K-Digital Training 스마트 팩토리 3기

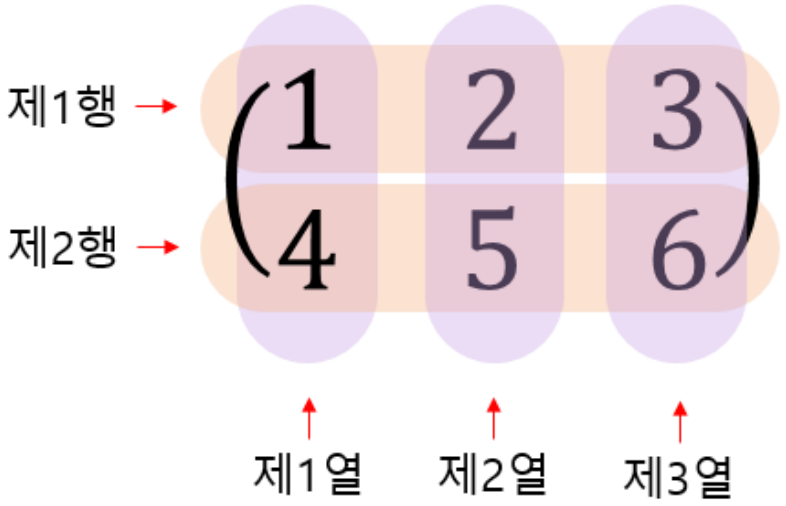
Numpy

Numpy

다차원 배열(행렬)을 쉽게 처리하고 효율적으로 사용할 수 있도록 지원하는 파이썬의 라이브러리

수치해석, 통계 관련 기능을 구현한다고 할 때 가장 기본이 되는 모듈

행렬



행렬의 성분을
가로로 배열한 줄을 **행**
세로로 배열한 줄을 **열**

→ 2 x 3 행렬 또는 2행 3열의 행렬

Numpy

- **Ndarray 타입**의 배열을 만들 수 있음.

[Ndarray 타입이란?]

N-dimension array 의 약자. 다차원 배열을 의미.

Numpy

Jupyter 실행하기

```
(base) C:\Users\[redacted]>conda env list
# conda environments:
#
base                  *  C:\Users\[redacted]\anaconda3
py38                     C:\Users\[redacted]\anaconda3\envs\py38
ver_3.6.2                C:\Users\[redacted]\anaconda3\envs\ver_3.6.2


(base) C:\Users\[redacted]>conda activate py38

(py38) C:\Users\[redacted]>jupyter notebook
[I 2023-05-23 14:38:40.426 LabApp] JupyterLab extension loaded from C:\Users\[redacted]\anaconda3\lib\site-packages\jupyterlab
[I 2023-05-23 14:38:40.426 LabApp] JupyterLab application directory is C:\Users\[redacted]anaconda3\share\jupyter\lab
[I 14:38:40.431 NotebookApp] The port 8888 is already in use, trying another port.
[I 14:38:40.432 NotebookApp] Serving notebooks from local directory: C:\Users\[redacted]
[I 14:38:40.432 NotebookApp] Jupyter Notebook 6.4.12 is running at:
[I 14:38:40.432 NotebookApp] http://localhost:8889/?token=9588437df73d346728549e71564d4b1534a597d65fa51bec
[I 14:38:40.433 NotebookApp]  or http://127.0.0.1:8889/?token=9588437df73d346728549e71564d4b1534a597d65fa51bec
[I 14:38:40.433 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 14:38:40.525 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///C:/Users/[redacted]/AppData/Roaming/jupyter/runtime/nbserver-27060-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/?token=9588437df73d346728549e71564d4b1534a597d65fa51bec
     or http://127.0.0.1:8889/?token=9588437df73d346728549e71564d4b1534a597d65fa51bec
```

Numpy 사용하기

In [7]:
```
import numpy as np #numpy 라이브러리 import
x = np.array([3, 1, 2]) #np.array : ndarray 타입의 배열 생성
print(x) # x 성분 출력
type(x) # x의 type 확인
```

```
[3 1 2]
```

Out[7]:
```
numpy.ndarray
```

Python List vs Numpy array(ndarray)

python 기본 자료형 list와 Numpy로 생성하는 ndarray의 차이점은?

1. 선언 형태의 차이

[List]

원소로 여러 가지 자료형을 허용한다.

[Ndarray]

원소로 한 가지 자료형만 허용한다.

```
a = [1, 2, 'a', 'b']
b = np.array([1, 2, 'a', 'b'])
print(a)
print(b)
```

```
[1, 2, 'a', 'b']
['1' '2' 'a' 'b']
```

Python List vs Numpy array(ndarray)

python 기본 자료형 list와 Numpy로 생성하는 ndarray의 차이점은?

2. 선언 형태의 차이

[List]

내부 배열의 원소 개수가 달라도 된다.

[Ndarray]

내부 배열의 원소 개수가 같아야 한다.

```
# list와 ndarray의 차이점
# 2. list는 내부 배열의 원소의 개수가 달라도 되고,
# ndarray는 내부 배열의 원소가 같아야 한다.

c = [[1], [2, 3], [4, 5, 6]]
d = np.array([[1], [2, 3], [4, 5, 6]])
```

```
C:\Users\Lily\AppData\Local\Temp\ipykernel_37604\2524811967.
py:6: VisibleDeprecationWarning: Creating an ndarray from ra
gged nested sequences (which is a list-or-tuple of lists-or-
tuples-or ndarrays with different lengths or shapes) is depr
ecated. If you meant to do this, you must specify 'dtype=obj
ect' when creating the nda
  d = np.array([[1], [2, 3
```

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

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

Python List vs Numpy array(ndarray)

python 기본 자료형 list와 Numpy로 생성하는 ndarray의 차이점은?

3. 연산의 차이

[List vs Ndarray] 코드로 확인해보기!

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

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

빼기, 곱하기, 나누기 연산은 불가

```
ndarray1 = np.array([1, 3, 5])
ndarray2 = np.array([2, 4, 6])
ndarray1 + ndarray2
```

```
array([ 3,  7, 11])
```

빼기, 곱하기, 나누기 연산도 동일하게 같은 위치의 원소끼리 연산

Python List vs Numpy array(ndarray)

python 기본 자료형 list와 Numpy로 생성하는 ndarray의 차이점은?

3. 연산의 차이

[List vs Ndarray] 코드로 확인해보기!

```
list3 = [1, 3, 5]
list4 = [2, 4]
print(list3 + list4)
```

```
[1, 3, 5, 2, 4]
```

```
ndarray3 = np.array([1, 3, 5])
ndarray4 = np.array([2, 4])
# print(ndarray3 + ndarray4) # 오류 발생
```

Numpy 관련 속성 및 메소드 (ndim, shape)

.ndim : 배열의 차원 수를 반환

.shape : 배열 각 차원의 크기를 튜플 형태로 표현

```
ndarray6 = np.array([[1, 2, 3], [4, 5, 6]])
ndarray6.ndim
```

```
2
```

```
ndarray6.shape
```

```
(2, 3)
```

Numpy 관련 속성 및 메소드 (dot)

np.dot(array1, array2) : 배열의 내적 연산

- array1과 array2가 모두 **0차원** 이면, 두 스칼라의 곱 연산
  하지만, 이경우엔 보통 * 연산자를 쓰겠죠?
- array1과 array2가 모두 1차원 이면, 두 벡터의 내적 연산
- array1과 array2가 모두 2차원 이면, 두 행렬의 행렬곱 연산
  하지만, 행렬곱 연산의 경우, matmul() 사용을 권장!

Numpy 관련 속성 및 메소드 (dot)

스칼라 곱 연산

```
np.dot(4, 2)
```

```
8
```

벡터 내적 연산

```
ndar1 = np.array([1, 2, 3])
ndar2 = np.array([4, 5, 6])
print(np.dot(ndar1, ndar2)) # 32
```

1, 2, 3 X 4, 5, 6

1 * 4 + 2 * 5 + 3 * 6 = 4 + 10 + 18 = 32

행렬곱 연산

```
ndar1 = np.array([[1, 2], [3, 4]])
ndar2 = np.array([[1, 2], [3, 4]])
print(np.dot(ndar1, ndar2))
```

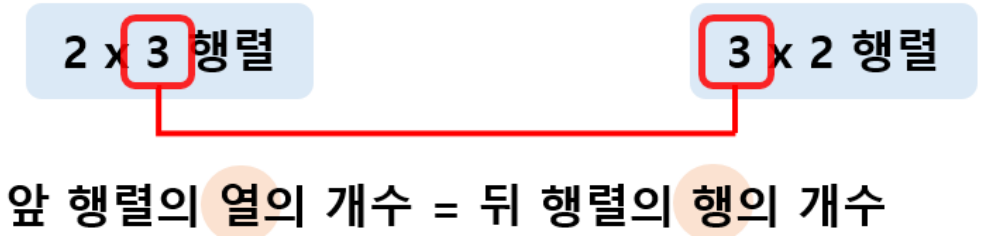
```
[[ 7 10]
 [15 22]]
```

Numpy 관련 속성 및 메소드 (matmul)

np.matmul(array1, array2) : 행렬의 행렬곱 연산

주의 ! A X B 연산을 진행할 때, A의 열의 개수와 B의 행의 개수가 동일해야 한다!

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



결과는?
2 x 2 행렬

Numpy 관련 속성 및 메소드 (matmul)

[2 X 2 행렬] X [2 X 2 행렬]

```
ndar1 = np.array([[1, 2], [3, 4]])
ndar2 = np.array([[1, 2], [3, 4]])
print(np.matmul(ndar1, ndar2))
```

```
[[ 7 10]
 [15 22]]
```

[2 X 3 행렬] X [3 X 2 행렬]

```
ndar1 = np.array([[1, 2, 3], [4, 5, 6]])
ndar2 = np.array([[6, 3], [5, 2], [4, 1]])
print(np.matmul(ndar1, ndar2))
```

```
[[28 10]
 [73 28]]
```

[1 X 3 행렬] X [3 X 2 행렬]

```
ndar1 = np.array([1, 2, 3])
ndar2 = np.array([[6, 3], [5, 2], [4, 1]])
print(np.matmul(ndar1, ndar2))
```

```
[28 10]
```

Numpy 관련 속성 및 메소드 (zeros)

np.zeros(m, n) : m x n 크기의 영행렬 생성

```
print(np.zeros((2,2)))  # 영행렬 생성
```

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

Numpy 관련 속성 및 메소드 (ones)

np.ones(m, n) : m x n 크기의 유닛행렬 생성

```
print(np.ones((2,3)))  # 유닛행렬
```

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

Numpy 관련 속성 및 메소드 (full)

np.full((m, n), k) : m x n 크기의 모든 원소가 k인 행렬 생성

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

```
[[5 5 5]
 [5 5 5]]
```

Numpy 관련 속성 및 메소드 (eye)

np.eye(n) : n x n 크기의 단위 행렬 생성

```
print(np.eye(3)) # 단위행렬
```

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

$$E = (1), \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \ldots$$

Numpy 관련 속성 및 메소드 (flatten)

.flatten() : 행렬의 평탄화 작업

평탄화 작업이란 ? N차원 배열을 1차원 배열로 변환시켜주는 작업

```
ndarr = np.array([[6, 3], [5, 2], [4, 1]])
ndarr.flatten()
```

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

Numpy 관련 속성 및 메소드 (reshape)

.reshape() : 배열의 차원을 변경

```
print(ndarr.shape)
ndarr.reshape(2,3)
```

```
(3, 2)

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

주의) 기존 원소의 개수와 변경될 차원에서의 원소 개수가 같아야 함!

```
print(ndarr.shape)
ndarr.reshape(2,4)
```

```
(3, 2)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~₩AppData₩Local₩Temp₩ipykernel_1772₩4039907857.py in <module>
      1 print(ndarr.shape)
----> 2 ndarr.reshape(2,4)

ValueError: cannot reshape array of size 6 into shape (2,4)
```

Numpy 관련 속성 및 메소드 (arrange)

.arange([start,] stop, [step,]) : 특정 배열을 만들어 줌.

```
print(np.arange(5))

print(np.arange(5,10))

print(np.arange(5,10,2))
```

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

Numpy 관련 속성 및 메소드 (random)

.random.randint(low, [high,] [size,])

```
np.random.randint(5) # 0 ~ 5까지의 난수
```

```
2
```

```
np.random.randint(5, 10) # 5 ~ 9까지의 난수
```

```
6
```

```
np.random.randint(5, size=(2, 4))
```

```
array([[1, 3, 0, 3],
       [0, 3, 2, 2]])
```

.random.rand(m, n) : m x n 크기의 배열 생성 및 0~1 사이의 난수로 초기화

```
np.random.rand(5)
```

```
array([0.77509011, 0.07569747, 0.48841539, 0.31223455, 0.98396833])
```

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

```
array([[0.36357133, 0.8512305 , 0.58088666],
       [0.52993306, 0.15884859, 0.30693899]])
```

Numpy 관련 속성 및 메소드 (조건식)

```
ndarr = np.array([[6, 3], [5, 2], [4, 1]])
ndarr > 4
```

```
array([[ True, False],
       [ True, False],
       [False, False]])
```

```
ndarr = np.array([[6, 3], [5, 2], [4, 1]])
ndarr[ndarr > 4]
```

```
array([6, 5])
```