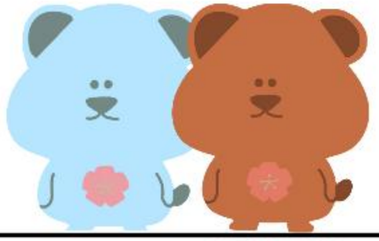
# 01. 파이썬 기반의 머신러닝과 생태계 이해

1팀 박혜린, 엄지민, 조승연

EWHA EURON

# 목차

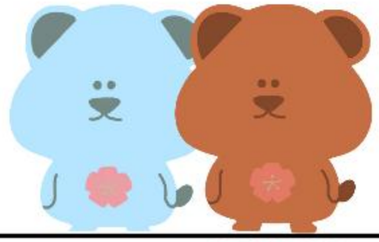




EWHA EURON

# 목차





EWHA EURON

# 01. 넘파이

EWHA EURON

# #1-1 넘파이 소개

## #1 파이썬 주요 패키지 중 하나

머신러닝 패키지 : 사이킷런

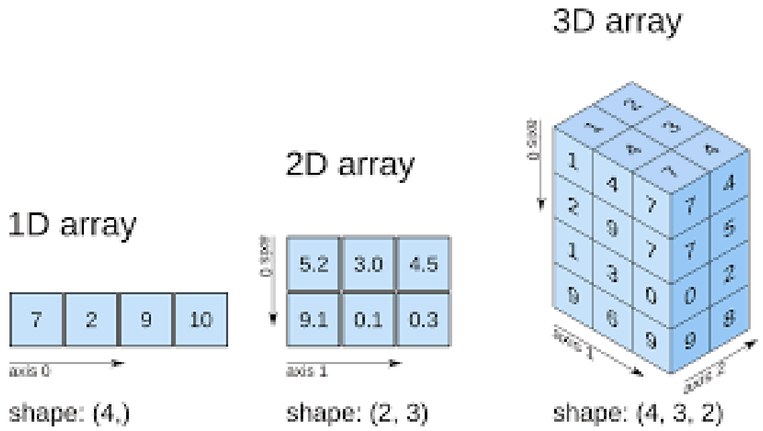
행렬/선형대수/통계 패키지 넘파이

데이터 핸들링 : 판다스

시각화 : 맷플롯립

등등…

## #2 배열 기반의 연산, 다양한 데이터 핸들링 (가공, 변환, 함수 적용 등은 판다스가 더 편리!)




기능 모음집
원하는 시점에 불러서 씀

큰 골격 제공
내가 코드를 쓰긴 해도 실행 흐름을 프레임워크가 주도

**+ 패키지란?**

프로그래밍 언어에서 머신러닝을 하기 위해 사용할 수 있는 **라이브러리나 프레임워크**
= 개발자가 바로 가져다 쓸 수 있는 라이브러리 모음

C / C++
-필요 기능 직접 구현해야 하거나
- 쓸 수 있는 패키지 적음

**파이썬**
- 개발자가 일일이 처음부터 코드를 짜지 않아도, 이미 만들어진 기능을 가져다 쓸 수 있게 해주는 도구 多
- 넘파이 – 수학 연산, 배열 계싼
  맷플롯립 – 시각화
  ……

EWHA EURON

# #1-2 넘파이 ndarray

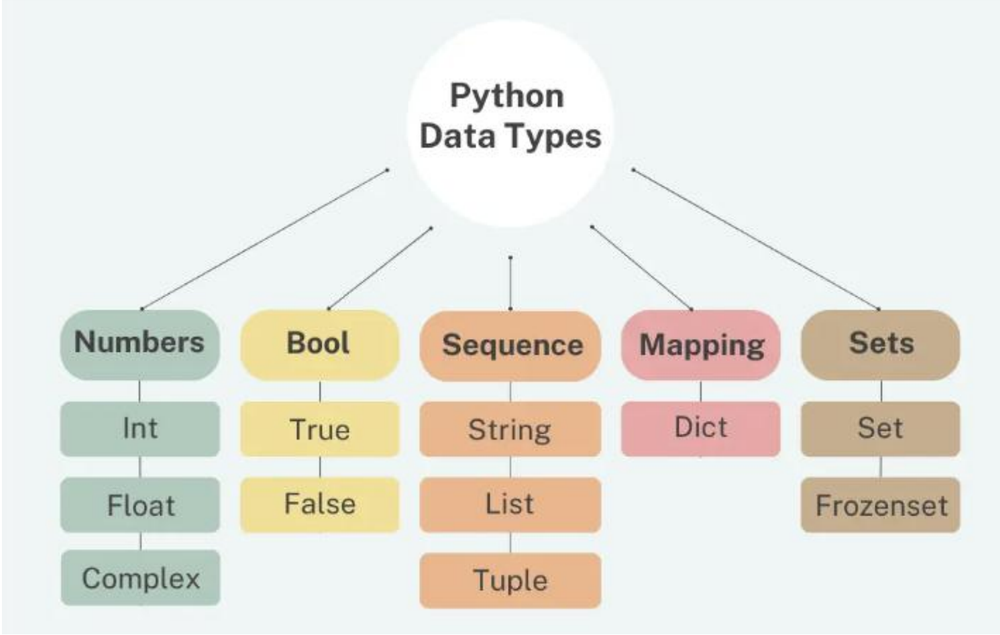
## #1 넘파이 모듈 임포트

```
Import numpy as np
```

약어로 모듈 표현 편리!

## #2 넘파이 기반 데이터 타입 : ndarray

: 다차원 배열 생성, 연산 수행



# #1-2 넘파이 ndarray

**#3 넘파이 array() 함수** : 인자 입력 -> ndarray로 변환

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

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

**#4 shape 변수** : ndarray의 크기를 튜플 형태로 반환

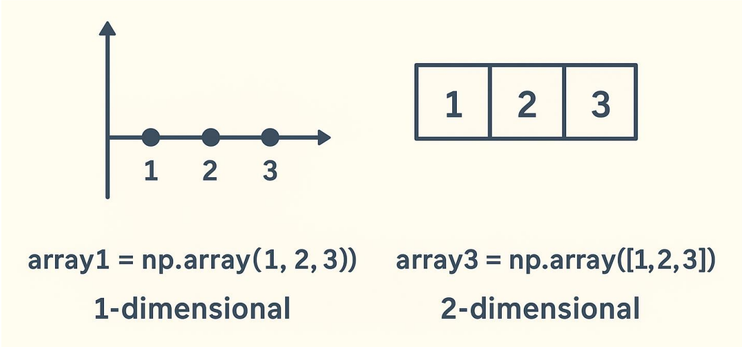
```
print(array1.shape)
```
```
(3, )
```

```
print(array2.shape)
```
```
(1, 3)
```

**#5 ndim 변수**

```
print(array1.ndim)
```
```
1
```

```
print(array2.ndim)
```
```
2
```



# #1-3 ndarray의 데이터 타입

**#1 숫자값, 문자값, 불값 등 모두 가능**

!! Ndarray 내 **같은 데이터 타입만** 가능 (ex. Int형 + float형 (X))

**#2 데이터 타입 확인하기**
**: dtype 속성**

```
print(array1.dtype)
```

**#3 다른 데이터 유형 섞인 리스트 -> ndarray 변환 시**
: **크기가 더 큰** 데이터 타입으로 일괄 형반환

**#4 데이터 타입 변경**
**: astype() 메서드**
인자에 원하는 타입을 문자열로 지정

```
array2 = array1.astype( 'float64' )
array2 = array1.astype( 'int32' )
```

**+ 속성 vs. 메서드?**

**속성** : 객체가 가지고 있는 **"값"**
그냥 꺼내 읽는 것
**메서드** : 객체가 가지고 있는 **"함수"**
해당 객체에 어떤 동작을 시키는 것이라 ()로 실행!

# #1-4 ndarray 편리하게 생성하기, 크기 바꾸기

| 함수/메서드 이름<br>0 | 쓰임 | Default 인자 | 기타 | 코드 예시 | 결과 예시<br>(print 시) |
|---|---|---|---|---|---|
| arange | 0 ~ (인자-1)까지 순차적 데이터값 생성 | stop 값 | start 값 부여해 0이 아닌<br>다른 값부터 시작 가능 | array1 = np.arrange(10) | [0 1 2 3 4 5 6 7 8 9] |
| zeros | 튜플 형태의 shape 값 입력,<br>모든 값을 0으로 채운<br>해당 shape을 가진 ndarray 반환 | (float64형 Dtype) | 인자로 dtype 지정 가능 | array2 = np.zeros( (3,2), dtype = ‘int32”) | [ [0 0]<br>[0 0]<br>[0 0] ] |
| ones | 튜플 형태의 shape 값 입력,<br>모든 값을 1으로 채운<br>해당 shape을 가진 ndarray 반환 | (float64형 Dtype) | 인자로 dtype 지정 가능 | array3 = np.ones( (3,2), dtype = ‘int32”) | [ [1 1]<br>[1 1]<br>[1 1] ] |
| reshape | 변환 원하는 크기를 인자로 입력,<br>ndarray 특정 차원 및 크기로 변환 | - | - 지정된 사이즈로 변경 불가 시 에러 | array4 = array1.reshape( 2,5 ) | [ [0 1 2 3 4]<br>[5 6 7 8 9] ] |
| | | | - 인자 -1 : 호환되는 새로운 shape 저절로 변환, 지정된 사이즈로 변경 불가 시 에러<br>- 인자 (-1, 1) : (n, 1)로 변환 | array5 = array1.reshape( 5, -1 ) | [ [0 1]<br>[2 3]<br>[4 5]<br>[6 7]<br>[8 9] ] |

# #1-5 ndarray 데이터 세트 선택 : 인덱싱, 슬라이싱

### #1 단일 값 추출 – 1차원

: 원하는 위치의 **인덱스** 값 [ ] 안 입력

```
value1 = array[6]
```

: **인덱스** 값 -1
맨 뒤의 값

응용 – 뒤에서 두 번 째 값

```
value2 = array[-2]
```

### #2 단일 값 변경

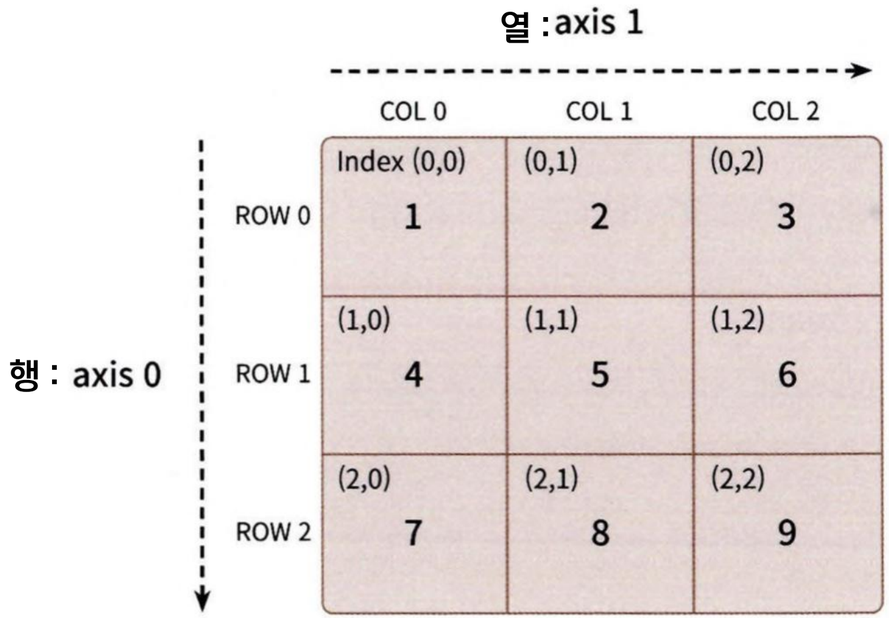
: 단일 인덱스 사용해 변수 선언처럼 간단하게 변경 가능

```
array[6] = 1
array[6] = 4
```

### #1-1 단일 값 추출 – 2차원

: 원하는 위치의 **인덱스** 값 [ ] 안 입력

```
value3 = array[ 0, 0]
value4 = array[ 0, 1]
```




※ 다차원 array는 축 기반 연산, axis 생략 시 행(axis0) 의미

EWHA EURON

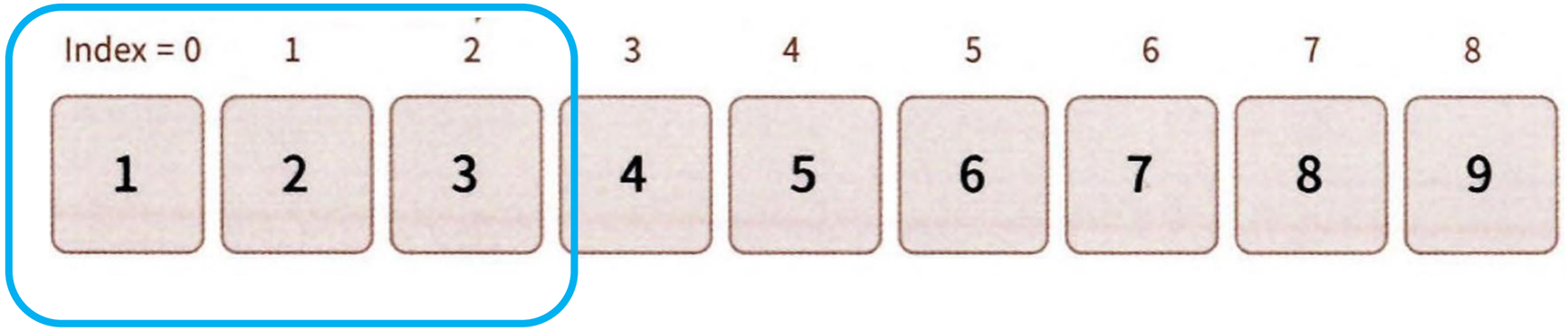
# #1-5 ndarray 데이터 세트 선택 : 인덱싱, 슬라이싱

## #3-1 슬라이싱 – 1차원

: “ : “ 기호 사이 시작 인덱스와 종료 인덱스 표시.
-> 시작 인덱스 ~ (종료 인덱스-1) 까지의 ndarry 반환

: 인덱스 생략 가능!
- 시작 인덱스 생략 : 맨 처음 인덱스 0 간주
- 종료 인덱스 생략 : 맨 마지막 인덱스 간주
- 모두 생략 : 0~ 마지막

```
value1 = array1[0:3]
```




## #3-2 슬라이싱 – 2차원

: 1차원과 유사, “ , “ 로 행 인덱스와 칼럼 인덱스 구분

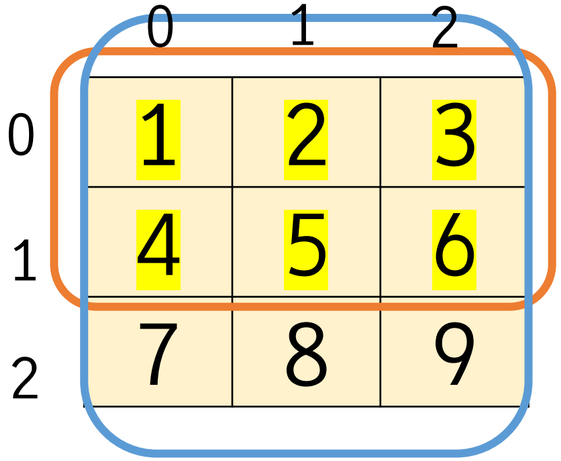
# #1-5 ndarray 데이터 세트 선택 : 인덱싱, 슬라이싱

## #3-2 슬라이싱 – 2차원

: 1차원과 유사, “ , “ 로 행 인덱스와 칼럼 인덱스 구분

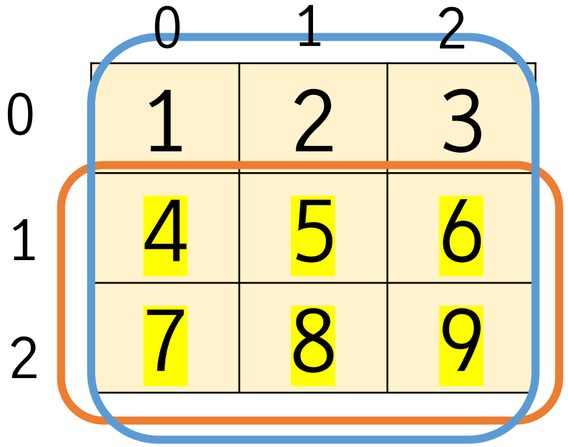
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

array[ 0:2, 0:3 ]




| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

array[ 1:3, : ]



# #1-5 ndarray 데이터 세트 선택 : 인덱싱, 슬라이싱

## #4 팬시 인덱싱

: **리스트나 ndarray로** 인덱스 집합 지정
-> 해당 위치 ndarray 반환

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

array[ [0, 1] , 2 ]

팬시 인덱싱 단일 인덱싱

**인덱스 (0,2), (1,2)**

[ 3, 6 ] 반환

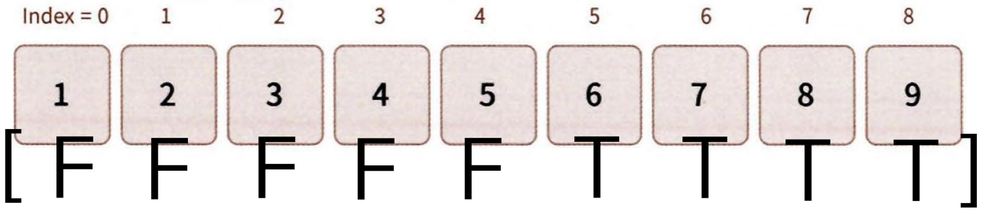
## #5 불린 인덱싱

: **조건 필터링 + 검색**
인덱스 지정 [ ] 내에 조건문 기재

array[ array > 5 ]

결과 : [ 6 7 8 9 ]

1. 조건식 부분 array > 5 에서 불린 배열이 생성됨




2. [ ]는 True 값이 있는 위치 인덱스 해당 값을 자동 변환해 반환
(False 값은 무시)

# #1-6 행렬의 정렬 : sort(), argsort()

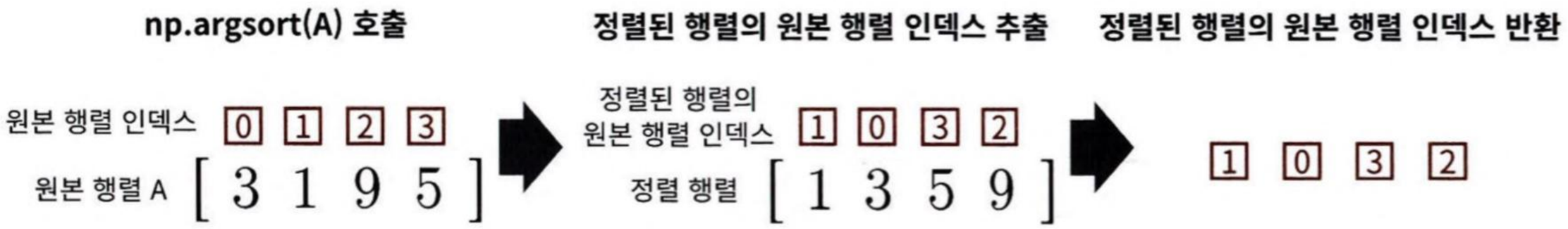
#1 sort()

| 이름 | 기능 | 특징 | 원래 행렬 | 반환값 | 차순 | 기타 |
|---|---|---|---|---|---|---|
| np.sort() | 행렬 정렬 | 넘파이에서 호출 | 원본 행렬 변경 X | 정렬된 행렬 | 기본 오름차순 | 내림차순 정렬 시 [::-1] 사용 |
| ndarray.sort() | | 행렬 자체에서 호출 | 정렬됨 (변경됨) | None | | |

# #1-6 행렬의 정렬 : sort(), argsort()

## #2 argsort()

: 행렬 정렬 후, 기존 원본 행렬의 인덱스 반환
+ 내림차순 정렬 시 [::-1] 사용




```
array1 = np.array( [ 3, 1, 9, 5] )
array_sort = np.argsort( array1 )

print(array_sort)
```

**결과 :** [1 0 3 2]

# #1-7 선형대수 연산 : 행렬 내적, 전치 행렬

| 이름 | 기능 | 코드 예시 | 결과 예시 |
|---|---|---|---|
| np.dot() | 인수로 받은 두 행렬의 내적(행렬곱) 수행 | A = np.array( [1,2,3], [4,5,6]])<br>B = np.array( [7,8],[9,10],[11,12]])<br><br>dot_product = np.dot( A, B)<br><br>print(dot_product) | [ [58 64]<br>[139, 154]] |
| np.transpose() | 인수로 받은 행렬을 전치 | A = np.array( [1,2], [3,4]] )<br>transpose = np.transpose(A)<br><br>print(transpose) | [ [1 3]<br>[2,4] ] |

# 2. 판다스 Part1

EWHA EURON

# #2-1 판다스 소개

## #1 판다스 vs 넘파이:

판다스는 데이터 처리를 위해 존재하는 가장 인기 있는 파이썬 '라이브러리'

기능 모음집
원하는 시점에 불러서 씀

2차원 데이터를 효율적으로 가공/처리할 수 있도록 함

행과 열로 이루어진 데이터를 말함

넘파이와 비교 1.
판다스는 넘파이보다 고수준 API로 더 유연함.

함수·메서드들의 모음

->더 편리한 데이터 핸들링 가능

넘파이와의 비교 2.
파이썬의 리스트,컬렉션, 넘파이 등의 내부 데이터
+ csv파일도 모두 'DataFrame' 으로 변환

->가공/분석이 쉬움

## #2 DataFrame :

| | survived | pclass | sex | age |
|---|---|---|---|---|
| 0 | 0 | 3 | male | 22.0 |
| 1 | 1 | 1 | female | 38.0 |
| 2 | 1 | 3 | female | 26.0 |

**Index:**
개별 데이터를
고유하게 식별하는
Key 값

**Series:**
칼럼이 하나 뿐인 데이터 구조체
*DataFrame은 여러 개의 Series로
이뤄졌다고 할 수 있음.

*Series와 DataFrame은 모두
Index를 key 값으로 가지고 있음.

# #2-1 판다스 소개

## #3 판다스 시작과 기본 API

판다스 시작:

```
import pandas as pd
```

판다스 데이터 로딩 API:

| 함수명 | 기본 구분자 | 특징 / 용도 | 예시 |
|---|---|---|---|
| read_csv() | 콤마(,) | 가장 범용적, sep로 다양한 구분자 지정 가능 | pd.read_csv("data.csv")pd.read_csv("data.txt", sep="\t") |
| read_table() | 탭(\t) | 사실상 read_csv(sep="\t")와 동일 | pd.read_table("data.txt") |
| read_fwf() | 없음(고정폭) | 고정 폭 열 파일 처리 | pd.read_fwf("data.fwf") |

* 실무에서는 거의 read_csv()만 사용
* sep을 활용해서 범용성 갖춤

# #2-1 판다스 소개

## #3 판다스 시작과 기본 API

판다스 DataFrame 탐색 & 요약:

| 기능 | 메서드 / 속성 | 설명 | 예시 |
|---|---|---|---|
| 일부 출력 | .head(n) | 앞의 n개 행 출력 (기본 5개) | df.head(3) |
| 크기 확인 | .shape | (행, 열) 튜플 반환 | (891, 12) |
| 데이터 구조 확인 | .info() | 행/열 수, 데이터 타입, Null 개수 | df.info() |
| 수치형 데이터 분포 확인 | .describe() | 숫자형 칼럼의 평균, 표준편차, 분포 | df.describe() |
| 카테고리형 데이터 분포 확인 | .value_counts() | 칼럼별 데이터 값의 분포도를 Series 객체로 반환<br>맨 왼쪽: 인덱스값<br>오른쪽: 데이터값 | df['Pclass'].value_counts() |
| 단일 칼럼 | df['칼럼명'] | Series 반환 (인덱스+값) | df['Pclass'] |

*Null 포함 여부는 dropna 인자로 판단

*기본은 dropna= True로 Null을 불포함하는 것이 기본.

*Null을 포함하고 싶으면 False로 바꾸면 됨.

# #2-2 DataFrame과 상호  변환 (list, dictionary, ndarray)

## #1 리스트/ndarray/딕셔너리 → DataFrame

CSV를 불러올 수도 있지만, 기본적으로 DataFrame은 파이썬의 리스트, 딕셔너리 그리고 넘파이 ndarray 등으로 변환이 가능함.

1. 리스트와 ndarray를 2차원인 DataFrame으로 변환하는 경우
   -> 1차원인지, 2차원인지에 따라 칼럼 개수만 조정해 주면 나머지는 동일

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

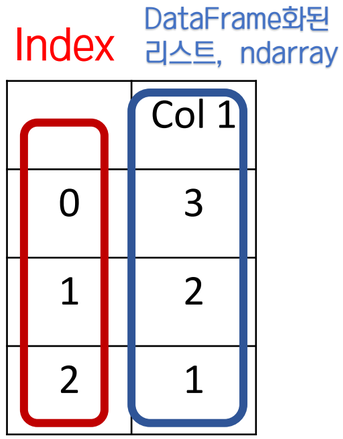
칼럼을 필요한 개수만큼의 리스트로 지정

```
# 리스트인 경우
df_list2 = pd.DataFrame(list2, columns=col_name2)

# ndarray인 경우
df_array2 = pd.DataFrame(array2, columns=col_name2)
```

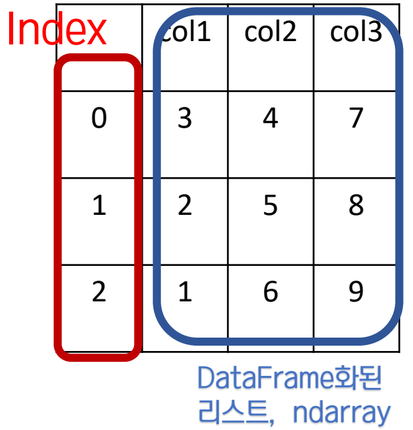
변환대상

DataFrame에서 칼럼 명




2. 딕셔너리를  DataFrame으로 변환하는 경우
   -> 딕셔너리의 키(Key)는 칼럼명으로, 딕셔너리의 값(Value)은 칼럼 데이터로 자동 매핑

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



# #2-2 DataFrame과 상호  변환 (list, dictionary, ndarray)

## #2 DataFrame -> 리스트/ndarray/딕셔너리

머신러닝 패키지가 기본 데이터 형으로 넘파이 ndarray를 많이 사용하여 반대로 변환하는 경우가 생김

1. 넘파이 ndarray로 변환하기  -> .values

```
array3 = df_dict.values
```

변환대상 DataFrame

2. 리스트로 변환하기 -> .tolist()

```
list3 = df_dict.values.tolist()
```

*2차원을 변환해야 하기 때문에 넘파이의 ndarray로 변환하는 과정이 필요한 것.
-> 따라서 1차원만 변환할 때는 바로 .tolist()사용 가능
Ex) list1 =df_series.tolist()

3. 딕셔너리로 변환하기 -> .to_dic()

```
dict3 = df_dict.to_dict('list')
```

*데이터 타입 부분에  'list' 라고 지정해주면 딕셔너리의 값이 리스트형으로 나옴.

```
df_dict.to_dict() 타입: <class 'dict'>
{'col1': [1, 11], 'col2': [2, 22], 'col3': [3, 33]}
```

# #2-3 DataFrame의 칼럼 데이터 세트 생성과 수정

## #1 데이터 세트 생성

```
titanic_df['Age_0']=0
```

코드 풀이:
기존의 'titanic_df' 데이터 프레임에 'Age_o' 이라는 칼럼이 생성된 것. 이 칼럼의 데이터 값은 일괄적으로 '0' 할당 됨.

```
titanic_df['Age_by_10'] = titanic_df['Age']*10
titanic_df['Family_No'] = titanic_df['SibSp'] + titanic_df['Parch']+1
titanic_df.head(3)
```

코드 풀이:
기존의 series를 가지고 새로운 series를 만들 수도 있음. 위의 코드들은 기존의 series를 활용해 'Age_by_10' 과 'Family_No' series를 만들어냄.

## #2 데이터 세트 수정 (업데이트)

```
titanic_df['Age_by_10'] = titanic_df['Age_by_10'] + 100
titanic_df.head(3)
```

코드 풀이:
해당 series 뒤에 연산을 추가하고, 이를 재지정하는 방식으로 수정, 업데이트가 가능함.

# #2-4 DataFrame 데이터 삭제

## #1 drop() 메서드의 원형

```
DataFrame.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')
```

원하는 칼럼명 입력

넘파이에서 배웠던 대로
axis=0 은 '로우'
axis=1 은 '칼럼 ' 임
따라서 axis =1로 이를 지정하는 경우 多

삭제할 때, 원형의 버전을 완전히 대체할 것인지 아닌지를 정하는 부분.
False가 디폴트이기 때문에, 대체되지 않는 것이 기본.
완전히 삭제하고 싶으면 True로 바꿔야 함.

## #2 drop()메서드 활용 inplace = False인 경우

```
titanic_drop = titanic_df.drop(['Age_0], axis=1, inplace=False)
```

->titanic_drop.head(3)와 titanic_df.head(3)의 결과값이 다름!!

| Ticket | Fare | Cabin | Embarked | Age_by_10 | Family_No |
|---|---|---|---|---|---|
| A/5 21171 | 7.2500 | NaN | S | 320.0 | 2 |
| PC 17599 | 71.2833 | C85 | C | 480.0 | 2 |
| STON/O2. 3101282 | 7.9250 | NaN | S | 360.0 | 1 |

titanic_drop.head(3) 출력
Age_0칼럼이 없음.

| Ticket | Fare | Cabin | Embarked | Age_0 | Age_by_10 | Family_No |
|---|---|---|---|---|---|---|
| A/5 21171 | 7.2500 | NaN | S | 0 | 320.0 | 2 |
| PC 17599 | 71.2833 | C85 | C | 0 | 480.0 | 2 |
| STON/O2. 3101282 | 7.9250 | NaN | S | 0 | 360.0 | 1 |

titanic_df.head(3) 출력
Age_0칼럼이 있음.

## #3 drop()메서드 활용 inplace = True인 경우

```
titanic_df.drop(['Age_0', 'Age_by_10', 'Family_No'], axis=1, inplace=True)
titanic_df.head(3)
```

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN | S |
| 1 | 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |
| 2 | 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26.0 | 0 | 0 | STON/O2. 3101282 | 7.9250 | NaN | S |

True로 설정했기 때문에 세 칼럼 모두 없어진 채로 출력됨.

*True는 반환 값이 None(아무 값도 아님)임.

*따라서 True로 설정한 채로 반환 값을 다시 자신의 DataFrame 객체로 할당하면 해당 객체 변수를 아예 None으로 만들어 버리기 때문에 안됨.

# #2-4 DataFrame 데이터 삭제

## #4 axis가 0으로 로우 없애기

```
titanic_df.drop([0,1,2], axis=0, inplace =True)
```

지우고 싶은 인덱스 값

로우를 없애라

**결과값:**

```
#### before axis 0 drop ####
   PassengerId  Survived  Pclass                Name     Sex   Age  SibSp  Parch
0            1         0       3  Braund, Mr....     male  22.0      1      0
1            2         1       1  Cumings, Mr...  female  38.0      1      0
2            3         1       3  Heikkinen, ...  female  26.0      0      0
#### after axis 0 drop ####
   PassengerId  Survived  Pclass                Name     Sex   Age  SibSp  Parch
3            4         1       1  Futrelle, M...  female  35.0      1      0
4            5         0       3  Allen, Mr. ...     male  35.0      0      0
5            6         0       3  Moran, Mr. ...     male   NaN      0      0
```

Index 0,1,2에 해당하는 로우가 삭제되고 3,4,5가 올라옴

## #5 정리

- axis : DataFrame의 로우를 삭제할 때는 axis=0, 칼럼을 삭제할 때는 axis=1으로 설정.

- 원본 DataFrame은 유지하고 드롭된 DataFrame을 새롭게 객체 변수로 받고 싶다면 inplace=False로 설정(디폴트 값이 False임).
예: titanic_drop_df = titanic_df.drop(' Age_0', axis=1, inplace=False )

- 원본 DataFrame에 드롭된 결과를 적용할 경우에는 inplace=True를 적용.
예: titanic_df.drop( 'Age_0', axis=1, inplace=True )

- 원본 DataFrame에서 드롭된 DataFrame을 다시 원본 DataFrame 객체 변수로 할당하면 원본 DataFrame에서 드롭된 결과를 적용할 경우와 같음(단, 기존 원본 DataFrame 객체 변수는 메모리에서 추후 제거됨).
예: titanic_df = titanic_df.drop(' Age_0', axis=1, inplace=False )

# #2-5 Index 객체

## #1 Index 객체란?

- DataFrame/Series 레코드를 고유하게 식별하는 객체
  - RDBMS의 Primary Key와 유사

-추출 방법: .index
ex) DataFrame.index, Series.index

-실제 값 확인: .values
ex) Index.values → 1차원 array 반환

*그러나 고유한 인덱스 값 변경을 불가함.
*Series 객체는 Index 객체를 포함하지만 Series 객체에 연산 함수를 적용할 때 Index는 연산에서 제외됩니다. Index는 오직 식별용으로만!

-단일값 반환 및 슬라이싱:

| | |
|---|---|
| indexes.values.shape | -> 크기 확인 |
| print(indexes[ : 5].values) | -> 슬라이싱 후에 array로 반환하기 |
| print(indexes.values[ : 5]) | -> array로 반환 후 슬라이싱<br>*위와 결과 값은 동일함. |
| print(indexes[6]) | []에 해당하는 index값 반환 |

# #2-5 Index 객체

## #2 reset_index()

- 새롭게 인덱스를 연속 숫자형으로 할당 .
- 기존 인덱스는 'index' 라는 새로운 칼럼 명으로 추가됨.
  ->이 경우, series는 DataFrame이 됨.
- 인덱스가 연속된 int 숫자형 데이터가 아닐 경우에 다시 이를 연속 int 숫자형 데이터로 만들 때 주로 사용.

【Output】

기존 index는 index라는 칼럼으로 추가됨

| | index | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN | S |
| 1 | 1 | 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |
| 2 | 2 | 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26.0 | 0 | 0 | STON/O2. 3101282 | 7.9250 | NaN | S |

새로운 index

-parameter 중 drop=True 로 설정하면 기존 인덱스는 새로운 칼럼으로 추가되지 않고 삭제(drop)됨.

```
titanic_reset_df = titanic_df.reset_index(drop=True)
```

# 03. 판다스 핵심

EWHA EURON

# #3-1 데이터 셀렉션 및 필터링

## #1 numpy의 [ ] 연산자

단일 값 추출, 슬라이싱, 팬시 인덱싱, 불린 인덱싱을 통해 데이터 추출
행의 위치, 열의 위치, 슬라이싱 범위 등을 지정해 데이터를 가져옴

## #2-1 pandas의 [ ] 연산자

iloc[ ], loc[ ] 연산자를 통해 동일한 작업 수행
칼럼명 문자, 인덱스로 변환가능한 표현식
입력값은 칼럼명(또는 칼럼의 리스트)을 지정해 칼럼 지정 연산에 사용하거나 불린 인덱스 용도로만 사용해야함
칼럼 지정 연산자
슬라이싱 연산으로 데이터 추출하는 방법은 비추천!

## #2-2 리스트 객체를 이용해 여러 칼럼의 데이터 추출

```
print('단일 칼럼 데이터 추출:\n', titanic_df['Pclass'].head(3))
print('\n여러 칼럼의 데이터 추출:\n', titanic_df[['Survived', 'Pclass']].head(3))
```

```
단일 칼럼 데이터 추출:
 0    3
1    1
2    3
Name: Pclass, dtype: int64

여러 칼럼의 데이터 추출:
   Survived  Pclass
0         0       3
1         1       1
2         1       3
```

EWHA EURON

# #3-1 데이터 셀렉션 및 필터링

## # 2-2 오류

칼럼명이 아닌 숫자 값 입력할 경우

```
print('[]안에 숫자 index는 KeyError 오류 발생:\n', titanic_df[0])
```

```
KeyError                                  Traceback (most recent call last)
Cell In[193], line 3
      1 print('단일 칼럼 데이터 추출:Wn', titanic_df['Pclass'].head(3))
      2 print('Wn여러 칼럼의 데이터 추출:Wn', titanic_df[['Survived', 'Pclass']].head(3))
----> 3 print('[]안에 숫자 index는 KeyError 오류 발생:Wn', titanic_df[0])
```

인덱스 형태로 변환 가능한 슬라이싱 사용

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

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 3 | Braund, Mr.... | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN | S |
| **1** | 2 | 1 | 1 | Cumings, Mr... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |

## #2-3 불린 인덱싱

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

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 3 | Braund, Mr.... | male | 22.0 | 1 | 0 | A/5 21171 | 7.250 | NaN | S |
| **2** | 3 | 1 | 3 | Heikkinen, ... | female | 26.0 | 0 | 0 | STON/O2. 31... | 7.925 | NaN | S |
| **4** | 5 | 0 | 3 | Allen, Mr. ... | male | 35.0 | 0 | 0 | 373450 | 8.050 | NaN | S |

EWHA EURON

# #3-1 데이터 셀렉션 및 필터링

## #3 DataFrame의 iloc[ ] 연산자

위치(Location) 기반 인덱싱으로 작동
=> 불린 인덱싱 제공 X
=> 정숫값, 정수형의 슬라이싱, 팬시 리스트 값을 입력해야함

**위치 기반 인덱싱?**

행과 열의 위치를 0을 출발점으로 하는
세로축, 가로축 좌표 정숫값으로 지정하는 방식

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

행과 열의 좌표 위치에 해당하는 값 입력

**오류**

```
data_df.iloc[0, 'Name']
data_df.iloc['one', 0]
```

열 위치에 위치 정숫값이 아닌 칼럼 명칭 입력
행 위치에 위치 정숫값이 아닌 인덱스 명칭 입력

EWHA EURON

# #3-1 데이터 셀렉션 및 필터링

## #3-1 DataFrame의 iloc[ ] 연산자- 정수 슬라이싱

| | Name | Year | Gender |
|---|---|---|---|
| **one** | Chulmin | 2011 | Male |
| **two** | Eunkyung | 2016 | Female |
| **three** | Jinwoong | 2015 | Male |
| **four** | Soobeom | 2015 | Male |

```
data_df.iloc[0:2, [0, 1]]
```

| | Name | Year |
|---|---|---|
| **one** | Chulmin | 2011 |
| **two** | Eunkyung | 2016 |

```
data_df.iloc[0:2, 0:3]
```

| | Name | Year | Gender |
|---|---|---|---|
| **one** | Chulmin | 2011 | Male |
| **two** | Eunkyung | 2016 | Female |
| **three** | Jinwoong | 2015 | Male |

전체 DataFrame 반환

```
data_df.iloc[:]
```

```
data_df.iloc[:, :]
```

| | Name | Year | Gender |
|---|---|---|---|
| **one** | Chulmin | 2011 | Male |
| **two** | Eunkyung | 2016 | Female |
| **three** | Jinwoong | 2015 | Male |
| **four** | Soobeom | 2015 | Male |

# #3-1 데이터 셀렉션 및 필터링

## #3-1 DataFrame의 iloc[ ] 연산자- 열 위치에 -1을 입력해 DataFrame의 가장 마지막 열 데이터 추출

넘파이와 마찬가지로 판다스 인덱싱에서도 -1은 마지막 데이터 값을 의미함

| | Name | Year | Gender |
|---|---|---|---|
| **one** | Chulmin | 2011 | Male |
| **two** | Eunkyung | 2016 | Female |
| **three** | Jinwoong | 2015 | Male |
| **four** | Soobeom | 2015 | Male |

타킷값

```
print("\n 맨 마지막 칼럼 데이터 [:, -1]\n", data_df.iloc[:,-1])
```

| | Gender |
|---|---|
| **one** | Male |
| **two** | Female |
| **three** | Male |
| **four** | Male |

피처값

```
print("\n 맨 마지막 칼럼을 제외한 모든 데이터 [:, :-1] \n", data_df.iloc[:,:-1])
```

| | Name | Year |
|---|---|---|
| **one** | Chulmin | 2011 |
| **two** | Eunkyung | 2016 |
| **three** | Jinwoong | 2015 |
| **four** | Soobeom | 2015 |

EWHA EURON

# #3-1 데이터 셀렉션 및 필터링

## #4 DataFrame의 loc[ ] 연산자

명칭(Label) 기반 인덱싱으로 작동
행 위치: DataFrame의 인덱스 값
열 위치: 칼럼의 명칭
명칭은 숫자형이 아닐 수 있음 => -1 사용 X

형식: loc[인덱스값, 칼럼명]

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

```
'Chulmin'
```

loc[시작점:종료점] 시작점 ~ 종료점

```
print('위치기반 iloc slicing\n', data_df.iloc[0:1,0], '\n')
print('명칭기반 loc slicing\n', data_df.loc['one':'two', 'Name'])
```

```
위치기반 iloc slicing
 one    Chulmin
Name: Name, dtype: object

명칭기반 loc slicing
 one      Chulmin
two     Eunkyung
Name: Name, dtype: object
```

위치기반은 행 1개 반환
명칭 기반은 2개의 행 반환

EWHA EURON

# #3-1 데이터 셀렉션 및 필터링

### #5 불린 인덱싱

가져올 값을 조건으로 [ ] 안에 입력하면 자동으로 원하는 값을 필터링함
loc[ ], [ ] 모두에서 지원함
형식: loc[조건]

조건: 나이가 60세 이상

```
titanic_boolean = titanic_df[titanic_df['Age'] > 60]
```

비트 연산자(~, &, |) 사용하면 복합 조건도 가능

조건: 나이가 60세 이상, 선실 1등급, 성별이 여성

```
titanic_df[ (titanic_df['Age']>60) & (titanic_df['Pclass']==1) & (titanic_df['Sex']=='female') ]
```

개별 조건을 변수에 할당하고
그 변수들을 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]
```

# #3-2 정렬, Aggregation 함수, GroupBy 적용

## #1 정렬 메소드; sort_values()

**by** 특정 칼럼을 입력하면, 해당 칼럼으로 정렬 수행

```
titanic_sorted = titanic_df.sort_values(by=['Name'])
titanic_sorted.head(3)
```

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **845** | 846 | 0 | 3 | Abbing, Mr.... | male | 42.0 | 0 | 0 | C.A. 5547 | 7.55 | NaN | S |
| **746** | 747 | 0 | 3 | Abbott, Mr.... | male | 16.0 | 1 | 1 | C.A. 2673 | 20.25 | NaN | S |
| **279** | 280 | 1 | 3 | Abbott, Mrs... | female | 35.0 | 1 | 1 | C.A. 2673 | 20.25 | NaN | S |

**ascending** True(default): 오름차순, False: 내림차순

```
titanic_sorted = titanic_df.sort_values(by=['Pclass', 'Name'], ascending=False)
titanic_sorted.head(3)
```

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **868** | 869 | 0 | 3 | van Melkebe... | male | NaN | 0 | 0 | 345777 | 9.5 | NaN | S |
| **153** | 154 | 0 | 3 | van Billiar... | male | 40.5 | 0 | 2 | A/5. 851 | 14.5 | NaN | S |
| **282** | 283 | 0 | 3 | de Pelsmaek... | male | 16.0 | 0 | 0 | 345778 | 9.5 | NaN | S |

**inplace**

False(default): 원본 유지, 정렬된 결과 반환
True: 원본 변경, 반환 값 없음

```
sorted_df = titanic_df.sort_values(by=['Name'], inplace = False)
titanic_sorted.head(3)
```

EWHA EURON

# #3-2 정렬, Aggregation 함수, GroupBy 적용

### #2 Aggregation 함수

min(), max(), sum(), count()

DataFrame에서 바로 aggregation을 호출하면 모든 칼럼에 해당 aggregation을 적용함
count()는 Null 값을 반영하지 않음

```
titanic_df.count()
```

```
PassengerId    891
Survived       891
Pclass         891
Name           891
Sex            891
Age            714
SibSp          891
Parch          891
Ticket         891
Fare           891
Cabin          204
Embarked       889
dtype: int64
```

특정 칼럼에 aggregation 함수를 적용
: DataFrame에 대상 칼럼들만 추출해 적용

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

```
Age      29.699118
Fare     32.204208
dtype: float64
```

# #3-2 정렬, Aggregation 함수, GroupBy 적용

## #3 GroupBy 적용

입력 파라미터 by에 칼럼 입력하면, 대상 칼럼으로 GroupBy 됨

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

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

Pclass 칼럼 기준으로 GroupBy된 DataFrameGroupby 객체를 반환함

groupby() 호출로 반환된 결과에 aggregation 함수를 호출
=> 대상 칼럼을 제외한 모든 칼럼에 aggregation 함수가 적용됨

```
titanic_groupby = titanic_df.groupby('Pclass').count()
titanic_groupby
```

| Pclass | PassengerId | Survived | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 216 | 216 | 216 | 216 | 186 | 216 | 216 | 216 | 216 | 176 | 214 |
| 2 | 184 | 184 | 184 | 184 | 173 | 184 | 184 | 184 | 184 | 16 | 184 |
| 3 | 491 | 491 | 491 | 491 | 355 | 491 | 491 | 491 | 491 | 12 | 491 |

# #3-2 정렬, Aggregation 함수, GroupBy 적용

groupby()에 특정 칼럼만 aggregation 함수 적용
=> 반환된 DataFrameGroupBy 객체에 해당 칼럼 필터링 -> aggregation 함수 적용

```
titanic_groupby = titanic_df.groupby('Pclass')[['PassengerId', 'Survived']].count()
titanic_groupby
```

| Pclass | PassengerId | Survived |
|---|---|---|
| 1 | 216 | 216 |
| 2 | 184 | 184 |
| 3 | 491 | 491 |

여러 개의 aggregation 함수 명을 객체의 agg() 내에 인자로 입력해서 사용

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

| Pclass | max | min |
|---|---|---|
| 1 | 80.0 | 0.92 |
| 2 | 70.0 | 0.67 |
| 3 | 74.0 | 0.42 |

여러 개의 칼럼을 서로 다른 aggregation 함수로 호출 시, 딕셔너리 형태로 처리

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

| Pclass | Age | SibSp | Fare |
|---|---|---|---|
| 1 | 80.0 | 90 | 84.154687 |
| 2 | 70.0 | 74 | 20.662183 |
| 3 | 74.0 | 302 | 13.675550 |

# #3-3 결손 데이터 처리하기

### #1 결손 데이터

칼럼에 값이 없는 상태; NULL인 경우
넘파이: NaN
함수 연산, 머신러닝에서는 NaN값을 처리하지 않기 때문에 다른 값으로 대체해야 함

**isna()** NaN 여부 확인
True, False로 알려줌

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

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | False | False | False | False | False | False | False | False | False | False | True | False |
| **1** | False | False | False | False | False | False | False | False | False | False | False | False |
| **2** | False | False | False | False | False | False | False | False | False | False | True | False |

**sum()** True는 1, False는 0으로 변환됨 => 결손 데이터의 개수를 구할 수 있음

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

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

# #3-3 결손 데이터 처리하기

fillna()

결손 데이터 대체

* 실제 데이터 세트 값 변경하는 방법 *
1. fillna()를 이용해 반환 값 다시 받기
2. inplace = True 파라미터 추가하기

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

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 3 | Braund, Mr.... | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | C000 | S |
| **1** | 2 | 1 | 1 | Cumings, Mr... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |
| **2** | 3 | 1 | 3 | Heikkinen, ... | female | 26.0 | 0 | 0 | STON/O2. 31... | 7.9250 | C000 | S |

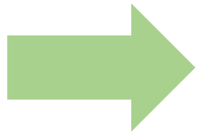
# #3-4 apply lambda 식으로 데이터 가공

## #1 lambda식

함수의 선언과 함수 내의 처리를 한 줄의 식으로 쉽게 변환하는 식
복잡한 데이터 가공이 필요할 경우에 사용

```
def get_square(a):
    return a**2

print('3의 제곱은:', get_square(3))
```



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

lambda x : x ** 2

입력 인자

입력 인자를 기반으로 한 계산식, 호출 시 계산 결과가 반환됨

map() 여러 개의 값을 입력 인자로 사용할 경우

```
a=[1,2,3]
squares = map(lambda x : x**2, a)
list(squares)
```

```
[1, 4, 9]
```

# #3-4 apply lambda 식으로 데이터 가공

if-else 절

: 의 오른쪽에 반환값이 있어야 하기 때문에 if 식보다 반환 값을 먼저 써야 함

```
titanic_df['Child_Adult'] = titanic_df['Age'].apply(lambda x : 'Child' if x <=15 else 'Adult')
titanic_df[['Age', 'Child_Adult']].head(8)
```

| | Age | Child_Adult |
|---|---|---|
| 0 | 22.000000 | Adult |
| 1 | 38.000000 | Adult |
| 2 | 26.000000 | Adult |
| 3 | 35.000000 | Adult |
| 4 | 35.000000 | Adult |
| 5 | 29.699118 | Adult |
| 6 | 54.000000 | Adult |
| 7 | 2.000000 | Child |

else if 지원 X => else 절을 ()로 내포에 () 안에서 다시 if else를 적용하면 사용 가능

```
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x : 'Child' if x<=15 else ('Adult' if x<=60 else 'Elderly'))
titanic_df['Age_cat'].value_counts()
```

| Age_cat | count |
|---|---|
| Adult | 786 |
| Child | 83 |
| Elderly | 22 |

# #3-4 apply lambda 식으로 데이터 가공

## if-else 절

else if가 많이 나와야 하는 경우나 switch case 문을 사용하는 경우에는 별도의 함수를 만들어 사용

```
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 식에서 위에서 생성한 get_category() 함수를 반환값으로 지정
# get_category(X)는 입력값으로 'Age'칼럼 값을 받아서 해당하는 cat 반환
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x : get_category(x))
titanic_df[['Age', 'Age_cat']].head()
```

# #3-5 요약

## #1 넘파이 - ndarray 편리하게 생성하기, 크기 바꾸기

| 함수/메서드 이름<br>0 | 쓰임 | Default 인자 | 기타 | 코드 예시 | 결과 예시<br>(print 시) |
|---|---|---|---|---|---|
| arange | 0 ~ (인자-1)까지 순차적 데이터값 생성 | stop 값 | start 값 부여해 0이 아닌<br>다른 값부터 시작 가능 | array1 = np.arrange(10) | [0 1 2 3 4 5 6 7 8 9] |
| zeros | 튜플 형태의 shape 값 입력,<br>모든 값을 0으로 채운<br>해당 shape을 가진 ndarray 반환 | (float64형 Dtype) | 인자로 dtype 지정 가능 | array2 = np.zeros( (3,2), dtype = ‘int32”) | [ [0 0]<br>[0 0]<br>[0 0] ] |
| ones | 튜플 형태의 shape 값 입력,<br>모든 값을 1으로 채운<br>해당 shape을 가진 ndarray 반환 | (float64형 Dtype) | 인자로 dtype 지정 가능 | array3 = np.ones( (3,2), dtype = ‘int32”) | [ [1 1]<br>[1 1]<br>[1 1] ] |
| reshape | 변환 원하는 크기를 인자로 입력,<br>ndarray 특정 차원 및 크기로 변환 | - | - 지정된 사이즈로 변경 불가 시 에러 | array4 = array1.reshape( 2,5 ) | [ [0 1 2 3 4]<br>[5 6 7 8 9] ] |
| | | | - 인자 -1 : 호환되는 새로운 shape 저절로 변환, 지정된 사이즈로 변경 불가 시 에러<br>- 인자 (-1, 1) : (n, 1)로 변환 | array5 = array1.reshape( 5, -1 ) | [ [0 1]<br>[2 3]<br>[4 5]<br>[6 7]<br>[8 9] ] |

# #3-5 요약

#1 넘파이 - 행렬의 정렬

| 이름 | 기능 | 특징 | 원래 행렬 | 반환값 | 차순 | 기타 |
|---|---|---|---|---|---|---|
| np.sort() | 행렬 정렬 | 넘파이에서 호출 | 원본 행렬 변경 X | 정렬된 행렬 | 기본 오름차순 | 내림차순 정렬 시 [::-1] 사용 |
| ndarray.sort() | | 행렬 자체에서 호출 | 정렬됨 (변경됨) | None | | |

# #3-5 요약

## #1 넘파이 - 행렬의 정렬

| 이름 | 기능 | 코드 예시 | 결과 예시 |
|---|---|---|---|
| np.dot() | 인수로 받은 두 행렬의 내적(행렬곱) 수행 | A = np.array( [1,2,3], [4,5,6]])<br>B = np.array( [7,8],[9,10],[11,12]])<br><br>dot_product = np.dot( A, B)<br><br>print(dot_product) | [ [58 64]<br>[139, 154]] |
| np.transpose() | 인수로 받은 행렬을 전치 | A = np.array( [1,2], [3,4]] )<br>transpose = np.transpose(A)<br><br>print(transpose) | [ [1 3]<br>[2,4] ] |

# #3-5 요약

**#2 판다스**

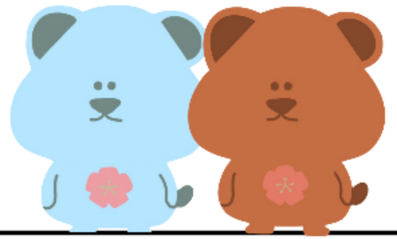
1. value_counts() 함수에서 null 불포함하는 것이 기본임.
2. 변환 시, 1차원인지 2차원인지를 고려해야 함.
   특히 dataframe을 리스트로 바꿀 때, 2차원은 ndarray를 거쳐야 함
3. 데이터 셋 삭제 시 axis로 로우와 칼럼 선택 가능, inplace로 완전히 대체될지 아닐지 선택 가능
4. 고유한 인덱스 변경은 불가
5. 새롭게 인덱스 할당 가능, 이때 drop으로 기존 인덱스 포함 여부 설정 가능

# #3-5 요약

**#3 판다스 핵심**

1. 개별 또는 여러 칼럼 값 추출 => DataFrame[ '칼럼명' ]
2. 행과 열을 함께 추출 => iloc[ ], loc[ ]
3. 명칭 기반 인덱싱: DataFrame의 인덱스나 칼럼명으로 데이터에 접근
4. 위치 기반 인덱싱: 0부터 시작하는 행, 열의 위치 좌표에 의존
5. iloc[ ]는 위치 기반 인덱싱만 가능 => 행, 열 위치 값으로 정수형 값을 지정해 원하는 데이터 반환
6. loc[ ]는 명칭 기반 인덱싱만 가능 => 행 위치에 DataFrame 인덱스가 옴, 열 위치에는 칼럼명을 지정해 원하는 데이터를 반환
7. 명칭 기반 인덱싱에서 슬라이싱을 '시작점:종료점' 으로 지정하면 시작점~종료점 데이터를 반환
8. apply lamda 식으로 데이터 가공하면 코드 가독성이 좋아서 프로젝트에서 활용하면 유용함

# THANK YOU



EWHA EURON