```
!pip list
```

숨겨진 출력 표시

```
!pip list | grep numpy     # 넘파이 설치 여부 확인
```

```
numpy                             2.0.2
```

넘파이(Numpy)

파이썬에서 수치 계산과 과학적 연산을 위한 핵심 라이브러리.

- 데이터 분석, 인공지능, 머신러닝, 통계 등 다양한 분야에서 사용
- 다차원 배열 객체(ndarray)를 중심으로 강력한 기능 제공.

📌 넘파이의 핵심 특징

- 다차원 배열 지원: 파이썬 리스트보다 훨씬 빠르고 효율적인 데이터 처리 가능
- 행렬 연산 최적화: 선형대수 등 고급 수학 기능 포함
- 활용 : pandas, TensorFlow, PyTorch 등 주요 패키지의 기반이 됨

🧱 관련 용어

- ndarray :넘파이의 기본 배열 객체. 같은 타입의 값들로 구성됨 `n-dimensional array 즉, n차원 배열 객체`
- shape :배열의 각 축(axis)별 크기. tuple 타입 리턴
- ndim :배열의 차원 수
- size :배열의 총 원소 개수
- dtype :배열의 데이터 타입 (예: int32, float64 등)
- axis :배열의 축. 데이터가 나열되는 방향
- rank :배열의 차원 수 (축의 개수)

```
# 리스트 복습
list1 = [1,2,3,4]
print(len(list1))
print(list1)
print('~~~~~~~~~~~~')
list2 = [[1,2,'😀',4],[5,'test',7,9],'hello','numpy']
print(len(list2))
print(list2)
```

```
4
[1, 2, 3, 4]
~~~~~~~~~~~~
4
[[1, 2, '😀', 4], [5, 'test', 7, 9], 'hello', 'numpy']
```

📐 ndarray 차원별 용어

NumPy나 딥러닝에서 자주 등장하는 배열(ndarray)의 차원에 따라 사용되는 용어들

- 차원별 정의

용어	차원 (Rank)	설명
스칼라 (Scalar)	0차원	단일 숫자. 예: `7`, `3.14`
벡터 (Vector)	1차원	숫자의 리스트. 예: `[1, 2, 3]`
행렬 (Matrix)	2차원	숫자의 2D 배열. 예: `[[1, 2], [3, 4]]`
텐서 (Tensor)	3차원 이상	다차원 배열. 예: `[[[1], [2]], [[3], [4]]]`

```
import numpy as np
```

스칼라

하나의 숫자. 차원이 없는 값. `x = np.array(5)`

```
# 넘파이 배열 생성하는 방법
# np.array() 메소드로 생성합니다. -> 데이터의 차원에 따라 부르는 용어
scalar = np.array(5)
print(scalar)
print(type(scalar))
# ndim -차원을 구하는 속성  : 0 -> 스칼라
print(scalar.ndim)
# shape - 넘파이 배열의 모양
print(scalar.shape)

```

```
5
<class 'numpy.ndarray'>
0
()
```

벡터

1차원 배열. 여러 개의 스칼라가 나열된 형태. `v = np.array([1, 2, 3])`

```
# 리스트를 넘파이 배열로 생성
# 길이가 4인 벡터
ndarr1 = np.array([1,2,3,4])
print(ndarr1)
print(type(ndarr1))
print(ndarr1.ndim)    # 1차원 : 벡터
print(ndarr1.shape)   # (4,) : 4는 배열의 첫 번째 축(axis 0)의 크기
```

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

- 넘파이 정수 타입 : `numpy.int64` (기본이 64비트)

```
print(type(ndarr1))      # <class 'numpy.ndarray'> -> 넘파이 배열 클래스 타입
print(type(ndarr1[0]))   # <class 'numpy.int64'> -> 넘파이 정수 타입 (기본이 64비트)
print(type(ndarr1[1]))
print(type(ndarr1[2]))
print(type(ndarr1[3]))
```

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

- 열 벡터 (n행 1열)

```
# 열벡터 (nx1): 행렬이지만 벡터처럼 취급 가능
# n= 3
col_vector = np.array([[10],
                       [20],
                       [30]])
print(col_vector)
print(col_vector.ndim)
print(col_vector.shape)

```

```
[[10]
 [20]
 [30]]
2
(3, 1)
```

- 행 벡터(1열 n행)

```
# 행벡터 (1 x n) :
# # n= 3
row_vector = np.array([[1,2,3]])
print(row_vector)
print(row_vector.ndim)
print(row_vector.shape)
```

```
[[1 2 3]]
2
(1, 3)
```

행렬 (matrix)

2차원 배열. 벡터들이 행과 열로 구성된 형태.

```
# matrix (행렬) : 2차원 배열 . m행 n열(m x n 행렬)
# matrix = np.array([[1,2],[4,5,6]])   # 오류
matrix = np.array([[1,2,3],[4,5,6]])
print(matrix)
print(matrix.ndim)    # 2차원
print(matrix.shape)   # 2행 3열
# shape 은 (2,3) 튜플을 리턴
# (행(axis=0)의 크기, 열(axis= 1)의 크기 -> 축의 크기)
```

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

텐서

3차원 이상 배열. 행렬을 여러 층으로 쌓은 형태.

```
tensor = np.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]]])
# 2 x 4 행렬이 3개
print(tensor.shape)   # (3,2,4)   len가 4인 1차원 배열이 2개, 2 x 4 행렬이 3개
print(tensor.ndim)
print(tensor)
#  shape 은 (3, 2, 4) 튜플을 리턴. ** 튜플의 인덱스와 axis 값이 같다. **
# (행(axis=1)의 크기, 열(axis= 2)의 크기 -> 축의 크기)
# 2x4 행렬이 3개. 가장 바깥쪽 축이 axis=0
```

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

 [[ 7  8  9  1]
  [10 11 12  4]]

 [[13 14 15  5]
  [16 17 18  6]]]
```

요약 : 예시 코드

```
import numpy as np

# 스칼라
a = np.array(5)          # shape: ()

# 벡터
b = np.array([1, 2, 3])  # shape: (3,)

# 행렬
c = np.array([[1, 2], [3, 4]])  # shape: (2, 2)

# 3차원 텐서
d = np.array([[[1], [2]], [[3], [4]]])  # shape: (2, 2, 1)
```

- 리스트를 ndarray 로 변환 : `ndarr1 = np.array(list1)`

```
# 리스트를 ndarray로 변환
list1 = [1, 2, 3, 4]
print(list1)
print(type(list1))
```

```
[1, 2, 3, 4]
<class 'list'>
```

```
ndarr1 = np.array(list1)
print(ndarr1)
print(type(ndarr1))

```

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

- ndarray를 리스트로 변환 : `list2 = ndarr1.tolist()`

```
# ndarray를 리스트로 변환
list2 = ndarr1.tolist()
print(list2)
print(type(list2))
```

```
[1, 2, 3, 4]
<class 'list'>
```

자동 변환

- 요소의 데이터 타입이 서로 다른 리스트를 넘파이로 변환할 때,
- 넘파이(Numpy) 배열의 요소들은 모두 같은 데이터 타입(dtype)을 가져야하므로 포괄적인 타입으로 자동 변환.

```
list1 = [1, 3.14, 'Python', '😊', True]
print(list1)
print(type(list1))
print(type(list1[0]))
print(type(list1[1]))
print(type(list1[2]))
print(type(list1[3]))
print(type(list1[4]))
```

```
[1, 3.14, 'Python', '😊', True]
<class 'list'>
<class 'int'>
<class 'float'>
<class 'str'>
<class 'str'>
<class 'bool'>
```

```
ndtest = np.array(list1)        # 원소의 데이터 타입이 같아야 하므로 모두 문자열로 변환
print(ndtest)
print(type(ndtest[0]))          # <class 'numpy.str_'>
```

```
['1' '3.14' 'Python' '😊' 'True']
<class 'numpy.str_'>
```

```
list2 = [1, 3.14, True]
ndtest = np.array(list2)        # 원소의 데이터 타입이 같아야 하므로 모두 실수로 변환
print(ndtest[0])
print(ndtest[1])
print(ndtest[2])
print(type(ndtest[0]))          # <class 'numpy.float64'>
```

```
1.0
3.14
1.0
<class 'numpy.float64'>
```

- 자동 변환하지 않고 직접 타입 지정

```
# dtype 속성으로 변환을 하려는 타입 지정
ndarr3 = np.array([1, 2, 3.64, True], dtype=int)
print(ndarr3)
```

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

```
ndarr4 = np.array(['1', 2, 3.14, True], dtype=int)
print(ndarr4)
```

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

```
# int로 변환 못하는 'a' 때문에 오류
ndarr4 = np.array(['a', 2, 3.14, True], dtype=int)
print(ndarr4)
```

숨겨진 출력 표시

ndarray 인덱싱과 슬라이싱

```
ndarr1 = np.array(['🍓', '🍉', '🍌', '🍒', '🍑'])
print(ndarr1)
print(ndarr1.shape)   # 해당 배열의 크기를 나타내는 튜플
4
```

```
['🍓' '🍉' '🍌' '🍒' '🍑']
(5,)
```

- 인덱싱

```
# 인덱싱
print(ndarr1[0])
print(ndarr1[4])
print(ndarr1[-1])
print(ndarr1[-2])
print(type(ndarr1[0]))
```

```
🍓
🍑
🍑
🍒
<class 'numpy.str_'>
```

- 슬라이싱

```
# 슬라이싱 - 새로운 ndarray 를 리턴.
print(ndarr1[0:3])
print(ndarr1[2:])
print(ndarr1[:3])
print(type(ndarr1[0:3]))   # 1차원 배열을 슬라이싱하면 1차원 리턴
```

```
['🍓' '🍉' '🍌']
['🍌' '🍒' '🍑']
['🍓' '🍉' '🍌']
<class 'numpy.ndarray'>
```

- 2차원 배열(행열) : `ndarr2d[row_start:row_end:row_step, col_start:col_end:col_step]`

```
# 2차원 배열
ndarr2d = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11,12]])
print(ndarr2d)
print(ndarr2d.shape)   # (3,4)
```

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

- `ndarr2d[0, :]` 는 0행만 추출. 결과는 벡터

```
0 은 인덱싱
: 는 슬라이싱
```

```
# 0행의 모든 요소 가져오기 - 아래 3가지 방법 모두 가능함.
print(ndarr2d[0, :])
print(ndarr2d[0,])
print(ndarr2d[0])
print(ndarr2d[0].shape)   # 2차원 배열에서 행만 가져오면 1차원
```

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

- `ndarr2d[:,0]` 는 0열만 추출하는 슬라이싱. 결과는 벡터

```
# 0열 가져오기 - 새로운 넘파이 배열 생성함.
print(ndarr2d[:, 0])
result = ndarr2d[:, 0]
print(result)
print(result.shape)    # 2차원 배열에서 열만 가져오면 1차원
```

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

- `ndarr2d[:, -1]` 는 모든 행에서 마지막 열만 추출하는 결과. (벡터)

```
import numpy as np

ndarr2d = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11,12]])
print(ndarr2d[:, -1])   #
```

```
print(ndarr2d[..., ::-1])
print(ndarr2d[..., -1])   # # Ellipsis
```

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

다양한 예시

- `ndarr2d[0, :]` 첫 번째 행 전체 [1 2 3 4]
- `ndarr2d[:, 1]` 두 번째 열 전체 [2 6 10]
- `ndarr2d[1:3, 2:4]` 2 ~ 3행, 3 ~ 4열 [[7 8], [11 12]]
- `ndarr2d[:2, :2]` 상단 2행, 좌측 2열 [[1 2], [5 6]]
- `ndarr2d[-1, :]` 마지막 행 전체 [9 10 11 12]
- `ndarr2d[:, -1]` 마지막 열 전체 [4 8 12]
- `ndarr2d[::2, ::2]` 행과 열을 2칸씩 건너뛰며 선택 [[1 3], [9 11]]
- `ndarr2d[::-1]` 역순 슬라이싱

```
ndarr2d[1:3, 2:4]
```

```
array([[ 7,  8],
       [11, 12]])
```

```
ndarr2d[::-1]
ndarr2d[-1]
```

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

- tensor (3 차원 배열) 슬라이싱

```

tensor = np.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[..., ::-1]   # 텐서의 마지막 축(axis)을 역순으로
```

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

       [[ 1,  9,  8,  7],
        [ 4, 12, 11, 10]],

       [[ 5, 15, 14, 13],
        [ 6, 18, 17, 16]]])
```

```
tensor[:,:,::-1]
```

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

       [[ 1,  9,  8,  7],
        [ 4, 12, 11, 10]],

       [[ 5, 15, 14, 13],
        [ 6, 18, 17, 16]]])
```

정수 배열 인덱싱

`ndarr2[[2,5,9]]`

```
# 가져오고 싶은 인덱스를 배열 또는 리스트로 전달
ndarr2 = np.array([10, 15, 2, 8, 20, 90, 85, 44, 23, 32])
print(ndarr2)
print(ndarr2[[2,5,9]])
```

```
[10 15  2  8 20 90 85 44 23 32]
[ 2 90 32]
```

```
idx = [2, 5, 9]
print(ndarr2[idx]) # ndarr1[[2, 5, 9]] 와 동일함
```

```
[ 2 90 32]
```

```
ndarr2d = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]])
print(ndarr2d[[0, 1], :])   # 0,1 행 전체 가져와서 배열 만들기
print(ndarr2d[[0, 1], :].shape)   # 0,1 행 전체 가져와서 배열 만들기
print('--------------------------')
print(ndarr2d)
print(ndarr2d.shape)
```

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

```
# 참고
print(ndarr2d[0][1])
print(ndarr2d[2][2])
print(ndarr2d[1][1])
print('------- 위와 같은 의미 -------')
print(ndarr2d[0,1])
print(ndarr2d[2,2])
print(ndarr2d[1,1])
```

```
2
11
6
------- 위와 같은 의미 -------
2
11
6
```

```
# 참고
ndarr2d = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]])
# 정수 배열 인덱싱
ndarr2d[[0, 2, 1], [1, 2 ,1]]   # 0행 1열, 2행 2열 , 1행 1열 값으로 새로운 1차원 배열 리턴
```

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

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

```
ndarr2d[[0, 2, 1,1], [1, 2 ,1, 3]]
# 첫번째 배열 [0, 2, 1, 1] 은 행을 선택하는 인덱스 값
# 두번째 배열 [1, 2 ,1, 3] 은 열을 선택하는 인덱스 값
```

```
array([ 2, 11,  6,  8])
```

불리언 인덱싱

- 배열과 크기가 같은 불리언(Boolean) 값 배열을 사용하여, True 값이 있는 위치의 요소만 선택하는 방식입니다.

```
ndarr3 = np.array(['🍓', '🍉', '🍌', '🍒🍒', '🍑'])
sel = [True, False, True, True, False]
ndarr3[sel]  # dtype='<U2' Unicode 최대 문자열 길이 2
```

```
array(['🍓', '🍌', '🍒🍒'], dtype='<U2')
```

```
# sel = [True, False, True]
# ndarr3[sel]   # IndexError: boolean index did not match indexed array along dimension 0; dimension is 5 but corresponding boolean dimension is 3
```

```
ndarr2d = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]])

print(ndarr2d > 7)       # ndarr2d 의 크기와 같은 boolean 배열 리턴
```

```
[[False False False False]
 [False False False  True]
 [ True  True  True  True]]
```

```
print(ndarr2d[ndarr2d > 7])    # 1차원 배열로 변환

# 차원 지정하려면 reshape 해야함
```

```
[ 8  9 10 11 12]
```