1장 파이썬 기반의 머신러닝과 생태계 이해

Chapter	1
날짜	@2025년 9월 9일

01. 머신러닝의 개념

머신러닝의 개념 ⭐

- 정의: 애플리케이션을 수정하지 않고도, 데이터를 기반으로 패턴을 학습하고 결과를 예측하는 알고리즘 기법
- 머신러닝은 데이터를 기반으로 숨겨진 패턴을 인지해 문제 해결
- 데이터 기반 + 수학적 기법 → 통계적인 신뢰도 강화 & 예측 오류 최소화
- Predictive Analysis 머신러닝 기반의 예측 분석

머신러닝의 분류 ⭐

: 지도학습(Supervised Learning) / 비지도학습(Un-supervised Learning) / 강화학습(Reinforcement Learning)

- **지도학습 - 분류, 회귀**, 추천시스템, 시각/음성 감지/인지, 텍스트 분석, NLP
- **비지도학습 - 클러스터링, 차원 축소, 강화학습**

데이터 전쟁

- 머신러닝 단점) 데이터에 의존적 Garbage In, Garbage Out
  → 좋은 품질의 데이터 필요
- 머신러닝 모델 개선) 최적의 머신러닝 알고리즘, 모델 파라미터 구축, 데이터 이해, 효율적으로 가공-처리-추출

파이썬 vs R 기반 머신러닝 비교

= 오픈 소스 프로그램

- R: 통계 전용
- Python: 직관적인 문법, 객체지향 함수형 프로그래밍 - 유연한 프로그램 아키텍처, 다양한 라이브러리 + 딥러닝 프레임워크인 텐서플로(TensorFlow), 케라스(Keras), 파이토치(PyTorch) 등에서 파이썬 우선 정책으로 python 지원

---

02. 파이썬 머신러닝 생태계 - 주요 패키지

- 머신러닝 패키지: Scikit-Learn 데이터 마이닝 기반
- 행렬/선형대수/통계 패키지 - Numpy(행렬 기반), SciPy

- 데이터 핸들링 - Pandas(2차원 데이터 처리 특화)
- 시각화 - matplotlib / seaborn(함축적)

+) IPython tool: Jupyter Notebook

파이썬 머신러닝을 위한 S/W 설치

- Anaconda 설치 (패키지 일괄 설치)

---

03. 넘파이 (numpy)

= Numerical Python

- 머신러닝의 주요 알고리즘은 선형대수 & 통계 기반
- 선형대수 기반의 프로그램을 쉽게 만들 수 있도록 지원하는 대표 패키지
- 루프 사용 x / 대량 데이터의 배열 연산 가능 → 빠른 배열 연산 속도
- C/C++과 같은 저수준 언어 기반의 호환 API 제공

넘파이 ndarray 개요

```
#넘파이 모듈 임포트
import numpy as np
```

- 넘파이의 기반 데이터 타임은 ndarray
- 넘파이에서 다차원(multi-dinemsion) 배열을 쉽게 생성 & 연산 수행

〈 넘파이 ndarray 배열의 차원들 〉

```
#array() 함수: ndarray로 변환
#shape 변수: ndarray의 크기 (행, 열) tuple

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

print('array1 type：', type(array1))
print('array 1 array 형태: ', array1.shape) #1차원 array, 3개 데이터


array2 = np.array([[1, 2, 3], [2, 3, 4]])

print('array2 type：  ', type(array2))
```

```
print('array2 array 형태 : ', array2.shape) #2차원 array, 2개의 row, 3개의 column


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

print('array3 type : ', type(array3))
print('array3 array 형태: ', array3.shape) #2차원 array, 1개의 row, 2개의 column
```

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

ndarray의 데이터 타입

- **ndarray 내 데이터값**

  숫자 값, 문자열 값, 불 값 모두 가능

  - int형 (8bit, 16bit, 32bit)
  - unsigned int형 (8bit, 16bit, 32bit)
  - float형 (16bit, 32bit, 64bit, 128bit)
  - complex 타입 (더 큰 숫자값이나 정밀도 필요 시)
- **특징**
  - ndarray 내의 데이터 타입은 모두 동일해야 함
  - `dtype` 속성으로 확인 가능

- **리스트 → ndarray 변환**
  - 리스트는 서로 다른 데이터 타입 허용하지만, ndarray는 **동일 타입만 허용**
  - 서로 다른 타입이 섞여 있는 리스트를 ndarray로 변환 시 **더 큰 데이터 타입으로 일괄 변환**
- **데이터 타입 변환 예시**
  - **int형 + string형 섞인 경우**
    - int값들이 문자열로 변환
    - 예: `[1, 2, "3"]` → 모두 문자열(유니코드)
  - **int형 + float형 섞인 경우**
    - int값들이 float으로 변환
    - 예: `[1, 2, 3.5]` → 모두 `float64`
- **astype( ) 메서드 ...** ndarray 내 데이터 타입 변경

```
#astype( ) 메서드 -> ndarray 내 데이터 타입 변경 가능

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)
```

ndarray 편리하게 생성하기 - arange, zeros, ones

```
# arange()
# 0부터 함수 인자 값 -1까지의 값을 순차적으로 ndarray의 데이터값으로 변환

sequence_array = np.arange(10)

print(sequence_array)
print(sequence_array.dtype, sequence_array.shape)
```

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

print(zero_array)
print(zero_array.dtype, zero_array.shape)

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

ndarray 의 차원과 크기를 변경하는 reshape()

- reshape() 메서드는 ndarray를 특정 차원 및 크기로 변환

```
array1 = np.arange(10)
print('array1 ：\n', array1)

array2 = array1.reshape(2, 5)
print('array2：\n', array2)
```

```
array3 = array1.reshape(5, 2)
print('array3 : \n', array3)
```

```
arrayl = np.arange(10)
print(array1)

array2 = arrayl.reshape(-1, 5)
print('array2 shape : ', array2.shape)

arrays = arrayl.reshape(5, -1)
print('array3 shape : ', array3.shape)
```

```
arrayl = np.arange(8)
array3d = arrayl.reshape((2, 2, 2))
print('array3d : \n', array3d.tolist()) #list로 변환

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

# 1 차원 ndarray를 2차원 ndarray로 변환
array6 = arrayl.reshape(-1, 1)
print('array6 : \n', array6.tolist())
print('array6 shape : ', array6.shape)
```

넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱(Indexing)

1️⃣ 단일 값 추출

- **1차원 ndarray**
  - 인덱스는 0부터 시작
  - 음수 인덱스: 뒤에서부터 선택 (-1: 마지막, -2: 뒤에서 두 번째)
  - 단일 데이터 추출 시 반환값은 **ndarray가 아닌 데이터 타입**
- **데이터 수정 가능**

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

- **다차원 ndarray**

```
#다차원 ndarray에서 단일값 추출 (2차원)

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

- row → axis 0 (행 방향), col → axis 1 (열 방향)
- 3차원 이상: axis 0, axis 1, axis 2 ...

2️⃣ 슬라이싱(Slicing)

- 연속된 데이터 슬라이싱해서 추출 → ndarray 타입
- **시작 인덱스, 종료 인덱스 표시**

```
array[start:stop]
```

  - `start` 생략 → 0부터 시작
  - `stop` 생략 → 마지막까지
  - `start`와 `stop` 모두 생략 → 전체 ndarray 반환
- **2차원 ndarray**

```
array2d[row_start:row_stop, col_start:col_stop]
```

  - 한쪽 축에만 슬라이싱, 다른 축은 단일 값 인덱스 가능
  - 뒤의 인덱스 생략 시: 차원 축 단축 (예: 2차원 → 1차원 반환)
- 슬라이싱으로 반환된 데이터는 항상 **ndarray 타입**

3️⃣ 팬시 인덱싱(Fancy Indexing)

- **리스트 또는 ndarray 형태의 인덱스 집합** 지정 가능

```
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('array2d[[0, 1], 0 : 2] ⇒', array4.tolist())
```

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

- **여러 위치를 동시에 선택 가능**

4️⃣ **불린 인덱싱(Boolean Indexing)**

- **조건 필터링과 데이터 선택** 동시 가능
- ex) 1차원 ndarray에서 5보다 큰 값 선택, 조건문 작성

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

# [ ] 안에 array 1d > 5 Boolean indexing을 적용
array3 = array1d[array1d > 5]
print('array 1d > 5 불린 인덱싱 결과 값 :', array3)
```

  - Step:
    1. 조건식 `array1 > 5` 필터링 조건을 [ ] 안에
       → True/False ndarray 생성
    2. False 무시, True 위치 인덱스만 저장
    3. 저장된 인덱스 위치 데이터만 반환 → 데이터세트로 ndarray 조

- 장점: **for loop/if 문 없이 간단 구현**
- ex)

```
array1 = np.array([1,2,3,4,5,6,7,8,9])
array1[array1 > 5]  # → [6,7,8,9]
```

행렬의 정렬 — sort()와 argsort()

넘파이에서 행렬을 정렬하고, 정렬된 원소의 인덱스를 얻는 방법

1️⃣ **행렬 정렬**

- **np.sort()**
  - 원본 행렬을 **그대로 유지**하며 **정렬된 행렬을 반환**
  - 기본: **오름차순 정렬**
  - 내림차순: `np.sort(array)[::-1]`
- **ndarray.sort()**
  - 원본 행렬 자체를 **정렬된 상태로 변경**
  - 반환값: `None`
  - 기본: **오름차순**

    - 내림차순: `array.sort(); array[::-1]`
- **2차원 이상 ndarray**
    - `axis` 값으로 정렬 방향 설정 가능
        - `axis=0` → 행 방향
        - `axis=1` → 열 방향

2️⃣ 정렬된 행렬의 인덱스를 반환하기

- `np.argsort(array)`
    - **원본 배열을 기준으로 정렬된 인덱스**를 ndarray로 반환
    - 내림차순: `np.argsort(array)[::-1]`
- **활용 포인트**
    - 넘파이는 DataFrame과 달리 **메타데이터를 가지지 않음**
    - 원본 값과 메타데이터를 분리하여 관리해야 함
    - `argsort()`와 **팬시 인덱싱**을 통해 정렬된 메타데이터 활용 가능

```
#예제
import numpy as np

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)
print('성적 오름차순 정렬 시 score_array의 인덱스:', sort_indices_asc)
print('성적 오름차순으로 name_array의 이름 출력:', name_array[sort_indices_asc])
```

선형대수 연산 - 행렬 내적과 전치 행렬 구하기

1️⃣ 행렬 내적(행렬 곱)

- **개념**
    - 두 행렬 A, B의 내적 = `np.dot(A, B)`
    - **왼쪽 행렬의 행(row) × 오른쪽 행렬의 열(column)**
    - 각 원소를 곱하고 더해 결과 행렬 생성
- **조건**
    - 왼쪽 행렬의 열 개수 = 오른쪽 행렬의 행 개수

```
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)
```

2️⃣ 전치 행렬

- **개념**
  : 원 행렬에서 **행과 열 위치를 교환**한 행렬

```
A = np.array([[1, 2],
 [3, 4]])

transpose_mat = np.transpose(A)
print ('A 의 전치 행렬:\n', transpose_mat)
```

---

04. 데이터 핸들링 - 판다스 (pandas)

Pandas

- 데이터 핸들링 프레임워크
- **핵심 객체: DataFrame** 여러 개의 행과 열로 이뤄진 2차원 데이터 담는 데이터 구조체
- **Index:** 개별 데이터를 고유하게 식별하는 Key값
- **Series**: 칼럼이 하나뿐인 데이터 구조체
  → **DataFrame:** 칼럼이 여러 개인 구조체 / 여러 개의 Series로 이뤄짐

판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API

```
import pandas as pd
```

1. 파일 로딩

판다스는 다양한 포맷을 **DataFrame**으로 불러올 수 있음.

```
read_csv(filepath_or_buffer, sep=',',...)
```

- `read_csv()` : CSV 파일 (기본적으로 콤마 구분)
- `read_table()` : 탭(`\t`) 구분 파일
- `read_fwf()` : 고정 길이 칼럼 포맷

→ `sep` 인자를 이용해 구분자 지정 가능

2. DataFrame 기본 확인

- `head(n)` : 앞에서 n개 행 반환 (default=5)

- `shape` : (행 개수, 열 개수) 반환
- `info()` : 데이터 건수, 타입, Null 개수 확인 (메타데이)
- `describe()` : 숫자형 칼럼의 기초 통계값 (평균, 표준편차, 최솟값, 분위수 등)

3. 인덱스 (Index)

- DataFrame은 생성 시 자동으로 **Index**를 가짐.
- RDBMS의 PK(Primary Key)처럼 각 행을 고유하게 식별.

4. 값 분포 확인

특정 칼럼의 값 분포 확인은 `value_counts()` 사용 (카테고리형 변)

```
#데이터 분포도 확인
value_counts = titanic_df['Pclass'].value_counts()
print(value_counts)
```

DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

- DataFrame은 다양한 데이터 구조(리스트, 딕셔너리, 넘파이 ndarray)로부터 생성 가능
- 반대로 DataFrame을 다른 구조로 변환할 수도 있음
- 특히 사이킷런(sklearn)은 주로 **ndarray** 입력을 요구하므로 변환이 자주 필요!

넘파이 ndarray, 리스트, 딕셔너리 ⇒ DataFrame으로 변환하기

(1) 리스트, ndarray → DataFrame

- **1차원 데이터**: 칼럼 1개 필요 → `columns=['col1']`
- **2차원 데이터**: 행렬 구조 반영 → 칼럼 수와 맞게 `columns` 지정

(2) 딕셔너리 → DataFrame

- **Key → 칼럼명**
- **Value → 칼럼 데이터 (리스트/ndarray 형태)**

DataFrame ⇒ 넘파이 ndarray, 리스트, 딕셔너리로 변환하기

(1) DataFrame → ndarray

```
df.values
df.to_numpy()
```

(2) DataFrame → 리스트

```
df.values.tolist()
```

(3) DataFrame → 딕셔너리

- 다양한 `orient` 옵션 제공
  - `dict` (기본값): {컬럼명: 값 리스트}
  - `list`: [{컬럼1:값, 컬럼2:값}, ...]

DataFrame의 칼럼 데이터 세트 생성과 수정

- **새로운 칼럼 생성**
  - `DataFrame['새칼럼명'] = 값` 형식으로 쉽게 추가 가능
  - 예: `titanic_df['Age_0'] = 0` → 모든 값이 0으로 채워진 칼럼 추가
  - 넘파이 ndarray에 상숫값을 할당하면 전체에 적용되는 것과 같은 원리
- **기존 칼럼 활용해 새로운 칼럼 생성**
  - 기존 칼럼 데이터를 가공해 새로운 칼럼 추가 가능
  - 예: `Age_by_10`, `Family_No` 등을 기존 데이터로부터 계산해 추가
- **기존 칼럼 값 일괄 수정**
  - 특정 칼럼을 지정 후 값을 다시 할당하면 전체 값이 업데이트됨
  - 예: `titanic_df['Age_by_10'] = titanic_df['Age_by_10'] + 100`

    → 기존 값에 +100이 적용되어 일괄 업데이트됨

DataFrame 데이터 삭제

- **drop() 메서드 기본 구조**
  - `labels`: 삭제할 행/열 이름
  - `axis`: 0 → 행(row), 1 → 열(column)
  - `inplace`: False → 원본 유지, 삭제된 결과 반환(기본값)

    True → 원본에서 직접 삭제

1. 칼럼 삭제 (axis=1)

- `titanic_df.drop('Age_0', axis=1)` → 'Age_0' 칼럼이 삭제된 **새로운 DataFrame 반환**
- 여러 칼럼 삭제: 리스트 형태로 전달

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

→ 원본 `titanic_df`에서 바로 세 칼럼이 삭제됨

2. 행 삭제 (axis=0)

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

→ 인덱스 0, 1, 2에 해당하는 로우가 원본에서 삭제됨

3. 정리

- **axis=0** → 행 삭제, **axis=1** → 열 삭제
- **inplace=False(기본값)** → 원본 유지, 삭제된 결과 반환
- **inplace=True** → 원본에서 직접 삭제, 반환값은 `None`

Index 객체

1. 정의

- **Index 객체**: DataFrame, Series의 레코드를 고유하게 식별하는 객체
- RDBMS의 **Primary Key(PK)**와 유사한 역할

2. 특징

- 추출: `DataFrame.index`, `Series.index`
- 실제 값: **1차원 ndarray** 형태 (`.values`로 확인 가능)
- ndarray와 유사 → 단일 값 반환, 슬라이싱 가능
- **변경 불가**: 한 번 생성된 Index는 임의 변경 불가능
- **연산 제외**: Series 연산 시 Index는 제외되고, 데이터 값만 연산

3. reset_index() 메서드

- 인덱스를 새롭게 **연속 숫자형**으로 재할당
- 기존 인덱스 → `index`라는 칼럼으로 추가됨
- **Series에 적용 시**: DataFrame으로 변환됨 (기존 인덱스가 칼럼으로 추가되므로 칼럼 2개가 됨)

4. drop 옵션

- `reset_index(drop=True)`
  - 기존 인덱스를 칼럼으로 추가하지 않고 삭제
  - **Series 그대로 유지**

데이터 셀렉션 및 필터링

- 판다스는 넘파이와 유사하지만, **DataFrame/Series마다 셀렉션 방식이 다름**
  넘파이: `[ ]` 안에서 **단일 값 추출, 슬라이싱, 팬시 인덱싱, 불린 인덱싱**
- 판다스: `iloc[]`, `loc[]`로 같은 기능 수행

DataFrame의 [ ] 연산자

- 넘파이의 `[ ]`와 가장 큰 차이점이 있음
- *DataFrame 뒤의 `[ ]`는 "칼럼 지정 연산자"**라고 이해하는 것이 안전
- 사용 가능
  - 단일 칼럼: `df['컬럼명']`
  - 여러 칼럼: `df[['컬럼1', '컬럼2']]`
  - 불린 인덱싱: `df[df['컬럼명'] 조건]`

- 사용 불가 (오류 발생)
  - 정수 인덱스: `df[0]`, `df[[0,1,2]]`
  - 넘파이식 행 지정 → 안 됨

DataFrame iloc[ ] 연산자

- 판다스는 두 가지 인덱싱 제공
  - **iloc[ ]** → 위치(Location) 기반 인덱싱
  - **loc[ ]** → 명칭(Label) 기반 인덱싱
- 과거 `ix[ ]`가 있었지만 혼동 때문에 폐지
- **iloc특징**
  - **정수 인덱스**만 허용 (행/열의 위치 좌표)
    → 불린 인덱싱 허용 X
  - 사용 형태

    ```
    df.iloc[행번호, 열번호]
    ```
- **잘못된 사용 (오류 발생)**:
  - 칼럼명 입력 → X
  - 인덱스 값(예: 'one') 입력 → X

iloc[ ] 연산 유형	설명 및 반환 값
data_df.iloc[1,0]	두번째 행의 첫번째 열 위치에 있는 단일 값 반환 반환 값: 'Eunkyung'
data_df.iloc[2,1]	세번째 행의 두번째 열 위치에 있는 단일 값 반환 반환 값: '2015'
data_df.iloc[0:2, [0,1]]	0:2 슬라이싱 범위의 첫번째에서 두번째 행과 첫번째, 두번째 열에 해당하는 DataFrame 반환 반환 값: Name Year one Chulmin 2011 two Eunkyung 2016
data_df.iloc[0:2, 0:3]	0:2 슬라이싱 범위의 첫번째에서 두번째 행의 0:3 슬라이싱 범위의 첫번째부터 세번째 열 범위에 해당하는 DataFrame 반환 반환 값: Name Year Gender one Chulmin 2011 Male two Eunkyung 2016 Female
data_df.iloc[:]	전체 DataFrame 반환 Name Year Gender one Chulmin 2011 Male two Eunkyung 2016 Female three Jinwoong 2015 Male four Soobeom 2015 Male
data_df.iloc[:, :]	전체 DataFrame 반환 Name Year Gender one Chulmin 2011 Male two Eunkyung 2016 Female three Jinwoong 2015 Male four Soobeom 2015 Male

DataFrame loc[] 연산자

- **명칭(Label) 기반 인덱싱** 방식
  - **행 위치** → DataFrame의 인덱스 값
  - **열 위치** → 칼럼명
- 기본 형식:

```
df.loc[행_인덱스, 열_칼럼명]
```

- 예: 인덱스가 `'one'`인 행에서 `'Name'` 칼럼 데이터 추출

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

- **특징**
  1. **행 위치는 인덱스 값**

- DataFrame의 인덱스 값을 명칭으로 사용.
- 숫자형 인덱스라도 위치가 아니라 명칭으로 판단.
- 잘못된 인덱스 값을 넣으면 오류 발생.

2. **열 위치는 칼럼명**
   - 직관적으로 이해 가능.
3. **슬라이싱 사용 시 주의**
   - 일반 슬라이싱과 다르게 **종료값 포함**

     예: `df.loc['one':'two']` → 'one'과 'two' 모두 포함
   - 숫자형 인덱스에서도 명칭 기반이므로 종료값 포함 방식이 적용

아래는 loc[ ] 연산의 다양한 수행 사례입니다.

loc[ ] 연산 유형	설명 및 반환 값
`data_df.loc['three', 'Name']`	인덱스 값 three인 행의 Name 칼럼의 단일값 반환 반환 값: 'Jinwoong'
`data_df.loc['one':'two', ['Name', 'Year']]`	인덱스 값 one부터 two까지 행의 Name과 Year 칼럼에 해당하는 DataFrame 반환 반환 값: Name Year one Chulmin 2011 two Eunkyung 2016
`data_df.loc['one':'three', 'Name':'Gender']`	인덱스 값 one부터 three까지 행의 Name부터 Gender 칼럼까지의 DataFrame 반환 반환 값: Name Year Gender one Chulmin 2011 Male two Eunkyung 2016 Female three Jinwoong 2015 Male
`data_df.loc[:]`	모든 데이터 값: Name Year Gender one Chulmin 2011 Male two Eunkyung 2016 Female three Jinwoong 2015 Male four Soobeom 2015 Male
`data_df.loc[data_df.Year >= 2014]`	iloc[ ]와 다르게 loc[ ]는 불린 인덱싱이 가능. Year 칼럼의 값이 2014 이상인 모든 데이터를 불린 인덱싱으로 추출 Name Year Gender two Eunkyung 2016 Female three Jinwoong 2015 Male four Soobeom 2015 Male

*** 정리 ***

DataFrame 인덱싱/셀렉션 비교

연산자	인덱싱 방식	행	열	슬라이싱 특징	주의/설명
`[ ]`	칼럼 지정 또는 불린 인덱싱	X	칼럼명 또는 칼럼명 리스트	슬라이싱 가능하지만 권장하지 않음	DataFrame 뒤 `[ ]`는 **칼럼 지정용** 또는 **불린 인덱싱용**으로만 사용
`iloc[]`	위치 기반	0부터 시작하는 정수	0부터 시작하는 정수	슬라이싱: 종료값 제외, 팬시 인덱싱 가능, -1 사용 가능	행/열 위치 값으로 **정수만 가능**, 불린 인덱싱 불가
`loc[]`	명칭 기반	DataFrame 인덱스 값	칼럼명	슬라이싱: 종료값 포함	행 위치에 **정확한 인덱스 값**, 열 위치에 **칼럼명** 필요

Tips

- 단일/여러 칼럼 선택만 필요하면 `[ ]`만으로 충분
- 행과 열을 함께 선택하려면 `iloc[]` 또는 `loc[]` 사용
- `iloc[]`는 위치 기반 → 정수 좌표 사용
- `loc[]`는 명칭 기반 → 인덱스/칼럼명 사용, 슬라이싱 종료값 포함

불린 인덱싱

- 개념
  - 불린 인덱싱은 **조건식을 이용해 원하는 데이터를 필터링**
  - `[ ]`와 `loc[]`에서 사용 가능하며, `iloc[]`에서는 지원되지 X
  - 조건식의 결과는 True/False로 반환되며, True인 행만 추출
- 복합 조건 - 여러 조건 결합 가능
  1. AND 조건: `&`
  2. OR 조건: `|`
  3. NOT 조건: `~`
  - 조건식은 반드시 `()`로 묶어야 함
- 특징
  - 다재다능하고 유연하며, 복잡한 조건도 쉽게 적용 가능
  - `[ ]`, `loc[]`에서 동일하게 사용 가능
  - `iloc[]`에서는 불린 인덱싱 사용 불가

정렬, Aggregation 흗h수% GroupBy 적용

1. DataFrame / Series 정렬 - `sort_values()`

- **기능**: 특정 칼럼 기준으로 데이터를 오름차순/내림차순 정렬

- **주요 파라미터**:
  - `by`: 정렬 기준 칼럼명
  - `ascending`: `True`(오름차순, 기본값), `False`(내림차순)
  - `inplace`: `False`(기본값, 반환된 DataFrame 사용), `True`(원본 DataFrame 적용)
- **예시**:
  - 단일 칼럼 기준: 오름차순 정렬

    ```
    titanic_df.sort_values(by='Name')
    ```

  - 여러 칼럼 기준: 내림차순 정렬

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

2. Aggregation 함수 적용

- **주요 함수**: `min()`, `max()`, `sum()`, `count()` 등
- **특징**:
  - DataFrame에서 바로 호출하면 모든 칼럼에 적용
  - `count()`는 **NaN 값은 제외**
- **특정 칼럼만 적용**:

```
titanic_df['Age'].max()
titanic_df[['Age', 'Fare']].sum()
```

3. `groupby()` 적용

- **기능**: 특정 칼럼 기준으로 데이터를 그룹화
- **반환 객체**: `DataFrameGroupBy` (별도의 DataFrame 아님)
- **사용법**:
  - 단일 칼럼 기준 그룹화:

    ```
    group_obj = titanic_df.groupby('Pclass
    ```

  - 그룹화 후 aggregation:

    ```
    group_obj['Age'].max()
    ```

- **특정 칼럼만 aggregation**:

```
group_obj[['PassengerId', 'Survived']].count()
```

- **여러 aggregation 함수 적용**:

```
group_obj['Age'].agg(['max', 'min'
```

- **칼럼별 서로 다른 aggregation 함수 적용**:

```
group_obj.agg({'Age': 'max', 'Fare': 'sum', 'SibSp': 'mean'})
```

- **SQL과 차이점**:
  - SQL: `SELECT max(Age), sum(SibSp) FROM titanic_table GROUP BY Pclass`
  - Pandas: `groupby()` + `agg()`로 구현, 칼럼별 다른 함수 적용 시 딕셔너리 사용 필요
  - 반환 객체가 `DataFrameGroupBy`이므로 aggregation 전에는 실제 값이 아님

결손 데이터 처리하기

- 값이 없는 데이터를 결손 데이터라고 하며, Pandas에서는 **NaN**으로 표시
- 머신러닝 알고리즘은 NaN 값을 처리하지 못하므로 다른 값으로 대체 필요
- NaN 값은 평균, 총합 등의 연산 시 자동으로 제외됨

Step1. `isna()` 로 결손 데이터여부 확인

- 데이터가 NaN인지 여부를 True/False로 반환

```
titanic_df.isna()
```

- **결손 데이터 개수 확인**:

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

  - True → 1, False → 0으로 계산되어 칼럼별 NaN 개수를 알 수 있음

Step2. `fillna()` 로 결손 데이터 대체하기

- NaN 값을 지정한 값으로 대체
- **사용법**:

```
# 반환값을 다시 받는 방식
titanic_df['Cabin'] = titanic_df['Cabin'].fillna('C000')

# inplace=True 사용 방식
titanic_df['Cabin'].fillna('C000', inplace=True)
```

- **주의**:
  - 반환값을 다시 변수에 할당하지 않으면 실제 DataFrame은 변경되지 않음

- `inplace=True`를 사용하면 반환값 없이도 원본 DataFrame이 변경됨

---

apply lambda 식으로 데이터 가공

1. 개념

- `apply()` 함수에 **lambda 식**을 결합하면 DataFrame 또는 Series의 **각 레코드별 데이터 가공**이가능
- 칼럼 단위 일괄 처리보다 속도는 느릴 수 있지만, 복잡한 가공 시 필요
- Lambda 식: **한 줄로 함수 정의 + 반환 가능**

```
# 일반 함수
def get_square(a):
    return a ** 2

# lambda 식으로 변환
lambda x: x ** 2
```

2. 여러 입력값 처리

- 여러 값을 입력으로 받을 때는 보통 **map()**과 결합