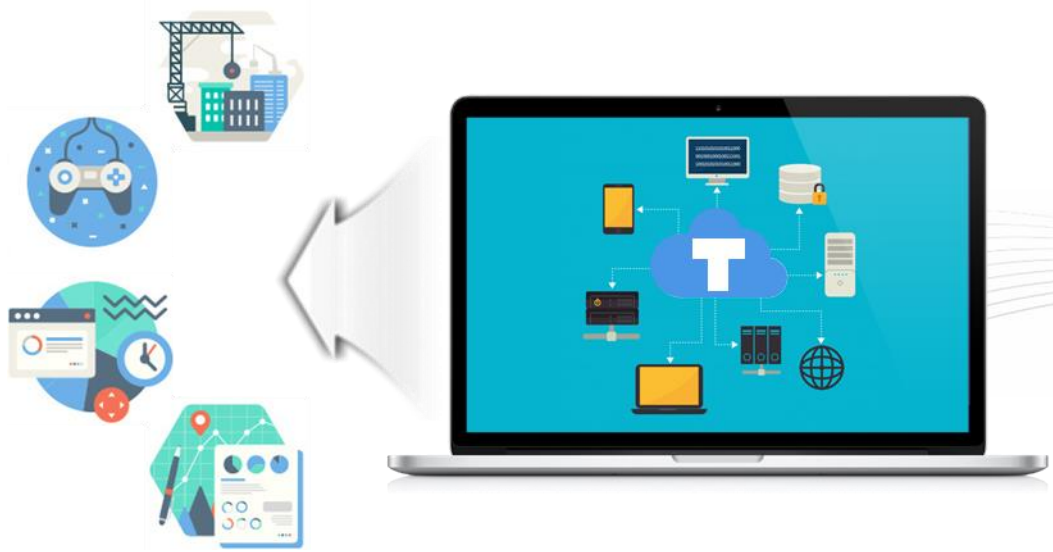
AIoT 데이터 시각화 및 대시보드 개발



Chapter 02.
Pandas 라이브러리 실전 꿀팁 대방출

Ⅰ Uber 데이터셋

Ⅱ German_credit 데이터셋

Ⅲ 임의 데이터셋 + APPL_price 데이터셋

Ⅳ weight_height 데이터셋

Chapter 02.
Pandas 라이브러리 실전 꿀팁 대방출

Ⅴ Covid19-US 데이터셋

Ⅵ product_inspection 데이터셋

Ⅶ product 데이터셋

- **각 데이터의 모양, 각 변수의 형태, 값들의 분포 등을 확인하는 것이 중요**
- **탐색적 데이터 분석 (EDA: Exploratory Data Analysis)**
  - 데이터의 구조 및 특징을 파악한 후 데이터 분석을 깊이 진행하고
  - 관련 통계 모델을 세우는 등 복잡한 데이터를 분석하는데 활용할 수 있음
- **Uber 데이터셋**
  - 미국, 스리랑카, 파키스탄의 2016년 1월부터 동년 12월까지 우버 운행 기록에 대한 데이터를 포함

데이터를 처음 만나면 하는 것들 (ch2–1.py)

```
import pandas as pd
import numpy as np

df = pd.read_csv('datasets/Uber/Uber.csv')
df.head()
```

	START_DATE*	END_DATE*	CATEGORY*	START*	STOP*	MILES*	PURPOSE*
0	1/1/2016 21:11	1/1/2016 21:17	Business	Fort Pierce	Fort Pierce	5.1	Meal/Entertain
1	1/2/2016 1:25	1/2/2016 1:37	Business	Fort Pierce	Fort Pierce	5.0	NaN
2	1/2/2016 20:25	1/2/2016 20:38	Business	Fort Pierce	Fort Pierce	4.8	Errand/Supplies
3	1/5/2016 17:31	1/5/2016 17:45	Business	Fort Pierce	Fort Pierce	4.7	Meeting
4	1/6/2016 14:42	1/6/2016 15:49	Business	Fort Pierce	West Palm Beach	63.7	Customer Visit

▲ 그림 32 Uber 데이터셋을 불러와 첫 5개 행을 살펴본 결과

■ Info()

- 각 열의 정보와 결측값의 유무를 확인

■ to_datetime()

- “START_DATE"와 "END_DATE*"열에 대해 데이터 형식을 datetime으로 바꾼 후,
  각각 unique 메서드를 사용하여 각 열의 고유값 확인
  → to_datetime 메서드에 errors 인자로 전달한 “coerce”는 전달된 데이터에 시간 형식이 아닌 데이터가 포함되어 있을 경우,
  NaT로 변환되게끔 함

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1156 entries, 0 to 1155
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   START_DATE*  1156 non-null   object
 1   END_DATE*    1155 non-null   object
 2   CATEGORY*    1155 non-null   object
 3   START*       1155 non-null   object
 4   STOP*        1155 non-null   object
 5   MILES*       1156 non-null   float64
 6   PURPOSE*     653 non-null    object
dtypes: float64(1), object(6)
memory usage: 63.3+ KB
```

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df['START_DATE*'] = pd.to_datetime(df['START_DATE*'], errors='coerce')
df['END_DATE*'] = pd.to_datetime(df['END_DATE*'], errors='coerce')
df = df.sort_values(['START_DATE*','END_DATE*'])

df['START_DATE*'].unique()

<DatetimeArray>
['2016-01-01 21:11:00', '2016-01-02 01:25:00', '2016-01-02 20:25:00',
 '2016-01-05 17:31:00', '2016-01-06 14:42:00', '2016-01-06 17:15:00',
 '2016-01-06 17:30:00', '2016-01-07 13:27:00', '2016-01-10 08:05:00',
 '2016-01-10 12:17:00',
 ...
 '2016-12-30 11:31:00', '2016-12-30 15:41:00', '2016-12-30 16:45:00',
 '2016-12-30 23:06:00', '2016-12-31 01:07:00', '2016-12-31 13:24:00',
 '2016-12-31 15:03:00', '2016-12-31 21:32:00', '2016-12-31 22:08:00',
                'NaT']
Length: 1155, dtype: datetime64[ns]
```

■ unique()

- “END_DATE" 열을 unique 메서드를 사용

→ "END_DATE "열 역시 NaT가 반환된 값이 있는 것으로 보아 “START DATE*"열과 동일하게 시간 형식이 아닌 데이터 존재

■ value_counts()

- 열의 고유값과 빈도를 확인

→ "CATEGORY** 열은 "Business"와 "Personal" 두 가지의 교유값을 가지고 있고,

→ “Business"가 “Personal" 대비 14배 높은 빈도를 가지고 있음

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df['END_DATE*'].unique()
```

```
<DatetimeArray>
['2016-01-01 21:17:00', '2016-01-02 01:37:00', '2016-01-02 20:38:00',
 '2016-01-05 17:45:00', '2016-01-06 15:49:00', '2016-01-06 17:19:00',
 '2016-01-06 17:35:00', '2016-01-07 13:33:00', '2016-01-10 08:25:00',
 '2016-01-10 12:44:00',
 ...
 '2016-12-30 11:56:00', '2016-12-30 16:03:00', '2016-12-30 17:08:00',
 '2016-12-30 23:10:00', '2016-12-31 01:14:00', '2016-12-31 13:42:00',
 '2016-12-31 15:38:00', '2016-12-31 21:50:00', '2016-12-31 23:51:00',
                'NaT']
Length: 1155, dtype: datetime64[ns]
```

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df['CATEGORY*'].value_counts()
```

```
CATEGORY*
Business    1078
Personal      77
Name: count, dtype: int64
```

■ value_counts()

- “START*"열과 "STOP*"열을 살펴보겠습니다. 이 열들은 object 형태의 범주형 데이터로, value_counts 메서드를 이용하여 각 고유값들과 각 값들의 빈도를 확인

  → 출발 장소에 해당하는 “START*" 열과 도착 장소를 나타내는 “STOP*"열 모두 가장 높은 빈도를 가지는 장소는 “Cary"

  → “Cary"라는 장소는 해당 데이터셋이 조사된 지역 중 가장 번화가일 것이라는 합리적 추론

  → “Unknown Location"이라는 결 측치가 150개가량 숨어있는 것을 확인

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df['START*'].value_counts()
```

```
START*
Cary                201
Unknown Location    148
Morrisville          85
Whitebridge          68
Islamabad            57
                   ...
Florence              1
Ridgeland             1
Daytona Beach         1
Sky Lake              1
Gampaha               1
Name: count, Length: 177, dtype: int64
```

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df['STOP*'].value_counts()
```

```
STOP*
Cary                203
Unknown Location    149
Morrisville          84
Whitebridge          65
Islamabad            58
                   ...
Daytona Beach         1
Sand Lake Commons     1
Sky Lake              1
Vista East            1
Ilukwatta             1
Name: count, Length: 188, dtype: int64
```

■ describe()

- “MILES** 열은 수치형 데이터인 운행 거리를 나타내고 있음
  → 평균 운행거리는 21.1마일인데 반해 표준편차가 359.3마일로 평균값에 비에 훨씬 높은 것을 확인
  → 위 열의 제3사분위수 (quantile (0.75)에 해당하는 값)가 10.4마일인데 반해 최대값은 12204.7마일로 최대값이 이상치에 해당할 만큼 큰 것을 확인
- "MILES**열의 최대값을 가지는 행이 어떤 데이터인지 확인
  → 인덱스가 1155인 것을 보니 가장 마지막 행인데, “START DATE*"열의 값이 “Totals"인 것 을 보아 해당 행은 나머지 행들의 주행거리를 모두 합한 열인 것이라 생각됨

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df['MILES*'].describe()
```

```
count    1156.000000
mean       21.115398
std       359.299007
min         0.500000
25%         2.900000
50%         6.000000
75%        10.400000
max     12204.700000
Name: MILES*, dtype: float64
```

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df[df['MILES*'] == df['MILES*'].max()]
```

	START_DATE*	END_DATE*	CATEGORY*	START*	STOP*	MILES*	PURPOSE*
1155	Totals	NaN	NaN	NaN	NaN	12204.7	NaN

▲ 그림 33 Uber 데이터셋에서 MILES*열이 최대값을 가지는 행

■ drop()

- 1155행을 drop 메서드를 통해 삭제한 후, 다시 describe 메서드를 확인

  → 총계 값을 제거하기 전에 비해 평균과 표준편차가 각각 10.6, 21.6마일로 크게 줄었고 최대값이 310으로 나타냄

  → 여전히 최대값이 크지만, 310이라는 값을 이상치라고 판단하기에는 그 근거가 충분하지 않으므로 해당 값을 지우기는 어려움

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df = df.drop(1155)
df['MILES*'].describe()
```

```
count    1155.000000
mean       10.566840
std        21.579106
min         0.500000
25%         2.900000
50%         6.000000
75%        10.400000
max       310.300000
Name: MILES*, dtype: float64
```

■ value_counts()

- “PURPOSE*"열 범주형 변수였으므로 value_counts 메서드로 확인

→ "PURPOSE" 열에는 총 열개의 고유값이 존재하며, 가장 높은 빈도수를 가지는

상위 3개의변수는 “Meeting", "Meal/Entertain", "Errand/Supplies

→ 각 빈도수의 합은 653으로 원데이터셋에서 마지막 행을 제외한 총 길이인 1155의 절반보다 조금 큰 수

데이터를 처음 만나면 하는 것들 (ch2–1.py)

```
df['PURPOSE*'].value_counts()
```

```
PURPOSE*
Meeting            187
Meal/Entertain     160
Errand/Supplies    128
Customer Visit     101
Temporary Site      50
Between Offices     18
Moving               4
Airport/Travel       3
Charity ($)          1
Commute              1
Name: count, dtype: int64
-------------------------------
```

■ “PURPOSE*"열의 결측치 개수를 isna와 sum 메서드를 통해 확인

- isna 메서드는 특정 값이 null 값인지 아닌지를 True/False로 반환
  → null 값이면 True를 반환하게 됨
- sum 메서드를 통해 이 결과의 총합을 구하면 결측치의 총 개수가 구해짐
  → True는 1에 해당하기 때문

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df['PURPOSE*'].isna().sum()
```

```
502
```

- 결측치가 아닌 값이 653이었고 결측치가 502개이므로 둘의 합은 데이터의 길이인 1155와 동일한 것을 알 수 있음

우버 이용 목적을 구분하는 “CATEGORY"과 "PURPOSE*“열이 있었는데, 이를 활용하여 각 이용 목적에 따른 이동거리와 ("MILES" 열) 운행 시간 (“START_DATE*" 열과 “END_DATE 열을 이용하여 계산)에 대한 통계치 확인**

■ “START_DATE*" 열과 “END_DATE** 열을 이용하여 운행 시간에 해당하는 “DURATION" 열을 생성

- 두 개의 datetime형식의 데이터에 대해 날짜/시간 차이를 구하기 위해서는 사칙연산 중 빼기를 이용할 수 있음
- timedelta 형의 데이터가 반환되게 되는데, 이를 초 단위로 바꾸기 위해서 아래처럼 total_seconds 메서드를 이용할 수 있음
- 운행 시간을 60으로 나눠 분 단위로 나타냄

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df['DURATION*'] = (df['END_DATE*'] - df['START_DATE*']).dt.total_seconds() / 60
df['DURATION*']
```

```
0         6.0
1        12.0
2        13.0
3        14.0
4        67.0
        ...
1150      7.0
1151     18.0
1152     35.0
1153     18.0
1154    103.0
Name: DURATION*, Length: 1155, dtype: float64
```

■ groupby 메서드를 이용하여 각 운행 목적에 대해 운행 거리와 운행 시간에 대한 평균, 표준편차, 데이터의 개수 확인

- 운행 목적이 “Business"인 경우 이동 거리가 긴 순서대로 상위 3개는 순서대로 “Customer Visit", "Meeting", "Between Offices"
- 이동 시간이 긴 순서대로 상위 3개의 값은 "Customer Visit", "Meeting", "Temporary Site“
- 반면 운행 목적이 “Personal 인 경우는 3개의 세부 목적이 있는데, 데이터의 개수가 각각 5개보다 적어 평균 및 표준편차에 대한 신뢰성이 높지 않음
- 특히, 세부 목적이 “Commute"인 경우 하나의 데이터가 있는데, 그 값이 180마일로 다른 값 대비 매우 커서 이상치로 간주하는 것을 고려할 필요가 있음

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
df.groupby(['CATEGORY*','PURPOSE*'])[['MILES*','DURATION*']].agg(['mean','std','count'])
```

		MILES*			DURATION*		
		mean	std	count	mean	std	count
CATEGORY*	PURPOSE*						
Business	Airport/Travel	5.500000	1.852026	3	26.000000	9.848858	3
	Between Offices	10.944444	8.458913	18	25.500000	15.553513	18
	Customer Visit	20.688119	40.632891	101	33.415842	42.891087	101
	Errand/Supplies	3.968750	3.464619	128	12.976562	9.656677	128
	Meal/Entertain	5.698125	5.019690	160	16.125000	10.477739	160
	Meeting	15.247594	25.093394	187	29.737968	26.662381	187
	Temporary Site	10.474000	7.757440	50	25.860000	18.233195	50
Personal	Charity ($)	15.100000	NaN	1	27.000000	NaN	1
	Commute	180.200000	NaN	1	185.000000	NaN	1
	Moving	4.550000	1.181807	4	15.000000	4.546061	4

▲ 그림 34 Uber 데이터셋에서 운행 목적에 대해 이동 거리와 이동 시간에 대한 통계치를 계산한 결과

■ 원 데이터셋의 운행 거리를 앞서 구한 운행 시간과 비교해서 두 변수 간의 상관관계를 구하기

- 상관관계는 시각화 라이브러리인 seaborn의 regplot을 이용해나타내 볼 수 있음

→ scipy 라이브러리의 linregress를 이용하여 “MILES" “DURATION" 두 변수 간 선형 회귀의 기울기, y 절편, 피어슨 상관계수를 계산

→ 데이터에 결측치가 있으면 선형 회귀 계산이 되지 않으므로 그 전에 dropna 메서드를 이용 하여 결측치를 제거

→ seaborn 라이브러리의 regplot (regression plot의 약 자)를 이용하여 두 변수를 각각 x축, y축으로 하여 선형 회귀선과 함께 시각화

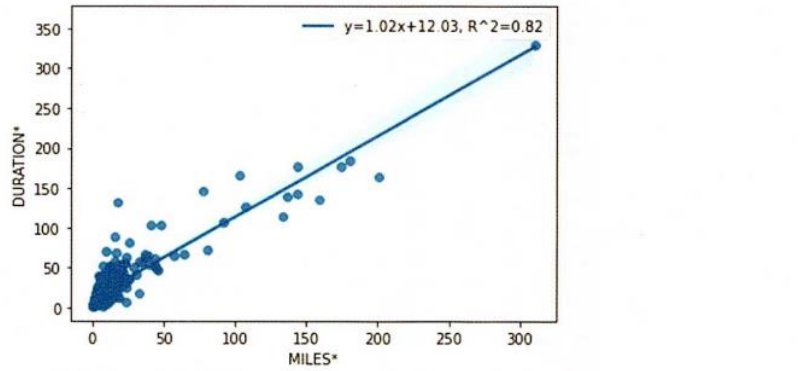
→ 결정계수 R 값이 0.82로, 두 변수는 서로 강한 상관관계를 가지고 있다고 할 수 있음

데이터를 처음 만나면 하는 것들 (ch2-1.py)

```
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import linregress

df = df.dropna()
s, i, r, _, _ = linregress(df['MILES*'], df['DURATION*'])

fig, ax = plt.subplots()
sns.regplot(x='MILES*', y='DURATION*', data=df, ax=ax,
            line_kws={'label':'y={:.2f}x+{:.2f}, R^2={:.2f}'.format(s, i, r**2)})
plt.legend()
```



▲ 그림 35 Uber 데이터셋에서 운행 거리와 운행 시간을 seaborn 라이브러리 regplot으로 시각화

■ Uber 데이터셋을 이용

- 각 변수의 분포와 형태를 살펴보았고, 특정 변수의 그룹별 데이터의 통계치를 확인
- 원 데이터셋의 변수들을 통해 새로운 변수(운행 시간)를 생성해 보았으며, 두 개의 수치형 변수에 대해 상관관계를 확인

■ German_credit 데이터셋

- 임의의 독일인에 대해 나이, 성별, 직업, 주택 보유 유무, 만기, 부채의 목적 및 크기 등을 나타냄

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
df = pd.read_csv('datasets/German_credit/German_credit.csv')
df.head()
```

	Age	Sex	Job	Housing	Saving accounts	Checking account	Credit amount	Duration	Purpose
0	67	male	2	own	NaN	little	1169	6	radio/TV
1	22	female	2	own	little	moderate	5951	48	radio/TV
2	49	male	1	own	little	NaN	2096	12	education
3	45	male	2	free	little	little	7882	42	furniture/equipment
4	53	male	2	free	little	little	4870	24	car

▲ 그림 36 German_credit 데이터셋을 불러와 head 메서드를 통해 처음 5개 행을 불러온 결과

- “Age" 열은 나이를 나타내는 변수로, 연속형 정수

  → 특정 나이 범위를 묶어서 그룹화하여 범주형 변수로 나타낼 수 있는 대표적인 항목임

■ Describe() → "Age" 열

- “Age" 열의 최소값은 19, 최대값은 75, 평균은 대략 36세

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
df['Age'].describe()
```

```
count    1000.000000
mean       35.546000
std        11.375469
min        19.000000
25%        27.000000
50%        33.000000
75%        42.000000
max        75.000000
Name: Age, dtype: float64
```

각 나이를 10 단위로 쪼개서 그룹화

- Cut()
  - 특정 수치 범위를 일정한 길이를 가지는 n개의 구간으로 나눌 때 유용함

    → “Age" 변수를 8개의 구간으로 나누기 → 구간을 나눌 데이터와 bins 인자에 8을 전달

    → bins 인자에 정수형 값이 전달되면 해당 정수만큼의 개수로 전달받은 데이터를 구간화

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
pd.cut(df['Age'], bins=8)
```

```
0        (61.0, 68.0]
1      (18.944, 26.0]
2        (47.0, 54.0]
3        (40.0, 47.0]
4        (47.0, 54.0]
             ...
995      (26.0, 33.0]
996      (33.0, 40.0]
997      (33.0, 40.0]
998    (18.944, 26.0]
999      (26.0, 33.0]
Name: Age, Length: 1000, dtype: category
Categories (8, interval[float64, right]): [(18.944, 26.0] < (26.0, 33.0] < (33.0, 40.0] < (40.0,
47.0] < (47.0, 54.0] < (54.0, 61.0] < (61.0, 68.0] < (68.0, 75.0]]
```

각 나이를 10 단위로 쪼개서 그룹화

- Cut()
  - 특정 수치 범위를 일정한 길이를 가지는 n개의 구간으로 나눌 때 유용함

    → “Age" 변수를 8개의 구간으로 나누기 → 구간을 나눌 데이터와 bins 인자에 8을 전달

    → bins 인자에 정수형 값이 전달되면 해당 정수만큼의 개수로 전달받은 데이터를 구간화

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
pd.cut(df['Age'], bins=8)
```

```
0        (61.0, 68.0]
1      (18.944, 26.0]
2        (47.0, 54.0]
3        (40.0, 47.0]
4        (47.0, 54.0]
              ...
995      (26.0, 33.0]
996      (33.0, 40.0]
997      (33.0, 40.0]
998    (18.944, 26.0]
999      (26.0, 33.0]
Name: Age, Length: 1000, dtype: category
Categories (8, interval[float64, right]): [(18.944, 26.0] < (26.0, 33.0] < (33.0, 40.0] < (40.0,
47.0] < (47.0, 54.0] < (54.0, 61.0] < (61.0, 68.0] < (68.0, 75.0]]
```

각 나이를 10 단위로 쪼개서 그룹화

- Cut() + groupby()
  - (26, 33] 구간에 가장 많은 데이터가 분포하고 있으며, 각 구간이 포함하는 데이터의 개수는 전부 상이한 것을 확인
  - 만약 구간을 직접 지정하고 싶다면 cut 메서드의 bins 인자에 해당 구간을 리스트 형태로 전달
    → 10부터 시작하여 80까지 10의 간격을 가지는 구간으로 Age 변수를 나누기

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
pd.cut(df['Age'], bins=8).reset_index().groupby('Age').size()
```

```
Age
(18.944, 26.0]    240
(26.0, 33.0]      276
(33.0, 40.0]      210
(40.0, 47.0]      123
(47.0, 54.0]       72
(54.0, 61.0]       41
(61.0, 68.0]       31
(68.0, 75.0]        7
dtype: int64
```

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
bins = [10, 20, 30, 40, 50, 60, 70, 80]
pd.cut(df['Age'], bins=bins)
```

```
0      (60, 70]
1      (20, 30]
2      (40, 50]
3      (40, 50]
4      (50, 60]
         ...
995    (30, 40]
996    (30, 40]
997    (30, 40]
998    (20, 30]
999    (20, 30]
Name: Age, Length: 1000, dtype: category
Categories (7, interval[int64, right]): [(10, 20] < (20, 30] < (30, 40] < (40, 50] < (50, 60] < (60,
70] < (70, 80]]
```

각 나이를 10 단위로 쪼개서 그룹화

- Cut() + groupby()
  - cut 메서드는 쪼개진 각 구간에서 왼쪽을 열린 구간, 오른쪽을 닫힌 구간으로 설정
  - 만약 구간의 왼쪽을 닫힌 구간, 오른쪽을 열린 구간으로 설정하고 싶다면 right 인자에 False를 전달

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
bins = [10, 20, 30, 40, 50, 60, 70, 80]
pd.cut(df['Age'], bins=bins, right=False)

0      [60, 70)
1      [20, 30)
2      [40, 50)
3      [40, 50)
4      [50, 60)
         ...
995    [30, 40)
996    [40, 50)
997    [30, 40)
998    [20, 30)
999    [20, 30)
Name: Age, Length: 1000, dtype: category
Categories (7, interval[int64, left]): [[10, 20) < [20, 30) < [30, 40) < [40, 50) < [50, 60) < [60,
70) < [70, 80)]
```

각 나이를 10 단위로 쪼개서 그룹화

- qcut()
  - bins 대신 q 인자를 전달

    → 각 구간의 길이가 일정하지 않은데, 이는 각 구간별로 데이터의 개수를 동일하게 하기 위함

    → German_credit 데이터셋의 "Age" 변수가 정수형 데이터이고, 각 값마다 (나이마다) 중복되는 데이터가 많아서 각 구간별로 동일한 데이터 개수를 가지는 특정 구간으로 나눌 수 없기 때문

    → 각 구간은 구간별 데 이터 개수의 편차가 가장 작은 방향

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
pd.qcut(df['Age'], q=8)
```

```
0      (49.125, 75.0]
1      (18.999, 24.0]
2      (42.0, 49.125]
3      (42.0, 49.125]
4      (49.125, 75.0]
           ...
995      (30.0, 33.0]
996      (36.0, 42.0]
997      (36.0, 42.0]
998    (18.999, 24.0]
999      (24.0, 27.0]
Name: Age, Length: 1000, dtype: category
Categories (8, interval[float64, right]): [(18.999, 24.0] < (24.0, 27.0] < (27.0, 30.0] < (30.0,
33.0] < (33.0, 36.0] < (36.0, 42.0] < (42.0, 49.125] < (49.125, 75.0]]
```

연속 데이터를 그룹화하여 범주형 데이터로 분석하기 (ch2-2.py)

```
pd.qcut(
    df['Age'], q=8, duplicates='drop'
).reset_index().groupby('Age').size()
```

```
Age
(18.999, 24.0]    149
(24.0, 27.0]      142
(27.0, 30.0]      120
(30.0, 33.0]      105
(33.0, 36.0]      111
(36.0, 42.0]      138
(42.0, 49.125]    110
(49.125, 75.0]    125
dtype: int64
```

■ 특정 조건을 만족하는 횟수가 얼마나 연속하여 나타나는지를 고려해야 할 때

- 주식 자동 거래 시스템을 제작하는 과정에서 특정 조건이 얼마동안 연속적으로 만족했을 때 매수 혹은 매도주문을 전달할지 설정하고자 할 때
- 공장의 제품 검수 시스템에서 불량품이 특정 횟수 이상 연속적으로 나왔을 때 알람을 울리도록 설정할 때

■ 사용될 임의 데이터 셋

- 정수형 Series 형태의 데이터셋

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
s = pd.Series([0, 0, 1, 1, 0, 1, 1, 1, 1, 0])
s
```

```
0    0
1    0
2    1
3    1
4    0
5    1
6    1
7    1
8    1
9    0
dtype: int64
```

■ Cumsum()

- 행을 따라 누적합을 구하여 sc라는 변수에 할당
- 원 데이터 s의 누적합인 sc를 다시 원 데이터 s와 곱
  → 원 데이터가 0이 아니던 구간만 0이 아닌 값을 갖게 됨
- diff 메서드를 사용하여 원 데이터를 한 칸씩 아래로 shift 시켜 원 데이터에서 빼기
  → 음수는 연속되어 나타난 1이 끝났음
- 음수만 남 기고 모두 NaN으로 만든 다음, 남은 음수 값을 ffill 메서드를 사용하여 아래쪽으로 전파
- 구한 값을 원 데이터 s의 누적합인 sc와 더해며, 이 때 NaN 값은 모두 0으로 치환

■ Cumsum()

- ***원 데이터인 s와 비교하여 확인해 봤을 때 원 데이터에서 1이 연속해서 나올 때 마 다 값이 1씩 더해지며 늘어나고, 1의 연속이 중단되면 값이 0으로 초기화되는 것을 확인***

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
sc = s.cumsum()
sc
```

```
0    0
1    0
2    1
3    2
4    2
5    3
6    4
7    5
8    6
9    6
dtype: int64
```

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
s.mul(sc)
```

```
0    0
1    0
2    1
3    2
4    0
5    3
6    4
7    5
8    6
9    0
dtype: int64
```

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
s.mul(sc).diff()
```

```
0    NaN
1    0.0
2    1.0
3    1.0
4   -2.0
5    3.0
6    1.0
7    1.0
8    1.0
9   -6.0
dtype: float64
```

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
s.mul(sc).diff().where(lambda x: x<0)
```

```
0    NaN
1    NaN
2    NaN
3    NaN
4   -2.0
5    NaN
6    NaN
7    NaN
8    NaN
9   -6.0
dtype: float64
```

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
s.mul(sc).diff().where(lambda x: x<0).ffill()
```

```
0    NaN
1    NaN
2    NaN
3    NaN
4   -2.0
5   -2.0
6   -2.0
7   -2.0
8   -2.0
9   -6.0
dtype: float64
```

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
s.mul(sc).diff().where(lambda x: x<0).ffill().add(sc, fill_value=0)
```

```
0    0.0
1    0.0
2    1.0
3    2.0
4    0.0
5    1.0
6    2.0
7    3.0
8    4.0
9    0.0
dtype: float64
```

■ APPL_price 데이터셋

- 1980년 12월부터 2022년 6월까지 애플 사의 주식 가격을 나타낸 데이터

■ 데이터에서 애플의 종가 기준 주식가격이 연속적으로 175불 이상이었던 주식거래일 기준 날짜들 중 가장 긴 연속일을 구해보기

- 애플의 종가 기준 주식가격이 175불인 날짜는 연속을 고려하지 않고 총 22일 인 것을 확인

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
df = pd.read_csv('datasets/APPL_price/APPL_price.csv')
s = df['Close'] > 175
s.sum()
```

```
22
```

■ 데이터에서 애플의 종가 기준 주식가격이 연속적으로 175불 이상이었던 주식거래일 기준 날짜들 중 가장 긴 연속일을 구해보기

- 종가가 연속적으로 175불인 주식거래일 중 가장 긴 연속일을 구하기

  → 가장 마지막에 max 메서드를 사용함으로써 가장 긴 연속일을 구할 수 있음

조건을 만족하는 최대 연속 횟수 구하기 (ch2-3.py)

```
sc = s.cumsum()
s.mul(sc).diff().where(lambda x: x<0).ffill().add(sc, fill_value=0).max()
```

```
9.0
```

→ 애플의 종가가 175불 이상이었던 날은 총 22일이며, 그 중 연속적으로 175불 이상이었던 주식거래일 중 가장 긴 연속은 9일인 것을 확인

이상치의 존재는 해당 데이터를 이용한 통계 분석에 큰 영향을 끼칠 수 있고,

분석 결과에 왜곡을 일으킬 수 있음

■ weight_height 데이터셋

- 임의의 사람의 성별, 키, 몸무게가 나와 있는 데이터셋
- describe()

  → eight 변수를 살펴보면, 평균값이 161.5 정도이고, 표준편차가 32.2

  → 최대값 이 390.2인데, 이는 평균에서 표준편차의 7배가량 큰 값

이상치를 다루는 방법: clip, quantile 메서드 (ch2-4.py)

```
df = pd.read_csv('datasets/weight_height/weight_height.csv')
df.describe()
```

	Height	Weight
count	10000.000000	10000.000000
mean	66.367560	161.452378
std	3.847528	32.171523
min	54.263133	64.700127
25%	63.505620	135.818051
50%	66.318070	161.212928
75%	69.174262	187.169525
max	78.998742	390.200000

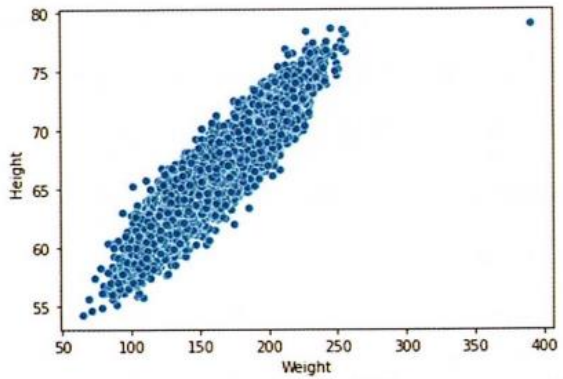
▲ 그림 37 weight_height 데이터셋을 불러와 describe 메서드를 적용한 결과

■ [seaborn 라이브러리]x축을 “Weight"로, y축을 “Height"로 하여 산점도 그래프

- weight height 데이터셋의 데이터가 어떤 형태로 분포되어 있는지 보기
- "Weight"가 대략 390, "Height"가 80정도인 데이터 한 포인트가 다 른 데이터들의 분포에 비해 유달리 떨어져 있는 것을 확인
  → 이상치 인 것으로 가정하고, 삭제

이상치를 다루는 방법: clip, quantile 메서드 (ch2-4.py)

```
import seaborn as sns
sns.scatterplot(x='Weight', y='Height', data=df)
```



▲ 그림 38 Seaborn 라이브러리를 이용한 weight_height 데이터셋의 산점도 그래프. 이상치를 눈으로 확인할 수 있습니다.

■ Query()

- 이상치가 다른 데이터에 비해 명확하게 떨어져 있는 경우

  → query 메서드를 이용하여 "Weight"가 350 이하인 데이터만 필터링

이상치를 다루는 방법: clip, quantile 메서드 (ch2-4.py)

```
df_new = df.query('Weight < 350')
df_new.describe()
```

	Height	Weight
count	9999.000000	9999.000000
mean	66.366297	161.429501
std	3.845646	32.091686
min	54.263133	64.700127
25%	63.505347	135.817009
50%	66.317899	161.201891
75%	69.172069	187.152394
max	78.621374	255.863326

▲ 그림 39 query 메서드를 이용한 weight_height 데이터셋의 이상치 제거. weight가 350 이하인 데이터만 필터링

■ quantile()

- 데이터가 보다 복잡하여 필터링 기준을 알기가 까다로운 경우가 많은데, 이럴때에는 quantile 메서드를 사용

  → 데이터셋에서 분위수를 구하는 것으로, 여기서는 "Weight" 변수가 상위 99.99% 이상인 데이터를 삭제

  → “Weight" 변수의 99.99% 지점의 값은 255.9

이상치를 다루는 방법: clip, quantile 메서드 (ch2-4.py)

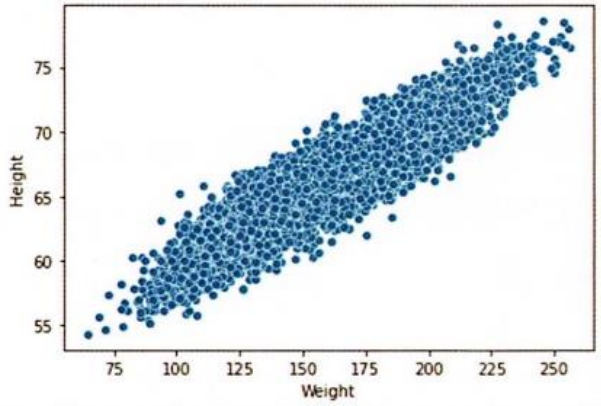
```
criteria = df['Weight'].quantile(0.9999)
criteria
```

```
255.876760167255
```

이상치를 다루는 방법: clip, quantile 메서드 (ch2-4.py)

```
df_new = df[df['Weight'] < criteria]

sns.scatterplot(x='Weight', y='Height', data=df_new)
```



▲ 그림 40 quantile 메서드를 이용하여 weight_height 데이터셋의 이상치 삭제

■ clip()

- 이상치를삭제하기 보다는 이상치를 사용자가 설정한 boundary 값으로 치환할 때 사용
- clip 메서드의 인자로 lower, upper bondary를 전달하게 되는데, 이 값들보다 각각 작거나 큰 데이터는 해당 boudary로 설정한 값으로 치환
- weight_height 데이터셋 Weight 변수의 lower, upper boundary를 각각 50, 300으로 설정
  → 2014번 인덱스의 “Weight" 값이 300으로 치환된 것을 확인

이상치를 다루는 방법: clip, quantile 메서드 (ch2-4.py)

```
df[df['Weight'] > 390].index
```

```
Index([2014], dtype='int64')
```

이상치를 다루는 방법: clip, quantile 메서드 (ch2-4.py)

```
df['Weight'] = df['Weight'].clip(50, 300)
df.iloc[2014]
```

```
Gender          Male
Height     78.998742
Weight         300.0
Name: 2014, dtype: object
```

■ 간단한 데이터셋

- 0부터 8까지의 연속된 인덱스를 가지며, 1부터 시작하여 1씩 증가하는 데이터셋
- 총 4개의 결측치 를 가지게끔 설정

결측치 내삽/외삽하기 (ch2-5.py)

```
df[df['Weight'] > 390].index
```

```
0    1.0
1    2.0
2    3.0
3    NaN
4    NaN
5    6.0
6    7.0
7    NaN
8    NaN
dtype: float64
```

■ Interpolate()

- 각 값들을 선형적으로 (1차 함수) 내삽 및 외삽
- method 인자의 “spline"은 scipy 라이브러리의 interpolate.UnivariateSpline를 통해 내삽 및 외삽 값을 계산
- method 인자에는“nearest", "quadratic", "polynomial" 등 다양한 값을 전달 가능
- limit_direction 인자에는 결측치를 채울 방향을 입력
  → "forward"를 입력하면, 결측치 앞에 결측 치가 아닌 값이 없을 때에는 값이 내삽 혹은 외삽되어 채워지지 않음
  → “backward" 가 전달되면 결측치 뒤쪽에 결측치가 아닌 값이 없다면 값이 채워지지 않음

결측치 내삽/외삽하기 (ch2-5.py)

```
s.interpolate(
    method="spline", order=1, limit_direction="forward", limit=2
)
```

```
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
5    6.0
6    7.0
7    8.0
8    9.0
dtype: float64
```

■ datetime 형식의 데이터나 문자열 형식의 데이터가 인덱스로 설정되어 사전식으로 정렬된 상태

- 데이터셋의 인덱스를 정확히 입력하지 않아도 데이터의 행을 슬라이싱 할 수 있음

■ Covid19-US 데이터셋

- 특정 기간동안 미국의 코로나19 바이러스의 확진 수를 지역으로 나누어 정리한 데이터
- 날짜에 해당하는 열인 "Date"를 datetime 형식으로 변경

정렬된 인덱스에서의 행 슬라이싱 (ch2-6.py)

```
df = pd.read_csv('datasets/Covid19-US/us_confirmed.csv')
df['Date'] = pd.to_datetime(df['Date'])
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Admin2          29948 non-null  object
 1   Date            30000 non-null  datetime64[ns]
 2   Case            30000 non-null  int64
 3   Country/Region  30000 non-null  object
 4   Province/State  30000 non-null  object
dtypes: datetime64[ns](1), int64(1), object(3)
memory usage: 1.1+ MB
```

정렬된 인덱스에서의 행 슬라이싱 (ch2-6.py)

```
df.head()
```

	Admin2	Date	Case	Country/Region	Province/State
0	Price	2020-08-10	33	US	Wisconsin
1	Garvin	2021-08-14	3929	US	Oklahoma
2	Butte	2020-01-24	0	US	Idaho
3	Lowndes	2021-01-25	10377	US	Georgia
4	Ringgold	2020-12-06	260	US	Iowa

▲ 그림 41 Covid19-US 데이터셋을 불러와 head 메서드로 상위 5개 행을 살펴본 결과

■ 날짜에 해당하는 "Date" 열을 인덱스로 지정한 후 인덱스를 정렬

- 2020년 1월과 2020년 2월까지의 데이터를 슬라이싱

정렬된 인덱스에서의 행 슬라이싱 (ch2-6.py)

```
df = df.set_index('Date').sort_values('Date')
df['2020-01':'2020-02']
```

Date	Admin2	Case	Country/Region	Province/State
2020-01-22	Winnebago	0	US	Illinois
2020-01-22	Barceloneta	0	US	Puerto Rico
2020-01-22	Montgomery	0	US	Indiana
2020-01-22	Johnson	0	US	Iowa
2020-01-22	Barron	0	US	Wisconsin
...	...	...	...	...
2020-02-29	Butte	0	US	California
2020-02-29	Essex	0	US	Massachusetts
2020-02-29	Alleghany	0	US	Virginia
2020-02-29	Cayey	0	US	Puerto Rico
2020-02-29	Del Norte	0	US	California

▲ 그림 42 Covid19-US 데이터셋에서 Date 열을 인덱스로 하여 2020년 1월부터 2020년 2월까지의 데이터를 슬라이싱

■ 정렬된 문자열을 인덱스로 하는 데이터셋 또한 사전식으로 데이터를 슬라이싱 할 수 있음

- Date 열을 인덱스로 지정한 것을 reset_index 메서드를 이용하여 초기화→"Province/State" 열을 인덱스로 지정 후 정렬

정렬된 인덱스에서의 행 슬라이싱 (ch2-6.py)

```
df = df.reset_index().set_index('Province/State').sort_index()
df.index.unique()
```

```
Index(['Alabama', 'Alaska', 'American Samoa', 'Arizona', 'Arkansas',
       'California', 'Colorado', 'Connecticut', 'Delaware', 'Diamond Princess',
       'District of Columbia', 'Florida', 'Georgia', 'Grand Princess', 'Guam',
       'Hawaii', 'Idaho', 'Illinois', 'Indiana', 'Iowa', 'Kansas', 'Kentucky',
       'Louisiana', 'Maine', 'Maryland', 'Massachusetts', 'Michigan',
       'Minnesota', 'Mississippi', 'Missouri', 'Montana', 'Nebraska', 'Nevada',
       'New Hampshire', 'New Jersey', 'New Mexico', 'New York',
       'North Carolina', 'North Dakota', 'Northern Mariana Islands', 'Ohio',
       'Oklahoma', 'Oregon', 'Pennsylvania', 'Puerto Rico', 'Rhode Island',
       'South Carolina', 'South Dakota', 'Tennessee', 'Texas', 'Utah',
       'Vermont', 'Virgin Islands', 'Virginia', 'Washington', 'West Virginia',
       'Wisconsin', 'Wyoming'],
      dtype='object', name='Province/State')
```

■ “California"부터 사전 순선대로 “Delaware"까지의 모든 행을 선택

- 인덱스의 모든 문자열을 입력하지 않더라도 행을 슬라이싱
- 각 인덱스의 첫 두 글자만 사용하여서 슬라이싱

  → “Delaware"가 “De"보다 사전식 정렬에서 뒤쪽에 위치하기 때 문에 “" 다음 알파벳인 "f"를 넣어서 “Df"로 검색하면 “Delaware"까지 포함되어 슬라이싱

정렬된 인덱스에서의 행 슬라이싱 (ch2-6.py)

```
df['Ca':'Df']
```

Province/State	Date	Admin2	Case	Country/Region
California	2020-06-04	Tuolumne	4	US
California	2020-06-06	San Benito	90	US
California	2021-12-18	Unassigned	330	US
California	2020-03-22	San Joaquin	33	US
California	2021-04-24	Sierra	63	US
...	...	...	...	...
Delaware	2021-08-04	Out of DE	0	US
Delaware	2020-03-22	Out of DE	0	US
Delaware	2020-06-11	Unassigned	13	US
Delaware	2020-03-16	New Castle	8	US
Delaware	2020-12-06	Out of DE	0	US

▲ 그림 43 Covid19-US 데이터셋에서 Province/State를 문자열 인덱스로 하여 사전식 행 슬라이싱

■ 시계열 데이터에서 특정 날짜나 시간 그룹으로 데이터를 업샘플링 혹은 다운샘플링 한 후, 그 안에서 또 다른 변수로 데이터를 그룹화하여 집계함수를 사용하는 방법

- Covid19-US 데이터셋
- “Date"열을 인덱스로 설정 및 정렬 → 해당 열에서 임의로 3개의 고유값을 뽑은 후 해당 값을 가지는 데이터 로만 데이터셋을 필터

Timestamp데이터를 포함한 여러 열을 기준으로 groupby하기 (ch2-7.py)

```
df = pd.read_csv('datasets/Covid19-US/us_confirmed.csv')
df['Date'] = pd.to_datetime(df['Date'])
df = df.set_index('Date').sort_values('Date')
```

Timestamp데이터를 포함한 여러 열을 기준으로 groupby하기 (ch2-7.py)

```
states = df['Province/State'].unique()[0:3]
df = df[df['Province/State'].isin(states)]
```

- 서브 데이터셋

  → 2020년 1월 22일부터 2022년 4월 16일까지 임의의 날짜에서 “Province/State" 열에 3개의 고유값을 가지고 있음

■ 특정 기간으로 그룹화하고, 해당 기간 그룹 내에서 다시 "Province/State" 열의 값으로 그룹화하여 각 그룹들의 코로나19 바이러스 확진자 수의 평균 구하기

- datetime 형식의 인덱스를 가지는 데이터셋에서 groupby 메서드에 pd. Grouper를 이용
- Grouper의 freq 인자에 resample 메서드에서와 동일한 방식으로 데이터를 샘플링 할 기간/날짜를 문자열 형태로 전달
  → resample 메서드와 동일한 방식으로 시계열 데이터를 샘플링 할 수 있음
- Grouper의 인자에 label이나 closed와 같은 메서드를 동일하게 사용
- groupby 인자에 시계열 인덱스를 6개월로 재구성하기 위한 Grouper와 함께 "Province/State" 열의 이름을 리스트 형태로 묶어서 전달하였고,
  그 결과 시계열 데이터의 샘플링과 특정 변수의 값에 따른 그룹화를 동시에 수행

Timestamp데이터를 포함한 여러 열을 기준으로 groupby하기 (ch2-7.py)

```
df.groupby([pd.Grouper(freq='6m'), 'Province/State'])['Case'].mean()
```

```
Date        Province/State
2020-01-31  Illinois              0.000000
            Indiana               0.000000
            Puerto Rico           0.000000
2020-07-31  Illinois            488.322727
            Indiana             222.080745
            Puerto Rico          18.608187
2021-01-31  Illinois           4996.807487
            Indiana            2683.175532
            Puerto Rico         526.294798
2021-07-31  Illinois           9314.986486
            Indiana            6138.547619
            Puerto Rico        1503.891156
2022-01-31  Illinois          32778.534653
            Indiana           13475.709677
            Puerto Rico        3225.673077
2022-07-31  Illinois          43229.678161
            Indiana           17653.965909
            Puerto Rico        5938.734177
Name: Case, dtype: float64
```

groupby로 각 그룹별 통계치를구하고, 그 통계치를 이용하여 각 그룹별로 데이터를 표준화하는 방법

■ product_inspection데이터

- 특정 제품의 생산 과정 중 “A”, “B", "C" 세 가지의 종류의 검수 공정에서 측정된 값과 날짜, 해당 검수 공정에서의 관리 spec을 함께 나타냄

데이터셋 내 특정 그룹별 데이터 표준화 (ch2-8.py)

```
df = pd.read_csv('datasets/product_inspection/product_inspection.csv')
df['date'] = pd.to_datetime(df['date'])
df.head()
```

	date	inspection_step	value	upper_spec	target	lower_spec
0	2022-01-01	A	21.2	22.0	21.3	20.6
1	2022-01-02	A	21.7	22.0	21.3	20.6
2	2022-01-03	A	21.4	22.0	21.3	20.6
3	2022-01-04	A	21.5	22.0	21.3	20.6
4	2022-01-05	A	21.5	22.0	21.3	20.6

▲ 그림 44 product_inspection 데이터셋을 로드하여 첫 5개 행을 살펴본 결과

- “inspection_step" 열에 대해서 groupby 메서드를 통해 그룹화를 진행하여 "value“ 열의 평균값 구하기

데이터셋 내 특정 그룹별 데이터 표준화 (ch2-8.py)

```
df.groupby('inspection_step')['value'].mean()
```

```
inspection_step
A    21.295105
B    31.628671
C    28.792308
Name: value, dtype: float64
```

■ 일반적으로 표준화는 각 데이터에서 평균을 뺀 값을 표준편차로 나누는 방식으로 진행

- 각 데이터의 평균을 빼고 표준편차를 나눠 제외하는 방법 → transform 메서드와 lambda 함수를 이용
- 임의의 사용자 정의 함수를 통해 그룹별 연산을 진행 후 원 데이터셋의 행 순서대 로 연산 값을 반환 → transform 메서드에 각 데이터에 평균을 빼서 그 값을 표준편차로 나누는 lambda 함수를 전달

데이터셋 내 특정 그룹별 데이터 표준화 (ch2-8.py)

```
df['normalized1'] = df.groupby('inspection_step')['value'].transform(lambda x: (x - x.mean())/
x.std())
df['normalized1']
```

```
0     -0.366795
1      1.561575
2      0.404553
3      0.790227
4      0.790227
         ...
424    1.356267
425   -1.502891
426    0.800320
427    0.482636
428   -1.344049
Name: normalized1, Length: 429, dtype: float64
```

■ “inspection_step" 의 고유값인 "A", "B", "C" 각 그룹별 "value"들에 대해 각 그룹에서 날짜가 가장 빠른 값을 빼는 방법으로 표준화 진행

- "inspection_step" 열과 “date" 열을 기준으로 오름차순 정렬
- drop_duplicates 메서드를 이용하여 “inspection_step" 열을 기준으로 첫 번째 값만 남기고 나머지 중복 값을 삭제
  → "inspection_step" 열을 기준으로 가장 날짜가 빠른 데이터만 남게 됨 → "temp"라는 변수에 저장
- "temp"에 저장된 서브 데이터셋에서 “inspection_step" 열을 인덱스로 설정한 후, 각 인덱스에 해당하는 값은 "value" 열만 남김

데이터셋 내 특정 그룹별 데이터 표준화 (ch2-8.py)

```
temp = df.sort_values(['inspection_step','date']).drop_duplicates('inspection_step')
temp
```

	date	inspection_step	value	upper_spec	target	lower_spec	normalized
0	2022-01-01	A	21.2	22.0	21.3	20.6	-0.366795
143	2022-01-01	B	31.6	32.1	31.6	31.1	-0.159778
286	2022-01-01	C	29.7	32.5	28.9	25.3	0.720899

▲ 그림 45 product_inspection 데이터셋에서 inspection_step의 고유값별로 date가 가장 빠른 행만 남긴 결과

데이터셋 내 특정 그룹별 데이터 표준화 (ch2-8.py)

```
temp = temp.set_index('inspection_step')['value']
temp
```

```
inspection_step
A    21.2
B    31.6
C    29.7
Name: value, dtype: float64
```

■ “inspection_step" 의 고유값인 "A", "B", "C" 각 그룹별 "value"들에 대해 각 그룹에서 날짜가 가장 빠른 값을 빼는 방법으로 표준화 진행

- 원 데이터셋에서도 “inspection_step" 열을 인덱스로 설정한 후, "value" 열을 위의 "temp"로 뺀 후, 그 값을 "normalized2" 열에 저장한 후 그 결과를 확인

데이터셋 내 특정 그룹별 데이터 표준화 (ch2-8.py)

```
df = df.set_index('inspection_step')
df['normalized2'] = df['value'] - temp
df = df.reset_index()
```

	inspection_step	date	value	upper_spec	target	lower_spec	normalized1	normalized2
0	A	2022-01-01	21.2	22.0	21.3	20.6	-0.366795	0.0
1	A	2022-01-02	21.7	22.0	21.3	20.6	1.561575	0.5
2	A	2022-01-03	21.4	22.0	21.3	20.6	0.404553	0.2
3	A	2022-01-04	21.5	22.0	21.3	20.6	0.790227	0.3
4	A	2022-01-05	21.5	22.0	21.3	20.6	0.790227	0.3
...	...	...	...	...	...	...	...	...
424	C	2022-05-19	30.5	32.5	28.9	25.3	1.356267	0.8
425	C	2022-05-20	26.9	32.5	28.9	25.3	-1.502891	-2.8
426	C	2022-05-21	29.8	32.5	28.9	25.3	0.800320	0.1
427	C	2022-05-22	29.4	32.5	28.9	25.3	0.482636	-0.3
428	C	2022-05-23	27.1	32.5	28.9	25.3	-1.344049	-2.6

▲ 그림 46 product_inspection 데이터셋에서 “inspection_step” 변수 그룹별 평균에서 표준편차를 나눈 표준화 결과와 (“normalized1” 열) 각 그룹별 날짜가 가장 빠른 값으로 빼준 표준화 결과 (“normalized2” 열)

- 원 데이터셋인 product_inspeciton의 데이터 길이와 "temp" 변수에 저장했던 데이터셋의 길이가 동일하지 않더라도 두 데이터셋 간의 빼기 연산이 정상적으로 수행되었다는 것을 확인
- 빼기 연산은 각 데이터셋의 인덱스의 값인 "A", "B", "C"에 대해 인덱스가 동일한 행끼리 수행 됨

■ product 데이터셋

- 두 개의 자동차 공장 “A1", “A2"에서 ("factory” 열) 자동차 조립 공정 순서 “P1", "P2", "P3" ("process" 열)에 따른 작업자 (“operator" 열)와 생산된 자동차의 차대번호 ("product_id" 열) 및 자동차 검수 통과 여부("passfail" 열)가 암호화되어 나타나 있는 데이터셋
- 자동차 생산 공정은 factory “A1”이나 “A2 "둘 중 하나에서 진행되며, 어느 한 공장에서 “P1", "P2", "P3" 공정을 모두 순서대로 진행하게 됨
- 각 공정마다 “operator"가 각각 한 명씩 배정되어 생산되는데, “P1" 공정의 “operator”는 "1" 또는 "2", "P2" 는 "V" 또는 "W", "P3"는 “X" 또는 "Y"중 한 명으로 배정됨

groupby 메서드를 이용한 문자열 연산 (ch2-9.py)

```
df = pd.read_csv('datasets/product/product.csv')
df.head()
```

	date	process	factory	operator	product_id	passfail
0	2023-02-01	P1	A2	1	D6523	P
1	2023-02-01	P2	A2	V	D6523	P
2	2023-02-01	P3	A2	Y	D6523	P
3	2023-02-01	P1	A2	1	D7123	P
4	2023-02-01	P2	A2	W	D7123	P

▲ 그림 47 product 데이터셋의 첫 5개 행

■ **하나의 행으로 합쳐서 (공장 이름)(P1 공정의 operator)_(P2 공정의 operator)_(P3 공정의 operator)와 같이 나타내려면?**

- groupby와 문자열 메서드 join을 이용

  → 각 자동차가 ("product_id" 열) 특정 공장 내에서 어떤 “operator"에 의해 제작되었는지를 "P1", "P2", "P3" 공정 순서대로 언더바로 구분하여 나타난 것을 확인

groupby 메서드를 이용한 문자열 연산 (ch2-9.py)

```
df['path'] = df.groupby('product_id')['operator'].transform(lambda x: '_'.join(x))
df['path']
```

```
0       1_V_Y
1       1_V_Y
2       1_V_Y
3       1_W_Y
4       1_W_Y
       ...
1057    1_V_Y
1058    1_V_Y
1059    1_V_Y
1060    1_V_Y
1061    1_V_Y
Name: path, Length: 1062, dtype: object
```

■ **하나의 행으로 합쳐서 (공장 이름)(P1 공정의 operator)_(P2 공정의 operator)_(P3 공정의 operator)와 같이 나타내려면?**

- "path" 열에 각 자동차가 생산된 공장명을 이어 붙인 후, drop_duplicates 메서드를 실행하여 각 "product_id" 값 당 하나의 행만 가지도록 데이터를 정리

groupby 메서드를 이용한 문자열 연산 (ch2-9.py)

```
df['path'] = df['factory'] + '_' + df['path']
df = df.drop_duplicates('product_id')
df = df[['date','product_id','passfail','path']]
df
```

	date	product_id	passfail	path
0	2023-02-01	A259721	P	A2_1_V_Y
3	2023-02-01	A100109	P	A2_1_W_Y
6	2023-02-01	A870944	P	A2_2_W_Y
9	2023-02-01	A587346	P	A2_2_V_Y
12	2023-02-01	A525921	P	A2_2_V_X
...	...	...	...	...
1047	2023-03-31	A917956	P	A1_2_W_Y
1050	2023-03-31	A860621	P	A2_1_V_X
1053	2023-03-31	A268064	P	A2_1_V_X
1056	2023-03-31	A817755	P	A2_1_V_Y
1059	2023-03-31	A332162	P	A1_1_V_Y

▲ 그림 48 product 데이터셋에서 각 자동차가 생산된 공장과 각 공정의 "operator"를 언더바로 구분하여 "path" 열에 저장한 결과

■ **하나의 행으로 합쳐서 (공장 이름)(P1 공정의 operator)_(P2 공정의 operator)_(P3 공정의 operator)와 같이 나타내려면?**

- value_counts 메서드를 이용하여 “passfail" 열의 값이 “F" (fail을 뜻함) 인 행들은 어떤 “path" 값들을 가지는지 확인
- “date " 에 대해서도 “passfail " 변수와의 연관성 확인

groupby 메서드를 이용한 문자열 연산 (ch2-9.py)

```
df.groupby('passfail')['path'].value_counts()
```

```
passfail  path
F         A2_2_W_X    3
          A2_2_V_X    3
          A2_1_V_X    2
          A2_1_W_X    1
P         A1_1_V_X    32
          A1_1_V_Y    26
          A2_1_W_Y    24
          A2_1_V_X    24
          A1_1_W_Y    23
          A1_2_W_Y    23
          A2_2_W_X    22
          A1_2_V_X    22
          A1_2_V_Y    22
          A2_2_W_Y    21
          A2_1_V_Y    20
          A2_1_W_X    20
          A2_2_V_Y    17
          A1_1_W_X    17
          A2_2_V_X    16
          A1_2_W_X    16
Name: count, dtype: int64
```

groupby 메서드를 이용한 문자열 연산 (ch2-9.py)

```
df.groupby(['passfail'])['date'].value_counts()
```

```
passfail  date
F         2023-03-10    3
          2023-03-12    2
          2023-03-11    2
          2023-03-09    2
P         2023-02-28    6
                        ..
          2023-02-22    6
          2023-03-12    4
          2023-03-11    4
          2023-03-09    4
          2023-03-10    3
Name: count, Length: 63, dtype: int64
```

■ 사용 데이터셋

- 언더바를 기준으로 첫 번째는 공장의 이름, 두 번째부터 네 번째 까지는 각각 자동차 생산 공정 “P1", "P2", "P3"에 대한 “operator"의 id를 나타냄

하나의 행을 여러 개의 행으로 쪼개기 (ch2-10.py)

```
df = pd.read_csv('datasets/product/product.csv')
df['path'] = df.groupby('product_id')['operator'].transform(lambda x: '_'.join(x))
df['path'] = df['factory'] + '_' + df['path']
df = df.drop_duplicates('product_id')
df = df[['date','product_id','passfail','path']]

df['path'].head()
```

```
0     A2_1_V_Y
3     A2_1_W_Y
6     A2_2_W_Y
9     A2_2_V_Y
12    A2_2_V_X
Name: path, dtype: object
```

■ 공장명을 “factory"열로 떼서 분리

- 언더바를 기준으로 첫 번째는 공장의 이름, 두 번째부터 네 번째 까지는 각각 자동차 생산 공정 “P1", "P2", "P3"에 대한 “operator"의 id를 나타내

하나의 행을 여러 개의 행으로 쪼개기 (ch2-10.py)

```
df['factory'] = df['path'].map(lambda x: x[0:2])
df['path'] = df['path'].map(lambda x: x[3:])

df.head()
```

	date	product_id	passfail	path	factory
0	2023-02-01	A259721	P	1_V_Y	A2
3	2023-02-01	A100109	P	1_W_Y	A2
6	2023-02-01	A870944	P	2_W_Y	A2
9	2023-02-01	A587346	P	2_V_Y	A2
12	2023-02-01	A525921	P	2_V_X	A2

▲ 그림 49 제품 id당 하나의 행으로 합쳐진 product 데이터셋에서 “factory id”를 따로 떼어 “factory” 열에 추가

■ path 열에 있는 값을 언더바를 기준으로 쪼개어 list 형태

- explode 메서드를 이용하여 이 값을 행으로 쪼갤 예정인데, 이를 위하여 리스트 형식으로 데이터를 준비
- explode 메서드를 이용하여 path 열의 데이터를 행으로 쪼개기

하나의 행을 여러 개의 행으로 쪼개기 (ch2-10.py)

```
df['path'] = df['path'].map(lambda x: x.split('_'))
df['path'].head()
```

```
0     [1, V, Y]
3     [1, W, Y]
6     [2, W, Y]
9     [2, V, Y]
12    [2, V, X]
Name: path, dtype: object
```

하나의 행을 여러 개의 행으로 쪼개기 (ch2-10.py)

```
df = df.explode('path')
df.head(9)
```

	date	product_id	passfail	path	factory
0	2023-02-01	A259721	P	1	A2
0	2023-02-01	A259721	P	V	A2
0	2023-02-01	A259721	P	Y	A2
3	2023-02-01	A100109	P	1	A2
3	2023-02-01	A100109	P	W	A2
3	2023-02-01	A100109	P	Y	A2
6	2023-02-01	A870944	P	2	A2
6	2023-02-01	A870944	P	W	A2
6	2023-02-01	A870944	P	Y	A2

▲ 그림 50 explode 메서드를 이용하여 리스트 형태의 데이터를 행으로 쪼개기

■ 각 행에 대해 어떤 공정인지를 나타내는 “process" 열을 추가한 후, “path" 열 이름을 “operator”로 변경

- explode 메서드를 이용하여 다시 세 개의 행으로 쪼개서 원 데이터셋과 거의 유사하게 나타난 것을 확인

하나의 행을 여러 개의 행으로 쪼개기 (ch2-10.py)

```
process_map = {
    '1':'P1',
    '2':'P1',
    'V':'P2',
    'W':'P2',
    'X':'P3',
    'Y':'P3'
}

df['process'] = df['path'].map(process_map)
df = df.rename({'path':'operator'}, axis=1)

df.head(9)
```

	date	product_id	passfail	operator	factory	process
0	2023-02-01	A259721	P	1	A2	P1
0	2023-02-01	A259721	P	V	A2	P2
0	2023-02-01	A259721	P	Y	A2	P3
3	2023-02-01	A100109	P	1	A2	P1
3	2023-02-01	A100109	P	W	A2	P2
3	2023-02-01	A100109	P	Y	A2	P3
6	2023-02-01	A870944	P	2	A2	P1
6	2023-02-01	A870944	P	W	A2	P2
6	2023-02-01	A870944	P	Y	A2	P3

▲ 그림 51 explode 메서드를 이용해 행으로 나뉘어진 product 데이터셋에서 process 행을 추가하고 path 열의 이름을 operator로 변경