넘파이, 배열과 벡터

이준원
SKT Data Scientist

학습 목표

1. numpy 배열 개념에 대해 설명할 수 있다.
2. numpy 배열을 조작할 수 있다.
3. numpy의 다양한 메소드를 사용할 수 있다.

K-Digital 훈련원

CONTENTS

Ⅰ _ numpy 기초

- Numpy 개념
- Numpy 데이터 생성 및 타입

Ⅱ _ numpy 조작하기

- numpy 연산
- numpy 색인 및 슬라이스

Ⅲ _ numpy 연산 메소드

- 유니버셜 메소드 활용
- 배열 연산 메소드 활용

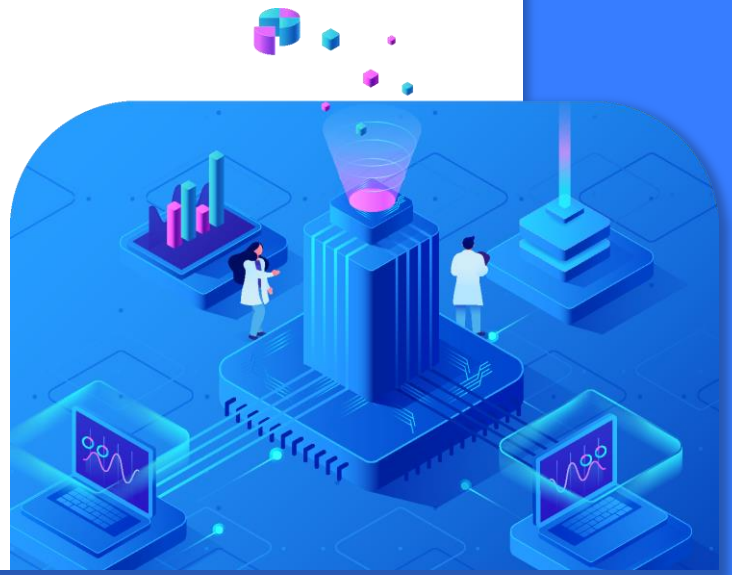
I

numpy 기초

CHAPTER

- Numpy 개념
- Numpy 데이터 생성 및 타입

Institute for
K-Digital Training
미래를 향한
인재를 양성합니다



numpy란?

- Numerical Python, 파이썬의 산술 계산을 위한 패키지
- 효율적인 다차원 배열 ndarray를 제공
- 전체 데이터 배열에 대해 빠른 계산을 할 수 있는 수학 메소드
- 배열을 디스크에 빠르게 읽고 쓸 수 있으며 효율적으로 가공할 수 있음
- 선형대수, 난수 생성, 푸리에 변환 등의 고급 수학 기능도 제공함

numpy를 왜 사용할까?

- 데이터는 이미지, 오디오, 텍스트, 숫자 등의 다양한 형태로 존재

- 컴퓨터가 이해할 수 있도록 데이터를 숫자 형식으로 변환해야 함
  - 모든 데이터는 숫자로 이루어진 배열의 형태 표현됨

- 빠르고 효율적인 배열 연산이 필요함
  - 기존 파이썬 list 자료구조도 가능하지만 데이터가 커질수록 비효율적임
  - numpy는 데이터의 크기가 커질수록 저장과 가공의 효율성을 보장함

numpy vs 파이썬 리스트

- numpy를 사용한 코드가 순수 파이썬보다 10~100배 이상 빠름

```
import numpy as np

np_array = np.arange(1000000)
python_list = list(np.arange(1000000))
```

```
%%time
for i in range(10):
    np_array = np_array * 2
```

```
CPU times: user 13.1 ms, sys: 3.28 ms, total: 16.3 ms
Wall time: 14.7 ms
```

```
%%time
for i in range(10):
    python_list = [x * 2 for x in python_list]
```

```
CPU times: user 2.25 s, sys: 116 ms, total: 2.37 s
Wall time: 2.37 s
```

ndarray 생성

```
import numpy as np

data1 = [1, 3, 5, 10, 2]
array1 = np.array(data1)
array1
```

```
array([ 1,  3,  5, 10,  2])
```

```
data2 = [
    [1, 2, 3, 4],
    [10, 9, 8, 7],
    [0.1, 0.2, 0.3, 0.4],
]

array2 = np.array(data2)
array2
```

```
array([[ 1. ,  2. ,  3. ,  4. ],
       [10. ,  9. ,  8. ,  7. ],
       [ 0.1,  0.2,  0.3,  0.4]])
```

```
array2.ndim
```

```
2
```

```
array2.shape
```

```
(3, 4)
```

```
array1.dtype, array2.dtype
```

```
(dtype('int64'), dtype('float64'))
```

- numpy의 다차원 배열은 ndarray
- np.array() 메소드를 통해 numpy 배열을 생성하며 인자로는 파이썬 list를 받음

ndarray 생성

```
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
np.zeros((4, 5))
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

```
np.ones(10)
```

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
np.ones((4, 5))
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# 단위행렬
np.eye(5)
```

```
array([[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.]])
```

- np.zeros: 모든 원소가 0인 배열
- np.ones: 모든 원소가 1인 배열
- np.arange: 주어진 범위에 따라 증가/감소하는 배열
- np.eye: 단위 행렬에 해당하는 배열

numpy 데이터 타입

```
array1 = np.array([1, 2, 3], dtype=np.float64)
array2 = np.array([1, 2, 3], dtype=np.int32)
```

```
array1
```

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

```
array2
```

```
array([1, 2, 3], dtype=int32)
```

```
array1.dtype, array2.dtype
```

```
(dtype('float64'), dtype('int32'))
```

```
int_array = np.array([1, 2, 3], dtype=np.int32)
float_array = int_array.astype(np.float64)
float_array
```

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

```
float_array = np.array([1.4, -2.3, 3.1], dtype=np.float64)
int_array = float_array.astype(np.int32)
int_array
```

```
array([ 1, -2,  3], dtype=int32)
```

```
string_arrray = np.array(['1.4', '-20', '0.2'], dtype=np.string_)
string_arrray
```

```
array([b'1.4', b'-20', b'0.2'], dtype='|S3')
```

```
float_array = string_arrray.astype(float)
float_array, float_array.dtype
```

```
(array([  1.4, -20. ,   0.2]), dtype('float64'))
```

- 배열은 다양한 데이터 타입을 가질 수 있음
  - int: 정수형
  - float: 실수형
  - bool: True / False
  - string_: 문자열
- 데이터 타입을 원하는 대로 변환할 수 있음

II

numpy 조작하기

CHAPTER

- numpy 연산
- numpy 색인 및 슬라이스

Institute for
K-Digital Training
미래를 향한
인재를 양성합니다

ndarray 산술연산

```
array = np.array([
    [1., 2., 3.],
    [10., 9., 8.]
])
```

```
array
```

```
array([[ 1.,  2.,  3.],
       [10.,  9.,  8.]])
```

```
array * array
```

```
array([[  1.,   4.,   9.],
       [100.,  81.,  64.]])
```

```
array + array
```

```
array([[ 2.,  4.,  6.],
       [20., 18., 16.]])
```

```
array + 10
```

```
array([[11., 12., 13.],
       [20., 19., 18.]])
```

```
array / 2
```

```
array([[0.5, 1. , 1.5],
       [5. , 4.5, 4. ]])
```

```
array ** 3
```

```
array([[   1.,    8.,   27.],
       [1000.,  729.,  512.]])
```

- numpy 산술연산의 특징은 for 문을 작성하지 않고 일괄처리할 수 있다는 점
- 같은 크기의 배열간의 연산은 각 원소 단위로 대응되어 적용됨

크기가 다른 배열간의 연산

```
array1 = np.array([
    [1., 2., 3.],
    [10., 9., 8.]
])

array2 = np.array([4., 5., 6.])
```

```
array1.shape, array2.shape
```

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

```
array1 + array2
```

```
array([[ 5.,  7.,  9.],
       [14., 14., 14.]])
```

```
array1 * array2
```

```
array([[ 4., 10., 18.],
       [40., 45., 48.]])
```

```
array1 / array2
```

```
array([[0.25      , 0.4       , 0.5       ],
       [2.5       , 1.8       , 1.33333333]])
```

- 크기가 다른 배열의 연산의 경우, 두 배열의 차원 크기가 중요함
- 차원의 크기가 맞으면 numpy가 알아서 차원의 크기에 맞게 연산을 수행함

크기가 다른 배열간의 연산

```
array1 = np.array([
    [1., 2., 3.],
    [10., 9., 8.]
])

array2 = np.array([4., 5., 6., 7., 8.])
```

```
array1.shape, array2.shape
```

```
((2, 3), (5,))
```

```
array1 + array2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-301-5a881dd2964f> in <module>
----> 1 array1 + array2

ValueError: operands could not be broadcast together with shapes (2,3) (5,)
```

```
array1 * array2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-302-2c9a55a9dd57> in <module>
----> 1 array1 * array2

ValueError: operands could not be broadcast together with shapes (2,3) (5,)
```

- 차원의 크기가 다른 경우 ValueError가 발생하며 연산이 이루어지지 않음

색인/슬라이스 기초

- 색인(index)이란 데이터를 저장되어 있는 위치를 의미하며 이를 이용하여 데이터를 선택하고 추출함
- 데이터의 일부 혹은 전체를 추출하기 위해 색인과 슬라이스를 사용함

```
array = np.arange(10)
```

```
array
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
array[4]
```

```
4
```

```
array[3:8]
```

```
array([3, 4, 5, 6, 7])
```

```
array[3:8] = 10
```

```
array
```

```
array([ 0,  1,  2, 10, 10, 10, 10, 10,  8,  9])
```

색인/슬라이스 기초

- numpy는 데이터 복사를 따로 하지 않는 이상 데이터가 복사되지 않음
- 데이터를 복사하기 위해서는 [:]로 슬라이스 해야 함

```
array_slice = array[3:8]
```

```
array_slice
```

```
array([10, 10, 10, 10, 10])
```

```
array_slice[1] = 100
```

```
array_slice
```

```
array([ 10, 100,  10,  10,  10])
```

```
array
```

```
array([  0,   1,   2,  10, 100,  10,  10,  10,   8,   9])
```

```
array_slice[:] = 99
```

```
array
```

```
array([ 0,  1,  2, 99, 99, 99, 99, 99,  8,  9])
```

ndarray 색인

- n차원 배열의 데이터를 선택하기 위해서는 재귀적으로 접근해야 함
  - d차원 배열의 개별 원소를 선택하기 위해서는 []를 d개 사용해야 함

```
array_2d = np.array([
    [1, 2, 3, 4],
    [10, 9, 8, 7],
    [0.1, 0.2, 0.3, 0.4]
])
```

```
array_2d.shape
```

```
(3, 4)
```

```
array_2d[2]
```

```
array([0.1, 0.2, 0.3, 0.4])
```

```
array_2d[2][3]
```

```
0.4
```

```
array_2d[2, 3]
```

```
0.4
```

ndarray 색인

```
array_3d = np.array([
    [[1, 2, 3, 4],
     [10, 9, 8, 7]],
    [[4, 3, 2, 1],
     [7, 8, 9, 10]]
])
```

```
array_3d
```

```
array([[[ 1,  2,  3,  4],
        [10,  9,  8,  7]],

       [[ 4,  3,  2,  1],
        [ 7,  8,  9, 10]]])
```

```
array_3d.shape
```

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

```
array_3d[0]
```

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

```
array_3d[0, 1]
```

```
array([10,  9,  8,  7])
```

```
array_3d[1, 1, 0]
```

```
7
```

ndarray 슬라이스

- 슬라이스를 사용하여 배열의 일부를 편리하게 선택할 수 있음

```
array_1d = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
array_1d[2:6]
```

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

```
array_2d = np.array([
    [1, 2, 3, 4],
    [10, 9, 8, 7],
    [0.1, 0.2, 0.3, 0.4]
])
```

```
array_2d
```

```
array([[ 1. ,  2. ,  3. ,  4. ],
       [10. ,  9. ,  8. ,  7. ],
       [ 0.1,  0.2,  0.3,  0.4]])
```

```
array_2d[:2]
```

```
array([[ 1.,  2.,  3.,  4.],
       [10.,  9.,  8.,  7.]])
```

```
array_2d[1:3]
```

```
array([[10. ,  9. ,  8. ,  7. ],
       [ 0.1,  0.2,  0.3,  0.4]])
```

ndarray 슬라이스

- 2차원 배열의 경우 슬라이스를 2개 차원에 대해 각각 적용할 수 있음

```
array_2d
```

```
array([[ 1. ,  2. ,  3. ,  4. ],
       [10. ,  9. ,  8. ,  7. ],
       [ 0.1,  0.2,  0.3,  0.4]])
```

```
array_2d[:2, :1]
```

```
array([[ 1.],
       [10.]])
```

```
array_2d[2, :3]
```

```
array([0.1, 0.2, 0.3])
```

```
array_2d[:2, 3]
```

```
array([4., 7.])
```

```
array_2d[:, 2]
```

```
array([3. , 8. , 0.3])
```

```
array_2d[:, 1:3]
```

```
array([[2. , 3. ],
       [9. , 8. ],
       [0.2, 0.3]])
```

ndarray 슬라이스

- n차원 배열도 동일하게 차원의 개수에 맞게 슬라이스 가능

```
array_3d = np.array([
    [[1, 2, 3, 4],
     [10, 9, 8, 7]],
    [[5, 6, 7, 8],
     [8, 7, 6, 5]],
    [[4, 3, 2, 1],
     [7, 8, 9, 10]]
])
```

```
array_3d[1:3]
```

```
array([[[ 5,  6,  7,  8],
        [ 8,  7,  6,  5]],

       [[ 4,  3,  2,  1],
        [ 7,  8,  9, 10]]])
```

```
array_3d[1:, 1]
```

```
array([[ 8,  7,  6,  5],
       [ 7,  8,  9, 10]])
```

```
array_3d[:, 1, 2:]
```

```
array([[ 8,  7],
       [ 9, 10]])
```

```
array_3d[1, :, :2]
```

```
array([[4, 3],
       [7, 8]])
```

ndarray 전치 (transpose)

- transpose로 행과 열을 뒤바꿈

```
array = np.arange(32).reshape(8, 4)
```

```
array
```

```
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],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

```
array.shape
```

```
(8, 4)
```

```
array.T
```

```
array([[ 0,  4,  8, 12, 16, 20, 24, 28],
       [ 1,  5,  9, 13, 17, 21, 25, 29],
       [ 2,  6, 10, 14, 18, 22, 26, 30],
       [ 3,  7, 11, 15, 19, 23, 27, 31]])
```

```
array.T.shape
```

```
(4, 8)
```

ndarray 전치 (transpose)

```
array = np.arange(32).reshape(2, 4, 4)
```

```
array
```

```
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],
        [24, 25, 26, 27],
        [28, 29, 30, 31]]])
```

```
array.shape
```

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

```
transpose_array = array.transpose((1, 0, 2))
```

```
transpose_array
```

```
array([[[ 0,  1,  2,  3],
        [16, 17, 18, 19]],

       [[ 4,  5,  6,  7],
        [20, 21, 22, 23]],

       [[ 8,  9, 10, 11],
        [24, 25, 26, 27]],

       [[12, 13, 14, 15],
        [28, 29, 30, 31]]])
```

```
transpose_array.shape
```

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

- n차원 배열에 대해서도 transpose를 사용할 수 있음
- 행과 열의 개념이 아닌 차원을 뒤바꾸는 개념으로 이해해야 함

III

numpy 연산 메소드

CHAPTER

- 유니버셜 메소드 활용
- 배열 연산 메소드 활용

Institute for
K-Digital Training
미래를 향한
인재를 양성합니다

유니버셜 메소드

- 하나 이상의 ndarray를 인자로 받아서 연산을 수행하는 numpy 제공 메소드

```
array = np.arange(10, 20)
array
```

```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

```
np.square(array)
```

```
array([100, 121, 144, 169, 196, 225, 256, 289, 324, 361])
```

```
np.sqrt(array)
```

```
array([3.16227766, 3.31662479, 3.46410162, 3.60555128, 3.74165739,
       3.87298335, 4.        , 4.12310563, 4.24264069, 4.35889894])
```

```
np.log10(array)
```

```
array([1.        , 1.04139269, 1.07918125, 1.11394335, 1.14612804,
       1.17609126, 1.20411998, 1.23044892, 1.25527251, 1.2787536 ])
```

```
array = np.array([0.2, 0.4, 0.6, 0.8, 1.])
```

```
np.ceil(array)
```

```
array([0., 0., 0., 0., 1.])
```

```
np.round(array)
```

```
array([0., 0., 1., 1., 1.])
```

유니버셜 메소드

- 2개의 배열을 인자로 받아서 연산을 수행하는 메소드도 존재함

```
array1 = np.arange(10)
array2 = np.arange(10, 20)
```

```
array1
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
array2
```

```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

```
np.add(array1, array2)
#np.subtract(array1, array2)
#np.multiply(array1, array2)
#np.divide(array1, array2)
```

```
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

```
np.maximum(array1, array2)
#np.minimum(array1, array2)
```

```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

```
np.power(array1, array2)
```

```
array([                    0,                     1,                 4096,
                     1594323,             268435456,          30517578125,
                2821109907456,     232630513987207,    18014398509481984,
       1350851717672992089])
```

numpy 조건절 사용

```
array1 = np.array([1, -1, -2, 3, 1, 2, -3])
array2 = np.array([1, 3, 2, 1, -1, 2, -3])
```

```
cond1 = array1 > 0
cond2 = array2 > 0
```

```
cond1
```

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

```
cond2
```

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

```
new_array = np.where(array1 > 0, 1, -1)
```

```
new_array
```

```
array([ 1, -1, -1,  1,  1,  1, -1])
```

```
new_array = np.where(array1 > 0, array1, 1)
```

```
new_array
```

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

```
new_array = np.where(array1 > 0, array1, array2)
```

```
new_array
```

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

- numpy는 boolean값이 들어있는 배열도 존재함
- np.where을 사용하여 if 연산을 배열 단위로 수행
  - x if condition else y
  - np.where(condition, x, y)

수학/통계 메소드

```
array = np.arange(32).reshape(8,4)
```

```
array
```

```
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],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

```
array.sum()
# np.sum(array)
```

```
496
```

```
array.mean()
# np.mean(array)
```

```
15.5
```

```
array.sum(axis=0)
```

```
array([112, 120, 128, 136])
```

```
array.mean(axis=0)
```

```
array([14., 15., 16., 17.])
```

```
array.mean(axis=1)
```

```
array([ 1.5,  5.5,  9.5, 13.5, 17.5, 21.5, 25.5, 29.5])
```

- ndarray의 전체 혹은 축을 따라서 수학/통계 연산을 수행하는 메소드
- axis를 인자로 받으면 해당 축을 기준으로 연산을 수행

수학/통계 메소드

```
array
```

```
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],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

```
array.max(axis=0)
```

```
array([28, 29, 30, 31])
```

```
array.min(axis=1)
```

```
array([ 0,  4,  8, 12, 16, 20, 24, 28])
```

```
array.argmax(axis=0)
```

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

```
array.argmin(axis=1)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0])
```

- max, min은 최대, 최소값을 반환
- argmax, argmin은 최대값, 최소값을 가지고 있는 index를 반환

numpy 정렬

- sort 메소드를 사용해 정렬, 새로운 배열을 반환하는 것이 아닌 해당 배열의 순서를 바꿈

```
array = np.random.randn(10)
```

```
array
```

```
array([ 0.22239961, -1.443217  , -0.75635231,  0.81645401,  0.75044476,
       -0.45594693,  1.18962227, -1.69061683, -1.35639905, -1.23243451])
```

```
array.sort()
```

```
array
```

```
array([-1.69061683, -1.443217  , -1.35639905, -1.23243451, -0.75635231,
       -0.45594693,  0.22239961,  0.75044476,  0.81645401,  1.18962227])
```

```
reverse_array = array[::-1]
```

```
reverse_array
```

```
array([ 1.18962227,  0.81645401,  0.75044476,  0.22239961, -0.45594693,
       -0.75635231, -1.23243451, -1.35639905, -1.443217  , -1.69061683])
```

numpy 정렬

- axis에 따라서 정렬을 수행하는 축이 달라짐

```
array = np.random.randn(5, 3)
```

```
array
```

```
array([[ 0.69012147,  0.68689007, -1.56668753],
       [ 0.90497412,  0.7788224 ,  0.42823287],
       [ 0.10887199,  0.02828363, -0.57882582],
       [-1.1994512 , -1.70595201,  0.36916396],
       [ 1.87657343, -0.37690335,  1.83193608]])
```

```
array.sort(axis=0)
array
```

```
array([[-1.1994512 , -1.70595201, -1.56668753],
       [ 0.10887199, -0.37690335, -0.57882582],
       [ 0.69012147,  0.02828363,  0.36916396],
       [ 0.90497412,  0.68689007,  0.42823287],
       [ 1.87657343,  0.7788224 ,  1.83193608]])
```

```
array.sort(axis=1)
array
```

```
array([[-1.70595201, -1.56668753, -1.1994512 ],
       [-0.57882582, -0.37690335,  0.10887199],
       [ 0.02828363,  0.36916396,  0.69012147],
       [ 0.42823287,  0.68689007,  0.90497412],
       [ 0.7788224 ,  1.83193608,  1.87657343]])
```

집합 메소드

- 배열 내 고유한 값을 추출할 수 있는 unique 메소드
- 배열의 원소가 다른 배열에 포함되어 있는지 알 수 있는 in1d 메소드

```
array = np.array(['a', 'b', 'c', 'a', 'b', 'c', 'e', 'f', 'g', 'c', 'f'])
```

```
np.unique(array)
```

```
array(['a', 'b', 'c', 'e', 'f', 'g'], dtype='<U1')
```

```
array1 = np.arange(10)
array2 = np.arange(5, 15)
```

```
array1
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
array2
```

```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
np.in1d(array1, array2)
```

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

행렬 연산 메소드

```
array = np.arange(8).reshape(4, 2)
```

```
array_transpose = array.T
# array_transpose = array.transpose()
```

```
array
```

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

```
array_transpose
```

```
array([[0, 2, 4, 6],
       [1, 3, 5, 7]])
```

```
np.dot(array, array_transpose)
```

```
array([[ 1,  3,  5,  7],
       [ 3, 13, 23, 33],
       [ 5, 23, 41, 59],
       [ 7, 33, 59, 85]])
```

```
array.shape, array_transpose.shape
```

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

```
np.dot(array_transpose, array)
```

```
array([[56, 68],
       [68, 84]])
```

- dot 메소드로 두 행렬을 곱함
- 두 행렬의 차원이 맞아야 함, 그렇지 않으면 ValueError 발생

내용 정리

- numpy를 통해 파이썬 내에서 데이터를 빠르고 효율적으로 계산하고 분석할 수 있음

- 효율적인 다차원 배열 ndarray를 제공, 배열을 쉽게 조작할 수 있음
  - 산술 연산, 색인/슬라이스, 전치

- 전체 데이터 배열에 대해 빠른 계산을 할 수 있는 다양한 메소드 사용 가능함
  - 수학/통계, 조건절 사용, 정렬, 행렬 연산

Thank you

junwonee@gmail.com