Week 1_예습과제_우정연

01. 머신러닝의 개념

- 머신러닝: 애플리케이션을 수정하지 않고도 데이터를 기반으로 패턴을 학습하고 결과를 예측하는 알고리즘 기법 통칭
    - 기존 소프트웨어 코드만으로 해결 어려웠던 문제점들을 머신러닝을 이요해 해결
- 머신러닝 기반의 예측 분석 (Predictive Analysis)
    - 데이터를 기반으로 숨겨진 패턴을 인지해 해결
    - 통계적 신뢰도를 강화, 다양한 수학적 기법을 적용
    - 데이터 내의 패턴을 스스로 인지하고 신뢰도 있는 예측 결과 도출

머신러닝의 분류

- 지도학습 (Supervised Learning)
    - 분류, 회귀, 추천 시스템, 시각/음성 감지/ 인지, 텍스트 분석, NLP
- 비지도학습 (Unsupervised Learning)
    - 클러스터링, 차원 축소, 강화학습
- 강화학습 (Reinforcement Learning)

데이터 전쟁

- 머신러닝 - 데이터에 매우 의존적

파이썬과 R 기반의 머신러닝 비교

02. 파이썬 머신러닝 생태계를 구성하는 주요 패키지

- 머신러닝 패키지
- 행렬/ 선형대수/ 통계 패키지
    - 넘파이

- 데이터 핸들링
  - 판다스
- 시각화
  - 맷플롯립
  - 시본(Seaborn)

03. 넘파이

- 머신러닝의 주요 알고리즘은 선형대수와 통계 등에 기반
- 넘파이(Numpy = Numerical Python) 사용
  - 루프를 사용하지 않고 대량 데이터의 배열 연산 가능
  - 빠른 배열 연산 속도
  - C/C++ 와의 호환 API 제공
  - 판다스가 더 편리

[넘파이 ndarray 개요]

- 넘파이의 기반 데이터 타임 - ndarray
  - 다차원 배열 쉽게 생성/ 다양한 연산 수행 가능
- array( ) 함수
  - 다양한 인자를 입력 받아서 ndarray로 변환하는 기능 수행
  - np.array() : ndarray로 변환을 원하는 객체를 인자로 입력하면 ndarray를 반환
  - 함수 인자로는 파이썬의 리스트 객체가 주로 사용
    - 리스트 [ ]는 1차원, [[ ]]는 2차원과 같이 배열의 차원과 크기 쉽게 표현 가능하기 때문
- shape 변수
  - ndarray의 크기 (행과 열의 수)를 튜플 형태로 가지고 있음
  - ndarray 배열의 차원을 알 수 있음
  - ndarray.shape : ndarray의 차원과 크기를 튜플(tuple)형태로 나타냄

ndarray의 데이터 타입

- ndarray내의 데이터 값
  - 숫자, 문자열, 불 값 등 모두 가능
- ndarray내의 데이터 타입
  - 그 연산의 특성상 같은 데이터 타입만 가능
- dtype 속성
  - ndarray내의 데이터 타입 확인 가능
  - e.g. array1.dtype
- list ↔ ndarray
  - list→ndarray 변경시 ndarray 내의 데이터 값은 모두 int32형
  - 다른 데이터 유형이 섞여 있는 list를 ndarray로 변경시 데이터 크기가 큰 데이터 타입으로 형 변환 일괄 적용
    - e.g. string > float > int
- astype( )
  - ndarray 내 데이터 값의 타입 변경시 이용
  - 인자로 원하는 타입을 문자열로 지정
    - e.g. array_int.astype('float64')
  - 메모리를 더 절약해야 할 때 보통 이용

ndarray를 편리하게 생성하기 - arange, zeros, ones

- 특정 크기와 차원을 가진 ndarray를 연속 값이나 0 또는 1로 초기화해 쉽게 생성해야 할 필요가 있는 경우
  - arange( ), zeros( ), ones( ) 사용
  - 주로 테스트용으로 데이터 만들거나 대규모 데이터를 초기화할 경우
- arange( ) 함수
  - 0부터 (함수 인자 값 - 1)까지의 값을 순차적으로 ndarray의 데이터 값으로 변환
    - e.g, np.arange(10)
  - default 함수 인자는 stop값
  - start 값도 부여해 0이 아닌 다른 값부터 시작 가능
    - e.g. np.arange(start=1, stop=10)

- zeros( ) 함수
  - 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 0으로 채운 해당 shape를 가진 ndarray를 반환
  - e.g, np.zeros((3,2), dtype='int32')
- ones( ) 함수
  - 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 1로 채운 해당 shape을 가진 ndarray 반환
  - 함수 인자로 dtype을 정하지 않으면 default로 float64 형의 데이터로 ndarray 채움
    - e.g, np.ones((3,2))

ndarray의 차원과 크기를 변경하는 reshape()

- reshape( )
  - ndarray를 특정 차원 및 크기로 변환
  - 함수 인자 - 변환을 원하는 크기 부여
    - e.g, array1.reshape(2, 5)
    - (row, column)
  - 지정된 사이즈로 변경 불가능하면 오류 발생
  - -1을 인자로 사용시 원래 ndarray와 호환되는 새로운 shape으로 변환
    - e.g, array2.reshape(-1, 5)
    - -1을 사용하더라도 호환될 수 없는 형태는 변환 불가능
- reshape(-1, 1)
  - 원본 ndarray가 어떤 형태라도 2차원
  - 여러 개의 row를 가지되 반드시 1개의 column을 가진 ndarray로 변환됨을 보장
  - 여러 개의 넘파이 ndarray는 stack이나 concat으로 결합할 때 각각의 ndarray의 형태를 통일해 유용하게 사용
- tolist( )
  - ndarray를 리스트 자료형으로 변환

[넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱 (indexing)]

1. **특정한 데이터만 추출**: 원하는 위치의 인덱스 값을 지정하면 해당 위치의 데이터가 반환됩니다.
2. **슬라이싱(Slicing)**: 슬라이싱은 연속된 인덱스상의 ndarray를 추출하는 방식입니다. ':' 기호 사이에 시작 인덱스와 종료 인덱스를 표시하면 시작 인덱스에서 종료 인덱스-1 위치에 있는 데이터의 ndarray를 반환합니다. 예를 들어 1:5라고 하면 시작 인덱스 1과 종료 인덱스 4까지에 해당하는 ndarray를 반환합니다.
3. **팬시 인덱싱(Fancy Indexing)**: 일정한 인덱싱 집합을 리스트 또는 ndarray 형태로 지정해 해당 위치에 있는 데이터의 ndarray를 반환합니다.
4. **불린 인덱싱(Boolean Indexing)**: 특정 조건에 해당하는지 여부인 True/False 값 인덱싱 집합을 기반으로 True에 해당하는 인덱스 위치에 있는 데이터의 ndarray를 반환합니다.

단일 값 추출

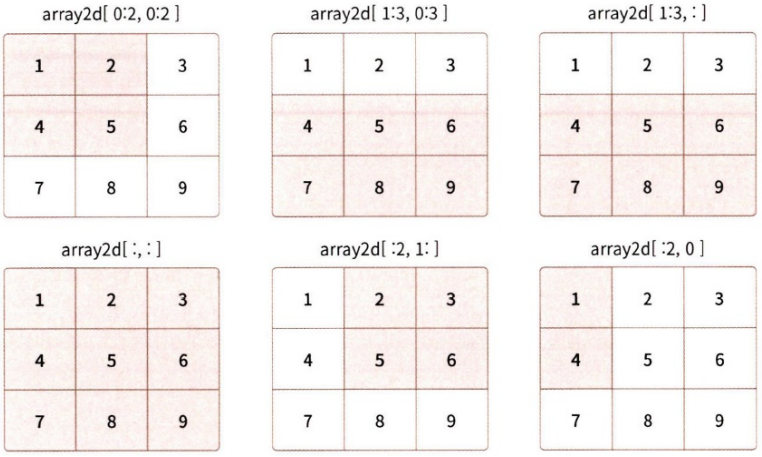
- 1개의 데이터 값 선택시
  - ndarray 객체에 해당하는 위치의 인덱스 값을 [ ] 안에 입력
  - 인덱스는 0부터 시작
  - e.g, array1[2]
    - array1[2]의 타입은 ndarray 타입이 아니고 ndaray내의 데이터 값 의미
  - 인덱스에 마이너스 기호 이용시 맨 뒤에서부터 데이터 추출 가능
  - 단일 인덱스를 이용해 ndarray 내의 데이터 값도 간단히 수정 가능
- 다차원 ndarray에서 단일 값 추출
  - e.g, array2d[row index, col index]
  - axis 0 - 로우 방향의 축
  - axis 1 - 칼럼 방향의 축

슬라이싱

- ':' 기호를 이용해 연속 데이터를 슬라이싱해서 추출 가능
- ndarray 타입
  - 단일 데이터 값 추출을 제외하고 슬라이싱, 팬시 인덱싱, 불린 인덱싱으로 추출된 데이터 세트는 모두 ndarray 타입
- ':' 사이에 시작 인덱스와 종료 인덱스를 표시

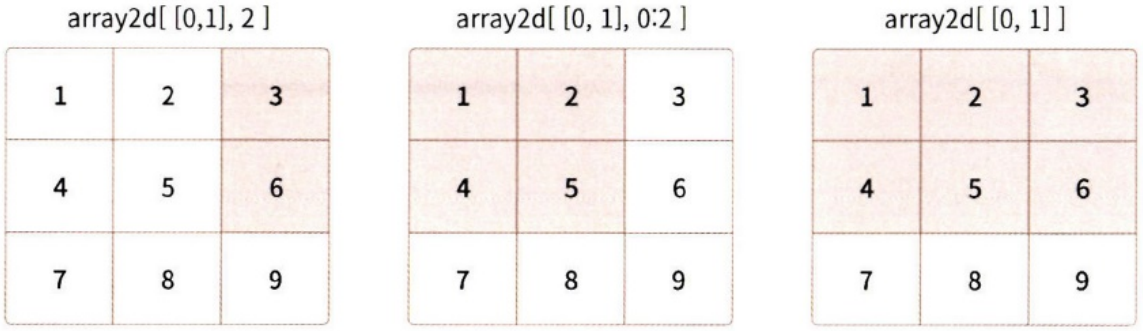
- 시작 인덱스에서 종료 인덱스-1 의 위치에 있는 데이터의 ndarray를 반환
- 시작/ 종료 인덱스 생략 가능

- 2차원 ndarray에서 슬라이싱
  - 콤마(,)로 로우와 칼럼 인덱스 지칭
  - 뒤에 오는 인덱스를 없애면 로우축(axis 0)의 첫번째 로우 1차원 ndarray를 반환



팬시 인덱싱(Fancy Indexing)

- 리스트나 ndarray로 인덱스 집합을 지정하면 해당 위치의 인덱스에 해당 ndarray 반환



불린 인덱싱(Boolean Indexing)

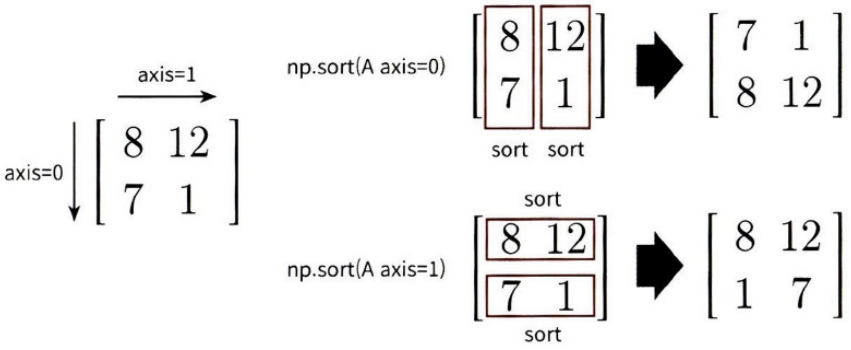
- 조건 필터링과 검색 동시에 가능
- ndarray의 인덱스를 지정하는 [ ] 내에 조건문을 그대로 기재
- 넘파이 ndarray 객체에 조건식 할당하면 True, False로 이루어진 ndarray 객체 반환 → False 값은 무시하고 True 값이 있는 위치 인덱스 값으로 자동 변환해 해당하는 인덱스 위치의 데이터만 반환

- Step 1: array1d 〉5와 같이 ndarray의 필터링 조건을 [ ] 안에 기재
- Step 2: False 값은 무시하고 True 값에 해당하는 인덱스값만 저장(유의해야 할 사항은 True값 자체인 1을 저장하는 것이 아니라 True값을 가진 인덱스를 저장한다는 것입니다)
- Step 3: 저장된 인덱스 데이터 세트로 ndarray 조회

[행렬의 정렬 - sort()와 argsort()]

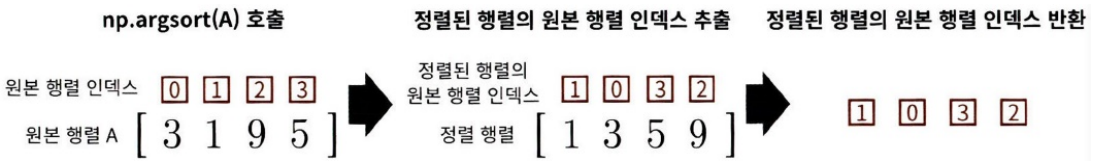
행렬 정렬

- np.sort( ) - 넘파이에서 sort( )를 호출하는 방식
  - 원행렬은 그대로 유지한 채 정렬된 행렬을 반환
- ndarray.sort( ) - 행렬 자체에서 sort( )를 호출하는 방식
  - 원행렬 자체를 정렬한 형태로 변환
  - 반환값은 None
- 오름차순 기본
- [::-1] - 내림차순 정렬
  - e.g. np.sort()[::-1]
- 2차원 이상의 행렬의 경우
  - axis 축 값 설정을 통해 로우 방향 또는 칼럼 방향으로 정렬 수행 가능



정렬된 행렬의 인덱스를 반환하기

- np.argsort( )
  - 원본 행렬이 정렬되었을 때 기존 원본 행렬의 원소에 대한 인덱스를 필요로 할 경우
  - 정렬 행렬의 원본 행렬 인덱스를 ndarray형으로 반환



- 내림차순 → np.argsort()[::-1]
- 메타 데이터를 가질 수 없는 넘파이에서 활용도 높음 - 넘파이의 데이터 추출에서 많이 사용

[선형대수 연산 - 행렬 내적과 전치 행렬 구하기]

행렬 내적(행렬 곱)

- np.dot( )
  - 두 행렬의 내적은 왼쪽 행렬의 로우와 오른쪽 행렬의 칼럼의 원소들을 순차적으로 곱한 뒤 그 결과를 모두 더한 값

전치 행렬(Transpose)

- 전치 행렬: 원행렬에서 행과 열의 위치를 교환한 원소로 구성한 행렬
- transpose( )

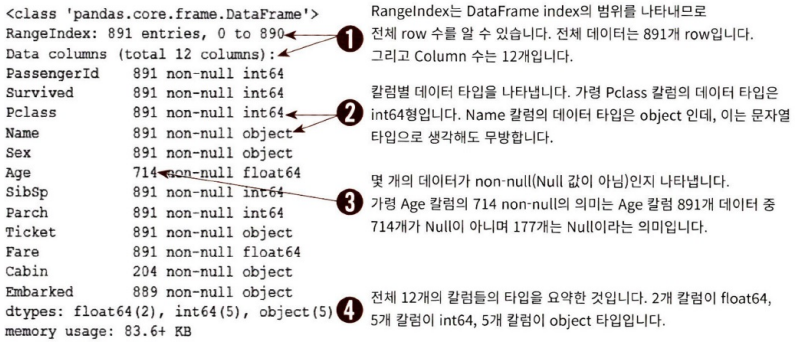
04. 데이터 핸들링 - 판다스

[판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API]

- 파이썬에서 데이터 처리를 위해 존재하는 가장 인기 있는 라이브러리
- 대부분의 데이터 세트는 2차원 데이터
- 판다스의 핵심 객체: DataFrame
  - 여러 개의 행과 열로 이루어진 2차원 데이터를 담는 데이터 구조체
- Index
  - 개별 데이터를 고유하게 식별하는 key값
  - Series와 DataFrame은 모두 Index를 key 값으로 가지고 있음
- Series: Index와 단 하나의 칼럼으로 구성된 데이터 구조체/ DataFrame: 칼럼이 여러 개인 데이터 구조체, 여러개의 Series로 이루어짐

판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API

- import pandas as pd
- read_csv( ), read_table( ), read_fwf( ) 등으로 된 파일을 DataFrame으로 로딩 가능
  - read_table( ) - 필드 구분 문자: 탭('\t')
  - read_csv( ) - 콤마(',')
  - read_fwf( ) : 고정 길이 기반의 칼럼 포맷을 로딩
- read_csv( )
  - filepath: 가장 중요한 인자 - 로드하려는 데이터 파일의 경로를 포함한 파일명 입력
- pd.read_csv( )
  - 호출 시 파일명 인자로 들어온 파일을 로딩해 DataFrame 객체로 반환
  - 별다른 파라미터 지정 없으면 파일의 맨 처음 로우를 칼럼면으로 인지하고 칼럼으로 변환
  - 맨 왼쪽 로우: 판다스의 Index 객체 값
    - 모든 DataFrame 내의 데이터는 생성되는 순간 고유의 Index 값을 가짐
- DataFrame.head( )
  - DataFrame의 맨 앞에 있는 N 개의 로우 반환
  - Default는 5개
- DataFrame.shape( )
  - DataFrame의 행과 열을 튜플 형태로 반환
- DataFrame.info( )
  - 총 데이터 건수와 데이터 타입, Null 건수를 알 수 있음



- DataFrame.describe( )

    - 칼럼별 숫자형 데이터값의 n-percentile 분포도, 평균값, 최댓값, 최솟값을 나타냄
        - count = Not Null인 데이터 건수
        - mean = 전체 데이터의 평균값
        - std = 표준편차
        - min/max = 최소/ 최댓값
        - 25%/50%/75% = 25/50/75 percentile값
    - int, float 등의 숫자형 칼럼의 분포도만 조사
    - 자동으로 object 타입의 칼럼은 출력에서 제외
    - 개략적인 수준의 분포도 확인 가능 - 유용
- DataFrame의 [ ] 연산자 내부에 칼럼명의 입력하면 Series형태로 특정 칼럼 데이터 세트가 반환됨
- value_count( )
    - 지정된 칼럼의 데이터값 건수를 반환
    - 데이터 분포도 확인시 유용한 함수

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

【Output】

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

    - Series 객체에서만 정의됨
    - 인덱스는 단순히 순차 값과 같은 의미 없는 식별자만 할당하는 것이 아님
    - 고유성이 보장된다면 의미있는 데이터 값 할당도 가능

[DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환]

- DataFrame ↔ 리스트, 딕셔너리 ndarray

넘파이 ndarray, 리스트, 딕셔너리를 DataFrame으로 변환하기

- DataFrame은 리스트와 넘파이 ndarray와 달리 칼럼명을 가지고 있음
  - 상대적으로 편하게 데이터 핸들링 가능
- DataFrame으로 변환 시 칼럼명 지정 (or 자동으로 할당)
- 리스트, 넘파이ndarray (1D) → DataFrame (2D)
  - 칼럼명 한 개만 필요

```
import numpy as np

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)
```

- 리스트, 넘파이ndarray (2D) → DataFrame (2D)
  - 2행 3열 형태의 리스트, ndarray → DataFrame
    - 칼럼명 3개 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)
```

- 딕셔너리 → DataFrame
  - 딕셔너리의 키(Key)는 칼럼명, 딕셔너리의 값(Value)는 키에 해당하는 칼럼 데이터로 변환됨
  - 키의 경우 문자열, 값의 경우 리스트(또는 ndarray) 형태로 딕셔너리 구성

```
# 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)
```

DataFrame을 넘파이 ndarray, 리스트, 딕셔너리로 변환하기

- DataFrame 객체의 values를 이용하여 ndarray로 변환

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

- DataFrame을 리스트와 딕셔너리로 변환
  - 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 df_dict.to_dict() 타입:', type(dict3))
print(dict3)
```

[DataFrame의 칼럼 데이터 세트 생성과 수정]

- [ ] 연산자를 이용
  - DataFrame[ ] 내에 새로운 칼럼명을 입력하고 값을 할당
    - DataFrame[ ] = 0 과 같이 Series에 상숫값을 할당하면 Series의 모든 데이터 세트에 일괄 적용
  - 업데이트를 원하는 칼럼 Series를 DataFrame[ ] 내에 칼럼 명으로 입력한 뒤에 값 할당
    - 기존 칼럼 값 일괄적으로 업데이트 가능

[DataFrame 데이터 삭제]

- drop( ) 메서드 이용

```
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=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)

- axis 파라미터
  - axis 값에 따라 특정 칼럼 또는 특정 행을 드롭
    - axis 0 : 로우 방향 축
    - axis 1 : 칼럼 방향 축
- axis = 1 입력시 칼럼 축 방향으로 드롭 수행 - 칼럼 드롭의 의미
  - labels에 원하는 칼럼명 입력 후 axis = 1을 입력하면 지정된 칼럼 드롭
  - 기존 칼럼 값을 가공해 새로운 칼럼을 만들고 삭제하는 경우
  - axis = 0 입력시 로우 축 방향으로 드롭 수행 - 특정 로우 드롭의 의미
    - axis를 0으로 지정시 DataFrame은 자동으로 labels에 오는 값을 인덱스로 간주 - DataFrame의 특정 로우를 가리키는 것은 인덱스이기 때문
    - 이상치 데이터를 삭제하는 경우 주로 사용
- inplace 파라미터
  - inplace = False (default)
    - 자신의 DataFrame의 데이터는 삭제하지 않음
    - 삭제된 경과 DataFrame을 반환
  - inplace = True
    - 자신의 DataFame 데이터 삭제

        - 반환 값이 None이 됨 → 반환 값을 다시 자신의 DataFrame 객체로 할당하면 안됨

- labels 파라미터
    - 여러 개의 칼럼 삭제 시 리스트 형태로 labels 파라미터에 칼럼명 입력

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

[Index 객체]

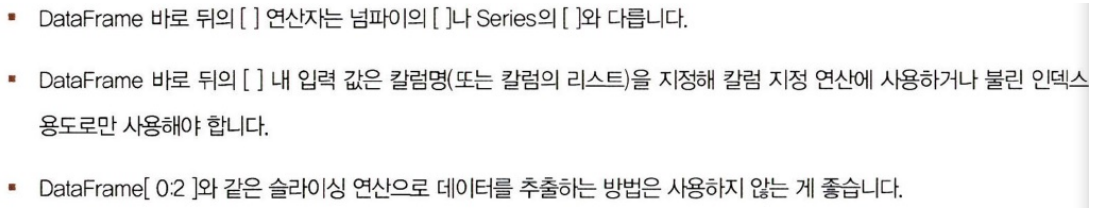
- 판다스의 Index 객체는 DataFrame, Series의 레코드를 고유하게 식별하는 객체
- DataFrame.index // Series.index 속성을 통해 Index 객체 추출 가능
- Index 객체
    - 식별성 데이터를 1차원 array로 가짐
    - ndarray와 유사하게 단일 값 반환 및 슬라이싱 가능
    - 한 번 만들어진 DataFrame 및 Series의 Index 객체는 변경 불가 - 오직 식별용으로만 사용됨
- reset_index( )
    - 새롭게 인덱스를 연속 숫자 형으로 할당
    - 기존 인덱스는 'index'라는 새로운 칼럼 명으로 추가
    - 인덱스가 연속된 int 숫자형 데이처가 아닐 경우 연속 int 숫자형 데이터로 만들 때 주로 사용
    - Series에 reset_index( )를 적용하면 Series가 아닌 DataFrame이 반환됨 - 기존 인덱스가 칼럼으로 추가돼 칼럼이 2개가 되기 때문
    - parameter 중 drop = True 로 설정하면 기존 인덱스는 새로운 칼럼으로 추가되지 않고 삭제(drop)됨 → Series 그대로 유지

[데이터 셀렉션 및 필터링]

- 넘파이: [ ] 연산자 내 단일 값 추출, 슬라이싱, 팬시 인덱싱, 불린 인덱싱을 통해 데이터를 추출
- 판다스: ix[ ], iloc[ ], loc[ ] 연산자를 통해 동일한 작업 수행

DataFrame의 [ ] 연산자

- 넘파이: 행의 위치, 열의 위치, 슬라이싱 범위 등을 지정해 데이터 가져올 수 있음
- DataFrame 바로 뒤의 [ ] 는 칼럼 명 문자(또는 칼럼 명의 리스트 객체), 또는 인덱스로 변환 가능한 표현식만 들어갈 수 있음
    - 칼럼 지정 연산자 로 이해하기
    - 숫자 값을 입력할 경우 오류 발생
    - 판다스의 인덱스 형태로 변환 가능한 표현식은 [ ] 내에 입력 가능
        - e.g. titanic_df [0:2] 가능
    - 불린 인덱싱 표현 가능



DataFrame ix[ ] 연산자

- ix[ ] 연산자의 칼럼 명칭(labels) 기반 인덱싱
- ix[ ] 연산자의 칼럼 위치(position) 기반 인덱싱
- 두 가지 방식을 모두 제공 → 혼돈을 주어 현재 판다스에서 사라짐(deprecated)

명칭 기반 인덱싱과 위치 기반 인덱싱의 구분

- 명칭(label) 기반 인덱싱
    - 칼럼의 명칭을 기반으로 위치를 지정하는 방식
    - '칼럼 명' 같이 명칭으로 열 위치 지정
- 위치(position) 기반 인덱싱
    - 0을 출발점으로 하는 가로축, 세로축 좌표 기반의 행과 열 위치를 기반으로 데이터 지정
    - 행, 열 값으로 정수가 입력됨
- DataFrame의 인덱스 값은 명칭 기반 인덱싱
    - ix[ ] 의 경우 명칭과 위치 기반 인덱싱 모두 허용 - 코드 작성에 혼선 초래 우려 있음 → loc[ ]과 iloc[ ] 연산자 도입

DataFrame iloc[ ] 연산자

- iloc[ ]
  - 위치 기반 인덱싱 → 행과 열 값으로 integer 또는 integer형의 슬라이싱, 팬시 리스트 값을 입력해야 함
    - e.g. data_df.iloc[0,0]
  - 위치 아닌 명칭 입력시 오류 발생
  - 명확한 위치 기반 인덱싱이 사용되어야 하는 제약 → 불린 인덱싱은 제공하지 않음

DataFrame loc[ ] 연산자

- loc[ ]
  - 명칭 기반 인덱싱 → 행 위치에는 DataFrame index 값을, 열 위치에는 칼럼 명을 입력
    - index가 숫자형일 경우도 있으므로 무조건 문자열을 입력하는 것은 아님
  - loc[ ]에 슬라이싱 기호 ':' 적용시
    - 일반적: '시작값:종료값' → 시작값~종료값-1 범위 의미
    - loc[ ]: 시작값~종료값(까지 포함) - 명칭 기반 인덱싱의 특성

1. 가장 중요한 것은 명칭 기반 인덱싱과 위치 기반 인덱싱의 차이를 이해하는 것입니다. DataFrame의 인덱스나 칼럼명으로 데이터에 접근하는 것은 명칭 기반 인덱싱입니다. 0부터 시작하는 행, 열의 위치 좌표에만 의존하는 것이 위치 기반 인덱싱입니다.
2. ix[ ]는 명칭 기반 인덱싱과 위치 기반 인덱싱을 모두 적용할 수 있습니다. DataFrame의 인덱스가 숫자 형일 경우 행 위치에 오는 숫자는 위치 기반 인덱싱이 아니라 명칭 기반 인덱싱의 DataFrame 인덱스를 가리킵니다.
3. iloc[ ]는 위치 기반 인덱싱만 가능합니다. 따라서 행과 열 위치 값으로 정수형 값을 지정해 원하는 데이터를 반환합니다.
4. loc[ ]는 명칭 기반 인덱싱만 가능합니다. 따라서 행 위치에 DataFrame 인덱스가 오며, 열 위치에는 칼럼 명을 지정해 원하는 데이터를 반환합니다.
5. 명칭 기반 인덱싱에서 슬라이싱을 '시작점:종료점'으로 지정할 때 시작점에서 종료점을 포함한 위치에 있는 데이터를 반환합니다.

불린 인덱싱

- 처음부터 가져올 값을 조건으로 [ ] 내에 입력하면 자동으로원하는 값을 필터링함
- iloc[ ] 는 불린 인덱싱 지원되지 않음
- 예시

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

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

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

- 복합 조건 결합 가능

1. and 조건일 때는 &
2. or 조건일 때는 |
3. Not 조건일 때는 ~

  - 개별 조건은 ( )으로 묶고, 복합 조건 연산자 사용
    - 예시

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

  - 개별 조건을 변수에 할당하고 변수를 결합해서 불린인덱싱 수행 가능
    - 예시

```
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( )

- sort_values( )
  - 주요 입력 파라미터: by, ascending, inplace
    - by로 특정 칼럼 입력 시 해당 칼럼으로 정렬 수행

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

    - ascending = True 설정시 오름차순으로 정렬 (기본)/ ascending = False 설정시 내림차순으로 정렬

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

- inplace = False (기본) 설정시 sort_values( )를 호출한 DataFrame은 그대로 유지하며 정렬된 DataFrame을 결과로 반환
- inplace = True 설정시 호출한 DataFrame의 정렬 결과를 그대로 적용

Aggregation 함수 적용

- DataFrame에서 바로 aggregation을 호출할 경우 모든 칼럼에 해당 aggregation을 적용
  - 예시

```
titanic_df.count()
```

- 특정 칼럼에 aggregation 함수를 적용하기 위해서는 DataFrame에 대상 칼럼들만 추출해 aggregation을 적용
  - 예시

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

groupby( ) 적용

- 입력 파라미터 by에 칼럼을 입력하면 대상 칼럼으로 groupby 됨
- DataFrame에 groupby( )를 호출하면 DataFrameGroupBy라는 또 다른 형태의 DataFrame을 반환함
- DataFrame에 groupby( )를 호출해 반환된 결과에 aggregation함수를 호출하면 groupby( ) 대상 칼럼을 제외한 모든 칼럼에 해당 aggregation 함수를 적용함
  - 예시

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

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

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

- 여러 개의 aggregation 함수명을 DataFrameGroupBy 객체의 agg( ) 내에 인자로 입력해서 사용

- 예시

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

- agg( ) 내에 입력 값으로 딕셔너리 형태로 aggregation이 적용될 칼럼들과 aggregation함수 입력
  - 예시

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

[결손 데이터 처리하기]

- 결손 데이터: 칼럼에 값이 없는 NULL의 경우 → 넘파이의 NaN으로 표시
  - NaN 값은 머신러닝 알고리즘이 처리하지 않아 다른 값으로 대체 필요
  - 평균, 총합 등의 함수 연산 시 제외됨
- isna( )
  - NaN 여부를 확인하는 API
- fillna( )
  - NaN 값을 다른 값으로 대체하는 API

isna( )로 결손 데이터 여부 확인

- DataFrame.isna( )
  - 모든 칼럼의 값이 NaN인지 아닌지를 True나 False로 알려줌
- DataFrame.isna( ).sum( )
  - True는 내부적으로 숫자 1로, False는 숫자 0으로 변환
  - 결손 데이터의 개수를 구할 수 있음

fillna( )로 결손 데이터 대체하기

- 결손 데이터 NaN 값을 편리하게 다른 값으로 대체 가능
  - 예시

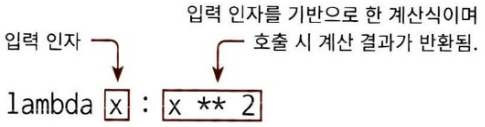
```
titanic_df['Cabin'] = titanic_df['Cabin'].fillna('C000')
```

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

[apply lambda 식으로 데이터 가공]

- 판다스는 apply 함수에 lambda 식을 결합해 DataFrame이나 Series의 레코드별로 데이터를 가공하는 기능 제공
- lambda 식
  - 파이썬에서 함수형 프로그래밍 지원 위해 만듬
  - 함수의 선언과 함수 내의 처리를 한 줄의 식으로 쉽게 변환하는 식
  - ':' 로 입력 인자와 반환될 입력 인자의 계산식을 분리

```
lambda_square = lambda x : x ** 2
```



    - ':' 의 왼쪽에 있는 x는 입력 인자
    - 오른쪽은 입력 인자의 계산식 = 반환 값
  - 여러 개의 값을 입력 인자로 사용해야 할 경우에는 map( ) 함수를 결합하여 사용

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

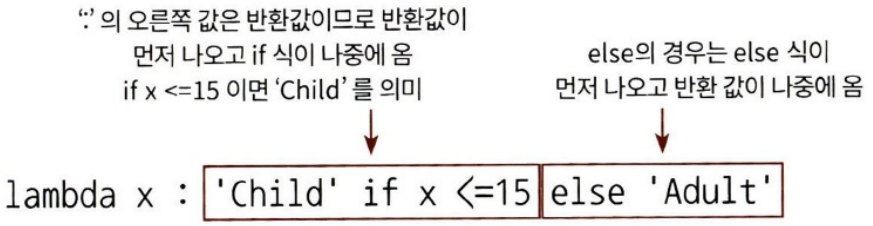
【Output】

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

- 판다스 DataFrame의 lambda 식
  - 예시

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

  - if 절의 경우 if 식보다 반환 값을 먼저 기술



- else if 지원 X
  - else 절을 ( )로 내포해 ( ) 내에서 다시 if else 적용해 사용

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

- else를 계속 내포해서 쓰기 부담스러운 경우
  - 별도의 함수 생성

05. 정리

- 넘파이, 판다스
- 시각화 패키지
  - 맷플롯립(Matplotlib)과 시본(Seaborn)