Week1_예습과제

제 1장. 파이썬 기반의 머신러닝과 생태계 이해

01. 머신러닝의 개념

- 애플리케이션을 수정하지 않고도 데이터를 기반으로 패턴을 학습하고 결과를 예측하는 알고리즘 기법을 통칭
- 데이터를 기반으로 다양한 수학적 기법을 적용해 데이터 내의 패턴을 인지하고 예측 결과를 도출해냄

머신러닝의 분류

: 머신러닝은 지도학습과 비지도학습, 강화학습으로 나뉨.

지도학습 (Supervised Learning)

- 분류, 회귀, 추천 시스템, 시각/음성 감지/인지, 텍스트 분석, NLP

비지도학습(Un-supervised Learning)

- 클러스터링, 차원 축소, 강화학습

02. 파이썬 머신러닝 생태계를 구성하는 주요 패키지

- 머신러닝 패키지: 사이킷런(Scikit-Learn)
- 행렬/선형대수/통계 패키지: 넘파이(Numpy), 사이파이(SciPy)
- 데이터 핸들링: 판다스
- 시각화: 맷플롭립(Matplotlib), 시본(Seaborn)

03. 넘파이

- Numerical Python을 의미하는 Numpy는 파이썬에서 선형대수 기반의 프로그램을 쉽게 만들 수 있도록 지원하는 대표적인 패키지
- 루프를 사용하지 않고 대량 데이터의 배열 연산을 가능하게 함 → 빠른 배열 연산 속도
- C/C++과 같은 저수준 언어 기반의 호환 API를 제공

넘파이 ndarray 개요

넘파이 모듈 임포트

```
import numpy as np
```

as np를 추가해 약어로 모듈을 표현하는 것이 관례

- 넘파이의 기반 데이터 타입은 ndarray, 이를 이용하여 넘파이에서 다차원 배열을 쉽게 생성하고 다양한 연산을 수행
- 넘파이 array()함수는 파이썬의 리스트와 같은 다양한 인자를 입력 받아서 ndarray로 변환하는 기능을 수행
- ndarray배열의 shape변수: ndarray의 크기 → 즉 행과 열의 수를 튜플 형태로 갖고 있으며 이를 통해 ndarray배열의 차원까지 알 수 있음

```
array1 = np.array([1,2,3])
print('array type:',type(array1))
print('array1 array 형태:',array1.shape)

array2 = np.array([[1,2,3],
                   [2,3,4]])
print('array2 type:',type(array2))
print('array2 array 형태:',array2.shape)

array3 = np.array([[1,2,3]])
```

```
print('array3 type:',type(array3))
print('array array 형태:',array3.shape)
```

- array1의 shape : (3,) → 1차원 array로 3개 데이터
- array2의 shape : (2,3) → 2차원 array로, 2개의 로우와 3개의 칼럼으로 구성. (2*3=6개의 데이터)
- array3의 shape : (1,3) → 1개의 로우와 3개의 칼럼으로 구성된 2차원 데이터

> 💡 **array1과 array3의 차이는?**
>
> array1과 array3의 동일한 데이터 건수를 갖고 있음
>
> But!
>
> array1은 1차원임을 (3,)으로 표현하였고 array3은 로우와 칼럼으로 이뤄진 2차원 데이터임을 (1,3)으로 표현한 것임

ndarray.ndim을 통해 각 array의 차원 확인하기

```
print('array1:{:0}차원, array2: {:1}차원, array3:{:2}차원'
.format(array1.ndim,array2.ndim,array3.ndim))
```

- array1 : 1차원, array2: 2차원, array3: 2차원
- array()함수의 인자로 파이썬의 리스트 객체가 주로 사용
- 리스크 [ ]→ 1차원, 리스트 [[ ]] → 2차원과 같은 형태로 배열의 차원과 크기를 쉽게 표현할 수 있기 때문

ndarray의 데이터 타입

- 데이터 값은 숫자 값, 문자열 값, 불 값 등 모두 가능
- int형(8, 16, 32bit), unsigned int형(8, 16, 32 bit), float형(16, 32, 64, 128bit), complex의 숫자형 타입 제공

- ndarray내의 데이터 타입은 같은 데이터 타입만 가능 → 즉 한 개의 ndarray객체에 int와 float가 함께 있을 수 없음
- dtype으로 데이터 타입 확인 가능

```
list1 = [1,2,3]
print(type(list1))
array1 = np.array(list1)
print(type(array1))
print(array1, array1.dtype)
```

: 리스트 자료형을 ndarray로 바꾸고 이렇게 변경된 ndarray내의 데이터값은 모두 int32

If, 다른 데이터 유형이 섞여 있는 리스트를 ndarray로 변경하면?

⇒ 데이터 크기가 더 큰 데이터 타입으로 형 변환 일괄 적용 !

```
list2 = [1,2,'test']
array2 = np.array(list2)
print(array2, array2.dtype)

list3 = [1,2,3.0]
array3 = np.array(list3)
print(array3, array3.dtype)
```

: array2는 숫자형 값 1,2가 모두 문자열 값인 '1,','2'로 변환

: array3은 int 1, 2가 모두 1. 2.인 float64형으로 변환

astype( )메서드 : ndarray 내 데이터값 타입 변경

- 인자로 원하는 타입을 문자열로 지정
- 메모리를 더 절약해야할 때 보통 이용(int형으로 충분할 때 float형이면 int형으로 바꿔 메모리 절약)

```
array_int = np.array([1,2,3])
array_float = array_int.astype('float64')
print(array_float, array_float.dtype)
```

```
array_int1 = array_float.astype('int32')
print(array_int1, array_int1.dtype)

array_float1 = np.array([1.1, 2.1, 3.1])
array_int2 = array_float1.astype('int32')
print(array_int2, array_int2.dtype)
```

: int32형 데이터→ float64형으로 변환→int32형으로 변환

: float형→int형으로 변환하는 경우 소수점 이하는 없어짐

ndarray를 편리하게 생성하기 - arange, zeros, ones

- 연속값이나 0 또는 1로 초기화해 생성해야할 필요가 있는 경우
- 테스트용으로 데이터를 만들거나 대규모 데이터를 일괄적으로 초기화해야할 경우

arange( ) 함수

- 파이썬 표준 함수인 range( )와 유사한 기능
- 0부터 함수 인자 값 -1까지의 값을 순차적으로 ndarray의 데이터값으로 변환해줌

```
sequence_array = np.arange(10)
print(sequence_array)
print(sequence_array.dtype, sequence_array.shape)
```

: default 함수 인자는 stop값이며, 0부터 stop값-1까지의 연속 숫자 값으로 구성된 1차원 ndarray생성

: 여기서는 stop값만 부여했으나 range와 유사하게 start값 부여 가능

zeros( ) 함수

- 함수 인자로 튜플 형태의 shape값을 입력→모든 값을 0으로 채운 해당 shape를 가진 ndarray를 반환

ones ( ) 함수

- 함수 인자로 튜플 형태의 shape값을 입력 → 1로 채운 해당 shape를 가진 ndarray를 반환

- 함수 인자로 dtype을 정해주지 않으면 default로 float64형으로 데이터 채움

```
zero_array = np.zeros((3,2), dtype='int32')
print(zero_array)
print(zero_array.dtype, zero_array.shape)

one_array= np.ones((3,2))
print(one_array)
print(one_array.dtype, one_array.shape) #float64
```

ndarray의 차원과 크기를 변경하는 reshape( )

- ndarray를 특정 차원 및 크기로 변환
- 변환을 원하는 크기를 함수인자로 부여

```
array1 = np.arange(10) # 0~9까지의 1차원
print('array1:\n',array1)

array2 = array1.reshape(2,5) #2 rows x 5 cols : 2차원
print('array2:\n',array2)

array3 = array1.reshape(5,2) # 5 rows x 2 cols : 2차원
print('array3:\n',array3)
```

▼ 지정된 사이즈로 변경이 불가능 하면 오류 발생

▼ ex) (10,)데이터를 (4,3) Shape 형태로 변경 불가능

```
array1.reshape(4,3) # 오류 발생
```

인자로 -1을 적용하는 경우

- -1을 인자로 사용하면 원래 ndarray와 호환되는 새로운 shape로 변환해줌

```
array1=np.arange(10)
print(array1)
array2 = array1.reshape(-1,5)
print('array2 shape:',array2.shape)
array3=array1.reshape(5,-1)
print('array3 shape:',array3.shape)
```

- array1 : 1차원 ndarray로 0~9까지 데이터를 가짐
- array2 : 로우 인자 -1, 칼럼 인자 5 → array1과 호환될 수 있는 2차원 ndarray로 변환 & 고정된 5개의 칼럼에 맞는 로우를 자동으로 생성해서 변환, 즉 2 X 5
- array3 : 10개의 1차원 데이터와 호환될 수 있는 고정된 5개의 로우에 맞는 칼럼은 2이므로 5 X 2의 2차원 ndarray로 변환

▼ 아까와 마찬가지로 호환될 수 없는 형태로 변환은 불가능

```
array1 = np.arange(10)
array4 = array1.reshape(-1,4)
```

: 10개의 1차원 데이터를 고정된 4개의 칼럼을 가진 로우로는 변경 불가능 → 에러

▼ -1 인자는 reshape(-1, 1)와 같은 형태로 자주 사용됨

- reshape(-1, 1)은 원본 ndarray가 어떤 형태라도 2차원이고, 여러 개의 로우와 반드시 1개의 칼럼을 가진 ndarray로 변환됨을 보장함
- 여러 개의 넘파이 ndarray는 stack이나 concat으로 결합할 때 각각의 ndarray의 형태를 통일해 유용하게 사용됨

```
array1 = np.arange(8)
array3d = array1.reshape((2,2,2))
print('array3d:\n',array3d.tolist())
#ndarray는 tolist()메서드를 이용해 리스트 자료형으로 변환가능

#3차원 ndarray를 2차원 ndarray로 변환
array5= array3d.reshape(-1,1)
print('array5:\n',array5.tolist())
print('array5 shape:',array5.shape)

#1차원 ndarray를 2차원 ndarray로 변환
```

```
array6 = array1.reshape(-1,1)
print('array6:\n',array6.tolist())
print('array6 shape:',array6.shape)
```

넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱(Indexing)

1. 특정한 데이터만 추출
2. 슬라이싱(Slicing): 연속된 인덱스상의 ndarray를 추출
3. 팬시 인덱싱(Fancy Indexing): 일정한 인덱싱 집합을 리스트 또는 ndarray 형태로 지정해 해당 위치에 있는 데이터의 ndarray로 반환
4. 불린 인덱싱(Boolean Indexing): 특정 조건에 해당하는지 여부인 True/False 값 인덱싱 집합을 기반으로 True에 해당하는 인덱스 위치에 있는 데이터의 ndarray를 반환

단일 값 추출

▼ ndarray 객체에 해당하는 위치의 인덱스 값을 [ ] 안에 입력

```
#1부터 9까지의 1차원 ndarray생성
array1 = np.arange(start=1, stop = 10)
print('array1: ',array1)
#index는 0부터 시작하므로 array1[2]는 3번째 index위치의 데이터 값을
value=array1[2]
print('value:',value)
print(type(value))
```

: array1[2]의 타입은 ndarray 내의 데이터 값을 의미

▼ 인덱스 -1은 맨 뒤의 데이터 값을 의미

```
print('맨 뒤의 값:',array1[-1],'맨 뒤에서 두 번째 값:',array1[-2
```

▼ 단일 인덱스를 이용해 ndarray 내의 데이터 값을 간단히 수정 가능

```
array1[0] = 9
array1[8] = 0
print('array1:',array1)
```

다차원 ndarray에서 단일 값 추출

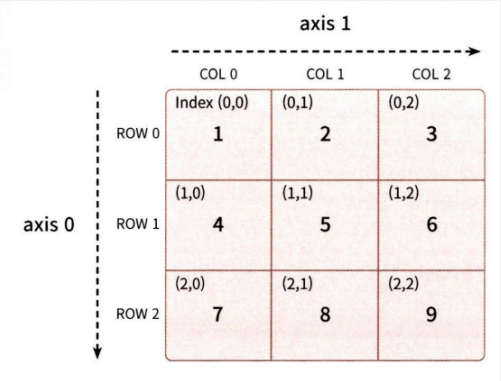
- 2차원의 경우 콤마로(,)로 분리된 로우와 칼럼의 위치의 인덱스를 통해 접근

```
array1d = np.arange(start=1, stop =10)
array2d = array1d.reshape(3,3)
print(array2d)

print('(row=0, col=0) index 가리키는 값:',array2d[0,0])
print('(row=0, col=1) index 가리키는 값:',array2d[0,1])
print('(row=1, col=0) index 가리키는 값:',array2d[1,0])
print('(row=2, col=2) index 가리키는 값:',array2d[2,2])
```

: 1차원을 2차원의 3 x 3으로 변환한 후 [row,col]을 이용해 2차원에서 데이터 추출

📌 **axis**



- **axis 0 : 로우 방향의 축**
- **axis 1 : 칼럼 방향의 축**

*넘파이 ndarray에서는 로우, 칼럼은 사용되지 않는 방식. axis가 정확한 표현

즉, [row=0, col=1] 인덱싱은 [axis0=0, axis1=1]이 정확한 표현임

*3차원의 경우 axis 0, axis 1, axis 2로 3개의 축을 가짐(행,열,높이).

*축 기반의 연산에서 axis가 생략되면 axis 0을 의미

슬라이싱

- ' : '기호를 이용해 연속한 데이터를 슬라이싱해서 추출
- 단일 데이터 값 추출을 제외하고 슬라이싱, 팬시 인덱싱, 불린 인덱싱으로 추출된 데이터 세트는 모두 ndarray타입
- ':'사이에 시작 인덱스와 종료인덱스 표시 : 시작 인덱스에서 종료 인덱스-1의 위치에 있는 데이터의 ndarray를 반환

```
array1 = np.arange(start=1,stop=10)
array3 = array1[0:3]
print(array3)
print(type(array3))
```

▼ 슬라이싱 기호인 ':'사이의 시작, 종료 인덱스는 생략 가능

1. ':'기호 앞에 시작 인덱스 생략 →맨 처음 인덱스인 0으로 간주
2. ':'기호 뒤에 종료 인덱스 생략 → 맨 마지막 인덱스로 간주
3. ':' 기호 앞/뒤에 시작/종료 인덱스 생략 → 맨 처음/마지막 인덱스로 간주

```
array1=np.arange(start=1, stop=10)
array4 = array1[:3]
print(array4)

array5 = array1[3:]
print(array5)

array6 = array1[:]
print(array6)
```

▼ 2차원 ndarray에서의 슬라이싱으로 데이터 접근

- 1차원과 유사하며 콤마(,)로 로우와 칼럼 인덱스를 지칭

```
array1d = np.arange(start=1,stop=10)
array2d = array1d.reshape(3,3)
print('array2d:\n',array2d)
```

```
print('array2d[0:2,0:2]\n',array2d[0:2,0:2])
print('array2d[1:3,0:3]\n',array2d[1:3,0:3])
print('array2d[1:3,:]\n',array2d[1:3,:])
print('array2d[:,:]\n',array2d[:,:])
print('array2d[:2,1:]\n',array2d[:2,1:])
print('array2d[:2,0]\n',array2d[:2,0]) #로우 축에만 슬라이싱,
```

- 2차원 ndarray에서 뒤에 오는 인덱스를 없애면 1차원 ndarray를 반환

```
print(array2d[0])
print(array2d[1])
print('array2d[0] shape:',array2d[0].shape,'array2d[1] sha
```

: array2d[0] → 로우 축(axis 0)의 첫 번째 로우 ndarray를 반환하게 됨.

: 반환되는 ndarray는 1차원

*3차원 ndarray에서 뒤에 오는 인덱스를 없애면 2차원 ndarray를 반환

팬시 인덱싱

리스트나 ndarray로 인덱스 집합을 지정하면 해당 위치의 인덱스에 해당하는 ndarray를 반환하는 인덱싱 방식

```
array1d = np.arange(start=1, stop = 10)
array2d = array1d.reshape(3,3)

array3 = array2d[[0,1],2]
print('array2d[[0,1],2] => ',array3.tolist())

array4 = array2d[[0,1],0:2]
print('arange2d[[0,1], 0:2] => ', array4.tolist())

array5 = array2d[[0,1]]
print('array2d[[0,1]] => ', array5.tolist())
```

- array3 : 로우 축에 팬시 인덱싱인 [0, 1], 칼럼 축에 단일 값 인덱싱 2 →(row, col)인덱스가 (0, 2), (1, 2)로 적용되어 [3, 6]반환

- array4 : ((0, 0), (0, 1)) , ((1, 0), (1, 1)) 인덱싱 적용
- array5 : ((0, :), (1, :))인덱싱 적용

불린 인덱싱

- 조건 필터링과 검색을 동시에 할 수 있음(자주 사용됨).
- 5보다 큰 데이터만 추출하기? → 불린 인덱싱을 통해 간단히 구현 가능
- 인덱스를 지정하는 [ ] 내에 조건문을 그대로 기재하면 됨

```
array1d = np.arange(start=1, stop=10)
#[ ]안에 array1d > 5 Boolean Indexing 을 적용
array3 = array1d[array1d > 5]
print('array1d > 5 불린 인덱싱 결과 값 : ', array3)
```

▼ 어떻게 조건 필터링이 간단히 일어나는가?

넘파이 ndarray 객체에 조건식을 할당했을 때 :

```
array1d > 5
#array([False, False, False, False, False,  True,  True,
```

: 5보다 큰 데이터의 위치에는 True값이, 그렇지 않은 경우 False값이 반환

- [ ]내에 입력하면 False값은 무시하고 True값의 위치 인덱스 값으로 자동 변환해 해당 인덱스 위치의 데이터만 반환

```
boolean_indexes = np.array([False,False,False,False,False,
array3=array1d[boolean_indexes]
print('불린 인덱스로 필터링 결과 :',array3)
```

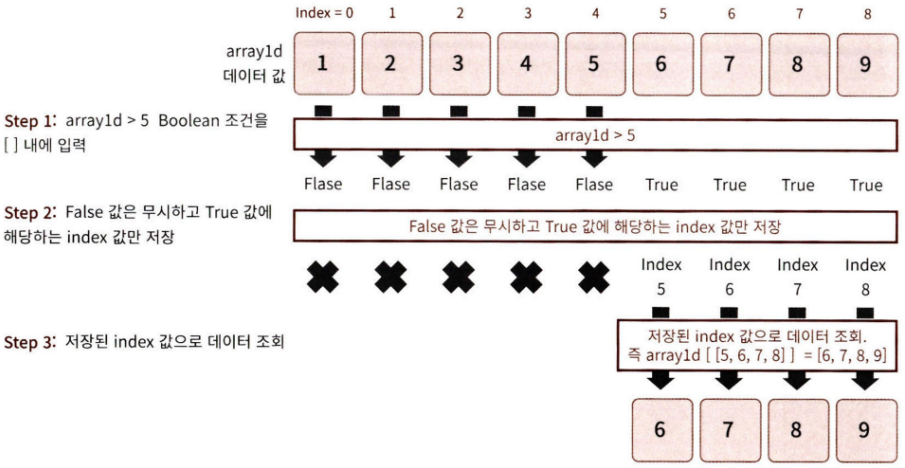
: 불린 ndarray를 만들고 이를 array1d[ ]내에 인덱스로 입력하면 동일한 데이터 세트가 반환됨을 알 수 있음

```
indexes = np.array([5,6,7,8])
array4 = array1d[indexes]
```

```
print('일반 인덱스로 필터링 결과:',array4)
```

: 이것과 같이 직접 인덱스 집합을 만들어 대입한 것과 위와 동일

▼ 불린 인덱싱이 동작하는 단계



행렬의 정렬 - sort( )와 argsort( )

- np.sort( )와 ndarray.sort( ) : 행렬을 정렬
- argsort( ) : 정렬된 행렬의 인덱스를 반환

행렬 정렬

- np.sort( ) : 원 행렬은 그대로 유지한 채, 원 행렬의 정렬된 행렬 반환
- ndarray.sort( ) : 원 행렬 자체를 정렬한 형태로 변환, 반환 값은 None

```
org_array = np.array([3,1,9,5])
print('원본 행렬:',org_array)
#np.sort()로 정렬 : 원본 행렬 변경X, 정렬된 형태로 반환
sort_array1 = np.sort(org_array)
print('np.sort()호출 후 반환된 정렬 행렬:',sort_array1)
print('np.sort()호출 후 원본 행렬:',org_array)
#ndarray.sort()로 정렬 : 원본 행렬 자체를 정렬한 값으로 변환
```

```
sort_array2 = org_array.sort()
print('org_array.sort()호출 후 반환된 행렬:',sort_array2)
print('org_array.sort()호출 후 원본 행렬:',org_array)
```

- 기본적으로 오름차순으로 행렬 내 원소 정렬

▼ 내림차순 정렬 : [::-1]

```
sort_array1_desc = np.sort(org_array)[::-1]
print('내림차순으로 정렬:',sort_array1_desc)
```

▼ 행렬이 2차원 이상인 경우 : axis축 값 설정을 통해 로우 방향, 또는 칼럼 방향으로 정렬

```
array2d = np.array([[8,12],
                    [7,1]])

sort_array2d_axis0 = np.sort(array2d, axis = 0)
print('로우 방향으로 정렬:\n',sort_array2d_axis0)

sort_array2d_axis1 = np.sort(array2d, axis = 1)
print('칼럼 방향으로 정렬:\n',sort_array2d_axis1)
```

정렬된 행렬의 인덱스를 반환하기

np.argsort( ) : 정렬 행렬의 원본 행렬 인덱스를 ndarray형으로 반환

```
org_array = np.array([3,1,9,5])
sort_indices = np.argsort(org_array)
print(type(sort_indices))
print('행렬 정렬 시 원본 행렬의 인덱스:',sort_indices)
```

▼ 내림차순 정렬 시 : np.argsort()[::-1]

```
org_array = np.array([3,1,9,5])
sort_indices_desc = np.argsort(org_array)[::-1]
print('행렬 내림차순 정렬 시 원본 행렬의 인덱스:',sort_indices_des
```

▼ 넘파이에서 활용도가 높음

- ndarray는 메타 데이터를 가질 수 없음 ⇒ 실제 값과 그 값이 뜻하는 메타 데이터를 별도의 ndarray로 각각 가져야 함

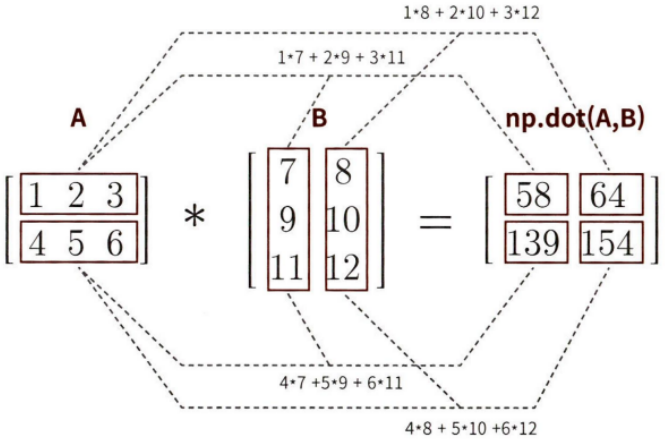
```
#argsort()가 유용하게 쓰이는 예제
name_array = np.array(['John', 'Mike','Sarah','Kate','Samu
score_array = np.array([78,95,84,98,88])

sort_indices_asc = np.argsort(score_array)
print('성적 오름차순 정렬 시 score_array의 인덱스:',sort_indices
print('성적 오름차순으로 name_array의 이름 출력:',name_array[sor
```

선형대수 연산 - 행렬 내적과 전치 행렬 구하기

행렬 내적(행렬 곱)

- 두 행렬 A와 B의 내적은 np.dot( )을 이용해 계산 가능
- 왼쪽 행렬의 로우(행)와 오른쪽 행렬의 칼럼(열)의 원소들을 순차적으로 곱한 뒤 그 결과를 더한 값




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

```
dot_product = np.dot(A,B)
print('행렬 내적 결과:\n',dot_product)
```

전치 행렬

- 원 행렬에서 행과 열 위치를 교환한 원소로 구성한 행렬
- 1행 2열의 원소 → 2행 1열의 원소 / 2행 1열의 원소 → 1행 2열의 원소
- 넘파이의 transpose( )를 이용해서 구할 수 있음

```
A = np.array([[1,2],
              [3,4]])
transpose_mat = np.transpose(A)
print('A의 전치 행렬:\n',transpose_mat)
```

04. 데이터 핸들링 - 판다스

- 파이썬에서 데이터 처리를 위해 존재하는 가장 인기 있는 라이브러리
- 일반적으로 대부분의 데이터 세트는 2차원 데이터 → Row X Column

  : 이해하기 쉬운 데이터 구조, 효과적으로 데이터를 담을 수 있는 구조
- 파이썬의 리스트, 컬렉션, 넘파이 등의 내부 데이터 뿐만 아니라 CSV 등의 파일을 쉽게 DataFrame으로 변경해 데이터의 가공/분석을 편리하게 수행하게 함
- 핵심 객체 : DataFrame → 여러 개의 행과 열로 이뤄진 2차원 데이터를 담는 구조체
- 다른 중요 객체인 Index와 Series
  - Index : 개별 데이터를 고유하게 식별하는 Key값
    - Series와 DataFrame 모두 Index를 Key값으로 가짐
  - Series: 칼럼이 하나뿐인 데이터 구조체(DataFrame은 칼럼이 여러개)

판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API

pandas를 pd로 에일리어스(alias)해 임포트하는 것이 관례

```
import pandas as pd
```

- read_csv( ) : CSV(칼럼을 ‘,’로 구분한 파일 포맷) 파일 포맷 변환을 위한 API
- CSV뿐만 아니라 어떤 필드 구분 문자 기반의 파일 포맷도 DataFrame으로 변환가능

  : read_csv( ) 인자인 sep에 해당 구분 문자를 입력하기
- read_csv( ) 함수에서 가장 중요한 인자는 filepath

```
titanic_df = pd.read_csv('/content/drive/MyDrive/Euron7_ML/da
titanic_df.head(3)
```

- pd.read_csv( )는 호출시 파일명 인자로 들어온 파일을 로딩해 DataFrame 객체로 반환

DataFrame살펴보기

- 데이터 파일의 첫 번째 줄에 있던 칼럼 문자열이 DataFrame의 칼럼으로 할당
- 콤마로 분리된 데이터 값들이 해당 칼럼에 맞게 할당
- 맨 왼쪽에 로우 순으로 순차적인 표시는?

  : 판다스의 Index 객체 값!(지금은 PK와 유사한 역할이라고 이해하자.)
- DataFrame.head()는 DataFrame의 맨 앞에 있는 N개의 로우 반환(Default는 5개)
- shape변수 이용 : DataFrame의 행과 열 크기 알아보기

```
print('DataFrame의 크기:',titanic_df.shape) #row:891, col:12
```

▼ 데이터 뿐만 아니라 칼럼의 타입, Null 데이터 개수, 데이터 분포도 등 메타 데이터 등도 조회 가능

: info( )와 describe( )

▼ info ( )

```
titanic_df.info()
```

- 데이터 건수, 데이터 타입, Null 건수를 알 수 있음

▼ describe( )

```
titanic_df.describe()
```

- 칼럼별 숫자형 데이터값의 n-percentile분포도, 평균값, 최댓값, 최솟값을 나타냄
- 오직 숫자형 칼럼만 조사하며 자동으로 object타입의 칼럼은 출력 제외
- 개략적인 수준의 분포도를 확인하는 경우 유용

▼ 상세 설명

- count : Not null인 데이터 건수
- mean : 전체 데이터의 평균값
- std : 표준편차
- min : 최솟값, max : 최댓값
- 25% : 25 percentile 값 ~~
- 숫자 칼럼이 숫자형 카테고리 칼럼인지를 판단하게 도움

- DataFrame의 [ ] 연산자 내부에 칼럼명을 입력하면 해당 칼럼에 해당하는 Series 객체 반환
- Series는 Index와 단 하나의 칼럼으로 구성된 데이터 세트
- value_counts( )
  - 해당 칼럼값의 유형과 건수를 확인가능
  - 지정된 칼럼의 데이터값 건수를 반환
  - 많은 건수 순서로 정렬되어 반환
  - Series 객체에서만 정의, DataFrame은 갖고 있지 않음
  - 반환된 Series 객체의 값을 보면 맨 왼쪽이 인덱스, 오른쪽이 데이터값

    : 단순 순차 값이 아님 ! ⇒ 인덱스는 **고유성**이 보장된다면 의미 있는 데이터값도 할당 가능
    - 인덱스는 만들어진 후에도 변경 가능
    - 숫자형 뿐만 아니라 문자열도 가능

DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호변환

DataFrame과 넘파이 ndarray 상호 간의 변환은 빈번히 발생

넘파이 ndarray, 리스트, 딕셔너리를 DataFrame으로 변환하기

- DataFrame은 리스트와 넘파이 ndarray와 다르게 컬럼명을 가짐

  ⇒ DataFrame으로 변환 시 이 칼럼명을 지정해야함
- 판다스 DataFrame 객체의 생성 인자 data는 리스트나 딕셔너리 또는 넘파이 ndarray 입력을 받고,
- 생성 인자 columns는 칼럼명 리스트를 입력 받아서 DataFrame생성
- 2차원 이하의 데이터들만 DataFrame으로 변환가능

```
col_name1 = ['col1']
list1=[1,2,3]
array1 = np.array(list1)
print('array1 shape: ',array1.shape)
#리스트를 이용해 DataFrame 생성
df_list1 = pd.DataFrame(list1, columns = col_name1)
print('1차원 list로 만든 DataFrame:\n',df_list1)
#넘파이 ndarray를 이용해 DataFrame 생성
df_array1 = pd.DataFrame(array1, columns = col_name1)
print('1차원 ndarray로 만든 DataFrame:\n',df_array1)
```

: 1차원 형태의 데이터를 기반으로 DataFrame을 생성 → 칼럼명이 한 개만 필요!

2차원 형태의 데이터를 기반으로 DataFrame생성해보기(2*3)

```
#3개의 칼럼명이 필요
col_name2=['col1','col2','col3']

#2행*3열 형태의 리스트와 ndarray 생성한 뒤 이를 DataFrame으로 반환
```

```
list2 = [[1,2,3],
         [11,12,13]]
array2=np.array(list2)
print('array2 shape:',array2.shape)
df_list2 = pd.DataFrame(list2,columns=col_name2)
print('2차원 리스트로 만든 DataFrame:\n',df_list2)
df_array2 = pd.DataFrame(array2,columns=col_name2)
print('2차원 ndarray로 만든 DataFrame:\n',df_array2)
```

딕셔너리를 DataFrame으로 변환

- 딕셔너리의 키(Key) → 칼럼명
- 딕셔너리의 값(Value) → 칼럼 데이터

```
#key는 문자열 칼럼명으로 매핑, vaule는 리스트 형(또는 ndarray) 칼럼 데
dict = {'col1':[1,11],'col2':[2,22],'col3':[3,33]}
df_dict = pd.DataFrame(dict)
print('딕셔너리로 만든 DataFrame:\n',df_dict)
```

DataFrame을 넘파이 ndarray, 리스트, 딕셔너리로 변환하기

DataFrame → 넘파이 ndarray

: DataFrame 객체의 values를 이용

```
#DataFrame을 ndarray로 변환
array3 = df_dict.values
print('df_dict.vaules 타입:',type(array3),'df_dict.vaules shap
print(array3)
```

DataFrame → 리스트와 딕셔너리

- 리스트로의 변환은 values로 얻은 ndarray에 tolist( )호출
- 딕셔너리로의 변환은 DataFrame 객체의 to_dict( ) 메서드 호출, 인자로 'list'를 입력하면 딕셔너리의 값이 리스트형으로 반환됨

```
#DataFrame을 리스트로 변환
list3 = df_dict.values.tolist()
print('df_dict.values.tolist()타입: ',type(list3))
print(list3)
#DataFrame을 딕셔너리로 변환
dict3 = df_dict.to_dict('list')
print('\n d_dict.to_dict()타입:',type(dict3))
print(dict3)
```

DataFrame의 칼럼 데이터 세트 생성과 수정

- [ ]연산자를 이용

```
titanic_df['Age_0']=0
titanic_df.head(3)
```

: 새로운 칼럼명 'Age_0'으로 모든 데이터 값이 0으로 할당된 Series가 기존 DataFrame에 추가됨

기존 칼럼 Series의 데이터를 이용해 새로운 칼럼 Series 만들기

```
titanic_df['Age_by_10']=titanic_df['Age']*10
titanic_df['Family_No']=titanic_df['SibSp']+titanic_df['Parch
titanic_df.head(3)
```

기존 칼럼 값 일괄적으로 업데이트하기

: 업데이트를 원하는 칼럼 Series를 DataFrame[ ]내에 칼럼 명으로 입력한 뒤, 값 할당

```
#Age_by_10칼럼 값을 일괄적으로 기존값+100으로 업데이트
titanic_df['Age_by_10']=titanic_df['Age_by_10']+100
titanic_df.head(3)
```

DataFrame 데이터 삭제

- **drop( )메서드** 이용
  - 원형 : DataFrame.drop(**labels=None**, **axis=0**, index=None, columns=None, level=None, **inplace=False**, errors='raise')
  - axis = 1 : 칼럼을 드롭
    - labels에 원하는 칼럼 명 입력
  - axis = 0 : 특정 로우 드롭
    - DataFrame의 특정 로우를 가리키는 것인 인덱스

    ⇒ labels에 오는 값을 인덱스로 간주

```
titanic_drop_df = titanic_df.drop('Age_0',axis=1) #Age_0 칼럼
titanic_drop_df.head(3)
```

그러나 원본 타이타닉 DataFrame을 다시 확인해보면 'Age_0'칼럼이 여전히 존재

⇒ inplace = False라고 설정했기 때문 ! (디폴트 값이 False)

: 자기 자신의 DataFrame의 데이터는 삭제 X, 삭제된 결과 DataFrame을 반환

```
#세 개의 칼람 삭제
drop_result=titanic_df.drop(['Age_0','Age_by_10','Family_No
print('inplace=True로 drop후 반환된 값:',drop_result)
titanic_df.head(3)
```

*inplace = True로 설정한 채로 반환 값을 다시 자신의 DataFrame객체로 할당하면 안 됨

⇒ 객체 변수를 아예 None으로 만들어 버림

axis = 0

```
#ㅏindex 0,1,2로우 삭제
pd.set_option('display.width',100)
pd.set_option('display.max_colwidth',15)
print('### before axis 0 drop ###')
print(titanic_df.head(3))
titanic_df.drop([0,1,2],axis=0,inplace=True)

print('### after axis 0 drop ###')
print(titanic_df.head(3))
```

Index 객체

- RDBMS의 PK와 유사하게 DataFrame, Series의 레코드를 고유하게 식별하는 객체
- DataFrame.index 또는 Series.index속성을 통해 Index 객체 추출

```
#Index 객체 추출
indexes = titanic_df.index
#Index 객체를 실제 값 array로 변환
print('Index객체 array 값:\n',indexes.values)
```

: Index 객체는 식별성 데이터를 1차원 array로 갖고 있음

- ndarray와 유사하게 단일 값 반환 및 슬라이싱도 가능

```
print(type(indexes.values))
print(indexes.values.shape)
print(indexes[:5].values)
print(indexes.values[:5])
print(indexes[6])
```

- DataFrame및Series의 Index 객체를 함부로 변경 불가

```
indexes[0]=5 #오류 발생
```

- Index는 오직 식별용으로만 사용

  : Series 객체는 Index객체를 포함하지만 Series 객체 연산 함수를 적용할 때 Index는 연산에서 제외

- reset_index( )

  : 새롭게 인덱스를 연속 숫자 형으로 할당하며 기존 인덱스는 'index'라는 새로운 칼럼 명으로 추가

  ```
  titanic_reset_df = titanic_df.reset_index(inplace=False)
  titanic_reset_df.head(3)
  ```

  - 인덱스가 연속된 int숫자형 데이터가 아닐 경우 주로 사용

  ```
  print('### before reset_index ###')
  value_counts = titanic_df['Pclass'].value_counts()
  print(value_counts)
  print('value_counts 객체 변수 타입:',type(value_counts))
  new_value_counts=value_counts.reset_index(inplace=False)
  print(' ### After reset_index ###')
  print(new_value_counts)
  print('new_value_counts 객체 변수 타입:',type(new_value_count
  ```

  *Series에 reset_index()를 적용하면 Series가 아닌 DataFrame으로 반환되니 유의

  - reset_index( )의 파라미터 중 drop = True로 설정하면 기존 인덱스는 새로운 칼럼으로 추가되지 않고 삭제(drop) 됨 → 새로운 칼럼이 추가되지 않으므로 그대로 Series유지

데이터 셀렉션 및 필터링

DataFrame의 [ ] 연산자

- DataFrame 뒤에 있는 [ ]는 칼럼만 지정할 수 있는 '칼럼 지정 연산자'로 우선 이해하자 !!

```
print('단일 칼럼 데이터 추출:\n',titanic_df['Pclass'].head(3))
print('\n여러 칼럼의 데이터 추출:\n',titanic_df[['Survived','Pclas
print('[]안에 숫자 index는 KeyError 오류 발생:\n',titanic_df[0])
```

: [ ]에는 칼럼명을 지정해야하는 0은 칼럼 명이 아니기 때문에 오류 발생

- 판다스의 인덱스 형태로 변환 가능한 표현식은 [ ]내에 입력 가능

```
titanic_df[0:2]
```

: 처음 2개 데이터를 추출하고자 슬라이싱을 이용

- 불린 인덱싱 표현도 가능

```
titanic_df[titanic_df['Pclass']==3].head(3)
```

: Pclass칼럼 값이 3인 데이터 3개 추출

📌혼돈 방지를 위한 가이드

1. DataFrame 바로 뒤의 [ ]연산자는 넘파이나 Series의 [ ]와 다름
2. DataFrame 바로 뒤의 [ ] 내 입력 값은 칼럼명(칼럼의 리스트)을 지정해 칼럼 지정 연산에 사용하거나 불린 인덱스 용도로만 사용
3. DataFrame[0:2]와 같은 슬라이싱 연산으로 데이터를 추출하는 방법은 지양

DataFrame ix[ ] 연산자

- 원하는 위치의 데이터 추출 가능
- ix[0, 'Pclass'] → 행 위치 지정으로 인덱스 값 0, 열 위치 지정으로 칼럼 명인 'Pclass' : 칼럼 명칭(label) 기반 인덱싱
- ix[0, 2] → 열 위치 지정은 칼럼 명 뿐만 아니라 칼럼의 위치 값 지정도 가능 : 칼럼 위치(position)기반 인덱싱

- 두 가지 방식을 제공하며 코드가 혼돈을 주거나 가독성이 떨어져서 ix[]는 현재 판다스에서 사라짐
- 이를 대신할 칼럼 명칭 기반 인덱싱 연자인 **loc [ ]**, 칼럼 위치 기반 인덱싱 연산자인 **iloc[]**
- ndarray의 [ ]연산자와 동일하게 단일 지정, 슬라이싱, 불린 인덱싱, 팬시 인덱싱 모두 가능

명칭 기반 인덱싱과 위치 기반 인덱싱의 구분

- 용어 정리
  - 명칭 기반 인덱싱 : 칼럼 명을 기반으로 위치 지정
  - 위치 기반 인덱싱 : 0을 출발점으로 하는 가로축, 세로축 좌표 기반의 행과 열 위치를 기반으로 데이터를 지정 → 행,열 값으로 정수 입력

DataFrame의 인덱스 값은 명칭 기반 인덱싱이라고 간주

DataFrame iloc[ ] 연산자

- 위치 기반 인덱싱만 허용 ⇒ 행과 열 값으로 integer 또는 integer 형의 슬라이싱, 팬시 리시트 값을 입력해줘야함

```
data_df.iloc[0,0]
```

- iloc[ ] 에 위치 인덱싱이 아닌 명칭을 입력하면 오류 발생
- 문자열 인덱스를 행 위치에 입력해도 오류 발생
- 슬라이싱과 팬시 인덱싱은 제공하나 명확한 위치 기반 인덱싱이 사용 되어야 하는 제약으로 인해 불린 인덱싱 제공 X

DataFrame loc [ ] 연산자

- 명칭 기반으로 데이터 추출 ⇒ 행 위치에 DataFrame index값을, 열 위치에 칼럼 명 입력

```
data_df.loc['one','Name']
```

- 명칭 기반이라고 index가 무조건 문자열 입력은 아님

```
data_df_reset.loc[1,'Name']
```

  - data_df_reset.loc[0,'Name'] 실행 시 : 인덱스 값이 0인 행이 없으므로류

- 슬라이싱 기호 유의점
  - loc [ ]에서 슬라이싱 기호 적용 → 종료 값 포함(ix[]에도 동일)
  - 명칭이 숫자 형이 아닐 수도 있어서 -1을 할 수 없음

```
print('위치 기반 iloc slicing\n',data_df.iloc[0:1,0],'\n')
print('명칭 기반 loc slicing\n',data_df.loc['one':'two','Nam
```

  : loc [ ]은 2개의 행 반환, iloc [ ]은 0번째 행 위치에 해당하는 1개의 행 반환

불린 인덱싱

- 매우 편리한 데이터 필터링 방식
- [ ], ix[ ], loc[ ]에서 공통으로 지원
- but, iloc [ ]는 정수형 값이 아닌 불린 값에 대해서는 지원하지 않기 때문에 불린 인덱싱 지원되지 않음

```
titanic_boolean = titanic_df[titanic_df['Age']>60]
print(type(titanic_boolean))
titanic_boolean
```

: 반환된 객체의 차입은 DataFrame타입 → 원하는 칼럼 명만 별도로 추출 가능

```
#60세 이상인 승객의 나이와 이름만 추출
titanic_df[titanic_df['Age']>60][['Name','Age']].head(3)
```

```
#loc[]으로동일하게 적용. 단, ['Name','Age']는 칼럼 위치에
titanic_df.loc[titanic_df['Age']>60,['Name','Age']].head(3)
```

- 여러 개 복합 조건도 가능
  1. and → &

2. or → |
3. Not → ~

```
#나이 60세 이상, 선실 등급은 1등급, 성별은 여성
titanic_df[ (titanic_df['Age']>60) & (titanic_df['Pclass']=
           (titanic_df['Sex']=='female')]
```

: 개별 조건은 ( )으로 묶고, 복합 조건 연산자를 사용

- 개별 조건을 변수에 할당하고 이들 변수를 결합해서 불린 인덱싱 수행 가능

```
cond1 = titanic_df['Age'] > 60
cond2 = titanic_df['Pclass'] == 1
cond3 = titanic_df['Sex'] =='female'
titanic_df[cond1&cond2&cond3]
```

정렬, Aggregation 함수, GroupBy 적용

DataFrame, Series의 정렬 - sort_values()

- RBDMS의 order by키워드와 유사
- 주요 입력 파라미터: by, ascending, inplace
  - by: 특정 칼럼을 입력하면 해당 칼럼으로 정렬
  - ascending=True → 오름차순(False로 하면 내림차순)
  - inplace=False → sort_values()를 호출한 DataFrame은 그대로 유지하며 정렬된 DataFrame을 결과로 반환 (True로 설정하면 호출한 DataFrame의 정렬 결과를 그대로 적용)

```
#Name 칼럼으로 오름차순 정렬해 반환
titanic_sorted = titanic_df.sort_values(by=['Name'])
titanic_sorted.head(3)
```

```
#Pclass와 Name을 내림차순으로 정렬한 결과 반환
titanic_sorted=titanic_df.sort_values(by=['Pclass','Name'],as
titanic_sorted.head(3)
```

Aggregation 함수 적용

- RBDMS의 aggregation함수 적용과 유사
- DataFrame의 경우 DataFrame에서 바로 호출하면 모든 칼럼에 적용함
- 특정 칼럼에 함수를 적용하기 위해서, DataFrame에 대상 칼럼만 추출해서 적용

```
titanic_df[['Age','Fare']].mean()
```

groupby( ) 적용

- RBDMS의 groupby 키워드와 유사하지만 다른 면이 있으므로 주의 !
- 입력 파라미터 by에 칼럼을 입력하면 대상 칼럼으로 groupby됨
- DataFrameGroupBy라는 또 다른 형태의 DataFrame을 반환

```
titanic_groupby =titanic_df.groupby(by='Pclass')
print(type(titanic_groupby))
```

- SQL의 groupby와 다르게 DataFrame에 groupby()호출해 반환된 결과에 aggregation함수를 호출하면 groupby()대상 컬럼을 제외한 모든 칼럼에 해당 aggregation함수 적용

```
titanic_groupby=titanic_df.groupby('Pclass').count()
titanic_groupby #Pclass를 제외하고 count됨
```

- 특정 칼럼만 aggregation함수를 적용하려면?

  : groupby()로 반환된 DataFrameGroupBy객체에 해당 컬럼을 필터링 한 뒤 aggregation함수 적용

  ```
  #titanic_df.groupby('Pclass')로 반환된 DataFrameGroupBy객체에
  titanic_groupby=titanic_df.groupby('Pclass')[['PassengerId
  ```

```
titanic_groupby
```

- 서로 다른 aggregation함수 적용할 시

  : 여러 개의 함수명을 DataFrameGroupBy 객체의 agg() 내에 인자로 입력

```
titanic_df.groupby('Pclass')['Age'].agg([max,min])
```

- 여러 개의 칼럼이 서로 다른 aggregation함수를 groupby에서 호출할 시

  : agg( )내에 입력 값으로 딕셔너리 형태로 aggregation이 적용될 칼럼들과 함수 입력

```
agg_format = {'Age':'max','SibSp':'sum','Fare':'mean'}
titanic_df.groupby('Pclass').agg(agg_format)
```

결손 데이터 처리하기

- 판다스는 결손 데이터를 처리하는 편리한 API 제공
- 결손 데이터: 칼럼에 값이 없는, 즉 NULL인 경우를 의미하며, 이를 넘파이의 NaN으로 표시
- 머신러닝에서 NaN값 처리 X → 다른 값으로 대체 필요
- NaN값은 평균,총합 등의 함수 연산 시 제외 (100개 데이터 중 10개가 NaN이면 이 칼럼의 평균 값은 90개 데이터의 평균)
- NaN 여부 확인하는 API : isna( )
- NaN 값을 다른 값으로 대체하는 API : fillna( )

isna( )로 결손 데이터 여부 확인

- 모든 칼럼의 값이 NaN인지 아닌지를 Truesk False로 알려줌

```
titanic_df.isna().head(3)
```

- sum()함수를 추가해 구할 수 있음

- True는 내부적으로 숫자 1로, False는 숫자 0으로 반환되어 결손 데이터의 개수를 구할 수 있음

```
titanic_df.isna().sum()
```

fillna( )로 결손 데이터 대체하기

타이타닉 데이터 세트의 'Cabin'칼럼의 NaN값을 'C000'으로 대체

```
titanic_df['Cabin'] = titanic_df['Cabin'].fillna('C000')
titanic_df.head(3)
```

*주의!: fillna()를 이용해 반환 값을 다시 받거나 inplace=True 파라미터를 fillna()에 추가해야 실제 데이터 세트 값이 변경됨

'Age' 칼럼의 NaN 값을 평균 나이로, 'Embarked'칼럼의 NaN값을 'S'로 대체해 모든 결손 데이터를 처리

```
titanic_df['Age']=titanic_df['Age'].fillna(titanic_df['Age'].
titanic_df['Embarked']=titanic_df['Embarked'].fillna('S')
titanic_df.isna().sum()
```

apply lambda 식으로 데이터 가공

- 판다스는 apply 함수에 lambda 식을 결합해 DataFrame이나 Series의 레코드별로 데이터를 가공하는 기능을 제공
- 복잡한 데이터 가공이 필요한 경우

lambda 식: 파이썬에서 함수형 프로그래밍을 지원하게 위해 짐

```
lambda_square = lambda x : x**2
print('3의 제곱은:',lambda_square(3))
```

: ':'의 왼쪽에 있는 x는 입력인자, 오른쪽은 입력 인자의 계산식. 계산식→반환값

```
#여러 개의 값을 입력 인자로 사용할 경우:map()함수를 결합
a=[1,2,3]
squares = map(lambda x : x**2, a)
list(squares)
```

lamda에서 if else 사용 시 주의사항

- if 절의 경우 if 식보다 반환값을 먼저 기술해야함
- if, else는 지원 BUT if, else if, else와 같이 else if 지원 X
  - else if 이용을 위해서는: else절을 ( )로 내포해 ( ) 내에서 다시 if else 적용해 사용

switch case의 경우

- 별도의 함수를 만들자

```
#나이에 따라 세분화된 분류를 수행하는 함수 생성
def get_category(age):
  cat =''
  if age<= 5: cat = 'Baby'
  elif age <= 12: cat = 'Child'
  elif age <= 18 : cat = 'Teenager'
  elif age <= 25: cat = 'Student'
  elif age <= 35: cat = 'Young Adult'
  elif age <=60 : cat = 'Adult'
  else : cat = 'Elderly'

  return cat

#lambda 식에 위에서 생성한 get_category()함수를 반환값으로 지정
#get_category(X)를 입력값으로 'Age' 칼럼 값을 받아서 해당하는 cat반환
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x : ge
titanic_df[['Age','Age_cat']].head()
```