• Numpy Pandas 강의자료 26페이지에 있는 (4,) / (1,4), (3,) / (3,1) 의 차이에 대해 작성하세요. •

a[1, :]와 같이 단일 행/열을 인덱싱할 때는 배열의 차원이 자동으로 축소된다. 따라서 이 경우 2차원 배열의 특정 행을 가져오면 1차원 배열로 축소된다. 즉, (1, n) 모양의 배열이 1차원 벡터로 변환되어 나오는 것

a[1:2, :]처럼 슬라이싱을 할 때는 차원이 유지된다. 슬라이싱 범위를 지정하는 경우(즉, 1:2처럼 범위를 사용한 경우), Numpy는 결과를 2차원 배열로 유지

(3,) / (3,1)도 이와 동일

단일 인덱스 (a[:, 1])는 특정 열만 선택하는 방식으로, Numpy는 이를 **벡터**로 취급하고 1차원으로 축소합니다.

슬라이싱 (a[:, 1:2])은 특정 열 범위를 선택하는 방식으로, Numpy는 원래의 배열 구조(즉, 2차원 배열)를 **유지**하려고 합니다. 따라서, 결과가 2차원 배열로 반환됩니다

본인 local anaconda에서 python 3.8.18 version으로 sda2024이라는 conda env을 생성하고 jupyterlab에서 해당 환경 사용 할 수 있 도록 셋팅 하고 화면 캡쳐

• Numpy를 이용해서 data matrix를 만들고 random number로 채워 넣는다. – row: 1000 / column 1000 •

In [15]:
```
import numpy as np

data_matrix = np.random.rand(1000, 1000)
print(data_matrix)
```

executed in 28ms, finished 18:18:59 2024-10-01

```
[[0.73679557 0.72245275 0.83892576 ... 0.14607237 0.11006653 0.01557541]
 [0.13624548 0.47630287 0.23794428 ... 0.9127024  0.35594216 0.14622679]
 [0.31661281 0.00332224 0.04311564 ... 0.36046877 0.7340491  0.11933729]
 ...
 [0.5733413  0.11591207 0.1392874  ... 0.20643274 0.38792175 0.58405949]
 [0.24039389 0.48011688 0.47972159 ... 0.41757289 0.30809378 0.5998473 ]
 [0.13297315 0.60342598 0.96445556 ... 0.05870765 0.34454325 0.20617705]]
```

i번째 sample과 j번째 sample의 거리를 구하기 위한 matrix를 만든다. – for loop을 사용하기 – vectorize 방법을 사용하기

In [11]:
```
import numpy as np

# 작은 크기의 데이터 예시 (10x10)
data_matrix = np.random.rand(100, 100)
print(data_matrix)
```

executed in 23ms, finished 18:17:44 2024-10-01

```
[[0.7968654  0.32202948 0.9877222  ... 0.78548935 0.63394408 0.50683314]
 [0.22056574 0.23970172 0.73060174 ... 0.6183376  0.75084916 0.60618363]
 [0.47297126 0.83497653 0.96988913 ... 0.97807924 0.71048407 0.60284193]
 ...
 [0.6025929  0.39713568 0.81450513 ... 0.00253682 0.78035955 0.66570158]
 [0.78539588 0.38027823 0.01405663 ... 0.83211899 0.50570258 0.61605566]
 [0.55194624 0.83880925 0.29135963 ... 0.32352128 0.15217757 0.38128236]]
```

In [14]:
```


# ----------------
# Method 1: Using for loop to compute distance matrix
# ----------------
distance_matrix_for_loop = np.zeros((100, 100))

# For loop to compute pairwise Euclidean distances
for i in range(100):
    for j in range(100):
        distance_matrix_for_loop[i, j] = np.linalg.norm(data_matrix[i] - data_matrix[j])

print(distance_matrix_for_loop)
```

executed in 135ms, finished 18:17:49 2024-10-01

```
[[0.         4.48800965 4.42150262 ... 4.06710807 4.00415852 4.12201682]
 [4.48800965 0.         4.62873542 ... 4.16133402 4.1923625  4.10843842]
 [4.42150262 4.62873542 0.         ... 4.23811794 3.98827153 4.43067619]
 ...
 [4.06710807 4.16133402 4.23811794 ... 0.         4.09652585 3.67430513]
 [4.00415852 4.1923625  3.98827153 ... 4.09652585 0.         3.85610401]
 [4.12201682 4.10843842 4.43067619 ... 3.67430513 3.85610401 0.        ]]
```

In [13]:

```
distance_matrix_vectorized = np.sqrt(np.sum((data_matrix[:, np.newaxis, :] - data_matrix[np.newaxis, :, :])**2, axis=2))
distance_matrix_vectorized
```

executed in 30ms, finished 18:17:47 2024-10-01

Out[13]:

```
array([[0.        , 4.48800965, 4.42150262, ..., 4.06710807, 4.00415852,
        4.12201682],
       [4.48800965, 0.        , 4.62873542, ..., 4.16133402, 4.1923625 ,
        4.10843842],
       [4.42150262, 4.62873542, 0.        , ..., 4.23811794, 3.98827153,
        4.43067619],
       ...,
       [4.06710807, 4.16133402, 4.23811794, ..., 0.        , 4.09652585,
        3.67430513],
       [4.00415852, 4.1923625 , 3.98827153, ..., 4.09652585, 0.        ,
        3.85610401],
       [4.12201682, 4.10843842, 4.43067619, ..., 3.67430513, 3.85610401,
        0.        ]])
```

- **두 방법의 실행 시간을 비교하여 캡쳐 후 업로드**

vectorize 방법이 평균적으로 3배 이상 빨랐습니다. 기준은 100 x 100 matrix입니다.

1000 x 1000으로 하고싶었는데 메모리에 가득차서 그런지 원할히 실행이 안되어 100 x 100으로 했습니다.