



단국대학교
SW중심대학

전공별 시활용

데이터 사전처리



CONTENTS

1. 누락 데이터 처리
2. 중복 데이터 처리
3. 데이터 표준화
 - 3-1. 단위 환산
 - 3-2. 자료형 변환
4. 범주형(카테고리) 데이터 처리
 - 4-1. 구간 분할
 - 4-2. 더미 변수
5. 정규화

- 데이터프레임에는 원소 데이터 값이 종종 누락되는 경우가 있음
 - 데이터를 파일로 입력할 때 빠뜨리거나 파일 형식을 변환하는 과정에서 데이터가 소실되는 것이 주요 원인
- 유효한 데이터 값이 존재하지 않은 누락 데이터를 NaN(Not a Number)로 표시
- 머신러닝 분석 모형에 데이터를 입력하기 전에 반드시 누락 데이터를 제거하거나 다른 적절한 값으로 대체하는 과정이 필요
 - 누락 데이터가 많아지면 데이터의 품질이 떨어지고, 머신러닝 분석 알고리즘을 왜곡하는 현상이 발생하기 때문

■ 누락 데이터 확인

[예제 5-1] 누락 데이터 확인

Seaborn 라이브러리의 'titanic' 데이터셋을 사용한다. 첫 5행을 출력하면 'deck' 열에 NaN값이 있다. 이 승객의 경우 몇 번 데크에 승선했는지 데이터가 없다는 뜻이다.

```
2
3 # 라이브러리 불러오기
4 import seaborn as sns
5
6 # titanic 데이터셋 가져오기
7 df = sns.load_dataset('titanic')
```

⟨Python 실행⟩ 코드 1~7라인을 부분 실행한 후 IPython 콘솔에서 head() 메소드 실행

```
In [2]: df.head()
Out[2]:
```

	survived	pclass	sex	age	...	deck	embark_town	alive	alone
0	0	3	male	22.0	...	NaN	Southampton	no	False
1	1	1	female	38.0	...	C	Cherbourg	yes	False
2	1	3	female	26.0	...	NaN	Southampton	yes	True
3	1	1	female	35.0	...	C	Southampton	yes	False
4	0	3	male	35.0	...	NaN	Southampton	no	True

[5 rows x 15 columns]

■ 누락 데이터 확인

[예제 5-1] 누락 데이터 확인

Info() 메소드로 데이터프레임의 요약 정보를 출력하면 각 열의 속하는 데이터 중에서 유효한 (non-null, 즉 NaN 값이 아닌) 값의 개수를 보여준다. RangeIndex를 보면 각 열의 891개의 데이터가 있다. 'deck' 열에는 203개의 유효한 범주형 데이터가 있다. (688개의 누락 데이터가 있음)

〈Python 실행〉 IPython 콘솔에서 info() 메소드 실행

```
In [3]: df.info()
Out [3]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
survived      891 non-null int64
pclass        891 non-null int64
sex           891 non-null object
age           714 non-null float64
sibsp         891 non-null int64
parch         891 non-null int64
fare          891 non-null float64
embarked      889 non-null object
class         891 non-null category
who           891 non-null object
adult_male    891 non-null bool
deck          203 non-null category
embark_town   889 non-null object
alive         891 non-null object
alone         891 non-null bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 63.0+ KB
```

■ 누락 데이터 확인

[예제 5-1] 누락 데이터 확인

value_count() 메소드를 이용하여 'deck'열에 688개의 누락 데이터가 있는 것을 파악할 수 있다. 단, 이때 누락 데이터의 개수를 확인하려면 반드시 **dropna = False** 옵션을 사용한다. 그렇지 않으면 NaN값을 제외하고 유효한 데이터의 개수만큼 구하기 때문이다.

```
9 # deck 열의 NaN 개수 계산하기
10 nan_deck = df['deck'].value_counts(dropna=False)
11 print(nan_deck)
```

〈실행 결과〉 코드 9~11라인을 부분 실행

NaN	688
C	59
B	47
D	33
E	32
A	15
F	13
G	4

Name: deck, dtype: int64

■ 누락 데이터 확인

[예제 5-1] 누락 데이터 확인

Notnull() 메소드를 적용하면 유효한 값이 있는 경우 True를 반환하고 누락 데이터가 있는 경우 False를 반환한다. 'deck' 열의 0행에 위치한 원소는 False 값을 갖기 때문에 누락 데이터이다.

```
16 # notnull() 메소드로 누락 데이터 찾기
17 print(df.head().notnull())
```

〈실행 결과〉 코드 16~17라인을 부분 실행

	survived	pclass	sex	age	...	deck	embark_town	alive	alone
0	True	True	True	True	...	False	True	True	True
1	True	True	True	True	...	True	True	True	True
2	True	True	True	True	...	False	True	True	True
3	True	True	True	True	...	True	True	True	True
4	True	True	True	True	...	False	True	True	True

[5 rows x 15 columns]

■ 누락 데이터 확인

- 누락 데이터를 찾는 직접적인 방법으로 `isnull()` 메소드와 `notnull()` 메소드가 있음

- `isnull()` : 누락 데이터면 `True`를 반환하고, 유효한 데이터가 존재하면 `False`를 반환한다.
- `notnull()` : 유효한 데이터가 존재하면 `True`를 반환하고, 누락 데이터면 `False`를 반환한다.

[예제 5-1] 누락 데이터 확인

`head()` 메소드를 적용하여 데이터프레임 객체를 반환하고 첫 5행의 원소들이 누락 데이터인지 여부를 `isnull()` 메소드를 적용하여 판별한다. 'deck' 열의 0행에 있는 원소는 `True` 값이므로 누락 데이터이다.

```
13 # isnull() 메소드로 누락 데이터 찾기
14 print(df.head().isnull())
```

〈실행 결과〉 코드 13~14라인을 부분 실행

	survived	pclass	sex	age	...	deck	embark_town	alive	alone
0	False	False	False	False	...	True	False	False	False
1	False	False	False	False	...	False	False	False	False
2	False	False	False	False	...	True	False	False	False
3	False	False	False	False	...	False	False	False	False
4	False	False	False	False	...	True	False	False	False

[5 rows x 15 columns]

1. 누락 데이터 처리

■ 누락 데이터 확인

[예제 5-1] 누락 데이터 개수 계산

누락 데이터의 개수를 구할 때 `isnull()` 메소드와 `notnull()` 메소드를 활용할 수 있다.
`isnull()` 메소드의 경우 반환되는 값이 참이면 1, 거짓이면 0으로 판별한다.
`isnull()` 메소드를 실행하고 **`sum(axis=0)` 메소드를 적용하면 참(1)의 합을 구한다.**
=> 각 열의 누락 데이터(NaN) 개수를 구할 수 있다. 'deck' 열에만 3개의 누락 데이터가 존재한다.

<실행 결과> 코드 13~14라인을 부분 실행

	survived	pclass	sex	age	...	deck
0	False	False	False	False	...	True
1	False	False	False	False	...	False
2	False	False	False	False	...	True
3	False	False	False	False	...	False
4	False	False	False	False	...	True

[5 rows x 15 columns]

```
19 # isnull() 메소드로 누락 데이터 개수 구하기
20 print(df.head().isnull().sum(axis=0))
```

<실행 결과> 코드 19~20라인을 부분 실행

survived	0
pclass	0
sex	0
age	0
sibsp	0
parch	0
fare	0
embarked	0
class	0
who	0
adult_male	0
deck	3
embark_town	0
alive	0
alone	0

dtype: int64

■ 누락 데이터 제거

- 열을 삭제하면 분석 대상이 갖는 특성(변수)를 제거하고, 행을 삭제하면 분석 대상의 관측값(레코드)을 제거하게 됨

[예제 5-2] 누락 데이터 확인

'titanic' 데이터셋의 각 열(변수)에 누락 데이터가 몇 개씩 포함되어 있는지 체크한다.
IsNull() 메소드와 value_counts() 메소드를 적용한 결과, 'age' 열에 177개, 'embarked' 열에 2개, 'deck' 열에 688개, 'embark_town' 열에 2개가 있다.

〈실행 결과〉 코드 1~17라인을 부분 실행

```
# 라이브러리 불러오기
import seaborn as sns

# titanic 데이터셋 가져오기
df = sns.load_dataset('titanic')

# for 반복문으로 각 열의 NaN 개수 계산하기
missing_df = df.isnull()
for col in missing_df.columns:
    missing_count = missing_df[col].value_counts(dropna = False) #각 열의 고유값별 갯수 파악
    try:
        print(col, ': ', missing_count[True]) # NaN 값이 있으면 개수를 출력
    except:
        print(col, ': ', 0) # NaN 값이 없으면 0개 출력
```

```
False    889
True       2
Name: embarked, dtype: int64
```

```
survived : 0
pclass : 0
sex : 0
age : 177
sibsp : 0
parch : 0
fare : 0
embarked : 2
class : 0
who : 0
adult_male : 0
deck : 688
embark_town : 2
alive : 0
alone : 0
```

■ 누락 데이터 제거

[예제 5-2] 누락 데이터 제거

전체 891명의 승객 중에서 688명의 데크(' deck ' 열) 데이터가 누락되어 있으며 차지하는 비율이 매우 높기 때문에 'deck'열의 누락 데이터를 삭제하여 분석에서 제외하도록 한다.
dropna() 메소드에 thresh=500 옵션을 적용하여 NaN값을 500개 이상 갖는 모든 열을 삭제한다.
=> 'deck' 열만 이 조건에 해당되어 제거된다.

```
19 # NaN값이 500개 이상인 열을 모두 삭제 - deck 열(891개 중 688개의 NaN값)  
20 df_thresh = df.dropna(axis=1, thresh=500)  
21 print(df_thresh.columns)
```

〈실행 결과〉 코드 19~21라인을 부분 실행

```
Index(['survived', 'pclass', 'sex', 'age', 'sibsp', 'parch', 'fare',  
      'embarked', 'class', 'who', 'adult_male', 'embark_town', 'alive',  
      'alone'],  
      dtype='object')
```

■ 누락 데이터 제거

[예제 5-2] 누락 데이터 제거

전체 891명의 승객 중에서 177명은 나이(age)에 대한 데이터가 없다.
승객의 나이가 데이터 분석의 중요한 변수라면 나이 데이터가 없는 승객의 레코드(행)을 제거하는 것이 좋다. `dropna()` 메소드에 기준이 되는 subset을 'age'열로 한정하면 'age' 열의 데이터 중에서 NaN값이 있는 행(axis=0)을 삭제한다. 기본값으로 `how='any'` 옵션이 적용되는데, 기준이 되는 모든 열들의 행 데이터 중에 NaN값이 하나라도 존재하면 삭제한다는 것이다. `how='all'` 옵션으로 입력하면 기준이 되는 모든 열들의 행 데이터가 모두 NaN값일 경우에만 삭제가 된다.
총 891개의 행 중 나이 데이터가 누락된 177행을 삭제하고 나머지 714개의 행을 `df_age`에 저장한다.

〈예제 5-2〉 누락 데이터 제거

(File: example/part5/5.2_dropna.py(이어서 계속))

~ ~ ~ 생략 ~ ~ ~

```
23 # age 열에 나이 데이터가 없는 모든 행 삭제 - age 열(891개 중 177개의 NaN값)
24 df_age = df.dropna(subset=['age'], how='any', axis=0)
25 print(len(df_age))
```

←왼쪽 코드와 비교

```
# age, embarked 열 기준, 나이 데이터, 정박 데이터가 모두 없는 행을 삭제)
df_all = df_thresh.dropna(subset=['age', 'embarked'], how='all', axis=0)
print(df_all)
print(len(df_all))
```

〈실행 결과〉 코드 23~25라인을 부분 실행

714

■ 누락 데이터 치환

– 평균(mean)으로 누락 데이터를 바꿔주는 방법

[예제 5-3] 평균으로 누락 데이터 바꾸기

승객의 나이 데이터가 누락된 행을 제거하지 않고 대신 'age' 열의 나머지 승객의 평균나으로 치환한다. 'age' 열에 들어있는 값들의 평균을 계산하여 mean_age에 저장한다. mean() 메소드를 적용하면 NaN을 제외하고 평균을 계산한다. fillna() 메소드에 mean_age를 인자로 전달하면 NaN을 찾아서 mean age 값으로 치환한다.

```
2
3 # 라이브러리 불러오기
4 import seaborn as sns
5
6 # titanic 데이터셋 가져오기
7 df = sns.load_dataset('titanic')
8
9 # age 열의 첫 10개 데이터 출력 (5행에 NaN값)
10 print(df['age'].head(10))
11 print('\n')
12
13 # age 열의 NaN값을 다른 나이 데이터의 평균으로 변경하기
14 mean_age = df['age'].mean(axis=0) # age 열의 평균 계산 (NaN값 제외)
15 df['age'].fillna(mean_age, inplace=True)
16
17 # age 열의 첫 10개 데이터 출력 (5행에 NaN값이 평균으로 대체)
18 print(df['age'].head(10))
```

<실행 결과> 코드 전부 실행

```
0    22.0
1    38.0
2    26.0
3    35.0
4    35.0
5     NaN
6    54.0
7     2.0
8    27.0
9    14.0
Name: age, dtype: float64
```

```
0    22.000000
1    38.000000
2    26.000000
3    35.000000
4    35.000000
5    29.699118
6    54.000000
7     2.000000
8    27.000000
9    14.000000
Name: age, dtype: float64
```

■ 누락 데이터 치환

– 가장 많이 나타나는 값으로 NaN을 치환하는 방법

[예제 5-4] 가장 많이 나타나는 값으로 바꾸기

승선도시를 나타내는 'embark_town' 열에 있는 NaN을 다른 값으로 바꾼다. 가장 많이 승선한 도시의 이름을 찾아서 NaN을 치환한다. `value_counts()` 메소드를 사용하여 승선도시별 승객 수를 찾고, `idxmax()` 메소드로 가장 큰 값을 갖는 도시(Southampton)를 찾는다. 실행 결과에서 829행의 NaN값을 포함해서 누락 데이터들은 Southampton으로 변경된다.

```
import seaborn as sns

# titanic 데이터셋 가져오기
df = sns.load_dataset('titanic')

# embark_town 열의 829행의 NaN 데이터 출력
print(df['embark_town'][825:830])
print('\n')

# embark_town 열의 NaN값을 승선도시 중에서 가장 많이 출현한 값으로 치환하기
print(df['embark_town'].value_counts(dropna=True))
most_freq = df['embark_town'].value_counts(dropna=True).idxmax()
print(most_freq)
print('\n')

df['embark_town'].fillna(most_freq, inplace=True)

# embark_town 열 829행의 NaN 데이터 출력 (NaN 값이 most_freq 값으로 대체)
print(df['embark_town'][825:830])
```

```
825    Queenstown
826    Southampton
827     Cherbourg
828    Queenstown
829              NaN
Name: embark_town, dtype: object
```

```
Southampton    644
Cherbourg       168
Queenstown       77
Name: embark_town, dtype: int64
Southampton
```

```
825    Queenstown
826    Southampton
827     Cherbourg
828    Queenstown
829    Southampton
Name: embark_town, dtype: object
```

■ 누락 데이터 치환

– 앞이나 뒤에서 이웃하고 있는 값으로 치환하는 방법

[예제 5-5] 이웃하고 있는 값으로 바꾸기

데이터셋의 특성상 서로 이웃하고 있는 데이터끼리 유사성을 가질 가능성이 높다. Fillna() 메소드에 method='ffill' 옵션을 추가하면 NaN이 있는 행의 **직전 행에 있는 값으로 바꿔준다**. method='bfill' 옵션을 사용하면 NaN이 있는 행의 **바로 다음 행에 있는 값을 가지고 치환한다**. 'ffill' 옵션을 사용하여 829행의 NaN값을 바로 앞에 위치한 828행의 Queenstown으로 변경한다.

```
3 # 라이브러리 불러오기
4 import seaborn as sns
5
6 # titanic 데이터셋 가져오기
7 df = sns.load_dataset('titanic')
8
9 # embark_town 열 829행의 NaN 데이터 출력
10 print(df['embark_town'][825:830])
11 print('\n')
12
13 # embark_town 열의 NaN값을 바로 앞에 있는 828행의 값으로 변경하기
14 df['embark_town'].fillna(method='ffill', inplace=True)
15 print(df['embark_town'][825:830])
```

〈실행 결과〉 코드 전부 실행

```
825    Queenstown
826    Southampton
827     Cherbourg
828    Queenstown
829             NaN
Name: embark_town, dtype: object

825    Queenstown
826    Southampton
827     Cherbourg
828    Queenstown
829    Queenstown
Name: embark_town, dtype: object
```

■ 중복 데이터 확인

[예제 5-6] 중복 데이터 확인

동일한 관측값이 중복되는지 여부, 즉 행의 레코드가 중복되는지 여부를 확인하려면 `uplicated()` 메소드를 이용한다. 이전에 나온 행들과 비교하여 중복되는 행이면 `True` 반환, 처음 나오는 행에 대해서 `False` 반환한다.

```
import pandas as pd

# 중복 데이터를 갖는 데이터프레임 만들기
df = pd.DataFrame({'c1': ['a', 'a', 'b', 'a', 'b'],
                   'c2': [1, 1, 1, 2, 2],
                   'c3': [1, 1, 2, 2, 2]})

print(df)
print('\n')

# 데이터프레임 전체 행 데이터 중에서 중복값 찾기
sr_dup = df.duplicated()
print(sr_dup)
print(type(sr_dup))
print('\n')
```

	c1	c2	c3
0	a	1	1
1	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

0	False
1	True
2	False
3	False
4	False

dtype: bool
<class 'pandas.core.series.Series'>

2. 중복 데이터 처리

▪ 중복 데이터 확인

[예제 5-6] 중복 데이터 확인

	c1	c2	c3
0	a	1	1
1	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

데이터프레임의 하나의 열에도, `duplicated()` 메소드를 적용할 수 있다.

데이터프레임 `df`의 'c2'열은 정수 1과 2로 구성된다. 1이 처음 나타난 0행과 2가 처음 나타난 행 3행을 제외하고 나머지 1, 2, 4행은 이전에 나온 행과 중복되므로 `True`가 된다.

1, 2행은 데이터 1을 가진 0행과 중복되고, 4행은 데이터 2를 가진 3행과 중복된다.

```
18 # 데이터프레임의 특정 열 데이터에서 중복값 찾기
19 col_dup = df['c2'].duplicated()
20 print(col_dup)
```

〈실행 결과〉 코드 18~20라인을 부분 실행

```
0    False
1     True
2     True
3    False
4     True
Name: c2, dtype: bool
```

▪ 중복 데이터 제거

[예제 5-7] 중복 데이터 제거

`drop_duplicates()` 메소드의 `subset` 옵션에 열 이름의 리스트를 전달할 수 있다. 데이터의 중복 여부를 판별할 때, `subset` 옵션에 해당하는 열을 기준으로 판단한다.
데이터프레임 `df`의 'c2', 'c3' 열을 기준으로 판별하면 0행과 1행, 3행과 4행의 데이터가 각각 중복된다. 0행과 3행은 처음 나타난 데이터라서 제외하고, 1행과 4행의 데이터만 중복으로 판별하고 삭제한다.

```
18 # c2, c3열을 기준으로 중복 행 제거
19 df3 = df.drop_duplicates(subset=['c2', 'c3'])
20 print(df3)
```

〈실행 결과〉 코드 18~20라인을 부분 실행

	c1	c2	c3
0	a	1	1
1	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

삭제

삭제

	c1	c2	c3
0	a	1	1
2	b	1	2
3	a	2	2

■ 중복 데이터 제거

[예제 5-7] 중복 데이터 제거

중복 데이터를 제거하는 명령에는 `drop_duplicates()` 메소드가 있다. 중복되는 행을 제거하고 고유한 관측값을 가진 행들만 남긴다. 원본 객체를 변경하려면 `inplace=True` 옵션을 추가한다. 1행 데이터는 앞에 이웃하고 있는 0행의 데이터와 중복되므로 제거된다.

```
3 # 라이브러리 불러오기
4 import pandas as pd
5
6 # 중복 데이터를 갖는 데이터프레임 만들기
7 df = pd.DataFrame({'c1':['a', 'a', 'b', 'a', 'b'],
8                    'c2':[1, 1, 1, 2, 2],
9                    'c3':[1, 1, 2, 2, 2]})
10 print(df)
11 print('\n')
12
13 # 데이터프레임에서 중복 행 제거
14 df2 = df.drop_duplicates()
15 print(df2)
16 print('\n')
```

〈실행 결과〉 코드 1~16라인을 부분 실행

	c1	c2	c3
0	a	1	1
1	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

	c1	c2	c3
0	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

■ 단위 환산

- 같은 데이터셋 안에서 서로 다른 측정 단위를 사용한다면, 전체 데이터의 일관성 측면에서 문제가 발생함 => 측정 단위를 동일하게 맞추어 필요 있음
- 외국 데이터를 가져오면 국내에서 잘 사용하지 않는 도량형 단위를 사용하는 경우가 많음
- 영미권에서 주로 사용하는 마일, 야드, 온스 등이 있는데 한국에서 사용하는 미터, 평, 그램 등으로 변환하는 것이 좋음
- UCI 자동차 연비 데이터셋에서의 'mpg' 열은 영미권에서 사용하는 '갤런당 마일(mile per gallon)' 단위로 연비를 표시하고 있음
 - => '리터당 킬로미터(km/L)' 단위로 변환해봄
 - (1마일은 1.60934km이고, 1갤런은 3.78541리터임) -> $1\text{mpg} = 0.425\text{km/L}$

■ 단위 환산

[예제 5-8] 단위 환산

0행에 들어 있는 차량의 연비는 18mpg이다. 한국식 연비 표현으로 변환한 'kpl'열의 0행에 해당하는 값은 리터당 7.65킬로미터이다. round(2) 명령은 소수점 아래 둘째자리 반올림을 뜻한다.

```
3 # 라이브러리 불러오기
4 import pandas as pd
5
6 # read_csv() 함수로 df 생성
7 df = pd.read_csv('./auto-mpg.csv', header=None)
8
9 # 열 이름 지정
10 df.columns = ['mpg','cylinders','displacement','horsepower','weight',
11               'acceleration','model year','origin','name']
12 print(df.head(3))
13 print('\n')
14
15 # mpg(mile per gallon)를 kpl(kilometer per liter)로 변환(mpg_to_kpl = 0.425)
16 mpg_to_kpl = 1.60934/3.78541
17
18 # mpg 열에 0.425를 곱한 결과를 새로운 열(kpl)에 추가
19 df['kpl'] = df['mpg'] * mpg_to_kpl
20 print(df.head(3))
21 print('\n')
```

반올림

```
22
23 # kpl 열을 소수점 아래 둘째자리에서 반올림
24 df['kpl'] = df['kpl'].round(2) => 둘째자리까지 반올림
25 print(df.head(3))
```

〈예제 5-8〉 코드 전부 실행

	mpg	cylinders	...	origin	name
0	18.0	8	...	1	chevrolet chevelle malibu
1	15.0	8	...	1	buick skylark 320
2	18.0	8	...	1	plymouth satellite

[3 rows x 9 columns]

	mpg	cylinders	...	name	kpl
0	18.0	8	...	chevrolet chevelle malibu	7.652571
1	15.0	8	...	buick skylark 320	6.377143
2	18.0	8	...	plymouth satellite	7.652571

[3 rows x 10 columns]

	mpg	cylinders	displacement	...	origin	name	kpl
0	18.0	8	307.0	...	1	chevrolet chevelle malibu	7.65
1	15.0	8	350.0	...	1	buick skylark 320	6.38
2	18.0	8	318.0	...	1	plymouth satellite	7.65

[3 rows x 10 columns]

■ 자료형 환산

[예제 5-9] 자료형 변환

숫자가 문자열(object)로 저장된 경우에 숫자형(int 또는 float)으로 변환해야 한다.
dtypes 속성을 사용하여 데이터프레임을 구성하는 각 열의 자료형을 확인한다.
dtypes 속성 대신 info() 메소드를 사용해도 각 열의 자료형을 확인할 수 있다.
엔진 출력을 나타내는 'horsepower'열은 숫자형이 적절하고, 출시연도를 나타내는 'model_year'열과 출시국가를 뜻하는 'origin'열은 카테고리를 나타내는 범주형이 적절하므로 변환한다.

```
3 # 라이브러리 불러오기
4 import pandas as pd
5
6 # read_csv() 함수로 df 생성
7 df = pd.read_csv('./auto-mpg.csv', header=None)
8
9 # 열 이름 지정
10 df.columns = ['mpg','cylinders','displacement','horsepower','weight',
11               'acceleration','model year','origin','name']
12
13 # 각 열의 자료형 확인
14 print(df.dtypes)
15 print('\n')
```

〈실행 결과〉 코드 1~15라인을 부분 실행

mpg	float64
cylinders	int64
displacement	float64
horsepower	object
weight	float64
acceleration	float64
model year	int64
origin	int64
name	object
dtype:	object

■ 자료형 환산

[예제 5-9] 자료형 변환

'horsepower' 열은 현재 object 자료형이나, 엔진 출력의 크기를 나타내는 데이터이므로 숫자형으로 변환해주는 것이 적절하다. 고유값 중 3번째 줄 중간에 문자열 '?'가 섞여 있어서, 이 때문에 CSV 파일을 데이터프레임으로 변환하는 과정에서 문자열로 인식된 것으로 보인다. '?'는 삭제하고 나머지 값은 모두 숫자형으로 변환하는 것이 적절하다.

```
17 # horsepower 열의 고유값 확인
18 print(df['horsepower'].unique())
19 print('\n')
```

〈실행 결과〉 코드 17~19라인을 부분 실행

```
['130.0' '165.0' '150.0' '140.0' '198.0' '220.0' '215.0' '225.0' '190.0'
 '170.0' '160.0' '95.00' '97.00' '85.00' '88.00' '46.00' '87.00' '90.00'
 '113.0' '200.0' '210.0' '193.0' '?' '100.0' '105.0' '175.0' '153.0'
 '180.0' '110.0' '72.00' '86.00' '70.00' '76.00' '65.00' '69.00' '60.00'
 '80.00' '54.00' '208.0' '155.0' '112.0' '92.00' '145.0' '137.0' '158.0'
 '167.0' '94.00' '107.0' '230.0' '49.00' '75.00' '91.00' '122.0' '67.00'
 '83.00' '78.00' '52.00' '61.00' '93.00' '148.0' '129.0' '96.00' '71.00'
 '98.00' '115.0' '53.00' '81.00' '79.00' '120.0' '152.0' '102.0' '108.0'
 '68.00' '58.00' '149.0' '89.00' '63.00' '48.00' '66.00' '139.0' '103.0'
 '125.0' '133.0' '138.0' '135.0' '142.0' '77.00' '62.00' '132.0' '84.00'
 '64.00' '74.00' '116.0' '82.00']
```

■ 자료형 환산

[예제 5-9] 자료형 변환

'horsepower' 열의 문자열 '?'를 NaN값으로 변환한다. `dropna(axis=0)` 메소드로 NaN값이 들어 있는 모든 행을 삭제한다. 'horsepower' 열에는 숫자형으로 변환 가능한 값들만 남는다. `astype('float')` 명령을 이용하여 문자열을 실수형으로 변환한다. 실수형 대신 정수형으로 변환하려면, 'float' 대신 'int'를 입력한다. `dtypes` 속성을 사용하여 변환된 결과가 맞는지 확인한다.

```
21 # 누락 데이터('?') 삭제
22 import numpy as np
23 df['horsepower'].replace('?', np.nan, inplace=True)      # '?'을 np.nan으로 변경
24 df.dropna(subset=['horsepower'], axis=0, inplace=True)  # 누락 데이터 행 삭제
25 df['horsepower'] = df['horsepower'].astype('float')     # 문자열을 실수형으로 변환
26
27 # horsepower 열의 자료형 확인
28 print(df['horsepower'].dtypes)
```

〈실행 결과〉 코드 21~28라인을 부분 실행

float64

■ 자료형 환산

[예제 5-9] 자료형 변환

'origin' 열에는 정수형 데이터 1, 2, 3이 들어 있지만, 실제로는 국가이름인 'USA, EU, JPN'을 뜻한다. `replace()` 메소드를 사용하여 각 숫자 데이터를 국가이름으로 바꿔주면 문자열을 나타내는 `object` 자료형으로 자동 변경된다.

```
31 # origin 열의 고유값 확인
32 print(df['origin'].unique())
33
34 # 정수형 데이터를 문자형 데이터로 변환
35 df['origin'].replace({1:'USA', 2:'EU', 3:'JPN'}, inplace=True)
36
37 # origin 열의 고유값과 자료형 확인
38 print(df['origin'].unique())
39 print(df['origin'].dtypes)
```

〈실행 결과〉 코드 31~39라인을 부분 실행

```
[1 3 2]
['USA' 'JPN' 'EU']
object
```

■ 자료형 환산

[예제 5-9] 자료형 변환

'origin' 열의 국가이름은 문자열 데이터이다. 값을 확인해보면 3개의 국가이름이 계속 반복된다. 유한 개의 고유값이 반복적으로 나타나는 경우에는 범주형 데이터로 표현하는 것이 효율적이다. `astype('category')` 메소드를 이용하면 범주형 데이터로 변환한다. 범주형을 다시 문자열로 변환하려면 `astype('str')` 메소드를 사용한다.

```
42 # 문자열을 범주형으로 변환
43 df['origin'] = df['origin'].astype('category')
44 print(df['origin'].dtypes)
45
46 # 범주형을 문자열로 다시 변환
47 df['origin'] = df['origin'].astype('str')
48 print(df['origin'].dtypes)
```

〈실행 결과〉 코드 42~48라인을 부분 실행

```
category
object
```

■ 자료형 환산

[예제 5-9] 자료형 변환

`sample()` 메소드로 'model year' 열에서 무작위로 3개의 행을 선택해서 출력해본다. 81, 71, 77과 같이 모델 출시연도가 숫자로 기록되어 있고, 자료형은 정수형을 나타내는 `int64`이다. 연도를 뜻하기 때문에 숫자형으로 남겨둬도 무방하지만 연도는 시간적인 순서의 의미는 있으나 숫자의 상대적인 크기는 별 의미가 없다. 데이터는 숫자 형태를 갖더라도 자료형은 범주형으로 표현하는 것이 적절하다.

```
50 # model year 열의 정수형을 범주형으로 변환
51 print(df['model year'].sample(3))
52 df['model year'] = df['model year'].astype('category')
53 print(df['model year'].sample(3))
```

〈실행 결과〉 코드 50~53라인을 부분 실행

```
365    81
44     71

218    77
Name: model year, dtype: int64

332    80
21     70
360    81
Name: model year, dtype: category
Categories (13, int64): [70, 71, 72, 73, ..., 79, 80, 81, 82]
```

■ 구간 분할

- 데이터 분석 알고리즘에 따라서는 연속 데이터를 그대로 사용하기 보다는 일정한 구간(bin)으로 나눠서 분석하는 것이 효율적인 경우가 있음
- 가격, 비용, 효율 등 연속적인 값을 일정한 수준이나 정도를 나타내는 이산적인 값으로 나타내어 구간별 차이를 드러내는 것임
- 연속 변수를 일정한 구간으로 나누고, 각 구간을 범주형 이산 변수로 변환하는 과정을 구간 분할(binning)이라 함
- 판다스 `cut()` 함수를 이용하면 연속 데이터를 여러 구간으로 나누고 범주형 데이터로 변환할 수 있음



[그림 5-1] 구간 분할 경계값 구하기

■ 데이터 구간 분할

[예제 5-10] 데이터 구간 분할

경계값을 구하는 방법 중에서 NumPy 라이브러리의 `histogram()` 함수를 활용하는 방법을 설명한다.
나누려는 구간(bin) 개수를 `bins` 옵션에 입력하면 각 구간에 속하는 값의 개수(count)와 경계값 리스트(`bin_dividers`)를 반환한다.
모두 4개의 경계값을 생성하고 3개의 구간이 만들어진다(46~107.3 구간, 107.3~168.6 구간, 168.6~230 구간)

```
# 라이브러리 불러오기
import pandas as pd
import numpy as np

# read_csv() 함수로 df 생성
df = pd.read_csv('./auto-mpg.csv', header=None)

# 열 이름을 지정
df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'model year', 'origin', 'name']

# horsepower 열의 누락 데이터('?') 삭제하고 실수형으로 변환
df['horsepower'].replace('?', np.nan, inplace=True)      # '?'를 np.nan으로 변경
df.dropna(subset=['horsepower'], axis=0, inplace=True)  # 누락데이터 행을 삭제
df['horsepower'] = df['horsepower'].astype('float')      # 문자열을 실수형으로 변환

# np.histogram 함수로 3개의 bin으로 나누는 경계 값의 리스트 구하기
count, bin_dividers = np.histogram(df['horsepower'], bins=3)
print(count, bin_dividers)
```

```
[257 103  32] [ 46.          107.33333333 168.66666667 230.          ]
```

■ 데이터 구간 분할

[예제 5-10] 데이터 구간 분할

판다스 `cut()` 함수를 이용하면 연속 데이터를 여러 구간으로 나누고 범주형 데이터로 변환할 수 있다. 판다스 `cut()` 함수의 옵션을 설정한다. 경계값의 리스트(`bin_dividers`)를 `bins` 옵션에 할당하고 각 구간의 이름(`bin_names`)을 `labels` 옵션에 할당한다.

```
23 # 3개의 bin에 이름 지정
24 bin_names = ['저출력', '보통출력', '고출력']
25
26 # pd.cut 함수로 각 데이터를 3개의 bin에 할당
27 df['hp_bin'] = pd.cut(x=df['horsepower'],          # 데이터 배열
28                       bins=bin_dividers,          # 경계값 리스트
29                       labels=bin_names,           # bin 이름
30                       include_lowest=True)         # 첫 경계값 포함
31
32 # horsepower 열, hp_bin 열의 첫 15행 출력
33 print(df[['horsepower', 'hp_bin']].head(15))
```

〈실행 결과〉 코드 23~33라인을 부분 실행

	horsepower	hp_bin
0	130.0	보통출력
1	165.0	보통출력
2	150.0	보통출력
3	150.0	보통출력
4	140.0	보통출력
5	198.0	고출력
6	220.0	고출력
7	215.0	고출력
8	225.0	고출력
9	190.0	고출력
10	170.0	고출력
11	160.0	보통출력
12	150.0	보통출력
13	225.0	고출력
14	95.0	저출력

■ 더미 변수

- 카테고리를 나타내는 범주형 데이터를 회귀분석 등 머신러닝 알고리즘에 바로 사용할 수 없는 경우가 있는데, 컴퓨터가 인식 가능한 입력값으로 변환해야 함

=> 숫자 0 또는 1로 표현되는 더미 변수(dummy variable)를 사용함

(여기서의 0과 1은 수의 크고 작음을 나타내지 않고, 어떤 특성이 있는지 없는지 여부만을 표시함)

4. 범주형(카테고리) 데이터 처리

■ 더미 변수

[예제 5-11] 더미 변수

판다스 `get_dummies()` 함수를 사용하면, 범주형 변수의 모든 고유값을 각각 새로운 더미 변수로 변환한다.

'hp_bin' 열의 고유값 3개가 각각 새로운 더미 변수 열의 이름이 된다. 각 행 별로, 범주형 변수가 본래 속해 있던 위치에는 1이 입력되고, 속하지 않았던 위치에는 0이 입력된다.

```
19 # np.histogram 함수로 3개의 bin으로 나누는 경계값의 리스트 구하기
20 count, bin_dividers = np.histogram(df['horsepower'], bins=3)
21
22 # 3개의 bin에 이름 지정
23 bin_names = ['저출력', '보통출력', '고출력']
24
25 # pd.cut으로 각 데이터를 3개의 bin에 할당
26 df['hp_bin'] = pd.cut(x=df['horsepower'],          # 데이터 배열
27                       bins=bin_dividers,          # 경계값 리스트
28                       labels=bin_names,           # bin 이름
29                       include_lowest=True)         # 첫 경계값 포함
30
31 # hp_bin 열의 범주형 데이터를 더미 변수로 변환
32 horsepower_dummies = pd.get_dummies(df['hp_bin'])
33 print(horsepower_dummies.head(15))
```

df['hp_bin']

0	보통출력
1	보통출력
2	보통출력
3	보통출력
4	보통출력
5	고출력
6	고출력
7	고출력
8	고출력
9	고출력
10	고출력
11	보통출력
12	보통출력
13	고출력
14	저출력

〈실행 결과〉 코드 전부 실행

	저출력	보통출력	고출력
0	0	1	0
1	0	1	0
2	0	1	0
3	0	1	0
4	0	1	0
5	0	0	1
6	0	0	1
7	0	0	1
8	0	0	1
9	0	0	1
10	0	0	1
11	0	1	0
12	0	1	0
13	0	0	1
14	1	0	0

▪ 더미 변수

[예제 5-12] 원핫-인코딩

sklearn 라이브러리를 이용해서 원핫인코딩을 처리할 수 있다.
데이터프레임 df의 'hp_bin'열에 들어 있는 범주형 데이터를 0, 1을 원소로 갖는 원핫벡터로 변환한다.
결과는 선형대수학에서 정의하는 희소행렬(sparse matrix)로 정리된다.
1차원 벡터를 2차원 행렬로 변환하고 다시 희소행렬로 변환한다. 희소행렬은 (행, 열) 좌표와 값 형태로 정리된다. (0,1)은 0행의 1열 위치를 말하고, 데이터 값은 숫자 1 또는 0이 입력된다.

```
31 # sklearn 라이브러리 불러오기
32 from sklearn import preprocessing
33
34 # 전처리를 위한 encoder 객체 만들기
35 label_encoder = preprocessing.LabelEncoder()      # label encoder 생성
36 onehot_encoder = preprocessing.OneHotEncoder()    # one hot encoder 생성
37
38 # label encoder로 문자열 범주를 숫자형 범주로 변환
39 onehot_labeled = label_encoder.fit_transform(df['hp_bin'].head(15))
40 print(onehot_labeled)
41 print(type(onehot_labeled))
42
43 # 2차원 행렬로 형태 변경
44 onehot_resaped = onehot_labeled.reshape(len(onehot_labeled), 1)
45 print(onehot_resaped)
46 print(type(onehot_resaped))
```

〈실행 결과〉 코드 전부 실행

```
[1 1 1 1 1 0 0 0 0 0 0 1 1 0 2]
<class 'numpy.ndarray'>
[[1]
 [1]
 [1]
 [1]
 [1]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [1]
 [1]
 [0]
 [2]]
<class 'numpy.ndarray'>
```

■ 더미 변수

[예제 5-12] 원핫-인코딩

sklearn 라이브러리를 이용해서 원핫인코딩을 편하게 처리할 수 있다.

데이터프레임 df의 'hp_bin'열에 들어 있는 범주형 데이터를 0, 1을 원소로 갖는 원핫벡터로 변환한다. 결과는 선형대수학에서 정의하는 희소행렬(sparse matrix)로 정리된다.

1차원 벡터를 **2차원 행렬로 변환하고 다시 희소행렬로 변환한다**. 희소행렬은 (행, 열) 좌표와 값 형태로 정리된다. (0,1)은 0행의 1열 위치를 말하고, 데이터 값은 숫자 1, 0이 입력된다.

```
48 # 희소행렬로 변환
49 onehot_fitted = onehot_encoder.fit_transform(onehot_reshaped)
50 print(onehot_fitted)
51 print(type(onehot_fitted))
```

```
[[1]
 [1]
 [1]
 [1]
 [1]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [1]
 [1]
 [0]
 [2]]
<class 'numpy.ndarray'>
```

???

```
(0, 1)      1.0
(1, 1)      1.0
(2, 1)      1.0
(3, 1)      1.0
(4, 1)      1.0
(5, 0)      1.0
(6, 0)      1.0
(7, 0)      1.0
(8, 0)      1.0
(9, 0)      1.0
(10, 0)     1.0
(11, 1)     1.0
(12, 1)     1.0
(13, 0)     1.0
(14, 2)     1.0
<class 'scipy.sparse._csr.csr_matrix'>
```

4. 범주형(카테고리) 데이터 처리

■ 더미 변수

[예제 5-12] 원핫-인코딩

sklearn 라이브러리를 이용해서 원핫인코딩을 편하게 처리할 수 있다.

데이터프레임 df의 'hp_bin'열에 들어 있는 범주형 데이터를 0, 1을 원소로 갖는 원핫벡터로 변환한다. 결과는 선형대수학에서 정의하는 희소행렬(sparse matrix)로 정리된다.

1차원 벡터를 **2차원 행렬로 변환하고 다시 희소행렬로 변환한다**. 희소행렬은 (행, 열) 좌표와 값 형태로 정리된다. (0,1)은 0행의 1열 위치를 말하고, 데이터 값은 숫자 1, 0이 입력된다.

```
48 # 희소행렬로 변환
49 onehot_fitted = onehot_encoder.fit_transform(onehot_reshaped)
50 print(onehot_fitted)
51 print(type(onehot_fitted))
```

```
[[1]
 [1]
 [1]
 [1]
 [1]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [1]
 [1]
 [0]
 [2]]
<class 'numpy.ndarray'>
```

```
[[0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [1. 0. 0.]
 [0. 0. 1.]]
<class 'numpy.ndarray'>
```

```
(0, 1) 1.0
(1, 1) 1.0
(2, 1) 1.0
(3, 1) 1.0
(4, 1) 1.0
(5, 0) 1.0
(6, 0) 1.0
(7, 0) 1.0
(8, 0) 1.0
(9, 0) 1.0
(10, 0) 1.0
(11, 1) 1.0
(12, 1) 1.0
(13, 0) 1.0
(14, 2) 1.0
<class 'scipy.sparse._csr.csr_matrix'>
```

■ 정규화

- 각 변수(데이터프레임의 열)에 들어 있는 숫자 데이터의 상대적 크기 차이 때문에 머신러닝 분석 결과가 달라질 수 있음

예) A변수 (0~1000범위의 값), B변수 (0~1범위의 값)

=> 상대적으로 큰 숫자 값을 갖는 A 변수의 영향이 더 커짐

- 숫자 데이터의 상대적인 크기 차이를 제거할 필요가 있음
- 정규화(normalization) : 각 열(변수)에 속하는 데이터 값을 동일한 크기 기준으로 나눈 비율로 나타내는 것
- 정규화 과정을 거친 데이터의 범위는 0~1 또는 -1~1이 됨

■ 정규화

[예제 5-13] 정규화

각 열(변수)의 데이터를 해당 열의 최대값(의 절대값)으로 나누는 방법이 있다.
어떤 열의 원소 값을 그 열의 최대값으로 나누면 가장 큰 값은 최대값 자기자신을 나눈 1이다.
'horsepower' 열의 원래 최대값은 230인데 최대값을 정규화하면 1이 된다.

```
3 # 라이브러리 불러오기
4 import pandas as pd
5 import numpy as np
6
7 # read_csv() 함수로 df 생성
8 df = pd.read_csv('./auto-mpg.csv', header=None)
9
10 # 열 이름 지정
11 df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
12              'acceleration', 'model_year', 'origin', 'name']
13
14 # horsepower 열의 누락 데이터('?') 삭제하고 실수형으로 변환
15 df['horsepower'].replace('?', np.nan, inplace=True) # '?'을 np.nan으로 변경
16 df.dropna(subset=['horsepower'], axis=0, inplace=True) # 누락 데이터 행 삭제
17 df['horsepower'] = df['horsepower'].astype('float') # 문자열을 실수형으로 변환
18
19 # horsepower 열의 통계 요약 정보로 최대값(max) 확인
20 print(df.horsepower.describe())
21 print('\n')
22
23 # horsepower 열의 최대값의 절대값으로 모든 데이터를 나눠서 저장
24 df.horsepower = df.horsepower/abs(df.horsepower.max())
25
26 print(df.horsepower.head())
27 print('\n')
28 print(df.horsepower.describe())
```

〈실행 결과〉 코드 전부 실행

```
count    392.000000
mean     104.469388
std       38.491160
min       46.000000
25%      75.000000
50%      93.500000
75%     126.000000
max      230.000000
Name: horsepower, dtype: float64

0    0.565217
1    0.717391
2    0.652174
3    0.652174
4    0.608696
Name: horsepower, dtype: float64

count    392.000000
mean      0.454215
std       0.167353
min       0.200000
25%      0.326087
50%      0.406522
75%      0.547826
max       1.000000
Name: horsepower, dtype: float64
```

정규화

[예제 5-14] 정규화

각 열별로 '각 데이터에서 해당 열의 최소값을 뺀 값을 분자'로 하고, '해당 열의 최대값과 최소값의 차를 분모'로 하여 계산하는 방법이다. 가장 큰 값은 1, 가장 작은 값은 0이 된다.

```
19 # horsepower 열의 통계 요약 정보로 최대값(max)과 최소값(min) 확인
20 print(df.horsepower.describe())
21 print('\n')
22
23 # horsepower 열의 최대값의 절대값으로 모든 데이터를 나눠서 저장
24 min_x = df.horsepower - df.horsepower.min()
25 min_max = df.horsepower.max() - df.horsepower.min()
26 df.horsepower = min_x/min_max
27
28 print(df.horsepower.head())
29 print('\n')
30 print(df.horsepower.describe())
```

〈실행 결과〉 코드 전부 실행

count	392.000000
mean	104.469388
std	38.491160
min	46.000000
25%	75.000000
50%	93.500000
75%	126.000000
max	230.000000

Name: horsepower, dtype: float64

0	0.456522
1	0.646739
2	0.565217
3	0.565217
4	0.510870

Name: horsepower, dtype: float64

count	392.000000
mean	0.317768
std	0.209191
min	0.000000
25%	0.157609
50%	0.258152
75%	0.434783
max	1.000000

Name: horsepower, dtype: float64

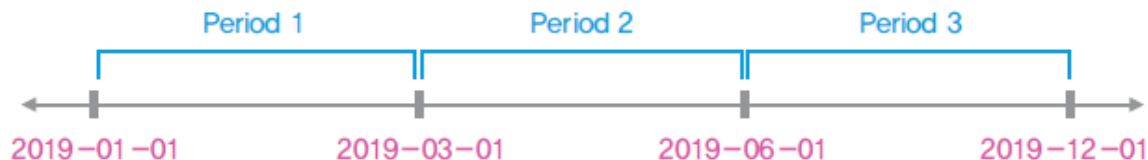
■ 시계열 데이터

주식, 환율 등 금융 데이터를 다루기 위해 개발된 판다스는 시계열(time series) 데이터를 다루는 여러 가지 유용한 기능을 제공한다. 특히 시계열 데이터를 데이터프레임의 행 인덱스로 사용하면, 시간으로 기록된 데이터를 분석하는 것이 매우 편리하다.

판다스의 시간 표시 방식 중에서 시계열 데이터 표현에 자주 이용되는 두 가지 유형을 알아보자. 특정한 시점을 기록하는 Timestamp와 두 시점 사이의 일정한 기간을 나타내는 Period가 있다.



(a) 타임스탬프(Timestamp)



(b) 피리어드(Period)

[그림 5-2] 판다스 시간 표시

■ 문자열을 Timestamp로 변환1

판다스 `to_datetime()` 함수를 사용하면 문자열 등 다른 자료형을 판다스 Timestamp를 나타내는 `datetime64` 자료형으로 변환 가능하다.

주식 시장에서 거래되는 A 종목의 거래 데이터를 정리한 CSV 파일^①을 `read_csv()` 함수를 이용하여 불러온다. `head()` 메소드로 데이터프레임의 일부를 살펴보면, 'Date' 열에 날짜 데이터가 들어 있다. `info()` 메소드로 해당 열의 자료형을 확인하면 문자열(object)임을 알 수 있다.

[예제 5-15]

```
4 import pandas as pd
5
6 # read_csv() 함수로 CSV 파일을 가져와서 df로 변환
7 df = pd.read_csv('stock-data.csv')
8
9 # 데이터 내용 및 자료형 확인
10 print(df.head())
11 print('\n')
12 print(df.info())
```

〈실행 결과〉 코드 1~12라인을 부분 실행

	Date	Close	Start	High	Low	Volume
0	2018-07-02	10100	10850	10900	10000	137977
1	2018-06-29	10700	10550	10900	9990	170253
2	2018-06-28	10400	10900	10950	10150	155769
3	2018-06-27	10900	10800	11050	10500	133548
4	2018-06-26	10800	10900	11000	10700	63039


```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20 entries, 0 to 19
Data columns (total 6 columns):
Date      20 non-null object
Close     20 non-null int64
Start     20 non-null int64
High      20 non-null int64
Low       20 non-null int64
Volume    20 non-null int64
dtypes: int64(5), object(1)
memory usage: 920.0+ bytes
None
```


■ 문자열을 Timestamp로 변환2

'Date' 열의 날짜 데이터를 판다스 Timestamp 객체로 바꿔본다. 'Date' 열을 `to_datetime()` 함수의 인자로 전달하면 문자열(object) 데이터를 `datetime64` 자료형으로 변환한다. 변환된 데이터를 'new_Date' 열에 담아서 데이터프레임 `df`에 추가한다.

[예제 5-15]

```
14 # 문자열 데이터 (시리즈 객체)를 판다스 Timestamp로 변환
15 df['new_Date'] = pd.to_datetime(df['Date']) # df에 새로운 열로 추가
16
17 # 데이터 내용 및 자료형 확인
18 print(df.head())
19 print('\n')
20 print(df.info())
21 print('\n')
22 print(type(df['new_Date'][0]))
```

<실행 결과> 코드 14~22라인을 부분 실행

	Date	Close	Start	High	Low	Volume	new_Date
0	2018-07-02	10100	10850	10900	10000	137977	2018-07-02
1	2018-06-29	10700	10550	10900	9990	170253	2018-06-29
2	2018-06-28	10400	10900	10950	10150	155769	2018-06-28
3	2018-06-27	10900	10800	11050	10500	133548	2018-06-27
4	2018-06-26	10800	10900	11000	10700	63039	2018-06-26

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 20 entries, 0 to 19
```

```
Data columns (total 7 columns):
```

```
Date      20 non-null object
```

```
Close     20 non-null int64
```

```
Start     20 non-null int64
```

```
High      20 non-null int64
```

```
Low       20 non-null int64
```

```
Volume    20 non-null int64
```

```
new_Date  20 non-null datetime64[ns]
```

```
dtypes: datetime64[ns](1), int64(5), object(1)
```

```
memory usage: 1.1+ KB
```

```
None
```

```
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

■ 문자열을 Timestamp로 변환3

'new_Date' 열을 데이터프레임 df의 행 인덱스로 설정하고, 'Date' 열을 제거한다. 이렇게 시계열 값을 행 인덱스로 지정하면 판다스는 DatetimeIndex로 저장한다. 이처럼 시계열 인덱스 클래스를 지원하기 때문에 시간 순서에 맞춰 인덱싱 또는 슬라이싱 하기가 편리하다.

[예제 5-15]

```
24 # 시계열 값으로 변환된 열을 새로운 행 인덱스로 지정. 기존 날짜 열은 삭제
25 df.set_index('new_Date', inplace=True)
26 df.drop('Date', axis=1, inplace=True)
27
28 # 데이터 내용 및 자료형 확인
29 print(df.head())
30 print('\n')
31 print(df.info())
```

<실행 결과> 코드 24~31라인을 부분 실행

new_Date	Close	Start	High	Low	Volume
2018-07-02	10100	10850	10900	10000	137977
2018-06-29	10700	10550	10900	9990	170253
2018-06-28	10400	10900	10950	10150	155769
2018-06-27	10900	10800	11050	10500	133548
2018-06-26	10800	10900	11000	10700	63039

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 20 entries, 2018-07-02 to 2018-06-01
Data columns (total 5 columns):
Close      20 non-null int64
Start      20 non-null int64
High       20 non-null int64
Low        20 non-null int64
Volume     20 non-null int64

dtypes: int64(5)
memory usage: 960.0 bytes
None
```

▪ Timestamp를 Period로 변환

판다스 to_period() 함수를 이용하면 Timestamp 객체를 일정한 기간을 나타내는 Period 객체로 변환할 수 있다.

freq 옵션을 D로 지정할 경우 1일의 기간, M은 1개월의 기간, A는 1년의 기간을 나타내는 데, 이때 1년은 12월에 끝나는 걸로 전제한다.

[예제 5-16]

```
4 import pandas as pd
5
6 # 날짜 형식의 문자열로 구성되는 리스트 정의
7 dates = ['2019-01-01', '2020-03-01', '2021-06-01']
```

```
10 ts_dates = pd.to_datetime(dates)
11 print(ts_dates)
12 print('\n')
13
14 # Timestamp를 Period로 변환
15 pr_day = ts_dates.to_period(freq='D')
16 print(pr_day)
17 pr_month = ts_dates.to_period(freq='M')
18 print(pr_month)
19 pr_year = ts_dates.to_period(freq='A')
20 print(pr_year)
```

〈실행 결과〉 코드 전부 실행

```
DatetimeIndex(['2019-01-01', '2020-03-01', '2021-06-01'], dtype='datetime64[ns]', freq=None)
PeriodIndex(['2019-01-01', '2020-03-01', '2021-06-01'], dtype='period[D]', freq='D')
PeriodIndex(['2019-01', '2020-03', '2021-06'], dtype='period[M]', freq='M')
PeriodIndex(['2019', '2020', '2021'], dtype='period[A-DEC]', freq='A-DEC')
```

▪ Timestamp 배열 만들기1

판다스 `date_range()` 함수를 이용하면 여러 개의 Timestamp가 들어 있는 배열 형태의 시계열 데이터를 만들 수 있다.

예제에서, 생성할 Timestamp들의 시작 옵션은 '2019-01-01'으로 설정하고, 끝 옵션은 None으로 설정하였다. `period` 옵션은 6으로 설정하여 Timestamp를 6개 생성하도록 하였다. `freq` 옵션은 MS로 설정하였는데 이는 월의 시작(Month start)이다.

정리하면, 2019년 1월1일을 시작으로 한달 간격으로 하여, 6개의 월별 시작 시점을 원소로 가지는 Timestamp 배열을 생성한다는 의미이다.

[예제 5-17]

```
4 import pandas as pd
5
6 # Timestamp의 배열 만들기 - 월 간격, 월의 시작일 기준
7 ts_ms = pd.date_range(start='2019-01-01',      # 날짜 범위 시작
8                       end=None,               # 날짜 범위 끝
9                       periods=6,              # 생성할 Timestamp 개수
10                      freq='MS',               # 시간 간격(MS: 월의 시작일)
11                      tz='Asia/Seoul')         # 시간대(timezone)
12 print(ts_ms)
```

〈실행 결과〉 코드 1~12라인을 부분 실행

```
DatetimeIndex(['2019-01-01 00:00:00+09:00', '2019-02-01 00:00:00+09:00',
               '2019-03-01 00:00:00+09:00', '2019-04-01 00:00:00+09:00',
               '2019-05-01 00:00:00+09:00', '2019-06-01 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Seoul]', freq='MS')
```

▪ Timestamp 배열 만들기2

freq옵션을 M으로 설정해보자. 이 경우 월의 마지막 Timestamp들을 원소로 생성한다.

freq옵션을 3M으로 설정해보자. 이 경우 3개월 간격의 Timestamp들을 원소로 생성한다.

[예제 5-17]

```
15 # 월 간격, 월의 마지막 날 기준
16 ts_me = pd.date_range('2019-01-01', periods=6,
17                       freq='M',          # 시간 간격(M: 월의 마지막 날)
18                       tz='Asia/Seoul')   # 시간대(timezone)
19 print(ts_me)
20 print('\n')
21
22 # 분기(3개월) 간격, 월의 마지막 날 기준
23 ts_3m = pd.date_range('2019-01-01', periods=6,
24                       freq='3M',        # 시간 간격(3M: 3개월)
25                       tz='Asia/Seoul')  # 시간대(timezone)
26 print(ts_3m)
```

〈실행 결과〉 코드 15~26라인을 부분 실행

```
DatetimeIndex(['2019-01-31 00:00:00+09:00', '2019-02-28 00:00:00+09:00',
               '2019-03-31 00:00:00+09:00', '2019-04-30 00:00:00+09:00',
               '2019-05-31 00:00:00+09:00', '2019-06-30 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Seoul]', freq='M')

DatetimeIndex(['2019-01-31 00:00:00+09:00', '2019-04-30 00:00:00+09:00',
               '2019-07-31 00:00:00+09:00', '2019-10-31 00:00:00+09:00',
               '2020-01-31 00:00:00+09:00', '2020-04-30 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Seoul]', freq='3M')
```

▪ Period 배열 만들기1

판다스 `period_range()` 함수를 이용하면 여러 개의 Period가 들어 있는 배열 형태의 시계열 데이터를 만들 수 있다.

예제에서, 생성할 Period들의 시작 옵션은 '2019-01-01'으로 설정하고, 끝 옵션은 None으로 설정하였다. `period` 옵션은 3으로 설정하여 Period를 3개 생성하도록 하였다. `freq` 옵션은 M으로 설정하였는데 이는 월을 의미한다.

정리하면, 2019년 1월1일을 시작으로 한달 간격으로 하여, 3개의 각 월(기간)을 원소로 가지는 Period 배열을 생성한다는 의미이다.

[예제 5-18]

```
4 import pandas as pd
5
6 # Period 배열 만들기 - 1개월 길이
7 pr_m = pd.period_range(start='2019-01-01',      # 날짜 범위 시작
8                          end=None,              # 날짜 범위 끝
9                          periods=3,              # 생성할 Period 개수
10                          freq='M')              # 기간의 길이(M: 월)
11 print(pr_m)
```

〈실행 결과〉 코드 1~11라인을 부분 실행

```
PeriodIndex(['2019-01', '2019-02', '2019-03'], dtype='period[M]', freq='M')
```


▪ Period 배열 만들기2

freq옵션을 H로 설정해보자. 이 경우 1시간 간격으로 Period 원소들을 총 3개 생성한다.

freq옵션을 2H로 설정해보자. 이 경우 2시간 간격으로 Period 원소들을 총 3개 생성한다.

[예제 5-18]

```
15 # 월 간격, 월의 마지막 날 기준
16 ts_me = pd.date_range('2019-01-01', periods=6,
17                       freq='M',          # 시간 간격(M: 월의 마지막 날)
18                       tz='Asia/Seoul')   # 시간대 (timezone)
19 print(ts_me)
20 print('\n')
21
22 # 분기(3개월) 간격, 월의 마지막 날 기준
23 ts_3m = pd.date_range('2019-01-01', periods=6,
24                       freq='3M',         # 시간 간격(3M: 3개월)
25                       tz='Asia/Seoul')   # 시간대 (timezone)
26 print(ts_3m)
```

〈실행 결과〉 코드 14~27라인을 부분 실행

```
PeriodIndex(['2019-01-01 00:00', '2019-01-01 01:00', '2019-01-01 02:00'],
            dtype='period[H]', freq='H')
```

```
PeriodIndex(['2019-01-01 00:00', '2019-01-01 02:00', '2019-01-01 04:00'],
            dtype='period[2H]', freq='2H')
```

▪ Period 배열 만들기2

freq옵션을 H로 설정해보자. 이 경우 1시간 간격으로 Period 원소들을 총 3개 생성한다.

freq옵션을 2H로 설정해보자. 이 경우 2시간 간격으로 Period 원소들을 총 3개 생성한다.

[예제 5-18]

```
15 # 월 간격, 월의 마지막 날 기준
16 ts_me = pd.date_range('2019-01-01', periods=6,
17                       freq='M',          # 시간 간격(M: 월의 마지막 날)
18                       tz='Asia/Seoul')   # 시간대 (timezone)
19 print(ts_me)
20 print('\n')
21
22 # 분기(3개월) 간격, 월의 마지막 날 기준
23 ts_3m = pd.date_range('2019-01-01', periods=6,
24                       freq='3M',         # 시간 간격(3M: 3개월)
25                       tz='Asia/Seoul')   # 시간대 (timezone)
26 print(ts_3m)
```

〈실행 결과〉 코드 14~27라인을 부분 실행

```
PeriodIndex(['2019-01-01 00:00', '2019-01-01 01:00', '2019-01-01 02:00'],
            dtype='period[H]', freq='H')
```

```
PeriodIndex(['2019-01-01 00:00', '2019-01-01 02:00', '2019-01-01 04:00'],
            dtype='period[2H]', freq='2H')
```

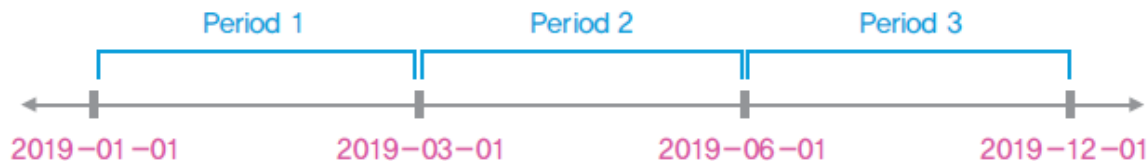

왜 Timestamp 또는 Period를 만드는가?

주식, 환율 등 금융 데이터를 다루기 위해 개발된 판다스는 시계열(time series) 데이터를 다루는 여러 가지 유용한 기능을 제공한다. 특히 시계열 데이터를 데이터프레임의 행 인덱스로 사용하면, 시간으로 기록된 데이터를 분석하는 것이 매우 편리하다.

판다스의 시간 표시 방식 중에서 시계열 데이터 표현에 자주 이용되는 두 가지 유형을 알아보자. 특정한 시점을 기록하는 Timestamp와 두 시점 사이의 일정한 기간을 나타내는 Period가 있다.



(a) 타임스탬프(Timestamp)



(b) 피리어드(Period)

[그림 5-2] 판다스 시간 표시

■ '날짜 데이터' 분리1

시계열 데이터를 원소로 가지는 판다스 시리즈 객체의 dt.year, dt.month, dt.day 속성을 이용하여 연, 월, 일을 개별적으로 추출하자.

[예제 5-19]

```
4 import pandas as pd
5
6 # read_csv() 함수로 파일 읽어와서 df로 변환
7 df = pd.read_csv('stock-data.csv')
8
9 # 문자열인 날짜 데이터를 판다스 Timestamp로 변환
10 df['new_Date'] = pd.to_datetime(df['Date']) # df에 새로운 열로 추가
11 print(df.head())
12 print('\n')
13
14 # dt 속성을 이용하여 new_Date 열의 연-월-일 정보를 연, 월, 일로 구분
15 df['Year'] = df['new_Date'].dt.year
16 df['Month'] = df['new_Date'].dt.month
17 df['Day'] = df['new_Date'].dt.day
18 print(df.head())
```

〈실행 결과〉 코드 1~18라인을 부분 실행

	Date	Close	Start	High	Low	Volume	new_Date
0	2018-07-02	10100	10850	10900	10000	137977	2018-07-02
1	2018-06-29	10700	10550	10900	9990	170253	2018-06-29
2	2018-06-28	10400	10900	10950	10150	155769	2018-06-28
3	2018-06-27	10900	10800	11050	10500	133548	2018-06-27
4	2018-06-26	10800	10900	11000	10700	63039	2018-06-26

	Date	Close	Start	High	...	new_Date	Year	Month	Day
0	2018-07-02	10100	10850	10900	...	2018-07-02	2018	7	2
1	2018-06-29	10700	10550	10900	...	2018-06-29	2018	6	29
2	2018-06-28	10400	10900	10950	...	2018-06-28	2018	6	28
3	2018-06-27	10900	10800	11050	...	2018-06-27	2018	6	27
4	2018-06-26	10800	10900	11000	...	2018-06-26	2018	6	26

■ '날짜 데이터' 분리2

추출한 날짜 정보를 데이터 프레임의 행인덱스로 지정하여 데이터를 관리할 수도 있다.

[예제 5-19]

계속

```
df.set_index('Year', inplace=True)  
print(df.head())
```

	Date	Close	Start	High	Low	Volume	new_Date	Month	Day
Year									
2018	2018-07-02	10100	10850	10900	10000	137977	2018-07-02	7	2
2018	2018-06-29	10700	10550	10900	9990	170253	2018-06-29	6	29
2018	2018-06-28	10400	10900	10950	10150	155769	2018-06-28	6	28
2018	2018-06-27	10900	10800	11050	10500	133548	2018-06-27	6	27
2018	2018-06-26	10800	10900	11000	10700	63039	2018-06-26	6	26

■ 날짜 인덱스의 활용1

Timestamp로 구성된 열을 행 인덱스로 지정하면 DatetimeIndex라는 고유 속성으로 변환된다. 마찬가지로 Period로 구성된 열을 행 인덱스로 지정하면 PeriodIndex라는 속성을 갖는다. 이와 같은 날짜 인덱스를 활용하면 시계열 데이터에 대한 인덱싱과 슬라이싱이 편리하다.

앞에서 사용한 주식 데이터를 불러와 to_datetime() 메소드를 이용하여 'Date' 열 데이터를 Timestamp로 변환하고 'new_Date' 열에 담는다. 그리고 set_index() 메소드로 'new_Date' 열을 데이터프레임의 행 인덱스로 지정한다. 행 인덱스 값들의 자료형은 datetime64[ns]이다.

[예제 5-20]

```
4 import pandas as pd
5
6 # read_csv() 함수로 파일 읽어와 df로 변환
7 df = pd.read_csv('stock-data.csv')
8
9 # 문자열인 날짜 데이터를 판다스 Timestamp로 변환
10 df['new_Date'] = pd.to_datetime(df['Date']) # 새로운 열에 추가
11 df.set_index('new_Date', inplace=True) # 행 인덱스로 지정
12
13 print(df.head())
14 print('\n')
15 print(df.index)
```

〈실행 결과〉 코드 1~15라인을 부분 실행

	Date	Close	Start	High	Low	Volume
new_Date						
2018-07-02	2018-07-02	10100	10850	10900	10000	137977
2018-06-29	2018-06-29	10700	10550	10900	9990	170253
2018-06-28	2018-06-28	10400	10900	10950	10150	155769
2018-06-27	2018-06-27	10900	10800	11050	10500	133548
2018-06-26	2018-06-26	10800	10900	11000	10700	63039

DatetimeIndex(['2018-07-02', '2018-06-29', '2018-06-28', '2018-06-27',
'2018-06-26', '2018-06-25', '2018-06-22', '2018-06-21',
'2018-06-20', '2018-06-19', '2018-06-18', '2018-06-15',
'2018-06-14', '2018-06-12', '2018-06-11', '2018-06-08',
'2018-06-07', '2018-06-05', '2018-06-04', '2018-06-01'],
dtype='datetime64[ns]', name='new_Date', freq=None)

■ 날짜 인덱스의 활용2

날짜 인덱스를 활용하는 장점은 연, 월, 일 기준으로 내가 원하는 정보를 필터링하거나 슬라이싱 하는 것.
loc 인덱서를 통해 원하는 정보에 접근할 수 있다.

[예제 5-20]

```
# 날짜 인덱스를 이용하여 데이터 선택하기
df_y = df.loc['2018']    #특정 연도 자료
print(df_y.head())
print('\n')
df_ym = df.loc['2018-07']    #특정 연월 자료
print(df_ym)
print('\n')
df_ymd = df.loc['2018-07-02']    #특정 연월일 자료
print(df_ymd)
print('\n')
df_ym_cols = df.loc['2018-07', 'Start':'High']    # 연월 자료 + 열 슬라이싱
print(df_ym_cols)
print('\n')
df_ymd_range = df.loc['2018-06-20':'2018-06-25']    # 연월일 자료 슬라이싱
print(df_ymd_range)
print('\n')
df_ymd_list = df.loc[['2018-06-20','2018-06-25']]    # 연월일 자료 필터링
print(df_ymd_list)
print('\n')
```

	Date	Close	Start	High	Low	Volume
new_Date						
2018-07-02	2018-07-02	10100	10850	10900	10000	137977
2018-06-29	2018-06-29	10700	10550	10900	9990	170253
2018-06-28	2018-06-28	10400	10900	10950	10150	155769
2018-06-27	2018-06-27	10900	10800	11050	10500	133548
2018-06-26	2018-06-26	10800	10900	11000	10700	63039

	Date	Close	Start	High	Low	Volume
new_Date						
2018-07-02	2018-07-02	10100	10850	10900	10000	137977

	Date	Close	Start	High	Low	Volume
2018-07-02	2018-07-02	10100	10850	10900	10000	137977

Name: 2018-07-02 00:00:00, dtype: object

	Start	High
new_Date		
2018-07-02	10850	10900

	Date	Close	Start	High	Low	Volume
new_Date						
2018-06-25	2018-06-25	11150	11400	11450	11000	55519
2018-06-22	2018-06-22	11300	11250	11450	10750	134805
2018-06-21	2018-06-21	11200	11350	11750	11200	133002
2018-06-20	2018-06-20	11550	11200	11600	10900	308596

	Date	Close	Start	High	Low	Volume
new_Date						
2018-06-20	2018-06-20	11550	11200	11600	10900	308596
2018-06-25	2018-06-25	11150	11400	11450	11000	55519