Lecture 3

NumPy 배열 마스킹

마스킹(Masking)

- 외부의 병균이나 미세먼지로부터 막기 위해 마스크를 착용하는 것처럼 **마스킹**은 불필요한 것들을 걸러내는 '**필터링**'과 유사한 기법이라고 생각하면 된다
- 넘파이 **배열 마스킹**은 마스크를 사용해서 **배열의 특정 부분만 표시**하는 것으로 보면 된다
- 마스크의 값들은 기본적으로 boolean 자료형으로 구성하며, 마스크의 원소값이 True면 대응되는 원본 배열의 원소를 가져오고 False면 걸러내는 방식을 취한다
- 대량의 데이터를 사용할 때 특정 데이터만을 추출하고자 하면 반복문보다는 이처럼 마스킹 기법이 보다 효율적으로 작동될 수 있기에 필요하다
- 잇따라 오는 예제 코드를 통해 마스킹에 대해 더 알아보자

마스킹(Masking)

- 넘파이 **배열 마스킹**은 마스크를 사용해서 **배열의 특정 부분만 표시**하는 것으로 보면 된다
- 마스크는 **ndarray 자료형**으로 만들고 그 안의 원소값들은 **boolean 자료형**으로 구성한다
- 마스크와 대응되는 값이 True이면 가져오고 False이면 걸러내는 것이다
- 사용법은 **마스킹대상배열[마스크배열]** 이처럼 기존의 인덱싱, 슬라이싱과 비슷하게 **대괄호[]** 안에 마스크 배열을 넣어주어 마스킹하고자 하는 타겟 배열의 마스킹을 수행한다

In [1]:
```
1  import numpy as np
2  my_first_mask = np.array([True, False, True, False])
3  my_second_mask = np.array([1, 0, 1, 0], dtype=bool)
4  print(my_first_mask)
5  print(my_second_mask) # my_first_mask와 동치
6  sample_arr = np.arange(12).reshape(4,3) # 4행 3열
7  print(sample_arr)
8  print(sample_arr[my_first_mask]) # (4,3) [(4,)] shape 성립;
   마스크의 정보에 따라 0행과 2행만을 가져와 새로운 배열을
   만들었다
9  print(sample_arr[my_second_mask]) # 위와 동치
```

```
[ True False  True
False]
[ True False  True
False]
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[[0 1 2]
 [6 7 8]]
```

마스킹(Masking)

- 마스크의 shape은 기존 배열의 shape의 차원 앞에서부터 대응이 되어야 한다; 기존 배열의 shape의 차원 앞에서부터 축의 길이가 동일해야 한다 (마스크 배열 사이즈 <= 기존 배열 사이즈)
- shape가 맞지 않은 경우 IndexError가 발생한다

In [2]:
```
1  mask1 = np.array([ [True, False, True],
                      [True, False, True],
                      [True, False, True],
                      [True, False, True] ])
2  print(sample_arr[mask1]) # (4,3) [(4,3)] shape 성립
3  print(type(sample_arr[mask1]))
```

```
[ 0  2  3  5  6  8  9 11]
<class 'numpy.ndarray'>
```

In [3]:
```
1  mask2 = mask1.flatten()
2  print(mask2)
3  sample_arr[mask2]
```

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

```
IndexError: boolean index did not match indexed array along dimension
0; dimension is 4 but corresponding boolean dimension is 12
```

잠깐! *flatten 메소드는 ndarray 배열 객체를 1D로 reshape 해준다; reshape(-1)과 동치*

마스킹(Masking)

- ndarray 배열 객체에 ~를 접두하면 내부 원소들에 비트 연산 NOT을 수행한다
- 이 점을 활용하여 마스크 배열 앞에 ~를 접두하여 마스크의 False 데이터만 추출할 수 있다

In [4]:
```
1 print(~np.array([True]))
2 print(~np.array([False]))
3 print(sample_arr[my_first_mask])
4 print(sample_arr[~my_first_mask]) # 올바른 False 마스킹
5 print(~sample_arr[my_first_mask]) # ~ 비트연산자 NOT
```

→

```
[False]
[ True]
[[0 1 2]
 [6 7 8]]
```

```
[[ 3  4  5]
 [ 9 10 11]]
[[-1 -2 -3]
 [-7 -8 -9]]
```

- 마스크 내부 원소값들을 **int 자료형**으로 구성할 수도 있다
- 다만 작동 방식은 bool 자료형과 다르게 되는데, 원소값은 **행의 인덱스 값으로 인식**이 된다

In [5]:
```
1 int_mask = np.array([1, 0, 1, 0], dtype=int)
2 print(sample_arr) # 원래 배열
3 print(sample_arr[int_mask]) # 마스킹된 배열
```

→

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

```
[[3 4 5]
 [0 1 2]
 [3 4 5]
 [0 1 2]]
```

마스킹(Masking)

- 그런데 매번 배열들에 대응되는 마스크 값을 일일히 지정할 수가 없다 (특히 배열의 크기가 커지면...)
- **배열들의 데이터를 필터링하는 기준**이 있을테니 이를 활용하는 것이 좋은 방식이다
- 원본 배열에서 각종 연산을 거쳐서 이를 마스크 배열로써 활용해보자

In [6]:
```
1 print(sample_arr)
2 print(sample_arr[sample_arr>=5]) # 비교 연산자
3 print(sample_arr[sample_arr%2==0]) # 나머지 연산자
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[ 5  6  7  8  9 10 11]
[ 0  2  4  6  8 10]
```