크기로 맞추고 다른 차원은 자동 계산

```
1 # 4열의 2차원 배열로 reshape.  tensor 에 저장된 값이 24. 4의 배수이므로 자동계산으로 reshape 가능함.
2 result = tensor.reshape(-1,4)
3 result
```

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

```
tensor.reshape(4,-1)
```

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

```
# 24의 약수로 차원의 값을 지정해야 함.
# tensor.reshape(-1, 5)  # ValueError: cannot reshape array of size 24 into shape (5)
```

- `nparr.transpose()` : 배열의 축(axes)의 순서를 재정렬

```
ndarr1 = np.arange(1,7)
ndarr1
```

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

```
ndarr2 = ndarr1.reshape(2,3)
ndarr2.shape
```

```
(2, 3)
```

```
ndarr2
```

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

```
ndarr3 =ndarr2.transpose()
ndarr3
```

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

3차원 transpose()

```
# 3차원 transpose
# 3차원 (z,y,x) 를 (x,y,z) 로 변경(기본)
#   옵션은 transpose 위치를 지정 (y,x,z) 이면
#   arr1.transpose(1,2,0)
```

```
ndarr3 = np.arange(24).reshape(2, 3, 4)  # shape: (2, 3, 4)
ndarr3

```

```
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]]])
```

- 원래 배열의 축 순서가 (0, 1, 2)였다면, transpose(1, 0, 2)는 첫 번째와 두 번째 축을 바꿈

```
1 transposed = ndarr3.transpose(1, 0, 2)   # shape: (3, 2, 4)
2 transposed
```

```
array([[[ 0,  1,  2,  3],
        [12, 13, 14, 15]],

       [[ 4,  5,  6,  7],
        [16, 17, 18, 19]],

       [[ 8,  9, 10, 11],
        [20, 21, 22, 23]]])
```

- 데이터 자체는 바뀌지 않음: 메모리 상의 값은 그대로이고, 보는 관점만 바뀌는 것.
- 고차원 배열에서 매우 중요: 특히 이미지 처리(CNN), 시계열 분석(RNN), 행렬 연산 등에서 자주 사용됨.
- 축 번호는 0부터 시작: 예를 들어 (batch, channel, height, width) 구조에서 transpose(0, 2, 3, 1)은 채널을 마지막으로 이동시킴.