Python
CH11: Pandas II

전종준 교수

서울시립대학교 통계학과/통계데이터사이언스대학원

Contents

- 고급데이터 정제 및 변환

- GroupBy를 통한 분석

- 데이터 병합

고급데이터 정제 및 변환

조건 기반 데이터 필터링 및 값 수정

- 조건을 이용해 특정 행만 선택할 수 있음
- 조건을 만족하는 값만 바꾸는 것도 가능

예시 코드:

```
import pandas as pd

df = pd.DataFrame({
    "이름": ["영희", "철수", "민수"],
    "점수": [80, 45, 90]
})

# 50점 미만인 학생만 필터링
print(df[df["점수"] < 50])

# 조건을 만족하는 값 수정
df.loc[df["점수"] < 50, "점수"] = 50
print(df)
```

복수 조건 결합: &, |, ~

- 여러 조건을 동시에 적용
- 괄호를 사용

예시 코드:

```
df = pd.DataFrame({
    "이름": ["영희", "철수", "민수", "지민"],
    "점수": [80, 45, 90, 60]
})

# 점수가 60 이상이고 이름이 '지민'이 아닌 경우
filtered = df[(df["점수"] >= 60) & (df["이름"] != "지민")]
print(filtered)
```

isin()을 이용한 다중 값 필터링

- 특정 값들의 리스트와 일치하는 행만 선택할 수 있음

예시 코드:

```
df = pd.DataFrame({
    "이름": ["영희", "철수", "민수", "지민"],
    "점수": [80, 45, 90, 60]
})

# '영희'와 '지민'만 선택
selected = df[df["이름"].isin(["영희", "지민"])]
print(selected)
```

lambda 함수란?

- 이름 없이 간단히 정의하는 '익명 함수'
- 주로 한 줄짜리 계산에 사용

예시 코드:

```
# 일반 함수
def square(x):
    return x * x

# 같은 기능을 하는 lambda
square2 = lambda x: x * x

print(square(3))   # 9
print(square2(3))  # 9
```

map() 함수로 리스트 일괄 처리

- 리스트나 시리즈의 각 요소에 함수를 적용할 수 있음
- 반복문 없이 간결하게 처리 가능

예시 코드:

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # [1, 4, 9, 16]
```

apply() 함수로 데이터프레임 처리

- 시리즈나 데이터프레임의 각 행/열에 함수를 적용할 수 있음.
- lambda와 함께 많이 써요.

예시 코드:

```python
import pandas as pd

df = pd.DataFrame({
    "이름": ["영희", "철수", "민수"],
    "점수": [80, 45, 90]
})

# 점수에 10점 추가
df["점수"] = df["점수"].apply(lambda x: x + 10)
print(df)
```

map() vs apply() 차이점

- `map()`: Series에 사용 가능, 성능 좋음
- `apply()`: 더 유연하고 복잡한 로직에 강함 (다음 예제 참고)

예시 코드:

```
df["이름길이"] = df["이름"].map(len)      # map으로 문자열 길이 계산
df["상태"] = df["점수"].apply(lambda x: "합격" if x >= 60 else "불합격")
```

apply(axis=1)로 행 단위 처리하기

- 각 행(row)을 하나의 딕셔너리처럼 다룰 수 있음
- 여러 열을 조합해서 새로운 값을 만들 때 유용.

예시 코드:

```
import pandas as pd
df = pd.DataFrame({
    "이름": ["영희", "철수", "민수"],
    "국어": [80, 60, 90],
    "수학": [70, 50, 95]})
# 평균 점수와 평가 결과 생성
def 평가(row):
    평균 = (row["국어"] + row["수학"]) / 2
    return "우수" if 평균 >= 75 else "보통"
df["평가"] = df.apply(평가, axis=1)
print(df)
```

re 모듈을 이용한 필터링

- 복잡한 정규식은 re 모듈을 써서도 처리할 수 있음
- apply()와 함께 쓰면 각 행마다 정규식 검사 가능!

예시 코드:

```
import pandas as pd
import re
df = pd.DataFrame({
    "이름": ["영희", "철수", "민수", "지민", "철민"]})
# 이름이 '민'으로 끝나는 경우만 선택
pattern = re.compile("민$")
filtered = df[df["이름"].apply(lambda x: bool(pattern.search(x)))]
print(filtered)
```

날짜 데이터 처리 (pd.to_datetime)

- 문자열로 된 날짜를 datetime 객체로 변환할 수 있어요.
- 날짜 기준 정렬이나 필터링을 위해 필요해요.

예시 코드:

```
import pandas as pd
df = pd.DataFrame({
    "이름": ["영희", "철수", "민수"],
    "가입일": ["2023-01-01", "2024-06-15", "2023-11-30"]})
# 문자열을 날짜 형식으로 변환
df["가입일"] = pd.to_datetime(df["가입일"])
# 2024년 이후 가입한 사람만 보기
recent = df[df["가입일"] > "2024-01-01"]
print(recent)
```

datetime 속성 활용하기

- pd.to_datetime()으로 변환한 열은 다양한 날짜 속성을 사용할 수 있어요.
- 연도, 월, 요일 등으로 필터링하거나 새 열을 만들 수 있어요.

예시 코드:

```
import pandas as pd

df = pd.DataFrame({
    "가입일": ["2024-01-01", "2023-12-15", "2025-07-01"]
})
df["가입일"] = pd.to_datetime(df["가입일"])

# 연도, 월, 요일 정보 추출
df["연도"] = df["가입일"].dt.year
df["월"] = df["가입일"].dt.month
df["요일"] = df["가입일"].dt.day_name()
print(df)
```

시간 간격 계산하기 (Timedelta)

- 두 날짜의 차이를 계산가능
- 며칠 지났는지, 며칠 남았는지 등을 쉽게 구할 수 있음

예시 코드:

```
from datetime import datetime
import pandas as pd
df = pd.DataFrame({
    "마감일": ["2025-07-30", "2025-08-15"]})
df["마감일"] = pd.to_datetime(df["마감일"])
# 오늘 날짜와의 차이 계산
t_day = pd.to_datetime(datetime.today().date())
df["D-남은일수"] = (df["마감일"] - t_day).dt.days

print(df)
```

최근 30일 데이터 추출

- 오늘 날짜 기준으로 최근 30일 이내의 데이터만 추출
- `pd.to_datetime`과 `datetime.timedelta`를 함께 활용

예시 코드:

```
from datetime import datetime, timedelta
import pandas as pd
df = pd.DataFrame({
    "작성일": ["2025-06-01", "2025-06-20", "2025-07-10", "2025-07-12"]})
df["작성일"] = pd.to_datetime(df["작성일"])
# 오늘 날짜와 30일 전 날짜
today = pd.to_datetime(datetime.today().date())
start_date = today - timedelta(days=30)
# 최근 30일 이내의 데이터 필터링
recent_df = df[df["작성일"] >= start_date]
print(recent_df)
```

날짜 문자열 포맷 지정하기

- `pd.to_datetime()`은 기본적으로 자동 추론
- 하지만 형식이 명확할 때는 `format=`을 지정하는 게 안전

예시 코드:

```
import pandas as pd

# 날짜 형식이 "일-월-연도"인 문자열
df = pd.DataFrame({
    "날짜": ["12-07-2025", "01-06-2025"]
})

# 포맷 지정해서 변환 (일-월-연도)
df["날짜"] = pd.to_datetime(df["날짜"], format="%d-%m-%Y")

print(df)
```

날짜 포맷 코드 요약

- %Y : 연도 (예: 2025)
- %y : 연도 두 자리 (예: 25)
- %m : 월 (01 12)
- %d : 일 (01 31)
- %H : 시 (00 23)
- %M : 분 (00 59)
- %S : 초 (00 59)

예:

- "2025-07-12" → %Y-%m-%d
- "12/07/25" → %d/%m/%y

GroupBy를 통한 분석

groupby 개념과 집계 함수

- `groupby()`는 데이터를 그룹별로 묶어 분석할 수 있게 해줌
- 집계 함수: 그룹별 평균, 합계, 개수 등을 계산할 수 있음

자주 쓰는 집계 함수:

- `mean()` : 평균
- `sum()` : 합계
- `count()` : 개수
- `agg()` : 여러 함수 한 번에 적용

groupby와 mean(), count() 예시

- 부서별 평균 급여, 성별 인원 수를 쉽게 계산할 수 있음

예시 코드:

```python
import pandas as pd

df = pd.DataFrame({
    "부서": ["인사", "인사", "개발", "개발", "마케팅"],
    "이름": ["영희", "철수", "민수", "지민", "서준"],
    "급여": [300, 320, 400, 420, 380]
})

# 부서별 평균 급여
print(df.groupby("부서")["급여"].mean())

# 부서별 인원 수
print(df.groupby("부서")["이름"].count())
```

agg()로 여러 집계 함수 한 번에

- agg()는 여러 함수를 동시에 적용할 수 있음
- 컬럼마다 다른 함수도 적용 가능

예시 코드:

```
# 부서별 급여 합계와 인원 수 한 번에 보기
result = df.groupby("부서").agg({
    "급여": ["mean", "sum"],
    "이름": "count"
})
print(result)
```

agg() 사용자 함수

- agg()에는 기본 함수뿐 아니라 직접 만든 함수도 쓸 수 있음

예시 코드:

```
import pandas as pd
df = pd.DataFrame({
    "부서": ["인사", "인사", "개발", "개발", "마케팅"],
    "급여": [300, 320, 400, 420, 380]})
# 영어로 된 사용자 정의 함수
def salary_range(x):
    return x.max() - x.min()
# 부서별 평균과 범위 집계
result = df.groupby("부서").agg({
    "급여": ["mean", salary_range]})
print(result)
```

groupby 결과를 저장하고 다루기

- groupby() + agg() 결과는 인덱스가 그룹값으로 설정
- 일반 데이터프레임처럼 다루려면 reset_index() 사용: index 로 설정된 그룹 구분값들이 데이터프레임의 컬럼값으로 들어옴
- sort_values(), rename() 등도 같이 사용 가능

예시 코드:

```
import pandas as pd
df = pd.DataFrame({
    "부서": ["개발", "개발", "인사", "인사", "마케팅"],
    "급여": [400, 420, 300, 320, 380]
})
# 그룹별 평균 계산 후 저장
result = df.groupby("부서")["급여"].mean()
# 인덱스를 열로 되돌리기
result = result.reset_index()
# 급여 내림차순 정렬
result = result.sort_values(by="급여", ascending=False)
print(result)
```

타이타닉 데이터 불러오기 (Seaborn 사용)

- seaborn 패키지를 통해 타이타닉 데이터를 바로 불러올 수 있음

예시 코드:

```
import seaborn as sns
import pandas as pd

# 타이타닉 데이터 불러오기
df = sns.load_dataset("titanic")

# 앞부분 확인
print(df.head())
```

성별 생존률 계산하기

- `groupby()`를 이용해 성별 생존률을 계산

예시 코드:

```
# 성별 생존률 계산
survival = df.groupby("sex")["survived"].mean().reset_index()
survival = survival.rename(columns={"survived": "생존률"})
print(survival)
```

선실 등급별 평균 나이와 요금

- `agg()`를 사용하면 여러 통계를 동시에 계산

예시 코드:

```
# 등급별 평균 나이, 평균 요금
pclass_stats = df.groupby("pclass").agg({
    "age": "mean",
    "fare": "mean"
}).reset_index().round(1)

print(pclass_stats)
```

성별 + 등급별 생존률 분석

- 복수 컬럼으로 그룹화할 수 있어요: `groupby([...])`

예시 코드:

```
# 성별 + 선실등급별 생존률
grouped = df.groupby(["sex", "pclass"])["survived"].mean().reset_index()

grouped = grouped.rename(columns={"survived": "생존률"}).round(2)
print(grouped)
```

데이터 병합

concat, merge, join의 차이

- `concat()`: 위/아래 또는 옆으로 단순히 붙이기
- `merge()`: SQL의 join처럼, 공통 열 기준으로 병합
- `join()`: 인덱스를 기준으로 병합 (Series나 DataFrame에서 편리)

주요 기준:

- `concat()`: 행 또는 열 방향 연결
- `merge()`: 키 값을 기준으로 정교한 병합
- `join()`: 인덱스를 기준으로 빠르게 병합

concat()로 행 방향 연결하기

- 같은 구조(컬럼)의 데이터 여러 개를 위아래로 붙일 때 사용

예시 코드:

```
import pandas as pd

df1 = pd.DataFrame({"이름": ["철수", "영희"], "점수": [80, 90]})
df2 = pd.DataFrame({"이름": ["민수", "지민"], "점수": [85, 95]})

result = pd.concat([df1, df2], ignore_index=True)
print(result)
```

merge()로 사용자와 거래 정보 병합

- merge()는 공통 컬럼(예: user_id)을 기준으로 병합

예시 코드:

```
users = pd.DataFrame({
    "user_id": [1, 2, 3],
    "name": ["철수", "영희", "민수"]
})

orders = pd.DataFrame({
    "order_id": [1001, 1002, 1003],
    "user_id": [1, 2, 1],
    "item": ["책", "연필", "지우개"]
})

# user_id 기준 병합
merged = pd.merge(users, orders, on="user_id")
print(merged)
```

merge()의 병합 방식들

- how="inner": 교집합 (기본값)
- how="left": 왼쪽 기준 보존
- how="outer": 전체 다 포함 (null 가능)

예시 코드:

```
pd.merge(users, orders, on="user_id", how="left")
pd.merge(users, orders, on="user_id", how="outer")
```

join()로 인덱스 기준 병합하기

- 인덱스를 기준으로 병합할 때 join()을 사용
- set_index()와 함께 자주 사용

예시 코드:

```
user_info = users.set_index("user_id")
order_info = orders.set_index("user_id")

result = user_info.join(order_info, how="inner")
print(result)
```

병합 후 꼭 알아야 할 후처리 팁

- **결측치 처리**
  - `fillna(0)`: 결측값을 기본값으로 채우기
  - `dropna()`: 결측값이 있는 행 제거
- **중복된 컬럼 처리**
  - 같은 이름의 컬럼이 병합되면 `_x`, `_y`로 붙음
  - `suffixes=("_left", "_right")` 옵션으로 명시 가능
- **인덱스 다루기**
  - `reset_index()`: 인덱스를 일반 컬럼으로 되돌리기
  - `set_index(...)`: 특정 컬럼을 인덱스로 설정

concat vs merge vs join 요약 비교

- **concat(df1, df2, axis=0)**
  - 단순 연결 (위아래 또는 옆으로 붙이기)
  - 인덱스 재설정: `ignore_index=True`
- **merge(df1, df2, on=...)**
  - 공통 키 기준 병합 (SQL의 JOIN처럼)
  - 병합 방식: `how="inner"`, `"left"`, `"outer"` 등
- **join() (인덱스 기반 병합)**
  - 인덱스를 기준으로 빠르게 병합
  - `set_index()`와 함께 사용

실습 예제: 사용자 정보와 주문 내역 병합하기

- seaborn의 `titanic` 데이터를 두 개로 나눠 병합 연습을 해볼게요.
- `passenger_id`를 기준으로 사용자 정보와 생존 정보를 연결해요.

예시 코드:

```
import seaborn as sns
import pandas as pd
# 전체 데이터 불러오기
df = sns.load_dataset("titanic")
# 사용자 정보 테이블
users = df[["survived", "class", "sex"]].copy()
users["passenger_id"] = users.index
# 탑승 정보 테이블
info = df[["age", "fare", "embarked"]].copy()
info["passenger_id"] = info.index
# 병합
merged = pd.merge(users, info, on="passenger_id")
print(merged.head())
```

실습 예제: concat()으로 데이터 연결하기

- seaborn의 tips 데이터를 둘로 나눈 뒤 다시 이어붙여 볼게요.

예시 코드:

```
tips = sns.load_dataset("tips")

# 데이터 둘로 나누기
tips1 = tips.iloc[:100]
tips2 = tips.iloc[100:]

# 다시 연결
combined = pd.concat([tips1, tips2], ignore_index=True)

print(combined.shape)  # 원래와 같은 크기
```