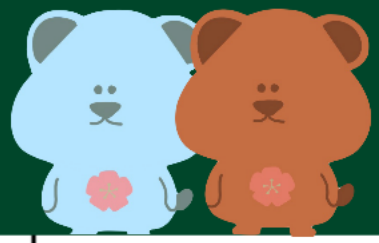


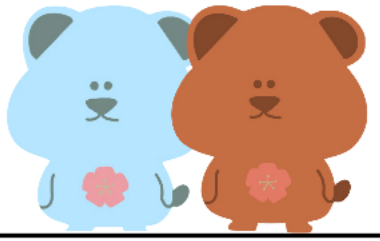
# 파이썬 기반의 머신러닝과 생태계 이해

1조 오지현 방민지 함예린

EWHA EURON

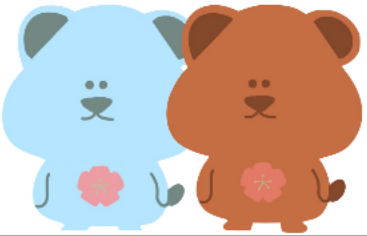
# 목차





EWHA EURON

# 넘파이



EWHA EURON

# #1.1 넘파이(numpy) 개요

#1 넘파이(numpy) 데이터 타입 : ndarray

```
array1 = np.array([1,2,3])
print(type(array1))  # numpy.ndarray
```

#2 ndarray 속성

```
ex) array1 = np.array([1,2,3])
array3 = np.array([[1,2,3]])
```

| 이름 | 설명 | 코드 예시 | 결과 예시 |
|---|---|---|---|
| np.array() | ndarray로 변환<br>* 주로 list 객체 | np.array([1,2,3])<br>np.array([[1,2,3]]) | [1,2,3]<br>[[1,2,3]] |
| ndarray.shape | ndarray 크기 (행, 열의 수)<br>* 차원 수 파악 가능함 | array1.shape<br>array3.shape | (3, )<br>(1,3) |
| ndarray.ndim() | ndarray 차원 | array1.ndim<br>array3.ndim | 1<br>2 |

# #1.2 ndarray 데이터 타입

#1 ndarray 데이터값 : 숫자, 문자열, 불 값 등

#2 ndarray 내 같은 데이터 유형만 가능
* if 다른 데이터 타입 존재
-> 데이터 크기 더 큰 타입으로 변환

```
list2 = [1,2,'test']  # list 내 다른 데이터 타입 섞여 있음
array2 = np.array(list2)  # list -> ndarray 변환
print(array2, array2.dtype)  # ['1','2','test'], <U11(유니코드 문자열)

list3 = [1,2,3.0]
array3 = np.array(list3)
print(array3, array3.dtype)  # [1.,2.,3.], float64
```

# #1.2 ndarray 데이터 타입

#3 astype() : 데이터 타입 변환
* 메모리 절약할 때 유용함

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

array_int1 = array_float.astype('int32')
print(array_int1, array_int1.dtype)  # [1,2,3], int32

array_float1 = np.array([1.1,2.1,3.1])
array_int2 = array_float1.astype('int32')  # 소수점 이하 제거
print(array_int2, array_int2.dtype)  # [1,2,3], int32
```

# #1.3 ndarray 생성하기

| 이름 | 설명 | 코드 예시 | 결과 예시 |
|---|---|---|---|
| np.arange() | 순차적으로 0부터 (stop-1) 값까지의 값 출력<br>* range(0, stop)과 유사함 | np.arange(10)<br>np.arange(start=1, stop=10) | [0,1,2,3,4,5,6,7,8,9]<br>[1,2,3,4,5,6,7,8,9] |
| np.zeros() | Tuple 형태 shape 입력 -> 0으로 채운 ndarray 반환 | np.zeros((3,2), dtype= 'int32' ) | [[0,0],[0,0],[0,0]] |
| np.ones() | Tuple 형태 shape 입력 -> 1로 채운 ndarray 반환 | np.ones((3,2)) | [[1.,1.],[1.,1.],[1.,1.]]<br>* default : float64 |

# #1.4 ndarray 차원과 크기 변경

#1 ndarray.reshape()

* np.stack(), np.concat() 이용 시,
  ndarray 형태 통일하기 위해 주로 사용함

```
ex) array1 = np.arange (10)
array2 = np.arange(8)
array3d = array2.reshape((2,2,2))
```

| 설명 | 코드 예시 | 결과 예시 |
|---|---|---|
| 1차원을 2차원 ndarray로 변환 | array1.reshape(2,5)<br>array1.reshape(-1,5) | [[0,1,2,3,4],<br>[5,6,7,8,9]] |
| 변환하지 못하는 경우 | array1.reshape(4,3)<br>array1.reshape(4,-1) | error |
| 1차원을 3차원 ndarray로 변환 | array2.reshape((2,2,2)) | [[[0,1],[2,3]],<br>[[4,5],[6,7]]] |
| 3차원을 2차원 ndarray로 변환 | array3d.reshape(-1,1) | [[0],[1],[2],[3],<br>[4],[5],[6],[7]] |

# #1.5 ndarray 인덱싱(indexing)

```
ex) array1 = np.arange(1,10)
array2d = array1.reshape(3,3)
```

* Index 이용하여 데이터 값 수정 ->

```
array1[0]=9
array1[8]=0
print(array1)  # [9,2,3,4,5,6,7,8,0]
```

| 이름 | 설명 | 코드 예시 | 결과 예시 |
|---|---|---|---|
| 단일 값 추출 | index 값을 [ ] 안에 입력<br>* (-) 기호로 맨 뒤에서부터 추출 | array1[2]<br>array1[-1] | 3<br>9 |
| | | array2d[0,0] | 1 |
| 슬라이싱 (Slicing) | : 이용하여 [ ] 안에 입력 | array1[ :3]<br>array1[3: ] | [1,2,3]<br>[4,5,6,7,8,9] |
| | | array2d[0:2,0:2]<br>array2d[ :2,0]<br>array2d[0] | [[1,2],[4,5]]<br>[1,4]<br>[1,2,3] |
| 팬시 인덱싱<br>(Fancy Indexing) | list나 ndarray를<br>[ ] 안에 입력 | array2d[[0,1],2]<br>array2d[[0,1]] | [3,6]<br>[[1,2,3],[4,5,6]] |
| 불린 인덱싱<br>(Boolean Indexing) | 조건문을 [ ] 안에 입력<br>* 조건 필터링 + 검색 | array1[array1>5] | [6,7,8,9] |

# #1.6 정렬

```
ex) org_array = np.array([3,1,9,5])
    array2d = np.array([[8,12],
                        [7,1]]
```

| 이름 | 설명 | 코드 예시 | 결과 예시 |
|---|---|---|---|
| np.sort() | 오름차순 정렬<br>* [::-1] 이면 내림차순 정렬 | np.sort(org_array)<br>np.sort(org_array)[::-1] | [1,3,5,9]<br>[9,5,3,1] |
| | | np.sort(array2d,axis=0) | [[7,1],<br>[8,12]] |
| ndarray.sort() | 오름차순 정렬 | org_array.sort() | [1,3,5,9] |
| ndarray.argsort() | 정렬된 행렬의 index 반환 | np.argsort(org_array)<br>np.argsort(org_array)[::-1] | [1,0,3,2]<br>[2,3,0,1] |

* np.sort() : 원행렬은 변하지 않음
  ndarray.sort() : 원행렬이 변함

```
sort_array1 = np.sort(org_array)
print(org_array)  # [3,1,9,5]
print(sort_array1)  # [1,3,5,9]
```

```
sort_array2 = org_array.sort()
print(sort_array2)  # None
print(org_array)  # [1,3,5,9]
```

EWHA EURON

# #1.6 정렬

* np.argsort()
  ex. 성적순으로 이름 정렬

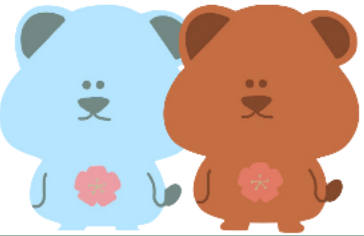
```
name_array = np.array(['John', 'Mike', 'Sarah','Kate','Samuel']
score_array = np.array([78,95,84,98,88])

sort_indices_asc = np.argsort(score_array)  # [0,2,4,1,3]
print(name_array[sort_indices_asc])  # 성적 오름차순으로 이름 정렬
```

# #1.7 선형대수 연산

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

| 이름 | 설명 | 코드 예시 | 결과 예시 |
|---|---|---|---|
| np.dot() | 행렬 내적 | np.dot(A,B) | [[58, 64],<br>[139, 154]] |
| np.transpose() | 전치 행렬 | np.transpose(A) | [[1,4],<br>[2,5],<br>[3,6]] |

# 판다스 part1



EWHA EURON

# #01 판다스 소개


- **데이터 처리를 위해 존재하는 가장 인기있는 라이브러리**
- 행과 열로 이뤄진 **2차원 데이터를 효율적으로 가공/처리**할 수 있는 다양하고 훌륭한 기능 제공
- RDBMS의 SQL이나 엑셀 시트에 버금가는 **고수준 API 제공**
- **파이썬의 리스트, 컬렉션, 넘파이 등 내부 데이터와 CSV 등의 파일을 쉽게 DataFrame으로 변경**해 데이터의 가공/분석을 편리하게 수행할 수 있게 해줌

# #01 판다스 소개 - DataFrame, Index, Series

## (1) Series

: 1차원 데이터, 칼럼이 하나인 구조체

| | |
|---|---|
| **이름** | 이화연 |
| **학교** | 이화여대 |
| **학과** | 컴퓨터공학과 |
| **학년** | 4 |

Index

데이터 값

## (2) DataFrame

: 2차원 데이터, 칼럼이 여러개인 구조체

| | **학생1** | **학생2** | **학생3** |
|---|---|---|---|
| **이름** | 이화연 | 최연수 | 김지우 |
| **학교** | 이화여대 | 이화여대 | 연세대 |
| **학과** | 컴퓨터공학과 | 수학과 | 국어국문학과 |
| **학년** | 4 | 2 | 4 |

Column 명

series series series

Index 명

EWHA EURON

# #02 판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API

## 1) 판다스 모듈 임포트

```
import pandas as pd
```

## 2) 파일을 DataFrame으로 로딩

| | |
|---|---|
| **read_csv( )** | - **csv(칼럼을 콤마로 구분한 파일 포맷) 파일 포맷 변환**<br>- read_csv( )의 인자인 sep에 구분 문자를 입력해서 설정 가능<br>**⇒ read_csv('파일명', sep='\t')**<br>- read_csv(filepath, sep=' ', …)에서 **filepath는 필수로 입력해야함**<br>- 별다른 파라미터 지정이 없으면 파일의 맨 처음 로우를 칼럼명으로 인지하고 칼럼으로 변환<br>- 모든 DataFrame 내의 데이터는 생성되는 순간 **고유의 Index** 값을 가지게 됨 |
| **read_table( )** | - 칼럼을 tab(\t)로 구분한 파일 포맷의 파일 포맷 변환<br>- 기능적으로 read_csv( )와 유사 |
| **read_fwf( )** | - 고정 길이 기반의 칼럼 포맷을 로딩 |

# #02 판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API

## 3) DataFrame 정보 확인

- **DataFrame.head( )**
  : DataFrame의 맨 앞 n개의 로우를 반환 (default: 5개)

```
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 |

- **DataFrame.shape**
  : DataFrame의 행과 열을 튜플 형태로 반환

```
print('DataFrame 크기: ', titanic_df.shape)
```

```
DataFrame 크기:  (891, 12)
```

- **DataFrame.info( )**
  : 총 데이터 건수와 데이터 타입, Null 건수를 알 수 있음

```
titanic_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

# #02 판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API

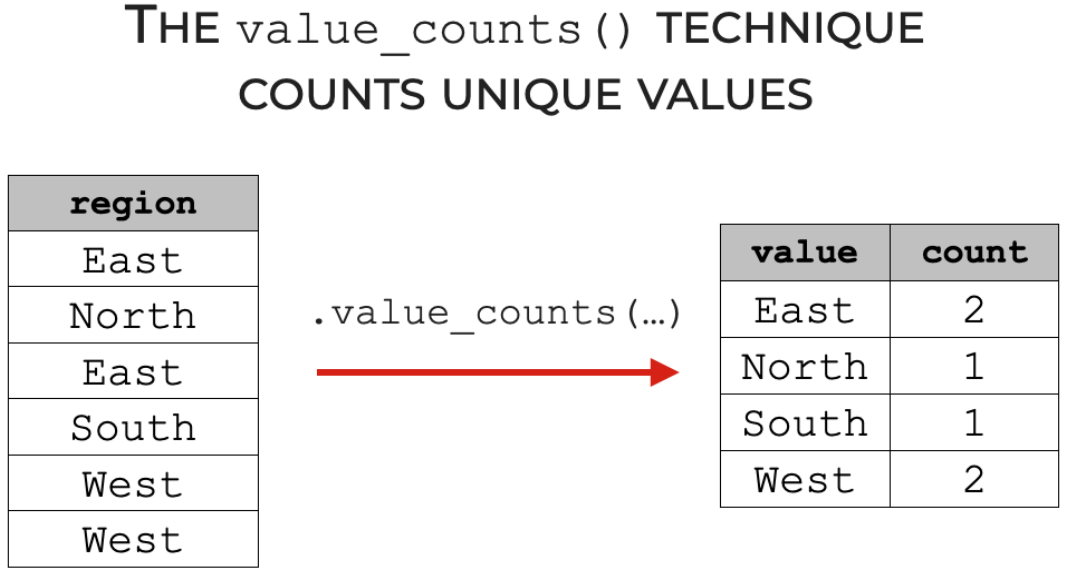
## 3) DataFrame 정보 확인

- **DataFrame.describe( )**
  : 칼럼별 숫자형 데이터값의 **n-percentile 분포도, 평균값, 최댓값, 최솟값**을 나타냄. 오직 **숫자형 칼럼**의 분포도만 조사
  ⇒ 숫자 형 칼럼에 대한 **개략적인 데이터 분포도**를 확인할 수 있음

- **DataFrame['열'].value_counts( )**
  : 칼럼값의 유형과 건수를 확인할 수 있음. Series 객체에서만 정의
  ⇒ **데이터의 분포도**를 확인하는 데 매우 유용한 함수.

```
value_counts = titanic_df['Pclass'].value_counts()
print(value_counts)
```

```
Pclass
3    491
1    216
2    184
Name: count, dtype: int64
```



# #03 DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

## ✓ 넘파이 ndarray, 리스트, 딕셔너리 → DataFrame

- DataFrame은 ndarray와 다르게 **칼럼명을 가지고 있음**

  => 일반적으로 DataFrame으로 변환할 때 **칼럼명을 지정**(지정하지 않으면 자동으로 할당)

- **data**(리스트/딕셔너리/ndarray 입력) + **columns**(칼럼명 리스트 입력) → DataFrame 생성
- **2차원 이하의 데이터들만 변환 가능**

# #03 DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

- **1차원 형태 데이터(리스트, ndarray) 변환**
  : pd.DataFrame(1차원 데이터 이름, columns = 칼럼 명 저장 리스트), 칼럼 명 1개만 필요

```
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 리스트로 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)
```

```
array1 shape:  (3,)
1차원 리스트로 만든 DataFrame:
    col1
0      1
1      2
2      3
1차원 ndarray로 만든 DataFrame:
    col1
0      1
1      2
2      3
```

# #03 DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

- **2차원 형태 데이터(리스트, ndarray) 변환**
  : pd.DataFrame(2차원 데이터 이름, columns = 칼럼 명 저장 리스트), 열의 수에 맞춰서 칼럼 수 필요

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

# 2행x3열 형태의 리스트와 ndarray 생성한 뒤 이를 DataFrame으로 변환
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)
```

```
array2 shape:  (2, 3)
2차원 리스트로 만든 DataFrame:
    col1  col2  col3
0     1     2     3
1    11    12    13
2차원 ndarray로 만든 DataFrame:
    col1  col2  col3
0     1     2     3
1    11    12    13
```

# #03 DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

- **딕셔너리 변환**
  : pd.DataFrame(딕셔너리 이름)

  키 → 칼럼명(문자열),
  값 → 키에 해당하는 칼럼 데이터(리스트/ndarray)

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

```
딕셔너리로 만든 DataFrame:
    col1  col2  col3
0     1     2     3
1    11    22    33
```

# #03 DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

## ✓ DataFrame → 넘파이 ndarray, 리스트, 딕셔너리

- **ndarray 변환**
  : 많은 머신러닝 패키지가 기본 데이터 형으로 넘파이 ndarray를 사용하기 때문에 빈번하게 변환
  ⇒ DataFrame 객체의 values를 이용해 쉽게 변환 가능

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

```
df_dict.values 타입:  <class 'numpy.ndarray'> df_dict.values shape:  (2, 3)
[[ 1  2  3]
 [11 22 33]]
```

# #03 DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

- **리스트 변환**
  : values로 얻은 ndarray에 tolist( ) 호출

- **딕셔너리 변환**
  : DataFrame 객체의 to_dict( )메서드 호출. 인자로 'list' 입력하면 딕셔너리의 값이 리스트형으로 반환

```
# DataFrame을 리스트로 변환
list3 = df_dict.values.tolist()
print('df_dict.value.tolist()타입: ', type(list3))
print(list3)

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

```
df_dict.value.tolist()타입:  <class 'list'>
[[1, 2, 3], [11, 22, 33]]

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

# #04 DataFrame 데이터 삭제

✓ **drop( )**

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

- **axis**
  : DataFrame의 로우를 삭제할 때는 axis=0, 칼럼을 삭제할 때는 axis=1
  ⇒ 주로 칼럼을 드롭. 로우는 이상치 데이터를 삭제하는 경우에 주로 사용

- **labels**
  : 삭제할 칼럼 명/인덱스 선택

  ⇒ axis =0 → labels: 인덱스 값
  DataFrame.drop(삭제할 인덱스 번호, axis=0)

  ⇒ axis =1 → labels: 칼럼 명
  DataFrame.drop(' 칼럼명' , axis=1)

| axis | labels |
|---|---|
| 0 => 로우 삭제 | 인덱스 값 |
| 1 => 칼럼 삭제 | 칼럼 명 |

# #04 DataFrame 데이터 삭제

✓ **drop( )**

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

- **inplace**
  : inplace=False면 자신의 DataFrame의 데이터는 삭제하지 않음,
  inplace=True면 자신의 DataFrame의 데이터 삭제

  ⇒ inplace = True일 때 반환 값을 자기 자신의 DataFrame 객체로 할당하면 안됨(None 반환)

- **여러 개의 칼럼 삭제**
  : 리스트 형태로 삭제하고자 하는 칼럼 명을 입력해 labels 파라미터로 입력

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

# #05 Index 개체

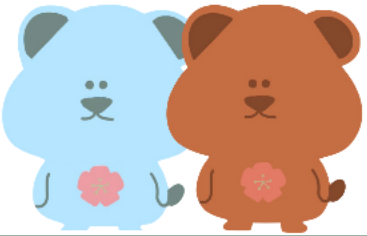
- **Index**
  : DataFrame, Series의 레코드를 **고유하게 식별하는 개체**

- **Index 개체 추출**
  : DataFrame.index, Series.index 속성 이용
  ⇒ 넘파이 1차원 ndarray로 볼 수 있음

- **Index 속성**
  1. ndarray와 유사하게 단일 값 반환 및 슬라이싱 가능함
  2. 한 번 만들어진 **Index 개체는 함부로 변경할 수 없음**
  3. Series 객체에 연산 함수를 적용할 때 **Index는 연산에서 제외됨. 식별용으로만 사용**

- **reset_index( )**
  : 새롭게 인덱스를 연속 숫자 형으로 할당, 기존 인덱스는 'index' 라는 새로운 칼럼명으로 추가
  ⇒ 인덱스가 연속된 숫자형 데이터가 아닐 경우에 다시 이를 연속 int 숫자형 데이터로 만들 때 주로 사용
  - Series에 reset_index( )를 적용하면 DataFrame이 반환

# 판다스 Part2



EWHA EURON

# #데이터 셀렉션 및 필터링

## 데이터 셀렉션 및 필터링 : 넘파이 vs 판다스

넘파이:
- 데이터 분석용으로 사용하기에는 편의성이 떨어짐
- '[ ]' 연산자 내 단일 값 추출, 슬라이싱, 팬시 인덱싱, 불린 인덱싱으로 데이터 추출
- '[ ]' 연산자는 행의 위치, 열의 위치, 슬라이싱 범위 등 지정해서 데이터 가져오기 가능

판다스:
- ix[ ], iloc[ ], loc[ ] 연산자로 데이터 추출
- DataFrame '[ ]' 안에는 칼럼명 문자 or 칼럼명의 리스트 객체, 인덱스로 변환 가능한 표현식 넣기 가능

-> DataFrame 뒤에 있는 [ ]는 '칼럼 지정 연산자 ' 로 이해하면 GOOD!

# #데이터 셀렉션 및 필터링

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

```
0    3
1    1
2    3
Name: Pclass, dtype: int64
```

['칼럼명']으로 '칼럼명'에 해당하는 칼럼 데이터 추출

```
print(titanic_df[['Survived','Pclass']].head(3))
```

| | Survived | Pclass |
|---|---|---|
| 0 | 0 | 3 |
| 1 | 1 | 1 |
| 2 | 1 | 3 |

['칼럼1', '칼럼2']로 여러 개의 칼럼 데이터 추출

```
titanic_df[0]
```

0은 칼럼명이 아니므로
오류 발생!

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

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked | Child_Adult |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | C000 | S | Adult |
| 1 | 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C | Adult |

인덱스로 표현 가능한 표현식은 입력 가능

EWHA EURON

# #데이터 셀렉션 및 필터링

**<DataFrame 뒤의 [ ] 연산자 혼동 방지 가이드>**

1. DataFrame 바로 뒤의 [ ] 연산자는 넘파이의 [ ]나 Series의 [ ]와 다름

2. DataFrame 바로 뒤의 [ ] 내 입력 값은 칼럼명(또는 칼럼의 리스트)을 지정해 칼럼 지정 연산에 사용하거나 불린 인덱스 용도로만 사용해야 함

3. DataFrame[0:2]와 같은 슬라이싱 연산으로 데이터를 추출하는 방법은 사용하지 않는 게 좋음

**명칭 기반 인덱싱과 위치 기반 인덱싱**

ix[ ] -> loc[ ], iloc[ ]

loc[ ] : 명칭 기반 인덱싱
iloc[ ] : 위치 기반 인덱싱

# #데이터 셀렉션 및 필터링

## iloc[ ] 연산자

행과 열 값으로 integer 또는 integer형의 슬라이싱, 팬시 리스트 값을 입력
(명확한 위치 기반 인덱싱이 사용되어야 해서 불린 인덱싱은 제공X)

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

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

:첫 번째 행, 첫 번째 열의 데이터를 추출

위치 인덱싱이 아니라 명칭을 입력하면 오류 발생! ex) data_df.iloc[0, 'Name' ]
문자열 인덱스를 행 위치에 입력해도 오류 발생! ex) data_df.iloc[ 'one' ,0]

# #데이터 셀렉션 및 필터링

## loc[ ] 연산자

행 위치에 DataFrame index 값, 열 위치에 칼럼 명 입력

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

```
'Chulmin'
```

:인덱스 값이 'one', 행의 칼럼 명이 'name' 인 데이터를 추출

| | old_index | Name | Year | Gender |
|---|---|---|---|---|
| 1 | one | Chulmin | 2011 | Male |
| 2 | two | Eunkyung | 2016 | Female |
| 3 | three | Jinwoong | 2015 | Male |
| 4 | four | Soobeom | 2015 | Male |

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

```
'Chulmin'
```

:인덱스가 숫자 형일 때의 데이터를 추출

인덱스 값이 아닌 숫자를 호출하면 오류 발생! ex) data_df_reset.loc[0, 'Name]'

# #데이터 셀렉션 및 필터링

loc[ ]에 슬라이싱 기호를 적용하면 종료 값-1이 아니라 종료 값까지 포함

```
data_df_reset.loc[1:2,'Name']
```

| | Name |
|---|---|
| 1 | Chulmin |
| 2 | Eunkyung |

dtype: object

VS

```
data_df_reset.iloc[1:2,1]
```

| | Name |
|---|---|
| 2 | Eunkyung |

dtype: object

# #데이터 셀렉션 및 필터링

## 불린 인덱싱

[ ], ix[ ], loc[ ]에 공통으로 지원 (iloc[ ]은 X)

and 조건일 때는 &
or 조건일 때는 |
Not 조건일 때는 ~

### 불린 인덱싱 적용

```
1  titanic_df[titanic_df['Age']>60][['Name','Age']].head(3)
```

| | Name | Age |
|---|---|---|
| 33 | Wheadon, Mr. Edward H | 66.0 |
| 54 | Ostby, Mr. Engelhart Cornelius | 65.0 |
| 96 | Goldschmidt, Mr. George B | 71.0 |

=

### loc[ ] 이용

```
1  titanic_df.loc[titanic_df['Age']>60,['Name','Age']].head(3)
```

| | Name | Age |
|---|---|---|
| 33 | Wheadon, Mr. Edward H | 66.0 |
| 54 | Ostby, Mr. Engelhart Cornelius | 65.0 |
| 96 | Goldschmidt, Mr. George B | 71.0 |

### 개별 조건 묶어서 복합 조건 연산자 사용

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

=

### 개별 조건을 변수에 할당

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

EWHA EURON

# #정렬, Aggregation 함수, GroupBy 적용

## DataFrame, Series의 정렬

sort_values( ) 매서드 이용
주요 입력 파라미터 by : 해당 칼럼으로 작업 수행
ascending : 오름차순, 내림차순
inplace : 호출한 DataFrame 유지 여부

내림차순으로 정렬

Pclass 와 Name으로 정렬

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

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked | Child_Adult |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **868** | 869 | 0 | 3 | van Melkebeke, Mr. Philemon | male | NaN | 0 | 0 | 345777 | 9.5 | C000 | S | Adult |
| **153** | 154 | 0 | 3 | van Billiard, Mr. Austin Blyler | male | 40.5 | 0 | 2 | A/5. 851 | 14.5 | C000 | S | Adult |
| **282** | 283 | 0 | 3 | de Pelsmaeker, Mr. Alfons | male | 16.0 | 0 | 0 | 345778 | 9.5 | C000 | S | Adult |

# #정렬, Aggregation 함수, GroupBy 적용

## Aggregation 함수 적용

min( ), max( ), sum( ), count( ) 등

```
1  titanic_df.count()
```

| | 0 |
|---|---|
| PassengerId | 891 |
| Survived | 891 |
| Pclass | 891 |
| Name | 891 |
| Sex | 891 |
| Age | 714 |
| SibSp | 891 |
| Parch | 891 |
| Ticket | 891 |
| Fare | 891 |
| Cabin | 891 |
| Embarked | 889 |
| Child_Adult | 891 |

dtype: int64

: 모든 칼럼에
count( ) 결과를 반환

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

| | 0 |
|---|---|
| Age | 29.699118 |
| Fare | 32.204208 |

dtype: float64

: 특정 칼럼에만
mean( ) 함수 적용

# # 정렬, Aggregation 함수, GroupBy 적용

## groupby( ) 적용

입력 파라미터 by : 대상 칼럼으로 groupby( )
DataFrame에 groupby( ) 호출하면 DataFrameGroupBy라는 DataFrame 반환

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

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

Aggregation 함수 호출하면 groupby( ) 대상 칼럼을 제외한 모든 칼럼에 함수 적용

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

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

# # 정렬, Aggregation 함수, GroupBy 적용

특정 칼럼만 aggregation 함수를 적용하려면 groupby( )로 반환된 DataFrameGroupBy 객체에 해당 칼럼을 필터링한 뒤 aggregation 함수를 적용

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

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

서로 다른 aggregation 함수를 적용할 경우, 함수 명을 agg( ) 내의 인자로 입력해서 사용

```
1  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 칼럼에 서로 다른 aggregation 함수를 적용하려면 칼럼들과 aggregation 함수를 딕셔너리 형태로 agg( ) 내에 입력

# #결손 데이터 처리하기

- isna( ) : 데이터의 NaN 여부 확인

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

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

+) sum( ) : 결손 데이터의 개수 확인

- fillna( ) : 결손 데이터를 다른 값으로 대체

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

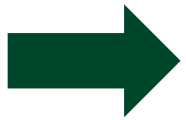
| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked | Child_Adult |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | C000 | S | Adult |
| **1** | 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C | Adult |
| **2** | 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26.0 | 0 | 0 | STON/O2. 3101282 | 7.9250 | C000 | S | Adult |

+) 반환 변숫값을 지정하지 않으면 inplace=True를  설정해야 실제 데이터 세트 값 변경!

# #apply lambda 식으로 데이터 가공

```
1  def get_square(a):
2      return a**2
3  print(get_square(3))
```

```
9
```



```
1  lambda_square =lambda x:x**2
2  print(lambda_square(3))
```

```
9
```

입력 인자

lambda **x** : **x**2**

입력 인자를 기반으로 한 계산식이며
호출 시 계산 결과가 반환됨

# #apply lambda 식으로 데이터 가공

map( ) : lambda 식을 이용할 때 여러 개의 값을 입력 인자로 사용해야 할 경우

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

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

# #apply lambda 식으로 데이터 가공

예시

- ‘Name’ 칼럼의 문자열 개수를 별도의 칼럼인 ‘Name_len’에 생성

```
1  titanic_df['Name_len']=titanic_df['Name'].apply(lambda x: len(x))
2  titanic_df[['Name','Name_len']].head(3)
```

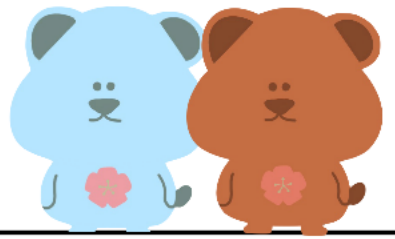
| | Name | Name_len |
|---|---|---|
| 0 | Braund, Mr. Owen Harris | 23 |
| 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | 51 |
| 2 | Heikkinen, Miss. Laina | 22 |

- 나이가 15세 미만이면 ‘Child’, 그 외는 ‘Adult’로 구분하는 칼럼 ‘Child_Adult

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

| | Age | Child_Adult |
|---|---|---|
| 0 | 22.0 | Adult |
| 1 | 38.0 | Adult |
| 2 | 26.0 | Adult |
| 3 | 35.0 | Adult |
| 4 | 35.0 | Adult |
| 5 | NaN | Adult |
| 6 | 54.0 | Adult |
| 7 | 2.0 | Child |

# THANK YOU



EWHA EURON