[베이직 반] 판다스 (3)

📎 **실습 데이터**

https://drive.google.com/file/d/1QUDuZ-hCMi4Y4yuCmQeNuD-xof0WPGgc/view?usp=sharing

---

데이터 전처리(판다스) (3)

0) 학습 목표

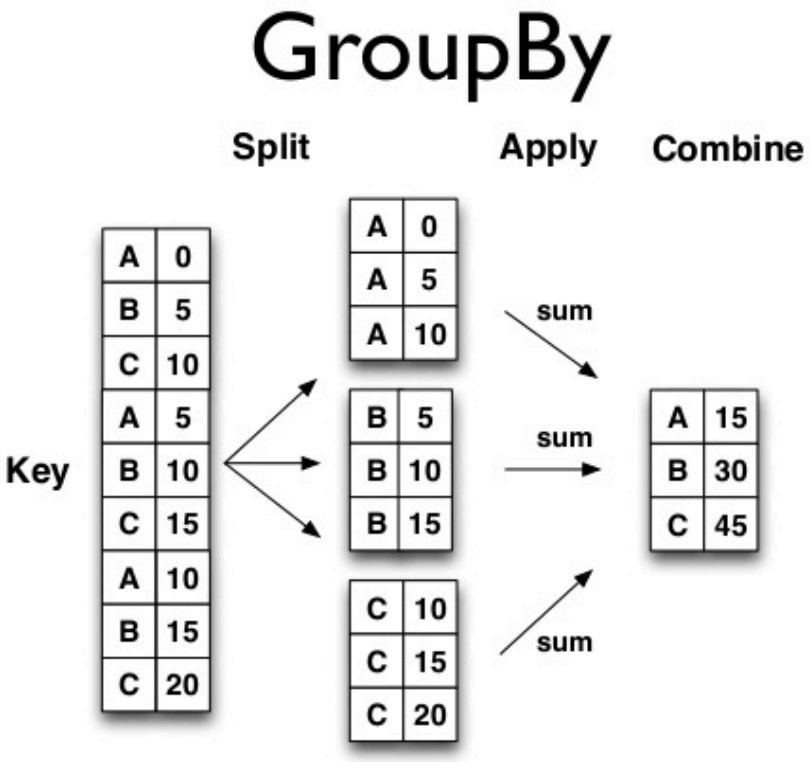
Titanic 승객 데이터(`train.csv`)를 활용하여 Pandas의 핵심 데이터 다루기 기법 5가지 학습

- **그룹화 및 Aggregation:** 데이터를 그룹별로 묶어 집계값(합계, 평균 등)을 계산하기 (`groupby`)
- **문자열 처리:** 데이터프레임에서 문자열 데이터를 다루는 기법 (`str` 액세서 활용)
- **시간 데이터 처리:** 날짜/시간 문자열을 datetime 타입으로 변환하고 다루기 (`pd.to_datetime`, `dt` 액세서 활용)

- **데이터 결합:** 여러 데이터프레임을 합치는 방법 (Merge/Join을 통한 열 병합, Concat을 통한 행/열 연결)
- **apply/map/lambda 활용:** 행이나 열 단위로 함수를 적용하거나 (`apply`), 매핑 사전으로 값 변환하기 (`map`), 그리고 `lambda` 함수를 활용한 데이터 처리

---

1) 그룹화 및 Aggregation



- 그룹화는 데이터를 특정 기준 열의 값에 따라 **묶는 것**을 의미. 예를 들어, 반별 시험 점수 데이터에서 반별 평균 점수를 구할 때 사용.
- Aggregation(집계)는 그룹별로 모아진 데이터에 대해 **합계(sum), 평균(mean), 개수(count)** 등의 요약 통계를 계산하는 것.
- Pandas에서는 `DataFrame.groupby(기준열)` 메서드를 사용해 데이터를 그룹화하고, 그 결과에 **집계 함수**를 적용하여 결과를 얻는다. 자주 쓰이는 집계 함수로는 `.mean()`, `.sum()`, `.count()`, `.max()`, `.min()` 등이 있다.
- 한 개 이상의 열을 그룹 기준으로 사용할 수 있다. (예: `df.groupby(['열1', '열2'])`)

- **집계 결과**는 보통 **Series 또는 DataFrame** 형태로 반환된다. 예를 들어 그룹별 평균을 구하면 각 그룹 값(index)과 그 평균이 담긴 Series가 나온다.
- 한 번에 여러 통계를 구하고 싶다면 `.agg()` 메서드를 사용하여 각 열에 대해 여러 함수를 적용할 수도 있다.

간단한 예제

```
import pandas as pd

# 예시 데이터프레임: 팀 별 점수
data = {'Team': ['Red', 'Blue', 'Red', 'Blue'],
        'Score': [5, 3, 8, 7]}
df = pd.DataFrame(data)
print("원본 데이터프레임:\n", df)
```

- **그룹별 합계:** `Team` 별로 점수를 묶은 후 합계를 계산하려면 아래와 같이 사용
  - `df.groupby('Team')['Score'].sum()` : Team 값별로 Score를 합산한 Series 반환

```
# 'Team'별 'Score' 합계
grouped_sum = df.groupby('Team')['Score'].sum()
print("팀별 점수 합계:\n", grouped_sum)
```

- **그룹별 평균:** 마찬가지로 `.mean()`을 사용하면 팀별 평균 점수를 구할 수 있음

```
grouped_mean = df.groupby('Team')['Score'].mean()
print("팀별 점수 평균:\n", grouped_mean)
```

- **다중 집계:** `.agg()`를 사용하여 여러 함수를 한꺼번에 적용 가능. 예를 들어, 점수의 평균과 합계를 모두 구하기:

```
multi_agg = df.groupby('Team')['Score'].agg(['mean', 'sum'])
print("팀별 점수 평균과 합계:\n", multi_agg)
```

Titanic 데이터 실습

- **예제 1: 객실등급(Pclass)에 따른 평균 요금(Fare)**

▼ **정답**

```
# Pclass별 Fare 평균 계산
class_fare_mean = titanic_df.groupby('Pclass')['Fare'].mean()
print("객실등급별 평균 요금:\n", class_fare_mean)
```

- **예제 2: 성별(Sex)에 따른 생존률** – **남성**과 **여성** 승객의 **생존률(생존자 비율)**을 비교해 봅시다.
- (힌트: Survived 열의 평균값은 생존자 비율과 같습니다. 0과 1로 구성된 값을 평균내면 1의 비율이 나오기 때문입니다.)

▼ **정답**

```
# 성별로 그룹화하여 생존률(mean of Survived) 계산
survival_rate_by_gender = titanic_df.groupby('Sex')['Survived'].mean()
print("성별 생존률:\n", survival_rate_by_gender)
```

- **예제 3: 복수 기준 그룹화** – 객실등급(Pclass)과 **성별(Sex)** 두 가지 기준으로 그룹을 나눈 뒤 **생존률**을 계산해보겠습니다. (예: 1등실 남성, 1등실 여성, 2등실 남성, ... 각 그룹별 생존률)

▼ **정답**

```
# Pclass와 Sex 두 열로 그룹화하여 생존률 계산
survival_rate_by_class_gender = titanic_df.groupby(['Pclass', 'Sex'])['Survived'].mean()
print("객실등급별 성별 생존률:\n", survival_rate_by_class_gender)
```

1. **퀴즈 1:** 각 **탑승항구(Embarked)** 별로 **생존률**은 어떻게 달랐을까요? Titanic 데이터프레임에서 Embarked 값(S, C, Q)에 따라 그룹화하고 생존률(Survived 열 평균)을 계산하는 코드를 작성해보세요.

2. **퀴즈 2: 객실등급(Pclass)별 최고령 승객의 나이**는 얼마일까요? Pclass별로 그룹화하여 **나이(Age)**의 **최댓값**을 구하는 코드를 작성해보세요.

▼ **정답**

```
# 퀴즈 1 정답: 탑승항구별 생존률
survival_rate_by_embark = titanic_df.groupby('Embarked')['Survived'].
mean()
print("탑승항구별 생존률:\n", survival_rate_by_embark)

# 퀴즈 2 정답: 객실등급별 최대 나이
max_age_by_class = titanic_df.groupby('Pclass')['Age'].max()
print("객실등급별 최대 나이:\n", max_age_by_class)
```

---

2) 문자열 처리

- **문자열 데이터 처리**는 매우 중요합니다. 이름, 주소, 코드 등 **텍스트 형태 데이터**에서 필요한 정보를 추출하거나 형식을 변환하는 작업이 잦습니다.
- Pandas에서는 문자열 전용 메서드를 사용하기 위해 `Series.str` 속성을 제공합니다.
- `df['컬럼'].str.메서드()` 형식으로 사용하며, Python의 문자열 메서드들을 Series 전체에 적용할 수 있습니다.
- 자주 사용하는 문자열 처리 기능:
  - **대소문자 변환:** `.lower()`, `.upper()`, `.capitalize()`, `.title()` 등
  - **공백 제거:** `.strip()` (문자열 양 끝 공백 제거), `.lstrip()`, `.rstrip()`
  - **길이 계산:** `.len()` 각 문자열의 길이 반환
  - **검색/포함 여부:** `.contains("문자열")` 부분 문자열 포함 여부 (True/False 시리즈 반환), `.startswith()`, `.endswith()`
  - **치환:** `.replace("기존", "새로운")` 문자열 치환 (정규표현식 사용 가능)
  - **분할:** `.split("구분자")` 구분자를 기준으로 문자열을 나누어 리스트로 반환; 이후 `[index]`로 각 부분 추출 가능
  - 이 외에도 `.find()`, `.isdigit()`, `.join()` 등 다양한 메서드 활용 가능
- **주의:** `Series.str` 메서드는 **NaN (결측치)**가 있으면 작업이 불가능하므로, 먼저 결측치를 처리하거나 제거해야 함.

간단한 예제

```
import pandas as pd

names = pd.Series(['Alice', ' Bob ', 'Charlie'])
print("원본 시리즈:\n", names)
```

- **소문자/대문자 변환:** 모든 이름을 소문자로 또는 대문자로 변환

```
print("모든 이름 소문자:\n", names.str.lower())
print("모든 이름 대문자:\n", names.str.upper())
```

- **공백 제거:** ' Bob '처럼 양쪽에 불필요한 공백이 있는 경우 `.strip()`으로 제거

```
print("양쪽 공백 제거:\n", names.str.strip())
```

- **문자열 길이 계산:** 각 이름의 글자 수 계산 (`공백 제외` 기준)

```
print("이름 길이:\n", names.str.strip().str.len())
```

- **부분 문자열 찾기:** 예를 들어 이름에 "li"가 들어가는지 확인

```
print("li 포함 여부:\n", names.str.contains("li"))
```

Titanic 데이터 실습

- **예제 1: 승객 이름에서 성(last name) 추출**
  - Titanic 데이터의 **Name** 열에는 `"성, 호칭. 이름"` 형태로 이름이 적혀 있습니다 (예: **"Braund, Mr. Owen Harris"**).
  - 각 승객의 **성을 추출**하여 새로운 열(`'LastName'`)로 추가해보겠습니다.

▼ **정답**

```
# Name 열에서 성(last name)은 콤마(,)를 기준으로 첫 번째 부분
titanic_df['LastName'] = titanic_df['Name'].str.split(',').str[0]
```

```
print(titanic_df[['Name', 'LastName']].head(3))
```

- **예제 2: 특정 문자열 포함 여부로 데이터 필터링**
  - **미혼 여성(Miss.) 승객**이 몇 명인지 확인해보겠습니다.
  - Name 열에 `"Miss."` 라는 문자열이 포함되는지를 검사하여 True/False 불린 시리즈를 얻은 뒤, True인 데이터만 추려서 개수를 세어봅시다.

▼ **정답**

```
# Name 열에 'Miss.' 문자열이 포함된 행 필터링
miss_mask = titanic_df['Name'].str.contains('Miss.')
miss_df = titanic_df[miss_mask]
print(miss_df[['Name', 'Sex']].head(3))
print("Miss.를 호칭으로 가진 승객 수:", len(miss_df))
```

- **예제 3: 문자열 치환 및 수정** – 이번에는 승객 이름에서 괄호를 없애보겠습니다.
- **정답**

```
# 괄호( (, ) ) 을 제거
titanic_df['Name_cleaned'] = titanic_df['Name'].str.replace('(', '').str.repl
ace(')', '')
titanic_df[['Name', 'Name_cleaned']]
```

1. **퀴즈 1:** Titanic 승객들의 이름에서 **호칭(Title)**을 추출해보세요. 예를 들어 "Braund, **Mr**. Owen Harris"에서 **"Mr"**, "Cumings, **Mrs**. John ..."에서 **"Mrs"**를 얻어야 합니다.
   a. (힌트: 이름은 **콤마(,)**와 **마침표(.)**를 기준으로 "성, 호칭. 이름" 구성입니다. 먼저 콤마로 분리한 뒤, 두 번째 부분에서 마침표 이전까지를 가져오면 됩니다.)
2. **퀴즈 2: Embarked** 열의 값('S', 'C', 'Q')을 모두 소문자로 바꾸는 코드를 작성해보세요.
   a. (대문자 'S','C','Q' → 소문자 's','c','q')

▼ **정답**

```
# 퀴즈 1 정답: Name 열에서 호칭 추출
titanic_df['Title'] = titanic_df['Name'].str.split(',').str[1].str.split('.').str[0].
str.strip()
print(titanic_df[['Name','Title']].head(3))
print("호칭 종류별 개수:\n", titanic_df['Title'].value_counts())

# 퀴즈 2 정답: Embarked 열 소문자로 변환
titanic_df['Embarked'] = titanic_df['Embarked'].str.lower()
print("Embarked 열 고유값:", titanic_df['Embarked'].unique())
```

---

3) 시간 데이터 처리

- **시간(Date/Time) 데이터**는 문자열로 존재할 때 이를 `datetime 타입`으로 변환하여 다뤄야 한다.
  - 날짜/시간을 제대로 처리하면 **연도, 월, 일 단위로 묶어 분석**하거나 **두 시점의 차이를 계산**하는 등 시간에 특화된 처리를 쉽게 할 수 있습니다.
- 문자열 날짜 변환
  - `pd.to_datetime()` 함수를 사용 → 다양한 형식의 날짜 문자열을 자동으로 인식하여 `datetime64` 타입으로 변환
  - 형식을 지정해야 하는 경우 `format` 인자를 사용
- datetime으로 변환된 열(또는 Series)에 대해서는 `.dt` 접근자로 연월일시 등 **각각의 시간 구성 요소**를 쉽게 추출 가능
  - 예: `df['date'].dt.year`, `df['date'].dt.month`, `df['date'].dt.day`, `df['date'].dt.hour` 등 ...
- 날짜 Series 간의 **뺄셈**을 하면 **시간 차이(`Timedelta`)**를 획득 가능
  - 예: `df['end_date'] - df['start_date']`를 하면 각 행마다 두 날짜의 차이가 `Timedelta` 형태로 계산

간단한 예제

```
import pandas as pd
```

```
date_strings = pd.Series(['2025-01-01', '2025-03-15', '2025-03-31'])
print("문자열 형태 날짜:\n", date_strings)
```

- **문자열을 datetime으로 변환:** `pd.to_datetime` 사용

```
dates = pd.to_datetime(date_strings)
print("datetime 형태:\n", dates)
print("데이터 타입:", dates.dtype)
```

- **연/월/일 추출:** `.dt` 접근자를 사용

```
print("연도:", dates.dt.year)
print("월:", dates.dt.month)
print("일:", dates.dt.day)
print("날짜:", dates.dt.date)
```

- **날짜 연산:** 첫 번째 날짜와 마지막 날짜의 차이

```
diff = dates.iloc[2] - dates.iloc[0]
print("첫 번째와 마지막 날짜 차이:", diff)
print("차이 타입:", type(diff))
```

Titanic 데이터 실습

Titanic 데이터에는 탑승 날짜나 생존 날짜 등의 **시간 정보가 직접 주어지지 않았습니다.** 따라서 학습을 위해 **가상의 시간 데이터를 추가**해보겠습니다. 실제 Titanic 항해 일정을 기반으로, 승객들의 **탑승일시(BoardTime)** 컬럼을 만들어보겠습니다:

- **Titanic 항해 일정 가정:**
  - Southampton(S) 항구 탑승: **1912-04-10 12:00:00**
  - Cherbourg(C) 항구 탑승: **1912-04-10 18:00:00** (같은 날 저녁)
  - Queenstown(Q) 항구 탑승: **1912-04-11 12:00:00** (다음날 정오)

이 가정에 따라 Embarked 값에 매핑되는 탑승 일시를 생성합니다.

```
# Embarked 값에 따른 탑승 일시 매핑
embark_to_time = {'S': '1912-04-10 12:00:00',
```

```
                'C': '1912-04-10 18:00:00',
                'Q': '1912-04-11 12:00:00'}
# BoardTime 열 추가 (현재는 문자열 상태)
titanic_df['BoardTime'] = titanic_df['Embarked'].map(embark_to_time)
print(titanic_df[['Embarked', 'BoardTime']].head(3))
```

- **예제 1: BoardTime 컬럼 datetime 변환** – 탑승 시간 컬럼인 BoardTime을 datetime 타입으로 변환해보겠습니다.

▼ **정답**

```
# 문자열을 datetime으로 변환
titanic_df['BoardTime'] = pd.to_datetime(titanic_df['BoardTime'])
print("BoardTime 타입:", titanic_df['BoardTime'].dtype)
```

- **예제 2: 기초적인 시간 연산 및 추출**
  1. **최초/마지막 탑승 시간 확인:** 모든 승객 중 가장 이른 탑승 시각과 가장 늦은 탑승 시각을 구해봅시다 (`min`, `max` 이용).
  2. **날짜별 탑승 인원 집계:** 날짜(일 단위-`dt.date`)별로 몇 명이 탑승했는지 계산해보겠습니다. (`value_counts()`)

▼ **정답**

```
# 가장 이른 탑승 시각과 가장 늦은 탑승 시각
first_time = titanic_df['BoardTime'].min()
last_time = titanic_df['BoardTime'].max()
print("최초 탑승 시각:", first_time)
print("마지막 탑승 시각:", last_time)

# 날짜별 탑승 인원 (연-월-일만 추출하여 그룹화)
titanic_df['BoardDate'] = titanic_df['BoardTime'].dt.date  # datetime에서 날짜만 추출
print("날짜별 탑승 인원:\n", titanic_df['BoardDate'].value_counts())
```

- **예제 3: 시간 간격 계산**

- 각 승객이 **최초 탑승 시각(4월 10일 12:00)** 이후 얼마나 후에 탑승했는지 계산해보겠습니다.

1. `BoardTime`에서 최초 탑승 시간을 빼서 `Timedelta`를 구해보겠습니다. (각 승객의 탑승 시각 - 최초 탑승 시각)
2. 이를 시간(hour) 단위로도 변환해 보겠습니다. → 초 단위로 나와있는 `timedelta`를 3600으로 나눠 시간으로 변환

▼ **정답**

```
# 각 승객의 탑승 시각 - 최초 탑승 시각
titanic_df['TimeSinceFirst'] = titanic_df['BoardTime'] - titanic_df['Board
Time'].min()
print(titanic_df[['Embarked','BoardTime','TimeSinceFirst']].head(5))

# TimeSinceFirst를 시간 단위로 환산 (초 단위로 나와있는 timedelta를 3600
으로 나눠 시간으로 변환)
titanic_df['HoursSinceFirst'] = titanic_df['TimeSinceFirst'].dt.total_seco
nds() / 3600
print(titanic_df[['Embarked','HoursSinceFirst']].head(5))
```

1. **퀴즈:** 방금 생성한 `BoardTime` 열에서 **시(hour)** 정보를 추출하여 `BoardHour`라는 새로운 열을 추가해보세요. (힌트: datetime Series에 대해 `dt.hour` 속성을 사용할 수 있습니다. 예: `df['datetime컬럼'].dt.hour`)

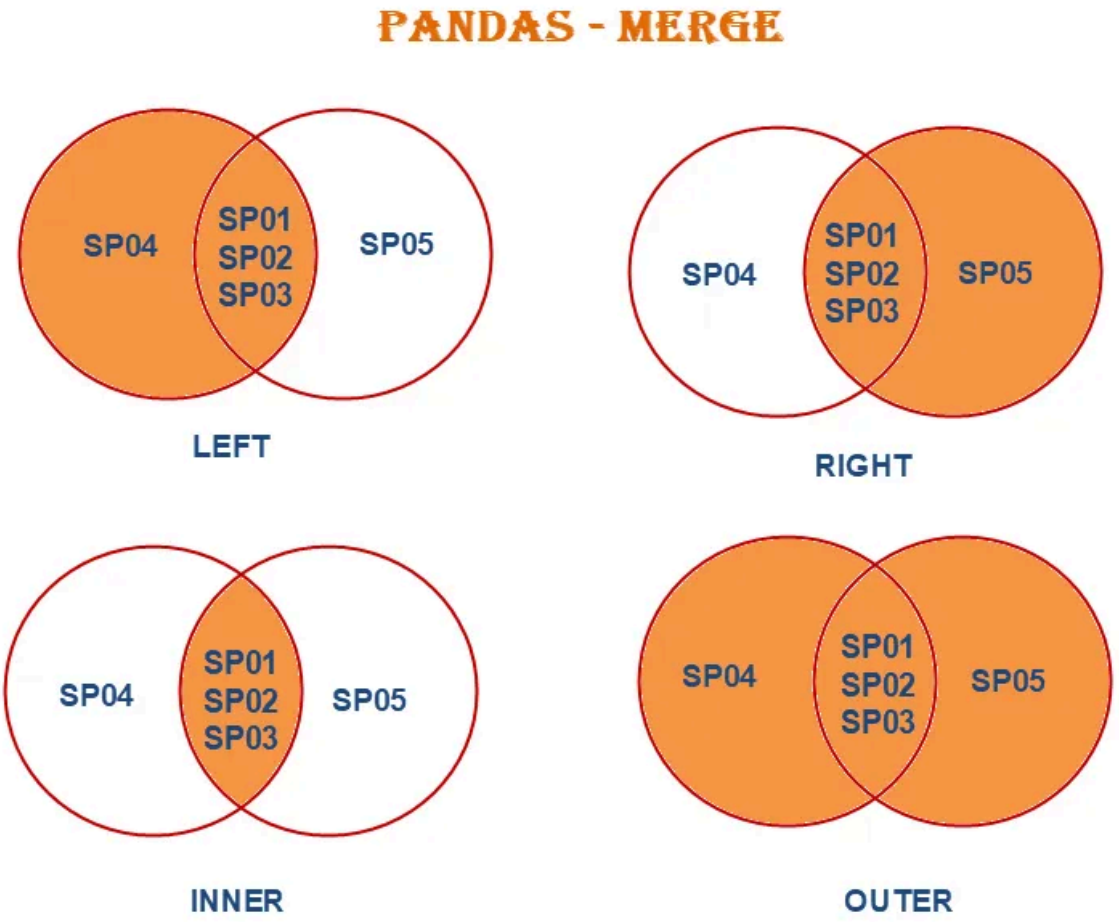
▼ **정답**

```
# 퀴즈 정답: 탑승 시각의 시(hour)를 추출하여 새로운 열 생성
titanic_df['BoardHour'] = titanic_df['BoardTime'].dt.hour
print(titanic_df[['Embarked', 'BoardTime', 'BoardHour']].head(5))
```

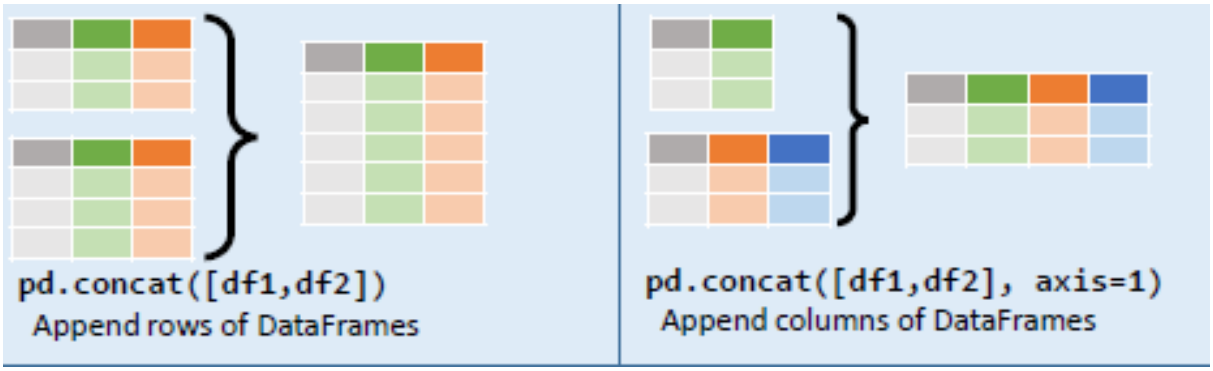
---

4) 데이터 결합 기법 (Merge, Join, Concat)

- **데이터 결합**은 여러 데이터 소스를 하나로 합치는 작업으로, **열 단위 결합(가로 방향)**과 **행 단위 결합(세로 방향)** 두 가지로 크게 나눌 수 있습니다.



- **Merge/Join (가로 방향 결합):** 두 데이터프레임에 공통된 키(key)가 있을 때, 그 키를 기준으로 하나의 데이터프레임으로 합치는 방법입니다.
  - `pd.merge(df1, df2, on='공통열')` 형식으로 사용하며, `how` 인자를 통해 **조인 방식**을 정할 수 있습니다.
    - 기본값 `how='inner'` : 교집합만
    - `how='left'` : 왼쪽 df 기준 모두 유지
    - `how='right'` : 오른쪽 df 기준
    - `how='outer'` : 합집합 전체)
  - DataFrame의 `.join()` 메서드도 있는데, 이는 **인덱스**를 기준으로 결합하거나 간단히 컬럼명을 지정해 결합할 때 쓰입니다. 기본 개념은 `merge`와 거의 동일하며, 내부적으로 merge를 사용합니다.
  - 여러 열을 복합 키로 사용할 수도 있습니다 (`on=['열1','열2']`).



- **Concat (세로/가로 방향 연결):** 동일한 구조의 데이터프레임을 **위아래로 이어붙이거나** (append) 또는 열 방향으로 이어붙일 때 사용합니다.
  - `pd.concat([df1, df2, ...], axis=0)`은 행 방향(세로)으로 연결, 즉 df1 아래에 df2의 행들을 추가. **이 때 열 구조가 동일해야 합니다.** 인덱스도 그대로 따라가므로, 기존 인덱스를 무시하고 싶다면 `ignore_index=True` 옵션을 사용합니다.
  - `pd.concat([df1, df2], axis=1)`은 열 방향(가로)으로 연결, 즉 df1 오른쪽에 df2의 열들을 붙입니다. 이 경우 두 데이터프레임의 **인덱스 기준으로 정렬되어 결합**됩니다. 인덱스가 다르면 NaN이 생길 수 있습니다.

간단한 예제

```
import pandas as pd

# 학생 시험 점수 예제
exam_df1 = pd.DataFrame({'ID': [1, 2, 3],
                         '국어': [90, 85, 80]})
exam_df2 = pd.DataFrame({'ID': [2, 3, 4],
                         '영어': [75, 80, 85]})
print("exam_df1:\n", exam_df1)
print("exam_df2:\n", exam_df2)
```

- **Merge 예제:** 두 데이터프레임을 학생 ID를 기준으로 합치기
  - ID 2, 3은 양쪽에 모두 존재하므로 결과에 포함. (inner join 기준) ID 1은 df2에 없고, ID 4는 df1에 없으므로 inner join 시 제외.

```
merged_inner = pd.merge(exam_df1, exam_df2, on='ID', how='inner')
print("Inner merge 결과:\n", merged_inner)
```

- 만약 `how='outer'`로 하면 모든 ID가 다 나오는 합집합 join

```
merged_outer = pd.merge(exam_df1, exam_df2, on='ID', how='outer')
print("Outer merge 결과:\n", merged_outer)
```

- **Concat 예제 (행 방향):** 점수 데이터가 두 분반으로 나뉘어 있었다고 가정하고, 이를 이어 붙이기

```
classA = pd.DataFrame({'ID': [1, 2, 3],
                       'Score': [50, 60, 70]})
classB = pd.DataFrame({'ID': [4, 5],
                       'Score': [80, 90]})
combined = pd.concat([classA, classB], axis=0, ignore_index=True)
print("행 방향 결합 결과:\n", combined)
```

- **Concat 예제 (열 방향):** 두 개의 Series를 옆으로 붙여 하나의 DataFrame으로 만들기

```
names = pd.Series(['Alice', 'Bob', 'Charlie'])
scores = pd.Series([85, 92, 88])
merged_df = pd.concat([names, scores], axis=1)
merged_df.columns = ['Name', 'Score']
print("열 방향 결합 결과:\n", merged_df)
```

Titanic 데이터 실습

- **예제 1: 항구 이름 정보 Merge**
  - Titanic 데이터의 **Embarked** 열에는 'S', 'C', 'Q'로만 표기되어 있습니다.
  - 각 문자가 어떤 항구를 의미하는지 풀어서 보여주는 열을 추가해보겠습니다. 이를 위해 **별도의 항구 정보 DataFrame**을 만들고 Titanic 데이터프레임과 **merge** 해 보겠습니다.
  - 항구 코드 -> 이름 대응: S = Southampton, C = Cherbourg, Q = Queenstown

1. 항구 코드 -> 이름 대응(S = Southampton, C = Cherbourg, Q = Queenstown) 되는 데이터프레임 생성
2. Titanic 데이터와 위에서 생성한 데이터프레임을 **Embarked** 열을 기준으로 병합 (기존 타이타닉 데이터의 열은 온전히 유지)

▼ **정답**

```
# 항구 코드와 이름 매핑 데이터프레임 생성
port_df = pd.DataFrame({
    'Embarked': ['S', 'C', 'Q'],
    'PortName': ['Southampton', 'Cherbourg', 'Queenstown']
})
print(port_df)

# Titanic 데이터와 port_df를 Embarked 열을 기준으로 병합
titanic_with_port = pd.merge(titanic_df, port_df, on='Embarked', how='l
eft')
print(titanic_with_port[['Embarked', 'PortName']].head(5))
```

- **예제 2: 데이터 세로 연결 (Concat)**
  - Titanic 데이터를 임의로 두 부분으로 나눈 뒤 다시 합쳐보겠습니다. 실제로는 하나의 데이터셋이지만, 데이터가 여러 파일로 나뉘어 제공되었을 때 이를 연결하는 연습입니다.

```
# 데이터프레임을 임의로 두 부분으로 분할
titanic_part1 = titanic_df.iloc[:445]   # 처음 445개 행
titanic_part2 = titanic_df.iloc[445:]   # 나머지 행
```

  - 임의의 두 데이터프레임(titanic_part1, titanic_part2)를 concat 을 수행해서 원래 타이타닉 데이터로 복원하세요.

▼ **정답**

```
# 세로 방향 결합으로 원래대로 복원
titanic_combined = pd.concat([titanic_part1, titanic_part2], axis=0)
```

```
print("원본 행 개수:", len(titanic_df))
print("결합 후 행 개수:", len(titanic_combined))
```

1. **퀴즈 1: 두 개의 데이터프레임 `df1`, `df2`가 공통 열 'ID'를 가지고 있을 때**, 이 열을 키(key)로 이용하여 **데이터를 가로로 합치는** 코드를 작성해보세요. (`pd.merge` 또는 `.merge()` 메서드를 활용하고 `on='ID'`를 지정)
2. **퀴즈 2: 데이터프레임 `df1`, `df2`가 동일한 컬럼 구조**를 가지고 있을 때, **세로로 이어붙이는** 코드를 작성해보세요. (`pd.concat` 사용)

▼ **정답**

```
# 퀴즈 1 정답: df1과 df2를 ID 열 기준으로 병합
merged_df = pd.merge(df1, df2, on='ID', how='inner')  # how는 필요에 따라 'left', 'right', 'outer' 등 변경
print("병합 결과:\n", merged_df)

# 퀴즈 2 정답: df1과 df2 세로 연결
concatenated_df = pd.concat([df1, df2], axis=0, ignore_index=True)
print("세로 연결 결과:\n", concatenated_df)
```

---

5) apply / map / lambda 활용

- **반복적인 데이터 처리**를 위해 **apply**와 **map** 메서드, 그리고 **lambda 함수**를 활용할 수 있습니다.
- **map**
  - `Series.map(func 또는 dict)`
  - func(함수)
    - Series의 각 원소를 주어진 함수에 적용한 결과로 변환한 Series를 반환
  - **dict(딕셔너리)**
    - **해당 값에 매핑(mapping)되는 값으로 변환**
    - **주로 값 치환/변환에 간편하게 사용**
- **apply**

- `Series.apply(func)`
  - map과 유사하지만 좀 더 범용적인 함수 적용이 가능
  - 람다 함수나 사용자 정의 함수를 넣어 Series의 각 값에 적용
- `DataFrame.apply(func, axis=0 또는 1)`
  - 데이터프레임 전체에 함수를 적용
  - `axis=0`이면 **각 열(Column)**에 함수를 적용하고, `axis=1`이면 **각 행(Row)**에 함수를 적용
  - 예를 들어 여러 열을 조합하여 하나의 값을 계산하거나, 모든 열에 대해 어떤 요약 통계를 낼 때 사용 가능

- **lambda(람다) 함수**
  - 파이썬의 익명 함수 기능으로, 함수를 한 줄로 간단히 정의할 때 사용
  - 예: `lambda x: x*2`는 입력값 `x`를 받아 `x*2`를 반환하는 함수와 같습니다.
  - 여러 줄이 필요한 복잡한 로직은 일반적인 `def`로 함수를 정의하고, 짧은 변환은 람다로 직접 `apply` 등에 넘기는 방식

- **주의**
  - `apply`나 `map`을 사용하면 편리하지만, 큰 데이터셋에 대해 반복문을 도는 것이므로 **성능 면에서는 Vectorized 연산보다 느릴 수 있습니다.**
  - 가능하다면 Pandas의 벡터화 연산이나 내장 메서드를 우선 사용하고, 복잡한 경우 어쩔 수 없이 apply를 활용하는 식으로 접근합니다.

간단한 예제

```
import pandas as pd

s = pd.Series([10, 20, 30])
print("원본 Series:\n", s)
```

- **Series.map 예제:** 값을 간단히 다른 값으로 치환

```
# 딕셔너리를 이용하여 값 매핑: 10->'A', 20→'B', 30→'C'
mapped = s.map({10: 'A', 20: 'B', 30: 'C'})
print("map 결과:\n", mapped)
```

- **Series.apply + lambda 예제:** 각 값에 임의의 연산 적용

```
# 각 값에 5를 더하고 문자열로 변환하는 lambda 함수 적용
applied = s.apply(lambda x: str(x + 5))
print("apply 결과:\n", applied)
print("apply 결과 타입:", applied.dtype)
```

- **DataFrame.apply 예제:** 각 행(row)에 대해 연산 적용

```
df = pd.DataFrame({'A': [1,2,3], 'B': [10, 20, 30]})
print("원본 DF:\n", df)
# 각 행별 합계를 계산하여 시리즈로 반환 (axis=1)
row_sum = df.apply(lambda row: row['A'] + row['B'], axis=1)
print("각 행의 A+B 합:\n", row_sum)
# 이 결과를 새로운 열로 추가할 수도 있음
df['A+B'] = row_sum
print("새 열 추가 후 DF:\n", df)
```

Titanic 데이터 실습

- **예제 1: map을 사용한 값 치환**
  - **생존여부(Survived)**를 현재는 0/1로 표현하고 있는데, 이해하기 쉽도록 **"사망"/"생존" 문자열로 변환**한 새로운 열을 추가해보겠습니다.

▼ **정답**

```
# Survived: 0→'사망', 1->'생존'으로 매핑하여 새로운 열 생성
titanic_df['SurvivalStatus'] = titanic_df['Survived'].map({0: '사망', 1: '생존'})
print(titanic_df[['Survived', 'SurvivalStatus']].head(5))
```

- **예제 2: apply와 lambda를 사용한 나이대 분류**
  - 승객들의 **나이(Age)**를 이용해 **미성년자/성인 분류**를 해보겠습니다.
  - 나이 기준은 편의상 **18세 미만**을 미성년자로 하겠습니다. 새로운 `AgeGroup` 열에 각 승객이 "미성년자" 또는 "성인"인지 표시합니다.

▼ **정답**

```
# 나이가 18 미만이면 '미성년자', 아니면 '성인'으로 분류하는 lambda 함수 적용
titanic_df['AgeGroup'] = titanic_df['Age'].apply(lambda x: '미성년자' if x
< 18 else '성인')
print(titanic_df[['Age', 'AgeGroup']].head(10))
```

- **예제 3: DataFrame.apply를 사용한 가족 규모 계산**
  - Titanic 데이터에는 형제/배우자 수(`SibSp`), 부모/자녀 수(`Parch`) 정보가 있습니다.
  - 이를 활용하여 각 승객의 **가족 규모(FamilySize)**를 계산해보겠습니다.
  - `가족 규모 = 본인을 포함한 가족 수 = SibSp + Parch + 1` 로 정의합니다.

▼ **정답**

```
# 각 행(row)에 대해 SibSp + Parch + 1 계산
titanic_df['FamilySize'] = titanic_df.apply(lambda row: row['SibSp'] + ro
w['Parch'] + 1, axis=1)
print(titanic_df[['SibSp', 'Parch', 'FamilySize']].head(5))
```

1. **퀴즈 1: 객실등급(Pclass)** 값을 **일등석/이등석/삼등석** 등의 문자열로 변환한 새로운 열 `PclassLabel`을 추가해보세요. (예: 1 -> "1등석", 2 -> "2등석", 3 -> "3등석") `map`을 사용하면 편리합니다.
2. **퀴즈 2: 이름(Name)의 길이**를 계산하여 새로운 열 `NameLength`에 저장해보세요. (`len()` 함수를 `apply`로 적용하거나, Pandas의 문자열 기능으로 쉽게 구할 수 있습니다.)

▼ **정답**

```
# 퀴즈 1 정답: Pclass를 등석 문자열로 매핑
titanic_df['PclassLabel'] = titanic_df['Pclass'].map({1: '1등석', 2: '2등석',
3: '3등석'})
```

```
print(titanic_df[['Pclass', 'PclassLabel']].head(5))

# 퀴즈 2 정답: Name 길이 계산하여 새로운 열 저장
titanic_df['NameLength'] = titanic_df['Name'].apply(lambda x: len(x))
# (또는 titanic_df['Name'].str.len()으로도 동일한 결과를 얻을 수 있습니다.)
print(titanic_df[['Name', 'NameLength']].head(3))
```

📎 미니 과제

1. H&M
    a. 데이터 :
    b. 실습 파일 :
    c. 정답 :
2. 서울시 부동산 데이터
    a. 데이터 :
    b. 실습 파일 :
    c. 정답 :

📎 자료

1. 자료 PDF :
2. 이전 전처리 라이브 세션 자료 : 🐼 CH2. 데이터 전처리 완벽 가이드(2)
