Lab. Using NumPy

1. Tool
   1)Microsoft Visual Studio Code
   2)Jupyter Notebook
   3)Google Colab
      #Google Colab 사용방법
         - 방향키 ↑↓: 셀 간 이동
         - Enter: 편집모드
         - Ctrl + Enter: 셀 실행
         - Shift + Enter: 셀 실행 + 다음 셀 선택
         - Ctrl +M D: 셀 삭제
         - Ctrl + M K: 셀 위로 이동
         - Ctrl + M J: 셀 아래로 이동

2. NumPy의 존재 이유
   1)NumPy는 Pandas, Scikit-learn, Tensorflow등 데이터 사이언스 분야에서 사용되는 라이브러리들의 토대가 되는 라이브러리.
   2)NumPy 그 자체로는 높은 수준의 데이터 분석 기능을 제공하지 않지만 NumPy를 활용해 데이터를 Python상에서 표현하고 다룰 줄 알아야만 데이터 분석이라는 그 이후 단계로 나아갈 수 있다.

3. NumPy 장점
   1)코어 부분이 C로 구현되어 동일한 연산을 하더라도 Python에 비해 속도가 빠름
   2)라이브러리에 구현되어있는 함수들을 활용해 짧고 간결한 코드 작성 가능
   3)효율적인 메모리 사용이 가능하도록 구현됨
   4)ndarray가 list보다 빠른 이유
      -Image 참조 https://image.slidesharecdn.com/numpy20160519-160516164831/95/numpy-8-638.jpg
   5)Python list가 느린 이유
      -Python list는 결국 포인터의 배열
      -경우에 따라서 각각 객체가 메모리 여기저기 흩어져 있음
      -그러므로 캐시 활용이 어려움
   6)NumPy ndarray가 빠른 이유
      -ndarray는 타입을 명시하여 원소의 배열로 데이터를 유지
      -다차원 데이터도 연속된 메모리 공간이 할당됨
      -많은 연산이 dimensions과 strides를 잘 활용하면 효율적으로 가능
      -가령 transpose는 strides를 바꾸는 것으로 거의 공짜

4. import NumPy
   import numpy as np

5. ndarray 배열 생성
   -Refer to https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html

   1)순차적인 객체를 받아 넘겨받은 data가 들어있는 새로운 NumPy 배열을 생성
   2)array() 함수 이용
      -Refer to https://cognitiveclass.ai/blog/nested-lists-multidimensional-numpy-arrays/
      -입력 data(list, tuple, 배열 또는 다른 순차형 data)를 ndarray로 변환
      -dtype이 명시되지 않은 경우에는 자료형을 추론하여 저장
      -기본적으로 입력 data는 복사됨.

```
data1 = [6,7.5, 8, 0, 1]
arr1 = np.array(data1)
arr1
---------------------------------
array([6. , 7.5, 8. , 0. , 1. ])

data2 = [[1,2,3,4], [5,6,7,8]]
arr2 = np.array(data2)
arr2
---------------------------------
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

arr2.ndim
---------------------------------
2

arr2.shape
---------------------------------
(2, 4)
```

   -[1, 2, 3] 배열을 생성.

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

   -X = [1, 2] 일 때(python list 상태) X를 배열로 변환.

```
X = [1,2]
np.array(X)
```

```
        # 다른 솔루션
        np.asarray(X)

    -X = [1, 2] 일 때(python list 상태) X를 'float'형 배열로 변환.
        X = [1, 2]
        np.array(X, float)

        # 다른 솔루션
        np.asarray(X, float)

        # 다른 솔루션
        np.asfarray(X)


3)zeros() / zeros_like()
    -주어진 dtype과 주어진 shape을 가지는 배열을 생성하고 내용을 모두 0으로 초기화한다.
    -zeros_like는 주어진 배열과 동일한 shape과 dtype을 가지는 배열을 새로 생성하여 내용을 모두 0으로 초기화한다.

        np.zeros(10)
        ------------------------
        array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

        np.zeros((3, 6))
        ------------------------
        array([[0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0.]])

        arr = np.zeros(10)
        arr2 = np.zeros_like(arr)
        print(arr)
        print(arr2)
        ---------------------------------------
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]

        #모든 원소가 0으로 채워진 3*2 실수형 다차원 배열을 생성하세요.
        np.zeros((3,2), float)
        ------------------------------------
        array([[0., 0.],
                [0., 0.],
                [0., 0.]])
    -모든 원소가 0으로 채워진 3*2 실수형 다차원 배열을 생성.
        np.zeros((3,2), float)

    -X = np.arange(4, dtype=np.int64) 일 때, X와 동일한 shape의 영행렬을 생성.
        X = np.arange(4, dtype=np.int64)
        np.zeros_like(x)


4)ones() / ones_like()
    -주어진 dtype과 주어진 shape을 가지는 배열을 생성하고 내용을 모두 1로 초기화한다.
    -ones_like는 주어진 배열과 동일한 shape과 dtype을 가지는 배열을 새로 생성하여 내용을 모두 1로 초기화한다.

        np.ones(10)
        ------------------------
        array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

        #모든 원소가 1로 채워진 3*2 실수형 다차원 배열을 생성
        np.ones([3,2], float)
        -------------------------------
        array([[1., 1.],
                [1., 1.],
                [1., 1.]])

    -모든 원소가 1로 채워진 3*2 실수형 다차원 배열을 생성.
        np.ones([3,2], float)

    -X = np.arange(4, dtype=np.int64) 일 때, X와 동일한 shape를 가지고 모든 원소가 1인 다차원 배열을 생성.
        X = np.arange(4, dtype=np.int64)
        np.ones_like(x)


5)empty() / empty_like()
    -memory를 할당하여 새로운 배열을 생성하지만 ones나 zeros처럼 값을 초기화하지 않는다.
    -즉, 0으로 초기화되지 않는 배열을 생성

        np.empty((2,3,2))
        ---------------------------------
        [[[3.19349014e-316, 3.61411312e-316],
          [3.95252517e-323, 3.61411312e-316],
          [3.95252517e-323, 3.61411312e-316]],
```

```
       [[4.44659081e-323, 0.00000000e+000],
          [3.61413842e-316, 3.61413842e-316],
          [3.61413842e-316, 0.00000000e+000]]]

       #2x2 integers 다차원 배열을 초기화 하지 않은 상태로 생성
       np.empty([2,2], int)
       ----------------------------------------
       [[4607182418800017408, 4607182418800017408],
          [4607182418800017408,                  0]]

       #X = np.array([1,2,3], [4,5,6], np.int32) 일 때, X와 동일한 shape를 가지는 다차원 배열을 초기화 하지 않은 상태로 생성하기
       X = np.array([[1,2,3], [4,5,6]], np.int32)
       np.empty_like(X)
       ----------------------------------------
       array([[1, 2, 3],
              [4, 5, 6]], dtype=int32)

   -2*2 integers 다차원 배열을 초기화 하지 않은 상태로 생성
       np.empty([2,2], int)

   -X = np.array([1,2,3], [4,5,6], np.int32) 일 때, X와 동일한 shape를 가지는 다차원 배열을 초기화 하지 않은 상태로 생성.
       X = np.array([[1,2,3], [4,5,6]], np.int32)
       np.empty_like(X)


6)arange()
   -Python의 range() 함수의 배열 version.
   -내장 range()와 유사하지만 list대신 ndarray를 반환

       np.arange(15)
       array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

       #x = np.arange(4, dtype=np.int64) 일 때, X와 동일한 shape를 가지고 모든 원소가 1인 다차원 배열을 생성하세요.
       x = np.arange(4, dtype=np.int64)
       np.ones_like(x)
       ----------------------------------
       array([1, 1, 1, 1])

       #x = np.arange(4, dtype=np.int64) 일 때, X와 동일한 shape의 영행렬을 생성하세요.
       x = np.arange(4, dtype=np.int64)
       np.zeros_like(x)
       ----------------------------------
       array([0, 0, 0, 0])

       #2, 4, 6, 8, ..., 100 을 원소로 가지는 배열을 생성하세요.
       np.arange(2, 101, 2)
       -----------------------------------
       array([  2,   4,   6,   8,  10,  12,  14,  16,  18,  20,  22,  24,  26,
                28,  30,  32,  34,  36,  38,  40,  42,  44,  46,  48,  50,  52,
                54,  56,  58,  60,  62,  64,  66,  68,  70,  72,  74,  76,  78,
                80,  82,  84,  86,  88,  90,  92,  94,  96,  98, 100])

       #3.0 에서 10.0 까지 50개의 원소가 균일하게 분포된 1차원 배열을 생성하세요.
       np.linspace(3., 10., 50)
       -------------------------------------------------
       array([ 3.        ,  3.14285714,  3.28571429,  3.42857143,  3.57142857,
                3.71428571,  3.85714286,  4.        ,  4.14285714,  4.28571429,
                4.42857143,  4.57142857,  4.71428571,  4.85714286,  5.        ,
                5.14285714,  5.28571429,  5.42857143,  5.57142857,  5.71428571,
                5.85714286,  6.        ,  6.14285714,  6.28571429,  6.42857143,
                6.57142857,  6.71428571,  6.85714286,  7.        ,  7.14285714,
                7.28571429,  7.42857143,  7.57142857,  7.71428571,  7.85714286,
                8.        ,  8.14285714,  8.28571429,  8.42857143,  8.57142857,
                8.71428571,  8.85714286,  9.        ,  9.14285714,  9.28571429,
                9.42857143,  9.57142857,  9.71428571,  9.85714286, 10.        ])

   -2, 4, 6, 8, ..., 100 을 원소로 가지는 배열을 생성.
       np.arange(2, 101, 2)

   -3.0 에서 10.0 까지 50개의 원소가 균일하게 분포된 1차원 배열을 생성.
       np.linspace(3., 10., 50)


7)eye(), identity()
   -N x N 크기의 단위 행렬(좌상단에서 우하단을 잇는 대각선은 1로 채워지고 나머지는 0으로 채운다)을 생성

       np.identity(3)  #크기가 3인 단위행렬
       ---------------------------
       array([[1., 0., 0.],
              [0., 1., 0.],
              [0., 0., 1.]])

   -크기가 3인 단위행렬을 생성.
       np.eye(3)
```

```
    np.identity(3)
```

8)full(), full_like()

```
#모든 원소가 6으로 채워진 2*5 uint형(부호없는 정수) 다차원 배열을 생성하세요.
np.full((2, 5), 6, dtype=np.uint)
----------------------------------------
array([[6, 6, 6, 6, 6],
       [6, 6, 6, 6, 6]], dtype=uint64)

#이것은 다음과 같이 할 수도 있다.
np.ones([2, 5], dtype=np.uint) * 6

#x = np.arange(4, dtype=np.int64)일 때, Create an array of 6's with the same shape and type as X.
x = np.arange(4, dtype=np.int64)
np.full_like(x, 6)
-----------------------------------
array([6, 6, 6, 6])

#또는
np.ones_like(x) * 6
```

9)asarray()

-입력 data를 ndarray로 변환하지만 입력 data가 이미 ndarray일 경우, 복사가 되지 않는다.

```
#[1, 2, 3] 배열을 생성하세요
np.array([1, 2, 3])
------------------------------
array([1, 2, 3])

#x = [1, 2] 일 때(python list 상태) x를 배열로 변환하세요.
x = [1,2]
np.array(x)
------------------------------
array([1, 2])

# 다른 솔루션
np.asarray(x)
------------------------------
array([1, 2])
```

6. ndarray의 자료형

1)자료형, dtype은 ndarray가 특정 data를 memory에서 해석하기 위해 필요한 정보를 담고 있는 특수한 객체이다.

```
arr1 = np.array([1,2,3], dtype=np.float64)
arr1.dtype
----------------------------------
dtype('float64')

arr2 = np.array([1,2,3], dtype=np.int32)
arr2.dtype
----------------------------------
dtype('int32')
```

2)정수형

-정수형의 default data type은 'int64'이다.
-진실: default data type은 운영체제에 따라 다르다.
-부호없는 정수형의 default data type은 'uint64'이다.
-int8 ,unit8(i1, u1)
  --부호가 있는 8bit(1Byte)와 부호가 없는 8bit 정수형
-int16, uint16(i2, u2)
  --부호가 있는 16bit 정수형과 부호가 없는 16bit 정수형
-int32, uint32(i4, u4)
  --부호가 있는 32bit 정수형과 부호가 없는 32bit 정수형
-int64, uint64(i8, u8)
  --부호가 있는 64bit 정수형과 부호가 없는 64bit 정수형

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

3)부호없는 정수형

```
uintArray = np.array([[1, 2], [3, 4]], dtype='uint')
uintArray
uintArray.dtype
```

4)실수형

-실수형의 default data type은 'float64'이다.

```
floatArray = np.array([[1.1, 2.2], [3.3, 4.4]])
```

```
    floatArray = np.array([[1.1, 2.2], [3.3, 4.4]], dtype='float64')
    floatArray
    floatArray.dtype

    -float16(f2)
      --반정밀도 부동소수점
    -float32(f4 or f)
      --단정밀도 부동소수점, C 언어의 float형과 호환
    -float64(f8 or d)
      --배정밀도 부동소수점, C 언어의 double형과 Python의 float 객체와 호환
    -float128(f16 or g)
      --확장 정밀도 부동소수점

5)복소수형
    complexArray = np.array([1+1j, 2+2j, 3+3j, 4+4j, 5+5j])
    complexArray
    complexArray.dtype

    -complex64, complex128, complex256(c8, c16, c32)
      --각각 2개의 32, 64, 128bit 부동소수점형을 가지는 복소수

6)bool
    -True, False 값을 저장하는 boolean형

      boolArray = np.array([True, False, True, True, False])
      boolArray
      boolArray.dtype

7)object(O)
    -Python 객체형

8)string_(S)
    -고정 길이 문자열형(각 글자는 1Byte).
    -길이가 10인 글자열의 dtype은 S10이 된다.

9)unicode_(U)
    -고정 길이 unicode(platform에 따라 글자별 byte 수는 다르다).
    -string_형과 같은 형식을 쓴다(예:U10)

10)dtype 변환하기
    -데이터가 정수로 입력되더라도 data type을 실수형으로 명시한다면 실수형으로 자동 형변환이 일어난다.

      floatArray2 = np.array([[1, 2], [3, 4]], dtype='float64')
      floatArray2
      floatArray2.dtype

    -정수형에서 실수형으로의 형변환 과정은 데이터 손실이 일어나지 않아 문제될 부분이 없지만 반대의 경우에는 문제가 발생할 수 있다.

      intArray2 = np.array([[1.1, 2.2], [3.3, 4.4]], dtype='int')
      intArray2

    -x = [1, 2] 일 때(python list 상태) x를 float형 배열로 변환하기.
      x = [1, 2]
      np.array(x, float)
      ----------------------------------
      array([1., 2.])

      # 다른 솔루션
      np.asarray(x, float)
      ----------------------------------
      array([1., 2.])

      # 다른 솔루션
      np.asfarray(x)
      ---------------------------------
      array([1., 2.])

11)astype()으로 변환하기

    arr = np.array([1,2,3,4,5])
    arr.dtype
    ---------------------------------------
    dtype('int64')

    float_arr = arr.astype(np.float64)
    float_arr.dtype
    ---------------------------------------
    dtype('float64')


    arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
    arr
    ------------------------------------------------
    array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
```

```
   arr.astype(np.int32)  #소수점 아랫자리를 버려진다.
   ---------------------------------------------
   array([ 3, -1, -2,  0, 12, 10], dtype=int32)

   -숫자 형식의 문자열을 담고 있는 배열이 있다면 astype을 사용하여 숫자로 변환 가능
      numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
      numeric_strings.astype(float)
      -----------------------------------------------------
      array([ 1.25, -9.6 , 42.  ])

   -형 변환이 실패하면 TypeError 예외 발생
   -원래 np.float64라고 해야 하는데, float라고 입력해도 변환가능

      int_array = np.arange(10)
      calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
      int_array.astype(calibers.dtype)
      ---------------------------------------------
      array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])

   -축약 code도 사용 가능

      empty_uint32 = np.empty(8, dtype='u4')
      empty_uint32
      ---------------------------------------------------
      array([         0, 1072693248,          0, 1072693248,          0,
              1072693248,          0,          0], dtype=uint32)



7. 배열 다루기 루틴
   1)X가 10 x 10 x 3의 다차원 배열일때, X의 두번째 차원이 150인 2차원 배열이 되도록 reshape하기.
      X = np.ones([10, 10, 3])
      X.reshape(2,150)

   2)X가 [[1, 2, 3], [4, 5, 6]]일 때, [1 4 2 5 3 6]로 변환.
      X = np.array([[1, 2, 3], [4, 5, 6]])
      np.ravel(X)
      X.flatten()

   3)X가 [[1, 2, 3], [4, 5, 6]]일 때, flatten한 후 5번째 원소를 가져오기.
      X = np.array([[1, 2, 3], [4, 5, 6]])
      out = X.flatten()
      out[4]

   4)다음 X와 Y를 연결해서 [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]] 를 만들기.
      X= [[ 1 2 3], [ 4 5 6]]
      Y = [[ 7 8 9],[10 11 12]]

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

   5)다음의 X와 Y를 연결해서 [[ 1  2  3], [ 4  5  6], [ 7  8  9], [10 11 12]]를 만들기.
      X = [[ 1 2 3], [ 4 5 6]
      Y = [[ 7 8 9], [10 11 12]]

      X = np.array([[1, 2, 3], [4, 5, 6]])
      Y = np.array([[7, 8, 9], [10, 11, 12]])
      np.concatenate((X, Y), 0)

   6)X가 [0, 1, 2]일 때, [0, 0, 1, 1, 2, 2]를 생성.
      X = np.array([0, 1, 2])
      np.repeat(X, 2)

   7)X가 [0, 0, 0, 1, 2, 3, 0, 2, 1, 0]일 때, 앞, 뒤의 0을 제거.
      X = np.array((0, 0, 0, 1, 2, 3, 0, 2, 1, 0))
      np.trim_zeros(X)



8. 배열과 Scala간의 연산
   1)for 반복문을 사용하지 않고 data를 일괄처리 가능 --> vector화
   2)같은 크기의 배열 간 산술연산은 배열의 각 요소 단위로 적용된다.

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

      arr * arr
      -------------------------------
      array([[ 1.,  4.,  9.],
```

```
                [16., 25., 36.]])

arr - arr
-------------------------------
array([[0., 0., 0.],
                [0., 0., 0.]])

1 / arr
-------------------------------
array([[1.        , 0.5       , 0.33333333],
          [0.25      , 0.2       , 0.16666667]])


arr ** 0.5
---------------------------------
array([[1.        , 1.41421356, 1.73205081],
                [2.        , 2.23606798, 2.44948974]])
```

9. Indexing과 Slicing

1)Slicing

-Python list와 유사하게, Numpy 배열도 슬라이싱이 가능하다.

-Numpy 배열은 다차원인 경우가 많기에, 각 차원별로 어떻게 슬라이스할건지 명확히 해야 한다

-Numpy 배열을 슬라이싱하면, 연속된 값을 가져오기에 결과로 얻어지는 배열은 언제나 원본 배열의 부분 배열이다.

```
# shape가 (3, 4)인 2차원 배열 생성
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

-다차원배열은 아래와 같은 방법으로 슬라이싱 한다.

```
a[ : , : ]
```

-a[행 슬라이싱 시작:행 슬라이싱 끝, 열 슬라이싱 시작:열 슬라이싱 끝]

-시작값부터 끝값전까지 슬라이싱 된다.

```
a[0:2, 0:4]
#0행부터 2행 전까지(= 1행 까지), 0열부터 4열 전까지(= 3열 까지) 슬라이스되었다.
```

-시작값이 0인 경우 생략 가능하며 끝 값이 shape의 값과 동일한 경우 생략 가능하다.

-a[0:2, 0:4]은 a[:2, :]로 표시할 수 있다.

```
a[:2, :]
```

-위 코드는 열을 슬라이싱 하지 않는 코드이며 이런 경우 열 부분은 전체 생략 가능하다.

-행을 슬라이싱 하지 않는다고 하더라도, 행부분을 생략할 순 없다.

```
a[:2]
```

-슬라이싱에 익숙해지기 전까진 생략하지말고 명시적으로 코드를 작성해주는것이 실수를 방지하는 방법이다.

2)Indexing

-Indexing을 통해 원소에 접근할 수 있다.

-두 가지 표현법이 있다.

```
# 선호하는 방식
a[0, 0]
a[0][0]
```

-Numpy 배열을 슬라이싱하면, 연속된 값을 가져오기에 결과로 얻어지는 배열은 언제나 원본 배열의 부분 배열이다.

-그러나 인덱싱을 한다면, 연속하지 않은 값을 가져올 수 있으니 원본과 다른 배열을 만들 수 있다.

```
# 0행, 2행만 인덱싱
a[[0, 2], ]

# 0열, 1열, 3열만 인덱싱
a[:, [0,1,3]]

arr2d = np.array([[1,2,3], [4,5,6], [7,8,9]])
arr2d[2]
---------------------------------------
array([7, 8, 9])

#아래의 2개의 표현을 같다.
arr2d[0][2]
---------------------------------
3

arr2d[0, 2]
---------------------------------
3
```

3)차원

-정수 Indexing과 Slicing을 혼합해서 사용하면 낮은 차원의 배열이 생성되지만,

-Slicing만 사용하면 원본 배열과 동일한 차원의 배열이 생성된다.

-0번째 행을 인덱싱하는 경우와 슬라이싱 하는 경우를 비교해보자.

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

# 슬라이싱만 사용
slicedRow = a[0:1, :]
print(slicedRow, slicedRow.shape, slicedRow.ndim)

# 아래와 동일한 코드
# 인덱싱&슬라이싱 혼합 사용
indexedRow = a[0]
print(indexedRow, indexedRow.shape, indexedRow.ndim)

# 위와 동일한 코드
# 인덱싱&슬라이싱 혼합 사용
indexedRow2 = a[0, :]
print(indexedRow2, indexedRow2.shape, indexedRow2.ndim)
```

-행이 아닌 열의 경우에도 마찬가지이다.

```
# 슬라이싱만 사용
slicedCol = a[:, 0:1]
print(slicedCol, slicedCol.shape, slicedCol.ndim)

# 인덱싱&슬라이싱 혼합 사용
indexedCol = a[:, 0]
print(indexedCol, indexedCol.shape, indexedCol.ndim)
```

-다차원배열간 연산에서 차원이 달라 문제가 발생하는 경우가 종종 있다.
-특별히 인덱싱을 써야 하는 상황이 아니라면, 인덱싱보다 슬라이싱을 추천.

4)Python의 list와 다른 점
-배열의 slice은 원본 배열의 view이다.
-즉, data는 값복사되는 것이 아니라 그대로 원본 배열에 반영된다.
-NumPy는 대용량 data 처리를 염두에 두고 설계되었기 때문에, 만약 NumPy가 data의 값복사를 남발한다면 성능과 memory문제에 직면할 것이기 때문이다.
-Indexing은 값을 복사한다.
-복사된 값을 변경해도 원본의 값은 변하지 않는다.
-Slicing된 배열은 원본 배열과 같은 데이터를 참조한다.
-만일 값복사를 하려면 arr[5:8].copy()를 사용해서 명시적으로 배열을 복사하면 된다.

```
# 인덱싱해서 b에 대입
b = a[0, 0]

# b값 수정
b = 100

print("a[0, 0]: {}".format(a[0, 0]))
print("b: {}".format(b))
```

-Slicing 배열은 원본 배열과 같은 데이터를 참조한다.
-즉 Slicing된 배열을 수정하면 원본 배열 역시 수정된다.

```
# a를 슬라이스하여 c 생성
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
c = a[1:3, 1:3]
print(c)

# c[0, 0]은 a[1, 1]과 같은 데이터.
c[0, 0] = 100

print(c)
print('------------------')
print(a)


arr[5:8] = 12
arr
----------------------------
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])

arr_slice = arr[5:8]
arr_slice[1] = 12345
arr
--------------------------------------
array([    0,     1,     2,     3,     4,    64, 12345,    64,     8, 9])

arr_slice[:] = 64
arr
----------------------------------------------
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

5)다차원 배열에서 마지막 index를 생략하면 반환되는 객체는 상위 차원의 data를 포함하고 있는, 한 차원 낮은 ndarray가 된다.

-즉, arr3d가 2 x 2 x 3크기의 배열이라면 arr3d[0]은 2 x 3 크기의 배열이다.

```
arr3d = np.array([[[1,2,3], [4,5,6]], [[7,8,9], [10,11,12]]])
arr3d
-----------------------------------------------
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])

arr3d[0]
-------------------------------------------------
array([[1, 2, 3],
       [4, 5, 6]])
```

-arr3d[0]에는 scala값과 배열 모두 대입할 수 있다.

```
old_values = arr3d[0].copy()
arr3d[0] = 42
arr3d
-------------------------------------
array([[[42, 42, 42],
        [42, 42, 42]],

       [[ 7,  8,  9],
        [10, 11, 12]]])


arr3d[0] = old_values
arr3d
--------------------------
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])

arr3d[1, 0]
------------------------------
array([7, 8, 9])
```

10. Boolean 배열 Indexing

1)Boolean 배열 Indexing을 통해 배열 속 요소를 취사선택할 수 있다.

2)Boolean 배열 Indexing은 특정 조건을 만족하게 하는 요소만 선택하고자 할 때 자주 사용된다.

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

bool_idx = (a > 2)  # 2보다 큰 a의 요소를 찾는다.
# 이 코드는 a와 shape가 같고 불리언 자료형을 요소로 하는 numpy 배열을 반환한다.
# bool_idx의 각 요소는 동일한 위치에 있는 a의 요소가 2보다 큰지를 말해준다.

print(bool_idx)      # 출력 "[[False False]
                     #      [ True  True]
                     #      [ True  True]]"
```

3)Boolean 배열 Indexing을 통해 bool_idx에서 참 값을 가지는 요소로 구성되는 rank 1인 배열을 구성할 수 있다.

```
print(a[bool_idx])  # 출력 "[3 4 5 6]"

# 위에서 한 모든것을 한 문장으로 할 수 있다.

print(a[a > 2])     # 출력 "[3 4 5 6]"
```

11. 정수 배열 Indexing

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

1)정수 배열 인덱싱의 예.

```
# 반환되는 배열의 shape는 (3,)
print(a[[0, 1, 2], [0, 1, 0]])  # 출력 "[1 4 5]"

# 위에서 본 정수 배열 인덱싱 예제는 다음과 동일하다.
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # 출력 "[1 4 5]"

# 정수 배열 인덱싱을 사용할 때 원본 배열의 같은 요소를 재사용할 수 있다.
print(a[[0, 0], [1, 1]])  # 출력 "[2 2]"

# 위 예제는 다음과 동일하다.
```

```
print(np.array([a[0, 1], a[0, 1]]))  # 출력 "[2 2]"
```

12. 전치(Transpose)
   1)종종 배열의 모양을 바꾸거나 데이터를 처리해야 할 때가 있다.
   2)가장 간단한 예는 행렬의 주 대각선을 기준으로 대칭되는 요소끼리 뒤바꾸는 것이다.
   3)이를 전치라고 하며 행렬을 전치하기 위해선, 간단하게 배열 객체의 'T' 속성을 사용하면 된다.

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

print(x.T)
```

   4)차원이 1인 배열을 전치할 경우 아무 일도 일어나지 않는다.

```
v = np.array([1,2,3])
print(v)    # 출력 "[1 2 3]"
print(v.T)  # 출력 "[1 2 3]"
```

13. Shape 변경
   1)numpy.reshape
      -Gives a new shape to an array without changing its data

```
np.arange(6)
np.arange(6).reshape((3, 2))
a = np.arange(6).reshape((3, 2))
a
np.reshape(a, (2,3))

# 2차원
np.reshape(a, (1,6))

# 1차원
np.reshape(a, 6)
```

   2)numpy.ravel()
      -Return a contiguous flattened array.

```
np.ravel(a)
a.ravel()
```

   3)numpy.ndarray.flatten()
      -Return a copy of the array collapsed into one dimension.

```
#np.flatten(a) # Numpy 모듈 함수가 아님
a.flatten()     # ndarray 객체의 메소드로만 사용 가능
```

   4)ravel()과 flatten() 차이점
      - flatten은 객체의 매소드로만 사용 가능
      - ravel()은 뷰를 반환, flatten은 복사본을 반환

   5)numpy.concatenate
      -연결
      -Refer to https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html#joining-arrays

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)
a
a[0:1].T
np.concatenate((a, b.T), axis=1)
```

14. 다차원 배열 연산
   1)기본적인 수학함수는 배열의 각 요소별로 동작하며 연산자를 통해 동작하거나 numpy 함수모듈을 통해 동작한다.
   2)다차원 배열간 연산시, shape가 맞아야 연산이 이루어진다.
   3)요소별 합, 차, 곱, 나눗셈의 경우 shape가 일치해야 한다.
   4)dot의 경우 앞 배열의 열과 뒤 배열의 행의 크기가 일치해야 한다.
   5)Refer to https://docs.scipy.org/doc/numpy/reference/routines.math.html

```
x = np.array([[1., 2.], [3., 4.]])
y = np.array([[5., 6.], [7., 8.]])
```

   6)요소별 합

```
x = [[ 6.0  8.0]
y = [10.0 12.0]]
print(x + y)
print(np.add(x, y))
```

7)요소별 차

```
x = [[-4.0 -4.0]
y = [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

8)요소별 곱

```
x = [[ 5.0 12.0]
y = [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))
```

9)요소별 나눗셈

```
x = [[ 0.2        0.33333333]
y = [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

10)요소별 제곱근

```
x = [[ 1.         1.41421356], [ 1.73205081  2.       ]]
print(np.sqrt(x))
```

11)Numpy에선 벡터의 내적, 벡터와 행렬의 곱, 행렬곱을 위해 `*`대신 `dot`함수를 사용한다.
12)dot은 Numpy 모듈 함수로서도 배열 객체의 메소드로서도 이용 가능한 합수이다

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# 벡터의 내적; 둘 다 결과는 219
print(v.dot(w))
print(np.dot(v, w))
```

13)행렬과 벡터의 곱

```
#둘 다 결과는 dimension 1인 배열 [29 67]
print(x.dot(v))
print(np.dot(x, v))
```

14)행렬곱

```
#둘 다 결과는 dimension 2인 배열
x = [[19 22]
y = [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

25. NumPy 고급 기능

1)Broadcasting은 Numpy에서 shape가 다른 배열 간에도 산술 연산이 가능하게 하는 메커니즘이다.
2)종종 작은 배열과 큰 배열이 있을 때, 큰 배열을 대상으로 작은 배열을 여러 번 연산하고자 할 때가 있다.
3)예를 들어, 행렬의 각 행에 상수 벡터를 더하는 걸 생각해보자.
4)이는 다음과 같은 방식으로 처리될 수 있습니다.

```
# 행렬 x의 각 행에 벡터 v를 더한 뒤,
# 그 결과를 행렬 y에 저장하고자 한다.
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
x =  "[[1  2  3]
      [4  5  6]
      [7  8  9]
      [10 11 12]]"

v = np.array([1, 0, 1])

x
v
x.shape
v.shape

y = np.empty_like(x)   # x와 동일한 shape를 가지며 비어있는 행렬 생성
y

# 명시적 반복문을 통해 행렬 x의 각 행에 벡터 v를 더하는 방법
for i in range(4):
    y[i, :] = x[i, :] + v

# 이제 y는 다음과 같다.
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

5)x가 매우 큰 행렬이라면, 파이썬의 반복문을 이용한 위 코드는 매우 느려질 수 있다.
6)벡터 v를 행렬 x의 각 행에 더하는 것은 v를 여러 개 복사해 수직으로 쌓은 행렬 vv를 만들고 이 vv를 x에 더하는것과 동일하다.
7)이 과정을 아래처럼 구현할 수 있습니다:

```
# 벡터 v를 행렬 x의 각 행에 더한 뒤,
# 그 결과를 행렬 y에 저장하고자 한다.
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])

vv = np.tile(v, (4, 1))  # v의 복사본 4개를 위로 차곡차곡 쌓은 것이 vv

vv.shape

print(vv)             # 출력 "[[1 0 1]
                      #      [1 0 1]
                      #      [1 0 1]
                      #      [1 0 1]]"

y = x + vv  # x와 vv의 요소별 합
print(y)    # 출력 "[[ 2  2  4
            #      [ 5  5  7]
            #      [ 8  8 10]
            #      [11 11 13]]"
```

8)Numpy 브로드캐스팅을 이용한다면 이렇게 v의 복사본을 여러 개 만들지 않아도 동일한 연산을 할 수 있다.
9)아래는 브로드캐스팅을 이용한 예시 코드입니다.

```
import numpy as np

# 벡터 v를 행렬 x의 각 행에 더한 뒤,
# 그 결과를 행렬 y에 저장하고자 한다.
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # 브로드캐스팅을 이용하여 v를 x의 각 행에 더하기
print(y)  # 출력 "[[ 2  2  4
          #      [ 5  5  7]
          #      [ 8  8 10]
          #      [11 11 13]]"
```

10)x의 shape가 (4, 3)이고 v의 shape가 (3,)라도 브로드캐스팅으로 인해 y = x + v는 문제없이 수행된다.
11)이때 `v`는 `v`의 복사본이 차곡차곡 쌓인 shape (4, 3)처럼 간주되어 `x`와 동일한 shape가 되며 이들 간의 요소별 덧셈연산이 y에 저장된다.
12)두 배열의 브로드캐스팅은 아래의 규칙을 따른다.
-두 배열이 동일한 dimension을 가지고 있지 않다면, 낮은 dimension의 1차원 배열이 높은 dimension 배열의 shape로 간주한다.
-특정 차원에서 두 배열이 동일한 크기를 갖거나, 두 배열 중 하나의 크기가 1이라면 그 두 배열은 특정 차원에서 compatible하다고 여겨진다.
-두 행렬이 모든 차원에서 compatible하다면, 브로드캐스팅이 가능하다.
-브로드캐스팅이 이뤄지면, 각 배열 shape의 요소별 최소공배수로 이루어진 shape가 두 배열의 shape로 간주한다.
-차원에 상관없이 크기가 1인 배열과 1보다 큰 배열이 있을 때, 크기가 1인 배열은 자신의 차원 수만큼 복사되어 쌓인 것처럼 간주한다.

13)브로드캐스팅을 지원하는 함수를 universal functions라고 한다.
14)Refer to https://docs.scipy.org/doc/numpy/reference/ufuncs.html
15)다음은 브로드캐스팅을 응용한 예시들이다.

```
import numpy as np

v = np.array([1,2,3])  # v의 shape는 (3,)
w = np.array([4,5])    # w의 shape는 (2,)
x = np.array([[1,2,3],
              [4,5,6]])

print(x)

# 벡터를 행렬의 각 행에 더하기
# x는 shape가 (2, 3)이고 v는 shape가 (3,)이므로 이 둘을 브로드캐스팅하면 shape가 (2, 3)인 아래와 같은 행렬이 나온다.
# [[2 4 6]
#  [5 7 9]]
print(x + v)

x.T
w
x.T + w
(x.T + w).T

# 벡터를 행렬의 각 행에 더하기
# x는 shape가 (2, 3)이고 w는 shape가 (2,)이다.
# x의 전치행렬은 shape가 (3,2)이며 이는 w와 브로드캐스팅이 가능하고 결과로 shape가 (3,2)인 행렬이 생긴다.
# 이 행렬을 전치하면 shape가 (2,3)인 행렬이 나오며
# 이는 행렬 x의 각 열에 벡터 w을 더한 결과와 동일하다.
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)

w
np.reshape(w, (2, 1))
```

```
    # 다른 방법은 w를 shape가 (2,1)인 열벡터로 변환하는 것이다.
    # 그런 다음 이를 바로 x에 브로드캐스팅해 더하면
    # 동일한 결과가 나온다.
    print(x + np.reshape(w, (2, 1)))
```

16)브로드캐스팅은 보통 코드를 간결하고 빠르게 한다.
17)그래서 권장한다.

26. 파일에서 데이터를 입력 받아 다차원 배열 생성하기
1)np.genfromtxt()을 이용하여 파일에 저장된 데이터를 입력받아 다차원 배열을 생성할 수 있다.
2)하지만 사용할 일은 많지 않다. 그 이유는,
-NumPy ndarray는 동일한 데이터타입만을 가질 수 있다.
-대부분 데이터 파일에는 하나의 데이터 타입만 있는게 아니라 정수 , 실수, 문자열이 섞여 있다.
3)파일에서 데이터를 읽어올땐 NumPy의 genfromtxt()보다는 Pandas의 read_csv()나 read_excel()을 주로 사용한다.

```
    #파일 다운로드 받기
    !wget -O 'mnist_train_super_small.csv'
    https://docs.google.com/spreadsheets/d/1lUBHgDlG5EVoZxfSF84X5jbcyynjnWSK8_weiQApt2M/export?format=csv

    import numpy as np
    fromCSVArray = np.genfromtxt('mnist_train_super_small.csv', delimiter=',')
    fromCSVArray
```

27. Google Drive에서 파일 가져오기
1)Google Drive Mount

```
    from google.colab import drive
    drive.mount('/content/drive')

    # Google Drive내에 있는 파일 경로 지정
    fromCSVArray = np.genfromtxt('/content/drive/My Drive/talk-on-seminar-numpy/resources/mnist_train_super_small.csv',
    delimiter=',')
    fromCSVArray
```

28. Statistics
1)Refer to https://docs.scipy.org/doc/numpy/reference/routines.statistics.html
2)numpy.amin
-Return the minimum of an array or minimum along an axis.

```
    A = np.arange(4).reshape((2,2))
    np.amin(A, 0)
    np.amin(A, axis=0)

    np.amin(A, 1)    # np.amin(A, axis=1)
    np.amin(A)
```

3)numpy.amax
-Return the maximum of an array or maximum along an axis.

```
    A = np.arange(4).reshape((2,2))
    np.amax(A, 0)    # np.amax(A, axis=0)

    np.amax(A, 1)   # np.amax(A, axis=1)
    np.amax(A, 1).shape
    np.amax(A)
```

4)numpy.ptp
-Range of values (maximum - minimum) along an axis.
-The name of the function comes from the acronym for ‘peak to peak’.

```
    A = np.arange(4).reshape((2,2))
    np.ptp(A, 0)
    np.ptp(A, 1)
    np.ptp(A)
```

5)numpy.median
-Compute the median along the specified axis.

```
    A = np.array([[10, 7, 4], [3, 2, 1]])
    np.median(A, 0)
    np.median(A, 1)
    np.median(A)
```

6)numpy.mean
-Compute the arithmetic mean along the specified axis.
-가중평균을 구하려면 numpy.average를 사용해야 한다.

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

```
np.mean(A, 0)
np.mean(A, 1)
np.mean(A)
```

7)numpy.var
-Compute the variance along the specified axis.

```
A = np.array([[1, 2], [3, 4]])
np.var(A, 0)
np.var(A, 1)
np.var(A)
```

8)numpy.std
-Compute the standard deviation along the specified axis.

```
A = np.array([[1, 2], [3, 4]])
np.std(A, 0)
np.std(A, 1)
np.std(A)
```

29. 선형대수학
1)다음 함수들은 넘파이에서 제공하는 선형대수(Linear Algebra) 함수들 중에서 자주 사용하는 함수들이다.
-행렬 곱: @ 또는 np.dot()
-역행렬(Inverse of a matrix): np.linalg.inv(x)
-단위행렬(Identity matrix): np.eye(n)
-대각합(Trace): np.trace(x)
-연립방정식 해 풀기(Solve a linear matrix equation): np.linalg.solve(a, b)
-고유값(Eigenvalue), 고유벡터 (Eigenvector): w, v = np.linalg.eig(x)
-대각행렬(Diagonal matrix): np.diag(x)
-내적(Dot product, Inner product): np.dot(a, b)
-행렬식(Matrix Determinant): np.linalg.det(x)
-특잇값 분해(Singular Value Decomposition): u, s, vh = np.linalg.svd(A)
-최소자승 해 풀기(Compute the Least-squares solution): m, c = np.linalg.lstsq(A, y, rcond=None)[0]

2)행렬의 곱
-행렬의 곱을 계산하려면 @ 연산자(파이썬 3.5 버전이상) 또는 dot() 함수를 사용하여 수행할 수 있다.

```
import   numpy   as np
a  =  np.array([[1.0,   2.0], [3.0, 4.0]])
a
---------------------------------------
array([[1., 2.],
           [3., 4.]])

j  =  np.array([[0.0,   -1.0],   [1.0, 0.0]])
j   @ j                      # 행렬의 곱
---------------------------------------------
array([[-1.,  0.],
            [ 0., -1.]])
```

3)행렬의 역행렬
-inv() 함수는 역행렬을 구한다.

```
np.linalg.inv(a)
--------------------------
array([[-2. ,  1. ],
          [ 1.5, -0.5]])
```

4)단위행렬만들기
-다음 코드는 2x2 단위행렬을 만든다.
-"eye"는 단위행렬 "I"를 의미한다.

```
u  =  np.eye(2)
u
------------------------
array([[ 1.,    0.],
           [  0.,    1.]])
```

5)행렬의 대각합
-trace() 함수는 대각합을 계산한다.

```
np.trace(u)
--------------------
2.0
```

6)선형 행렬 방정식
-solve() 함수는 선형 행렬 방정식을 구한다.

-ax = b의 정확한 해 x를 계산한다.

```
y  = np.array([[5.],  [7.]])
np.linalg.solve(a,  y)
---------------------------
array([[-3.],
       [ 4.]])
```