

서울대학교 삼성전자 AI전문가 과정

Data Analysis with Numpy

강의 & 실습목표

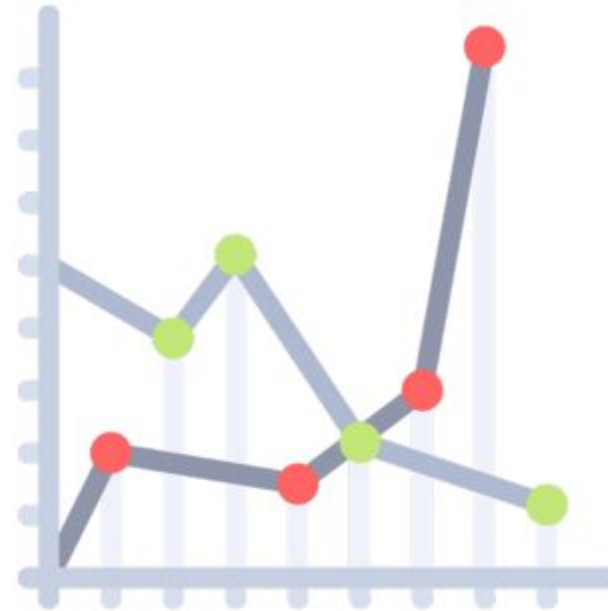
- 사전교육 1주차에 배웠던 NumPy를 복습하고, NumPy의 사용법에 익숙해지는데 초점을 맞춤
- 강의와 함께 간단한 실습 문제들을 풀면서 강의/실습 진행
- NumPy를 사용하면서 헛갈리기 쉬운 부분과 개발(ML, DL)을 진행하면서 자주 사용하는 기능을 위주로 실습을 진행
- 오늘 코드 실습은 Google Colab을 통해 진행

NumPy란?

- NumPy = Numerical Python의 약자
- 숫자로 이루어진 배열을 다루기 위한 전문 도구
- NumPy의 배열 vs Python의 list
 - 규모가 커질수록 저장 및 처리에 훨씬 효율적
 - 다양한 데이터 처리 **library** 혹은 모듈에서 사용됨
 - Pandas
 - SciPy

NumPy 배열

- 시계열 자료



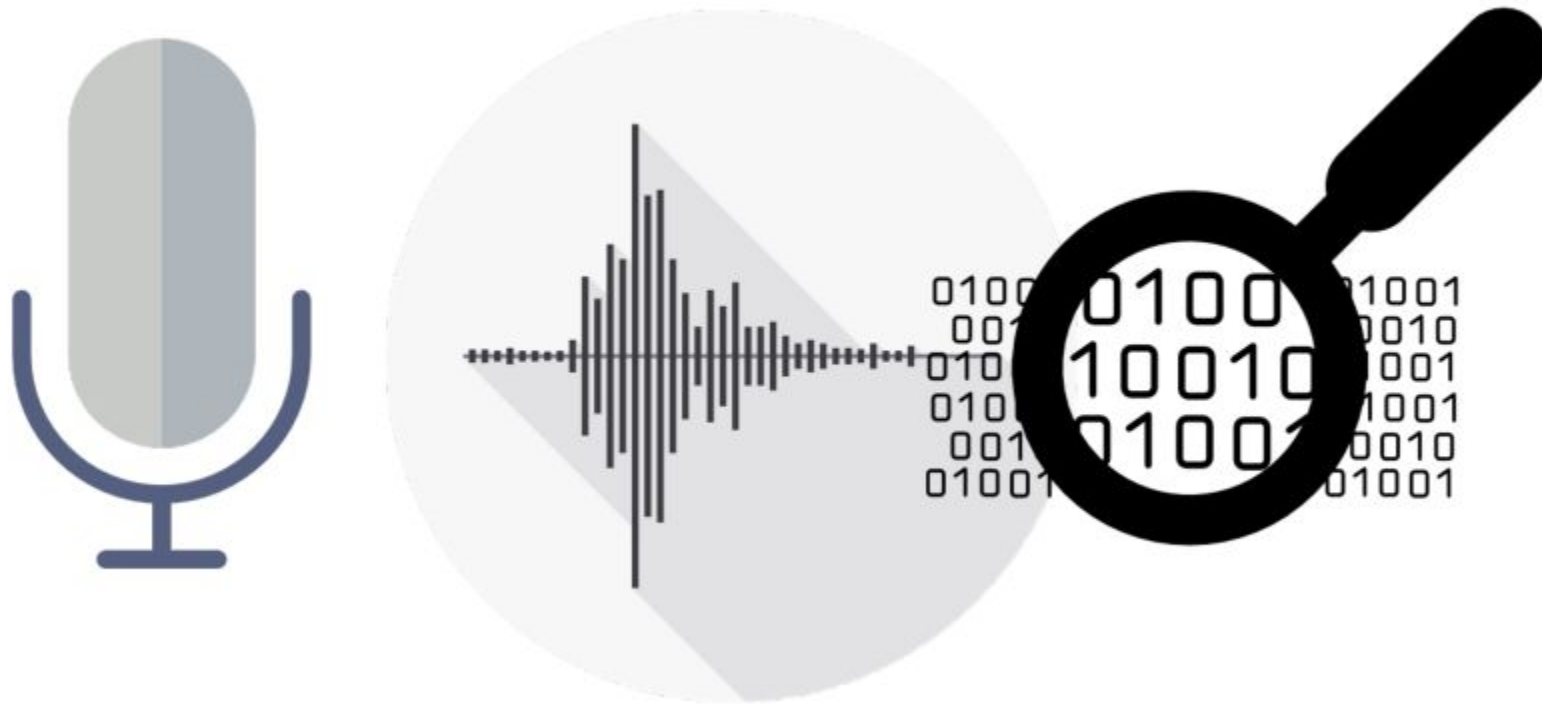
NumPy 배열

- 디지털 이미지 처리 및 분석



NumPy 배열

- 소리 신호 처리 및 분석



NumPy 버전 확인

- NumPy를 import한 후 `__version__`으로 확인

```
In [1]: import numpy  
        numpy.__version__  
  
Out[1]: '1.13.3'
```

- Import시에 `as` 문을 사용해서 축약어 사용 가능

```
In [2]: import numpy as np  
        np.__version__  
  
Out[2]: '1.13.3'
```

NumPy 배열의 생성

```
In [1]: import numpy as np
arr = [1, 4, 2, 5, 3]
n_arr = np.array(arr)
print(arr)
print(type(arr))
print(n_arr)
print(type(n_arr))

[1, 4, 2, 5, 3]
<class 'list'>
[1 4 2 5 3]
<class 'numpy.ndarray'>
```


NumPy 배열의 특징

- 배열의 모든 요소가 같은 타입이어야 함
- 타입이 일치하지 않을 경우 가능한 상위 타입을 취함

```
In [2]: import numpy as np

arr = [3.14, 2, 5, 3]
n_arr = np.array(arr)
print(arr)
print(type(arr))
print(n_arr)
print(type(n_arr))

[3.14, 2, 5, 3]
<class 'list'>
[ 3.14  2.    5.    3. ]
<class 'numpy.ndarray'>
```

```
In [4]: import numpy as np

arr = ['3.14', 2., 5, 3]
n_arr = np.array(arr)
print(arr)
print(type(arr))
print(n_arr)
print(type(n_arr))
```

NumPy 배열의 데이터 타입

- 명시적으로 데이터 타입을 설정하기 위해서는 `dtype`를 사용


```
In [5]: import numpy as np
arr = [1, 4, 2, 5, 3]
n_arr = np.array(arr, dtype='float32')
print(arr)
print(type(arr))
print(n_arr)
print(type(n_arr))

[1, 4, 2, 5, 3]
<class 'list'>
[ 1.  4.  2.  5.  3.]
<class 'numpy.ndarray'>
```


NumPy 배열의 속성

- Rank: 차원의 수
 - ex) `np.array([1, 2, 3, 4, 5])`의 rank = 1
 - `ndim` 속성을 통해 확인 가능

```
In [1]: import numpy as np
        nums = np.array([1, 4, 2, 5, 3])
        print(nums)
        print(nums.ndim)
        print(nums.shape)
        print(len(nums.shape))
        print(nums.size)
```




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




NumPy 배열의 속성

- Shape: 배열의 모양
 - ex) `np.array([1, 2, 3, 4, 5])`의 `shape = (5,)`
 - `shape` 속성을 통해 확인 가능 (tuple로 정보 반환)

```
In [1]: import numpy as np
        nums = np.array([1, 4, 2, 5, 3])
        print(nums)
        print(nums.ndim)
        print(nums.shape)
        print(len(nums.shape))
        print(nums.size)
```

 [1 4 2 5 3]

 (5,)

1
5

NumPy 배열의 속성

- Shape의 응용
 - len() 함수를 사용할 경우 **차원의 수**를 산출 가능
 - ex) len(nums.shape)

```
In [1]: import numpy as np
        nums = np.array([1, 4, 2, 5, 3])
        print(nums)
        print(nums.ndim)
        print(nums.shape)
        print(len(nums.shape))
        print(nums.size)
```

[1 4 2 5 3]

1

(5,)


1

5

NumPy 배열의 속성

- 배열에 있는 **element**의 수
 - **size** 속성을 사용
 - **len()** 함수를 사용할 경우, 다차원 배열에서 문제 발생

```
In [1]: import numpy as np
        nums = np.array([1, 4, 2, 5, 3])
        print(nums)
        print(nums.ndim)
        print(nums.shape)
        print(len(nums.shape))
        print(nums.size)
```



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

```
1
```

```
(5,)
```

```
1
```

```
5
```



NumPy 배열의 값 접근

- NumPy 배열 내부의 값을 가져오기 위해 첨자 사용

```
In [1]: import numpy as np
        nums = np.array([1, 4, 2, 5, 3])
        print(nums[1])
        print(nums[:3])
        print(nums[2:4])

4
[1 4 2]
[2 5]
```

NumPy 배열의 값 접근

- 슬라이싱
 - 표준 Python 리스트의 구문을 그대로 따름
 - `Array[start:stop]`
 - `Array[start:stop:step]`
 - 값이 지정되지 않은 경우 기본값 사용
 - `start = 0, stop = 차원의 크기, step = 1`

```
In [1]: import numpy as np
        nums = np.array([1, 4, 2, 5, 3])
        print(nums[::2])

[1 2 3]
```


NumPy의 2차원 배열

- 1차원 배열

1	4	2	5	3
---	---	---	---	---

- 2차원 배열

1	4	2
7	5	3

NumPy의 2차원 배열

- `array()` 함수를 사용한 2차원 배열의 선언

```
In [1]: import numpy as np
        nums = np.array([[1, 4, 2], [7, 5, 3]])
        print(nums)

[[1 4 2]
 [7 5 3]]
```

- 1차원 array 2개를 원소로 갖는 array 사용

$$\left. \begin{array}{l} [1, 4, 2] \\ [7, 5, 3] \end{array} \right\} \quad [[1, 4, 2], [7, 5, 3]]$$

NumPy의 2차원 배열

- 아래와 같은 경우 어떤 결과가 나올지 직접 작성해보세요 (실습)

```
In [1]: import numpy as np
        nums = np.array([[1, 4, 2], [7, 5, 3]])
        print(nums)
        print(nums.ndim)
        print(nums.shape)
        print(len(nums.shape))
        print(nums.size)
```

NumPy의 2차원 배열

- 아래와 같은 경우 어떤 결과가 나올지 직접 작성해보세요

```
In [1]: import numpy as np
        nums = np.array([[1, 4, 2], [7, 5, 3]])
        print(nums)
        print(nums.ndim)
        print(nums.shape)
        print(len(nums.shape))
        print(nums.size)
```

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

NumPy 2차원 배열의 값 접근

- Array[행, 열]
- Array[행][열]

```
In [1]: import numpy as np
        nums = np.array([[1, 4, 2], [7, 5, 3]])
        print(nums)
        print(nums[0,2])
        print(nums[0][2])

[[1 4 2]
 [7 5 3]]
2
2
```

NumPy 2차원 배열의 값 접근

- Array[슬라이스, 슬라이스]

1	4	2
7	5	3

1	4	2
7	5	3

1	4	2
7	5	3

1	4	2
7	5	3

```
In [2]: import numpy as np
nums = np.array([[1, 4, 2], [7, 5, 3]])
print(nums)
print(nums[0:1,])
print(nums[0:1,:])
print(nums[:,1:2])
print(nums[1,1:])
print(nums[0:,1:])
```

[[1 4 2]
 [7 5 3]]
[[1 4 2]]
[[1 4 2]]
[[4]
 [5]]
[5 3]
[[4 2]
 [5 3]]

NumPy 2차원 배열의 값 접근

- 범위로 슬라이싱 할 때와 특정 행/열을 슬라이싱할 때의 Shape가 다름에 유의!

```
In [3]: import numpy as np
nums = np.array([[1, 4, 2], [7, 5, 3]])
print(nums[0:1,])
print(nums[0:1,:])
print(nums[0,:])
```

```
[[1 4 2]]
[[1 4 2]]
[1 4 2]
```

1	4	2
7	5	3

NumPy 2차원 배열의 값 접근

- 이번엔 다음의 색칠된 영역을 선택해보세요 (실습)

1	4	2
7	5	3
1	4	2
7	5	3
1	4	2
7	5	3
1	4	2
7	5	3
1	4	2
7	5	3

```
In [1]: import numpy as np  
nums = np.array([[1, 4, 2], [7, 5, 3]])
```

```
5  
[7 5]  
[[2]  
 [3]]  
[[1 4]  
 [7 5]]
```

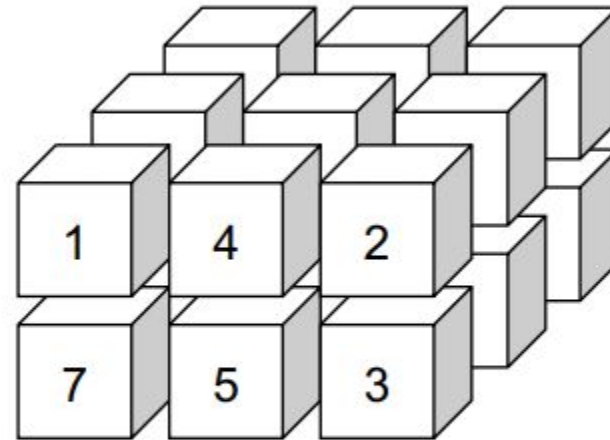

NumPy의 3차원 배열

• 3차원 배열

1	4	2
7	5	3

0	4	8
6	9	1

7	6	9
4	0	8



NumPy의 3차원 배열

- `array()` 함수를 사용하여 아래와 같은 3차원 `array`를 선언해봅시다

1	4	2
7	5	3

0	4	8
6	9	1

7	6	9
4	0	8

```
In [1]: import numpy as np
nums = np.array([[[1,4,2],[7,5,3]],
                 [[0,4,8],[6,9,1]],
                 [[7,6,9],[4,0,8]]])

print(nums)

[[[1 4 2]
  [7 5 3]]

 [[0 4 8]
  [6 9 1]]

 [[7 6 9]
  [4 0 8]]]
```

NumPy 배열 사용시 주의사항

- 배열에 대한 검색/슬라이싱 결과는 참조만 할당
 - 기존 배열이 변경될 경우 값이 같이 변경 됨
- 독립적으로 사용해야할 경우 `copy()` 함수 사용

```
In [1]: import numpy as np
        nums = np.array([1, 4, 2, 5, 3])
        ref = nums[1:4]
        cpy = nums[1:4].copy()
        print(ref)
        print(cpy)
        nums[2] = 10
        print(ref)
        print(cpy)
```

[4 2 5]
[4 2 5]
[4 10 5]
[4 2 5]

NumPy 내장 함수를 사용한 배열 생성

- `zeros(shape)`
 - 모든 값이 0인 배열 생성
 - ex) `np.zeros((3, 3))`
- `ones(shape)`
 - 모든 값이 1인 배열 생성
 - ex) `np.ones((2, 3))`
- `full(shape, value)`
 - 모든 값이 `value`로 초기화된 배열 생성
 - ex) `np.full((2, 2), 7)`

NumPy 내장 함수를 사용한 배열 생성

- `random.random(shape)`
 - 0과 1사이의 임의의 숫자로 초기화된 배열 생성
 - ex) `np.random.random((2, 2))`
- `random.normal(loc, scale, size)`
 - 정규분포를 따르는 난수 배열 생성 (*loc* = center of distribution, *scale* = standard deviation)
 - ex) `np.random.normal((2, 2))`
- `random.randint(low, high, shape)`
 - *low*부터 *high*사이의 정수형 난수 (*high*는 포함되지 않음)
 - ex) `np.random.random((2, 2))`

NumPy 내장 함수를 사용한 배열 생성

- `arange(start, stop, step, dtype=?)`
 - start부터 stop까지 step 단위로 배열에 값을 생성 (슬라이싱과 동일)
 - ex) `arange(1, 3, 1)`인 경우 `[1, 2]`
 - `dtype`으로 자료형 지정

```
In [1]: import numpy as np
        print( np.arange(0.4, 1.1, 0.1) )

[ 0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

NumPy 내장 함수를 사용한 배열 생성

- linspace(*start*, *stop*, *num*=?, *endpoint*=?, *retstep*=?)
 - start부터 stop까지 num만큼 나눠서 배열 생성
 - endpoint는 stop을 범위에 포함할지 여부를 결정
 - retstep은 생성된 배열과 함께 step의 크기를 return할지 결정

```
In [1]: import numpy as np
print( np.linspace(0, 1, num=5, endpoint=True) )
print( np.linspace(0, 1, num=5, endpoint=False) )
print( np.linspace(0, 1, num=5, endpoint=False, retstep=True) )

[ 0.    0.25  0.5   0.75  1. ]
[ 0.    0.2  0.4  0.6  0.8]
(array([ 0. ,  0.2,  0.4,  0.6,  0.8]), 0.20000000000000001)
```

NumPy 내장 함수를 사용한 배열 생성

- identity (*size*)
 - size x size인 단위행렬 생성
- eye(*size*, k=n)
 - size x size인 단위행렬 생성 후 k만큼 이동
 - ex) np.eye(3, k=1)

```
In [1]: import numpy as np
print( np.identity(3) )
print( np.eye(3) )
print( np.eye(3, k=1) )
print( np.eye(3, k=-2) )

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


배열의 전환과 형태 변형

- reshape(*shape*)
 - 기존의 배열을 다른 형태로 변형할때 사용
 - ex) .reshape(2,4) => 기존 배열을 (2, 4) 형태로 변형
 - 기존 배열과 새로운 배열의 **element** 수는 동일해야함
 - ex) (1,7)을 (2,4)로 변형 불가

```
In [1]: import numpy as np
sap = np.array(["MMM", "ABT", "ABBV", "ACN", "ACE", "ATVI", "ADBE", "ADT"])
print(sap)
sap2d = sap.reshape(2, 4)
print(sap2d)

['MMM' 'ABT' 'ABBV' 'ACN' 'ACE' 'ATVI' 'ADBE' 'ADT']
[['MMM' 'ABT' 'ABBV' 'ACN']
 ['ACE' 'ATVI' 'ADBE' 'ADT']]
```

배열의 전환과 형태 변형

- transpose (전치)
 - 함수 호출이 아닌 속성 T를 사용
 - ex) arr가 (2, 4) 배열일 경우 arr.T는 (4, 2)

```
In [1]: import numpy as np
sap = np.array(["MMM", "ABT", "ABBV", "ACN", "ACE", "ATVI", "ADBE", "ADT"])
sap2d = sap.reshape(2, 4)
print(sap2d)
print(sap2d.T)

[[ 'MMM' 'ABT' 'ABBV' 'ACN' ]
 [ 'ACE' 'ATVI' 'ADBE' 'ADT' ]]
[[ 'MMM' 'ACE' ]
 [ 'ABT' 'ATVI' ]
 [ 'ABBV' 'ADBE' ]
 [ 'ACN' 'ADT' ]]
```

배열의 전환과 형태 변형

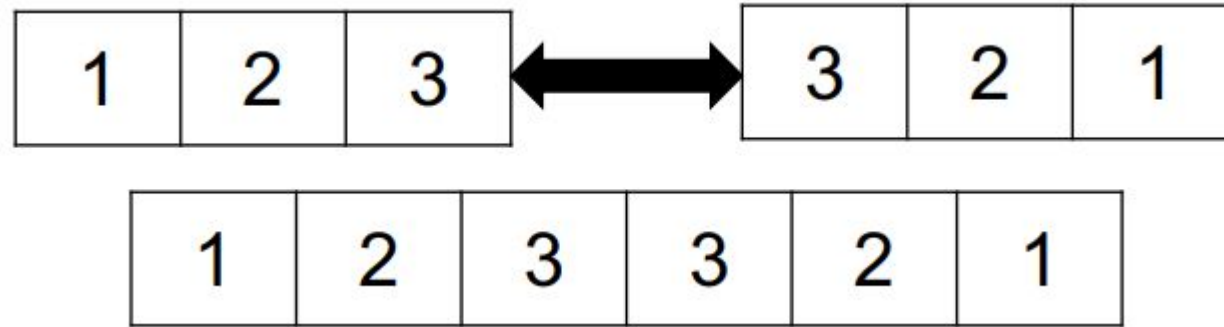
- `transpose()`
 - 배열의 축을 바꿀때 사용하는 함수
 - 변형 후의 축의 `index`를 순서대로 기재
- `swapaxes(axis_1, axis_2)`
 - 배열의 축을 바꿀때 사용하는 함수
 - 맞바꿀 축을 넣어줌
 - ex) `arr.swapaxes(0, 1) == arr.T`

```
import numpy as np
sap = np.array(["MMM", "ABT",
               "ABBV", "ACN",
               "ACE", "ATVI",
               "ADBE", "ADT"])
sap2d = sap.reshape(2, 4)
print(sap2d)
print(sap2d.T)
print(sap2d.swapaxes(0, 1))
```

```
[[ 'MMM' 'ABT' 'ABBV' 'ACN' ]
 [ 'ACE' 'ATVI' 'ADBE' 'ADT' ]]
[[ 'MMM' 'ACE' ]
 [ 'ABT' 'ATVI' ]
 [ 'ABBV' 'ADBE' ]
 [ 'ACN' 'ADT' ]]
[[ 'MMM' 'ACE' ]
 [ 'ABT' 'ATVI' ]
 [ 'ABBV' 'ADBE' ]
 [ 'ACN' 'ADT' ]]
```

배열의 연결

- concatenate ([arr1, arr2, ...])
 - Numpy 배열을 연결할 때 사용하는 함수



배열의 연결

- concatenate ([arr1, arr2, ...])
 - 2개 이상의 배열들을 연결할 때

```
In [1]: import numpy as np
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
z = np.array([4, 5, 6])
print(np.concatenate([x, y, z]))

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

배열의 연결

- concatenate ([arr1, arr2, ...])
 - 2차원 배열을 연결할 때
 - 2차원 배열은 연결할 수 있는 축이 2개 (기본 axis = 0)

```
In [1]: import numpy as np
        grid = np.array([[1, 2, 3],
                          [4, 5, 6]])
        print(np.concatenate([grid, grid]))
```

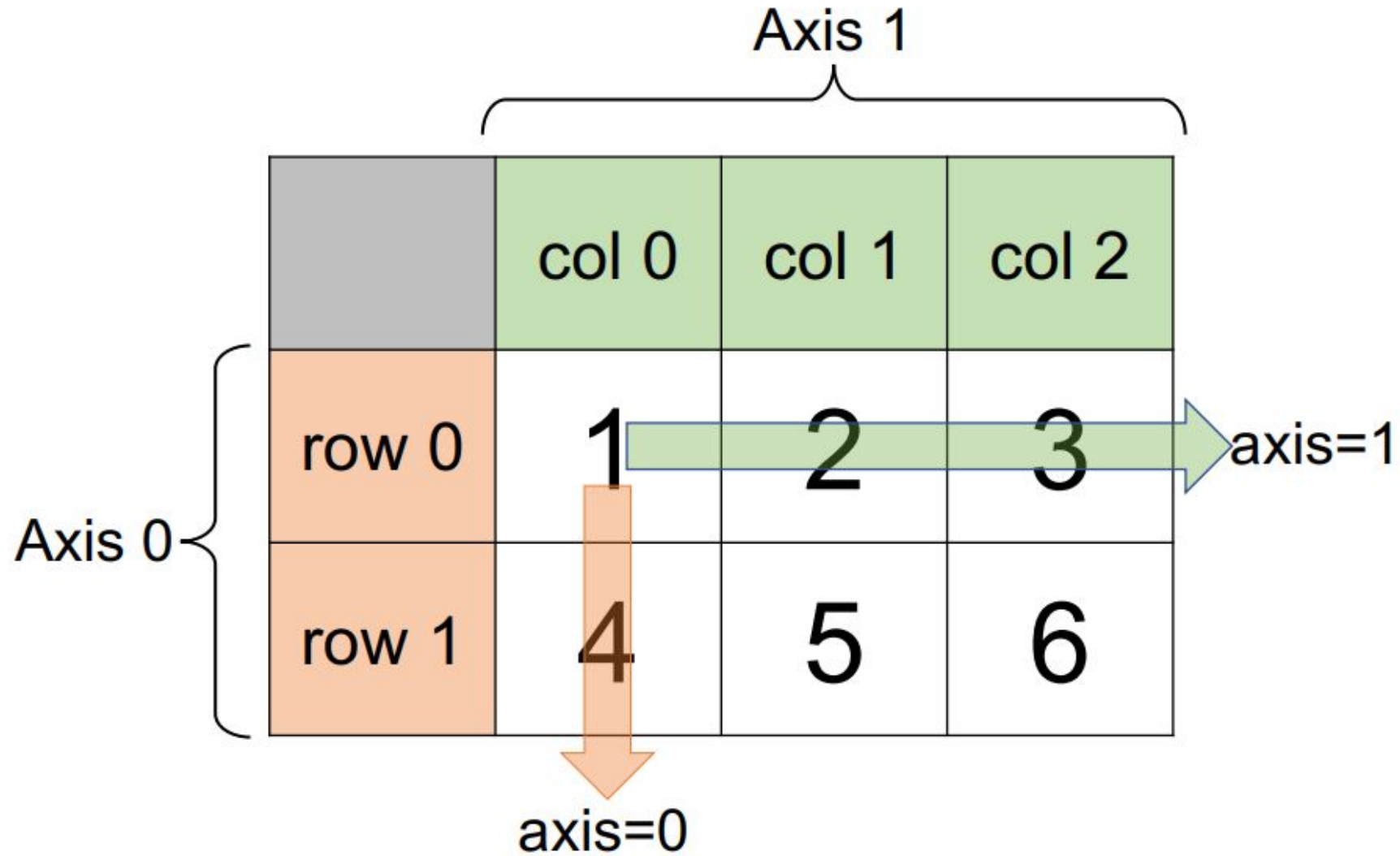
```
[[1 2 3]
 [4 5 6]
 [1 2 3]
 [4 5 6]]
```

1	2	3
4	5	6



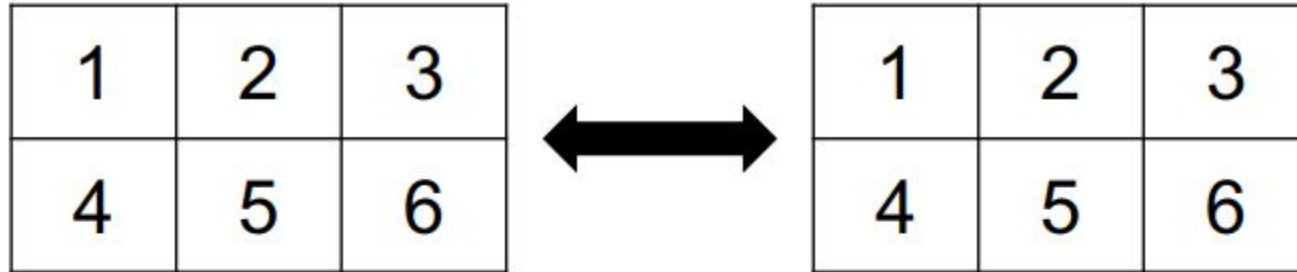
1	2	3
4	5	6

NumPy 배열의 축



배열의 연결

- 아래와 같이 2차원 배열을 연결하려면 어떻게 해야할지 생각해봅시다



1	2	3	1	2	3
4	5	6	4	5	6

배열의 연결

- 아래와 같이 2차원 배열을 연결하려면 어떻게 해야할지 생각해봅시다

```
In [1]: import numpy as np
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
print(np.concatenate([grid, grid], axis=1))
```

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



1	2	3	1	2	3
4	5	6	4	5	6

배열의 연결

- vstack ([arr1, arr2, ...])
 - 수직 방향으로 배열을 병합

```
In [1]: import numpy as np
x = np.array([9, 8, 7])
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
print(np.vstack([x, grid]))

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

배열의 연결

- hstack ([arr1, arr2, ...])
 - 수평 방향으로 배열을 병합

```
In [1]: import numpy as np
y = np.array([[9],
              [8]])
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
print(np.hstack([y, grid]))

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

배열의 분할

- split (arr, 분할할 지점, axis)
 - ex) np.split(arr, [3, 5])

```
In [1]: import numpy as np
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

배열의 분할

- split (arr, 분할할 지점, axis)
 - 2차원 배열의 분할

```
In [1]: import numpy as np
        grid = np.arange(16).reshape((4, 4))
        upper, lower = np.split(grid, [2])
        print(upper)
        print(lower)

[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

배열의 분할

- `split (arr, 분할할 지점, axis)`
 - 다른 방향으로 배열을 분할하려면 어떻게 해야 할지 생각해봅시다

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

배열의 분할

- vsplit()
 - split의 axis=0과 동일한 작업 수행
 - 첫번째 축을 따라가면서 분할

```
In [1]: import numpy as np
        grid = np.arange(16).reshape((4, 4))
        upper, lower = np.vsplit(grid, [1])
        print(upper)
        print(lower)
```

```
[[0 1 2 3]]
[[ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

배열의 분할

- `hsplit()`
 - `split`의 `axis=1`과 동일한 작업 수행
 - 두번째 축을 따라가면서 분할

```
In [1]: import numpy as np
        grid = np.arange(16).reshape((4, 4))
        left, right = np.hsplit(grid, [1])
        print(left)
        print(right)
```

```
[[ 0]
 [ 4]
 [ 8]
[12]]
[[ 1  2  3]
 [ 5  6  7]
 [ 9 10 11]
[13 14 15]]
```


NumPy 배열 연산

- 반복문을 활용하여 배열에 저장된 화씨 온도를 섭씨로 바꿔봅시다 (실습)

$$C = (F - 32) \times \frac{5}{9}$$

```
In [1]: arr = [100, 80, 70, 90, 110]
```

```
[37, 26, 21, 32, 43]
```

NumPy 배열 연산

- NumPy 배열의 경우 반복문에서 성능이 좋지 않음 (물론 python보다는 좋음)
 - NumPy에서는 반복문 없이 벡터화 연산을 사용하여 한번에 연산을 수행할 수 있음

```
arr = np.array([100, 80, 70, 90, 110])
arr_d = np.array([])

for i in arr_f:
    arr_d = np.append(arr_d, np.array([(i * 2)]))

print(arr_d)

[200. 160. 140. 180. 220.]

print(arr * 2)

[200 160 140 180 220]
```

NumPy 배열 연산

- 벡터화 연산을 활용하여 배열에 저장된 화씨 온도를 섭씨로 바꿔봅시다 (실습)

$$C = (F - 32) \times \frac{5}{9}$$

```
In [1]: arr = [100, 80, 70, 90, 110]
```

```
[37, 26, 21, 32, 43]
```

Broadcasting

- NumPy는 두 배열들의 **shape**가 같은 경우 각 요소 단위로 연산을 수행

```
In [1]: import numpy as np  
a = np.array([0, 1, 2])  
b = np.array([5, 5, 5])  
print(a + b)  
  
[5 6 7]
```

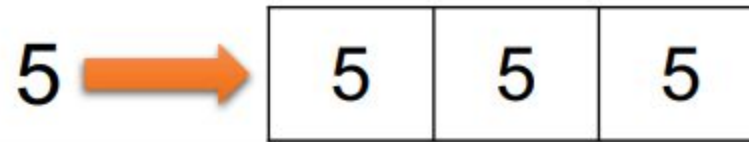
Broadcasting

- 아래와 같은 경우에는 어떻게 처리가 될까요?

```
In [2]: import numpy as np  
a = np.array([0, 1, 2])  
b = 5  
print(a + b)
```

Broadcasting

- NumPy에서는 연산 대상 둘 중 하나가 스칼라 값인 경우 배열로 확장 혹은 복제 후 연산을 처리



```
In [2]: import numpy as np  
a = np.array([0, 1, 2])  
b = 5  
print(a + b)  
[5 6 7]
```

Broadcasting

- 아래의 코드를 실행시켜보고 결과를 살펴봅시다

```
In [1]: import numpy as np
noise = np.eye(4) + 0.01 * np.ones((4,))
print(np.eye(4))
print(np.ones((4,)))
print(noise)
```

```
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
[ 1.  1.  1.  1.]
[[ 1.01  0.01  0.01  0.01]
 [ 0.01  1.01  0.01  0.01]
 [ 0.01  0.01  1.01  0.01]
 [ 0.01  0.01  0.01  1.01]]
```

Broadcasting Rules

- 두 배열의 차원 수가 다르면 더 작은 수의 차원을 가진 배열의 차원을 늘린다
 - 배열의 앞쪽(왼쪽)을 1로 채운다
- 두 배열의 형상이 어떤 차원에서도 일치하지 않는다면 해당 차원의 형상이 1인 배열이 다른 형상과 일치하도록 늘어난다
- 임의의 차원에서 크기가 일치하지 않고 1도 아니라면 오류가 발생한다

Broadcasting Rules

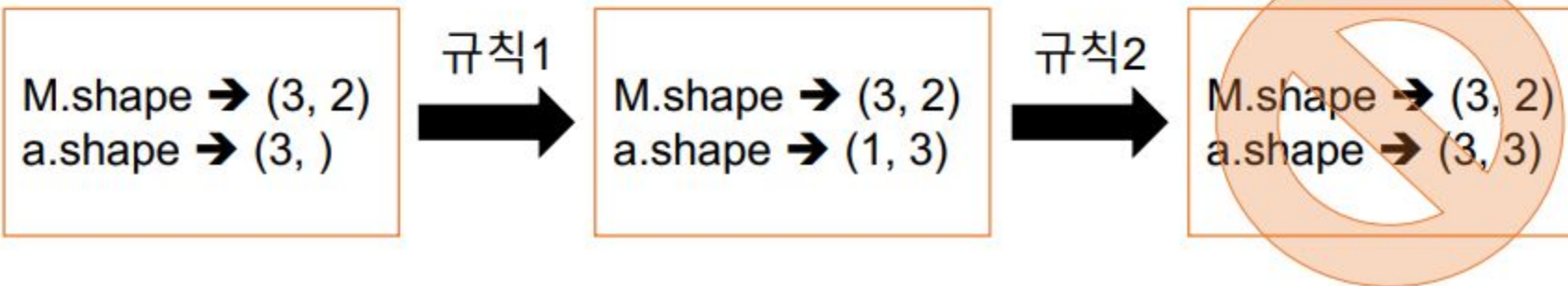
- Broadcasting이 되지 않는 경우

```
In [1]: import numpy as np
M = np.ones((3, 2))
a = np.arange(3)
print(M)
print(a)
```

```
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
[0 1 2]
```

```
In [1]: import numpy as np
M = np.ones((3, 2))
a = np.arange(3)
print(M.shape)
print(a.shape)
```

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

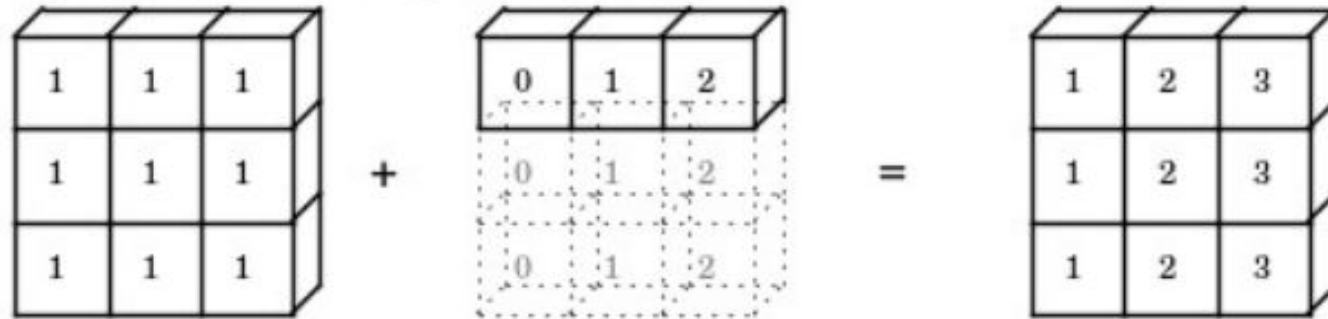
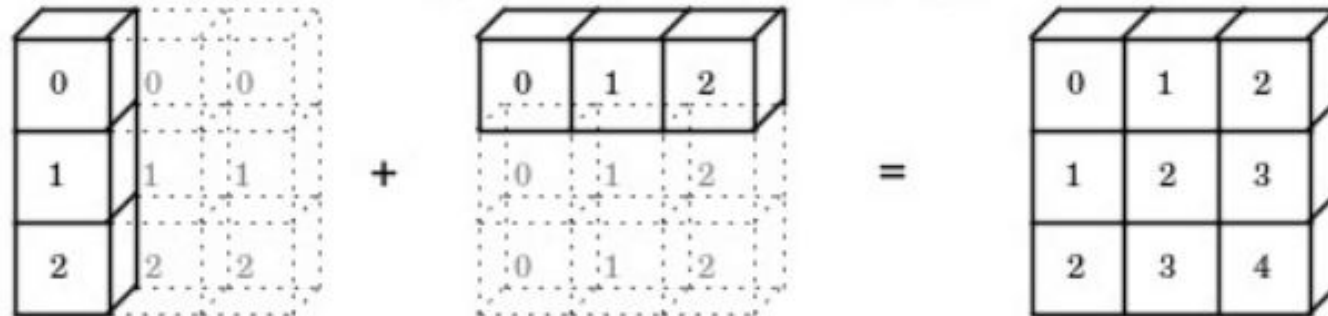


Broadcasting Rules

- 두 배열에 대해 operation을 적용할 때 shape를 element-wise로 비교
- 각 배열은 최소한 한 차원(dimension)을 가지고 있어야 함
- 두 배열의 차원을 뒤쪽부터 시작해서 비교하였을 때, 크기가 동일하거나, 둘 중 하나가 1이거나, 둘 중 하나가 존재하지 않으면 broadcastable하다.

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):    7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

Broadcasting

 $\text{np.arange}(3) + 5$  $\text{np.ones}((3, 3)) + \text{np.arange}(3)$  $\text{np.arange}(3).reshape((3, 1)) + \text{np.arange}(3)$ 

Universal Functions

- `abs()`
 - 절댓값 함수 (python 내장 함수)

```
In [1]: import numpy as np  
a = np.array([0, -1, 2])  
print(abs(a))  
[0 1 2]
```

- `np.absolute()`
 - NumPy 모듈의 절댓값 함수
 - 복소수도 연산 가능

Universal Functions

- 사칙연산
 - `add()`, `multiply()`, `negative()`, `exp()`, `log()`, `sqrt()`
- 삼각함수
 - `sin()`, `cos()`, `hypot()`
- 비트단위
 - `bitwise_and()`, `left_shift()`
- 기타
 - `maximum()`, `minimum()`

Aggregation Functions

- 원소 전체의 합계 구하기
 - `sum()`
 - `python` 내장함수와는 달리 다차원 배열에 대해서도 적용 가능
- 최솟값과 최댓값
 - `min()`, `max()`
- 평균, 표준편차, 분산, 중앙값, 백분위수
 - `mean()`, `std()`, `var()`, `median()`, `percentile()`...

Aggregation Functions

- 아래와 같은 배열을 선언하고 평균을 구해보세요

1	2	3
4	6	2

Aggregation Functions

- 아래와 같은 배열을 선언하고 평균을 구해보세요

1	2	3
4	6	2

```
In [1]: import numpy as np
        grid = np.array([[1, 2, 3],
                          [4, 6, 2]])
        print(np.mean(grid))
```

```
3.0
```


Aggregation Functions

- 이번에는 **axis** 파라미터를 사용하여 각 행, 열별 평균을 구해봅시다

```
In [2]: import numpy as np
        grid = np.array([[1, 2, 3],
                          [4, 6, 2]])
        print(np.mean(grid, axis=0))

[ 2.5  4.   2.5]
```

1	2	3
4	6	2

```
In [3]: import numpy as np
        grid = np.array([[1, 2, 3],
                          [4, 6, 2]])
        print(np.mean(grid, axis=1))

[ 2.  4.]
```

Boolean 배열 (Masked array)

- NumPy 배열에는 산술연산자 외에 비교 연산자 활용 가능
 - ex) <, >, <=, >=, ==, !=

```
In [1]: import numpy as np
x = np.array([1, 2, 3, 4, 5])
print(x < 3)
print(x >= 3)
print(x == 3)
print(x != 3)

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

Boolean 배열 (Masked array)

- 2차원 배열에도 적용 가능

```
In [1]: import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
print(x < 6)
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
[[ True  True  True  True]
 [False False  True  True]
 [ True  True False False]]
```

Boolean 배열 (Masked array)

- 6보다 작은 값이 몇 개인지 찾기
 - True는 1, False는 0으로 해석
 - `count_nonzero()` 함수를 통해 True의 숫자 파악

```
In [1]: import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
print(np.count_nonzero(x < 6))
```

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

Boolean 배열 (Masked array)

- 6보다 작은 값이 몇 개인지 찾기
 - True는 1, False는 0으로 해석
 - sum() 함수를 활용하여 True의 숫자 파악 가능

```
In [1]: import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
print(np.sum(x < 6))
```

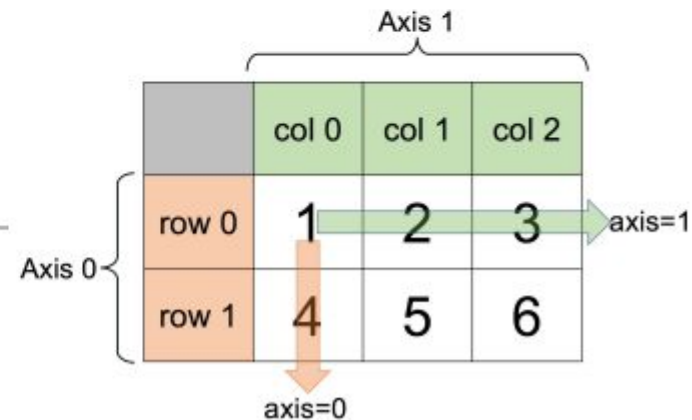
```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
8
```

Boolean 배열 (Masked array)

- 각 행, 열별로 6보다 작은 값은 몇 개가 있는지 구해보세요 (실습)

```
In [1]: import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
```

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



Boolean 배열 (Masked array)

- 값 중 하나라도 참이 있는지 확인 -> `any()` 함수
- 모든 값이 참인지 확인 -> `all()` 함수

```
In [1]: import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
print(np.any(x > 8))
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
True
```

```
In [1]: import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
print(np.all(x < 6, axis=1))
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
[ True False False]
```

Boolean 배열과 bitwise 연산자

- Python의 비트 단위 연산자를 활용하여 여러 가지 조건을 결합 가능
 - & (and), | (or), ^ (xor), ~ (not)
- ex) 3이상 6미만인 값이 몇 개 인지 찾기

```
In [1]: import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
print(np.sum((x >= 3) & (x < 6)))

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

- 3 이하 또는 6 이상인 값이 몇 개인지 찾아봅시다 (실습)

Boolean 배열 (Masked array)

- 배열에서 조건에 맞는 값들만 선택하고자 할 때, Boolean 배열을 사용
 - 조건에 맞는 값들로 채워진 1차원 배열 도출
- Boolean 배열과 bitwise 연산자를 사용하여 3 이하 또는 6 이상인 값을 출력해봅시다 (실습)

```
In [1]: import numpy as np
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x[x < 5])

[0 3 3 3 2 4]
```

Summary

- NumPy 소개
- NumPy 배열의 기초
 - dtype, rank, shape, slicing
 - zeros, ones, full, eye, identity
 - random, linspace, arange, reshape, transpose, swapaxes
- NumPy 배열 연산
 - 벡터화 연산, broadcasting
 - Aggregation Functions
- Boolean 배열 (Masked array)
 - Bitwise 연산