In [1]:

```
#설정변경코드
#변수 명이 두번 이상 출력되어도 모두 콘솔에서 보여줄 것
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity="all"

# InteractiveShell.ast_node_interactivity : 'all' | 'last' | 'last_expr' | 'none' (기본값은 'last_ex
```

In [1]:

```
#모듈 import
import numpy as np
import pandas as pd
```

pandas 데이터처리 및 변환관련 함수

데이터 개수 세기

- 가장 간단한 분석은 개수를 세기 이다. count()함수 이용
  - NaN값은 세지 않는다.

In [2]:

```
# 시리즈에서 개수 세기
s = pd.Series(range(10))
s[3] = np.nan
s
```

Out[2]:

```
0    0.0
1    1.0
2    2.0
3    NaN
4    4.0
5    5.0
6    6.0
7    7.0
8    8.0
9    9.0
dtype: float64
```

In [3]:

```
s.count() # nan은 제외하고 count
```

Out[3]:

```
9
```

데이터 프레임에 count()함수 적용하기

- 각 열마다 데이터 개수를 세기때문에 누락된 부분을 찾을 때 유용

난수 발생시켜 dataframe 생성

- 난수 seed(값)라는 함수를 사용할 수 있음
- seed의 의미 : 난수 알고리즘에서 사용하는 기본 값으로
  - 시드값이 같으면 동일한 난수가 발생함
  - 난수 함수 사용시 매번 고정시켜야 함
- 계속 변경되는 난수를 받고 싶으면 함수등을 이용해서 시드값이 매번 변하게 작업해야 함. Time.tiem()

In [4]:

```
np.random.randint(5) # 정수 0~4사이에서 난수 발생
np.random.randint(5, size =4) # 난수 4개를 생성
```

Out[4]:

```
array([2, 2, 3, 2])
```

In [5]:

```
# 난수 발생시 항상 고정된 값을 발생시킬 수도 있음
np.random.seed(3)
np.random.randint(5, size =4) # 난수 4개를 생성
```

Out[5]:

```
array([2, 0, 1, 3])
```

In [6]:

```
# 위 셀에서 고정시켰지만 한번 사용했으므로  seed 는 리셋됨
np.random.randint(5, size =4)
```

Out[6]:

```
array([0, 0, 0, 3])
```

In [7]:

```
# 시드값이 매번 다르게 전달되도록 코드를 작성
import time
np.random.seed(int(time.time()))
np.random.randint(5, size=4)
```

Out[7]:

```
array([3, 3, 1, 3])
```

In [8]:

```
np.random.randint(5,size=(4,4))
```

Out[8]:

```
array([[1, 2, 4, 3],
       [1, 4, 3, 1],
       [1, 4, 4, 3],
       [3, 2, 0, 2]])
```

In [9]:

```
np.random.seed(3) # 시드값 고정되어서 동일한 값의 난수가 발생
df1=pd.DataFrame(np.random.randint(5,size=(4,4))) #기본 정수
df1.iloc[2,3] =np.nan # 2행3열의 value는 NaN으로 변경 - 3열 데이터는 실수형으로 변환됨
df1
# 컬럼(피처)은 동일한 타입의 데이터로만 구성할 수 있다
```

Out[9]:

	0	1	2	3
0	2	0	1	3.0
1	0	0	0	3.0
2	2	3	1	NaN
3	2	0	4	4.0

In [10]:

```
# 각 열에 대한 원소의 개수 반환
# 3열은 3개라면 결측치가 있음
df1.count()
```

Out[10]:

```
0    4
1    4
2    4
3    3
dtype: int64
```

count 함수 사용 예제

- 타이타닉 승객 데이터 사용
  - seaborn 패키지 내에 data로 존재
  - 데이터셋 읽어오기 : 패키지명.load_dataset("data명")

In [11]:

```
import seaborn as sns # 그래프 패키지

# 타이타닉 승객 데이터 로드
# titanic = sns.load_dataset('titanic')
titanic = pd.read_csv('./data/test.csv')
del titanic['Unnamed: 0']
titanic.head(10)
titanic.tail(2)
```

Out[11]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	dec
889	1	1	male	26.0	0	0	30.00	C	First	man	True	
890	0	3	male	32.0	0	0	7.75	Q	Third	man	True	Na

In [12]:

```
titanic.to_csv('./data/test.csv')
```

In [13]:

```
titanic.count()
# 대부분의 열이 891개의 원소가 있는데 반해
# age는 714, deck 203개의 원소가 있음
# embarked 889개의 원소
```

Out[13]:

```
survived       891
pclass         891
sex            891
age            714
sibsp          891
parch          891
fare           891
embarked       889
class          891
who            891
adult_male     891
deck           203
embark_town    889
alive          891
alone          891
dtype: int64
```

카테고리 값 세기

- 시리즈의 값이 정수,문자열 등 카테고리 값인 경우에
- 시리즈.value_counts()메서드를 사용해 각각의 값이 나온 횟수를 셀 수 있음
- 파라미터 normalize=True 를 사용하면 각 값 및 범주형 데이터의 비율을 계산
  - 시리즈.value_counts(normalize=True)

In [14]:

```
np.random.seed(1) #항상 같은 값이 나오게 설정
s2=pd.Series(np.random.randint(6,size=100))
s2.tail() # 시리즈의 뒷부분 데이터 5개를 추출
s2.head() # 시리즈의 앞부분 데이터 5개를 추출
s2.head(10) # 해당 개수 만큼의 데이터를 추출
len(s2)
```

Out[14]:

```
100
```

In [15]:

```
s2.value_counts() #0,1,2,3,4,5의 데이터가 몇번 나왔는지
s2.value_counts(normalize=True)
```

Out[15]:

```
1    0.22
0    0.18
4    0.17
5    0.16
3    0.14
2    0.13
dtype: float64
```

범주형 데이터에 value_counts() 함수 적용

- 범주형 데이터 : 관측 결과가 몇개의 범주 또는 항목의 형태로 나타나는 자료
  - ex. 성별(남,여), 선호도(좋다, 보통, 싫다), 혈액형(A,B,O,AB) 등

In [16]:

```
# titanic 데이터 alive열 : 생존여부가 yes/no로 표시되어 있음
titanic['alive'].dtype # 해당 열의 data type을 반환
titanic['alive'].value_counts()
titanic['alive'].value_counts(normalize=True)
```

Out[16]:

```
no     0.616162
yes    0.383838
Name: alive, dtype: float64
```

In [17]:

```
titanic['alive'].value_counts(normalize=True) *100
```

Out[17]:

```
no     61.616162
yes    38.383838
Name: alive, dtype: float64
```

In [18]:

```
# 타이타닉호에 승선한 승객의 남여 비율을 확인
# 성별 피처 : sex column
titanic['sex'].value_counts()
titanic['sex'].value_counts(normalize=True)*100
```

Out[18]:

```
male      64.758698
female    35.241302
Name: sex, dtype: float64
```

데이터프레임에 value_counts() 함수 사용

- 행을 하나의 value로 설정하고 동일한 행이 몇번 나타났는지 반환
- 행의 경우가 인덱스로 개수된 값이 value로 표시되는 Series 반환

In [19]:

```
titanic[['sex','alive']].head(2)
```

Out[19]:

	sex	alive
0	male	no
1	female	yes

In [20]:

```
titanic[['sex','alive']].value_counts()
```

Out[20]:

```
sex     alive
male    no       468
female  yes      233
male    yes      109
female  no        81
dtype: int64
```

In [21]:

```
type(titanic[['sex','alive']].value_counts())
```

Out[21]:

```
pandas.core.series.Series
```

In [22]:

```
pd.DataFrame(titanic[['sex','alive']].value_counts())
```

Out[22]:

		0
sex	**alive**	
male	**no**	468
female	**yes**	233
male	**yes**	109
female	**no**	81

정렬함수 -데이터 정렬 시 사용

- sort_index(ascending=True/False) : 인덱스를 기준으로 정렬
  - ascending 생략하면 오름차순 정렬
- sort_value(ascending=True/False) : 데이터 값을 기준으로 정렬

시리즈 정렬

In [23]:

```
#예제 시리즈
s2
```

Out[23]:

```
0     5
1     3
2     4
3     0
4     1
     ..
95    4
96    5
97    2
98    4
99    3
Length: 100, dtype: int32
```

In [24]:

```
# s2 시리즈에 대해 value_counts()
s2.value_counts() # 반환결과는 빈도값을 기준으로 내림차순 정렬
```

Out[24]:

```
1    22
0    18
4    17
5    16
3    14
2    13
dtype: int64
```

In [25]:

```
# # s2 시리즈에 대해 value_counts()
s2.value_counts().sort_index() # 인덱스 기준 정렬
```

Out[25]:

```
0    18
1    22
2    13
3    14
4    17
5    16
dtype: int64
```

In [26]:

```
s2.value_counts().sort_index(ascending=False) # 인덱스 기준 정렬
```

Out[26]:

```
5    16
4    17
3    14
2    13
1    22
0    18
dtype: int64
```

In [27]:

```
s2.value_counts().sort_values() #빈도값을 기준으로 오름차순 정렬
```

Out[27]:

```
2    13
3    14
5    16
4    17
0    18
1    22
dtype: int64
```

In [28]:

```
s2.value_counts().sort_values(ascending=False)
```

Out[28]:

```
1    22
0    18
4    17
5    16
3    14
2    13
dtype: int64
```

데이터 프레임 정렬

- df.sort_values() : 특정열 값 기준 정렬
  - 데이터프레임은 2차원 배열과 동일하기 때문에
    - 정렬시 기준열을 줘야 한다. by 인수 사용 : 생략 불가
    - by = 기준열, by=[기준열1,기준열2]
  - 오름차순/내림차순 : ascending = True/False (생략하면 오름차순)
- df.sort_index() : DF의 INDEX 기준 정렬
  - 오름차순/내림차순 : ascending = True/False (생략하면 오름차순)

In [29]:

```
df1
```

Out[29]:

	0	1	2	3
0	2	0	1	3.0
1	0	0	0	3.0
2	2	3	1	NaN
3	2	0	4	4.0

In [30]:

```
# df1.sort_values()  #TypeError  by
```

In [31]:

```
df1.sort_values(by=0)
df1.sort_values(by=0,ascending=False)
```

Out[31]:

	0	1	2	3
0	2	0	1	3.0
2	2	3	1	NaN
3	2	0	4	4.0
1	0	0	0	3.0

In [32]:

```
df1.sort_values(by=[0,2],ascending=False)
```

Out[32]:

	0	1	2	3
3	2	0	4	4.0
0	2	0	1	3.0
2	2	3	1	NaN
1	0	0	0	3.0

In [33]:

```
# 예제 df
df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
                   'num_wings': [2, 0, 0, 0]},
                  index=['falcon', 'dog', 'cat', 'ant'])
df
```

Out[33]:

	num_legs	num_wings
falcon	2	2
dog	4	0
cat	4	0
ant	6	0

- 데이터프레임의 index를 기준으로 정렬

In [34]:

```
df.sort_index() # 인덱스 기준 오름차순 정렬
df.sort_index(ascending=False) # 인덱스 기준 내림차순 정렬
```

Out[34]:

	num_legs	num_wings
falcon	2	2
dog	4	0
cat	4	0
ant	6	0

행/열 합계

- df.sum() 함수 사용
- 행과 열의 합계를 구할때는 sum(axis=0/1) - axis는 0이 기본

- 각 열의 합계를 구할때는 sum(axis=0)
- 각 행의 합계를 구할때는 sum(axis=1)

In [35]:

```
# 예제 DF 생성
#4행 8열의 데이터프레임 작성, 난수를 발생시키고
#0-9범위에서 매번 같은 난수 발생되어 반환되도록 설정
np.random.seed(1)
df2=pd.DataFrame(np.random.randint(10,size=(4,8)))
df2
```

Out[35]:

	0	1	2	3	4	5	6	7
0	5	8	9	5	0	0	1	7
1	6	9	2	4	5	2	4	2
2	4	7	7	9	1	7	0	6
3	9	9	7	6	9	1	0	1

In [36]:

```
# df2의 각 행의 합계를 구함
# 4행이므로 결과는 4개의 원소값을 값는 시리즈 반환
df2.sum(axis=1)
```

Out[36]:

```
0    35
1    34
2    41
3    42
dtype: int64
```

In [37]:

```
# df2의 각 열의 합계를 구함
# 8열이므로 결과는 8개의 원소값을 값는 시리즈 반환
df2.sum(axis=0)
```

Out[37]:

```
0    24
1    33
2    25
3    24
4    15
5    10
6     5
7    16
dtype: int64
```

In [38]:

```
df2.sum() # axis 인수 생략하면 기본값이 0 이므로 각 열의 합계를 구함
```

Out[38]:

```
0    24
1    33
2    25
3    24
4    15
5    10
6     5
7    16
dtype: int64
```

In [39]:

```
# 각 행의 합계를 표시하는 합계 열을 추가하시오
# 여러번 실행하면  total값까지 누적되어서 잘못된 결과가 발생할 수 있다!!!!
# df2['total']=df2.sum(axis=1)
df2['total']=df2[[0,1,2,3,4,5,6,7]].sum(axis=1)
df2
```

Out[39]:

	0	1	2	3	4	5	6	7	total
0	5	8	9	5	0	0	1	7	35
1	6	9	2	4	5	2	4	2	34
2	4	7	7	9	1	7	0	6	41
3	9	9	7	6	9	1	0	1	42

df의 기본 함수

- mean(axis=0/1)
- min(axis=0/1)
- max(axis=0/1)

In [40]:

```
# 예제 DF 확인
df2
```

Out[40]:

	0	1	2	3	4	5	6	7	total
0	5	8	9	5	0	0	1	7	35
1	6	9	2	4	5	2	4	2	34
2	4	7	7	9	1	7	0	6	41
3	9	9	7	6	9	1	0	1	42

In [41]:

```
# 각 열의 평균
df2.mean(axis=0)
# 각 열의 평균
df2.mean(axis=1)
```

Out[41]:

```
0    7.777778
1    7.555556
2    9.111111
3    9.333333
dtype: float64
```

In [42]:

```
# 각 열의 최소값
df2.min(axis=0)
# 각 행의 최소값
df2.min(axis=1)
```

Out[42]:

```
0    0
1    2
2    0
3    0
dtype: int64
```

In [43]:

```
# 각 열의 최대값
df2.max(axis=0)
# 각 행의 최대값
df2.max(axis=1)
```

Out[43]:

```
0    35
1    34
2    41
3    42
dtype: int64
```

In [44]:

```
# 각 열의 최대값을 구해서 max_data 라는 행을 추가
# 새로운 행 추가(loc 인덱서 사용)
# df.loc['추가되는행이름'] = data
#  동일한 코드를 반복실행하면 max_data까지에서 최대값을 구하므로 의미가 달라질 수 있다
# df2.loc['max_data']=df2.max(axis=0)
df2.loc['max_data']=df2[[0,1,2,3,4,5,6,7]].max(axis=0)
df2
```

Out[44]:

	0	1	2	3	4	5	6	7	total
0	5.0	8.0	9.0	5.0	0.0	0.0	1.0	7.0	35.0
1	6.0	9.0	2.0	4.0	5.0	2.0	4.0	2.0	34.0
2	4.0	7.0	7.0	9.0	1.0	7.0	0.0	6.0	41.0
3	9.0	9.0	7.0	6.0	9.0	1.0	0.0	1.0	42.0
max_data	9.0	9.0	9.0	9.0	9.0	7.0	4.0	7.0	NaN

행/열 삭제 - drop() 사용 예제

- df.drop('행이름',0) : 행삭제
  - 행삭제 후 df로 결과를 반환
- df.drop('행이름',1) : 열 삭제
  - 행삭제 후 df로 결과를 반환

- 원본에 반영되지 않으므로 원본수정하려면 저장 해야 함
  - inplace=True

In [45]:

```
df2
```

Out[45]:

	0	1	2	3	4	5	6	7	total
0	5.0	8.0	9.0	5.0	0.0	0.0	1.0	7.0	35.0
1	6.0	9.0	2.0	4.0	5.0	2.0	4.0	2.0	34.0
2	4.0	7.0	7.0	9.0	1.0	7.0	0.0	6.0	41.0
3	9.0	9.0	7.0	6.0	9.0	1.0	0.0	1.0	42.0
max_data	9.0	9.0	9.0	9.0	9.0	7.0	4.0	7.0	NaN

In [46]:

```
# total 열 삭제
df2.drop('total',1,inplace=True)
df2
df2.drop('max_data',0)
df2
```

```
C:₩Users₩bbobbee₩AppData₩Local₩Temp₩ipykernel_11836₩2354260271.py:2: FutureWarning:
In a future version of pandas all arguments of DataFrame.drop except for the argumen
t 'labels' will be keyword-only.
  df2.drop('total',1,inplace=True)
C:₩Users₩bbobbee₩AppData₩Local₩Temp₩ipykernel_11836₩2354260271.py:4: FutureWarning:
In a future version of pandas all arguments of DataFrame.drop except for the argumen
t 'labels' will be keyword-only.
  df2.drop('max_data',0)
```

Out[46]:

	0	1	2	3	4	5	6	7
0	5.0	8.0	9.0	5.0	0.0	0.0	1.0	7.0
1	6.0	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	4.0	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	9.0	7.0	4.0	7.0

NaN 값 처리 함수

- df.dropna(axis=0/1)
  - NaN값이 있는 열 또는 행을 삭제
  - 원본 반영되지 않음
- df.fillna(0)
  - NaN값을 정해진 숫자로 채움
  - 원본 반영 되지 않음

In [47]:

```
df2
```

Out[47]:

	0	1	2	3	4	5	6	7
0	5.0	8.0	9.0	5.0	0.0	0.0	1.0	7.0
1	6.0	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	4.0	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [48]:

```
# 결측치값 적용 - NaN값 생성 : np.nan  속성
df2.iloc[0,0] = np.nan
df2
```

Out[48]:

	0	1	2	3	4	5	6	7
0	NaN	8.0	9.0	5.0	0.0	0.0	1.0	7.0
1	6.0	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	4.0	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [49]:

```
# NaN이 포함된 행을 삭제
df2.dropna(axis=0)
df2 # 원본반영되지 않음
```

Out[49]:

	0	1	2	3	4	5	6	7
0	NaN	8.0	9.0	5.0	0.0	0.0	1.0	7.0
1	6.0	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	4.0	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [50]:

```
# NaN이 포함된 열을 삭제
df2.dropna(axis=1)
df2 # 원본반영되지 않음
```

Out[50]:

	0	1	2	3	4	5	6	7
0	NaN	8.0	9.0	5.0	0.0	0.0	1.0	7.0
1	6.0	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	4.0	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [51]:

```
# NaN이 포함된 열을 삭제
df2.dropna(axis=1,inplace=True)
```

In [52]:

```
df2
```

Out[52]:

	1	2	3	4	5	6	7
0	8.0	9.0	5.0	0.0	0.0	1.0	7.0
1	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [53]:

```
df2.iloc[0,0] = np.nan
```

In [54]:

```
df2
# NaN값을 0으로 대체
df2.fillna(0)
df2
df2.fillna(1)
```

Out[54]:

	1	2	3	4	5	6	7
0	1.0	9.0	5.0	0.0	0.0	1.0	7.0
1	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [55]:

```
df2.fillna(1,inplace=True) # 원본반영 파라미터 inplace=True
```

In [56]:

```
df2
```

Out[56]:

	1	2	3	4	5	6	7
0	1.0	9.0	5.0	0.0	0.0	1.0	7.0
1	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	7.0	4.0	7.0

데이터프레임의 형변환

- df.astype(자료형)
- 전체 데이터에 대해서 형변환을 진행

In [57]:

```
df2.iloc[0,0]=np.nan
df2
```

Out[57]:

	1	2	3	4	5	6	7
0	NaN	9.0	5.0	0.0	0.0	1.0	7.0
1	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [58]:

```
# 결측치를 0으로 채우면서 df의 데이터를 정수형으로 형변환
df2.fillna(0).astype(int)
df2.fillna(0).astype(float)
```

Out[58]:

	1	2	3	4	5	6	7
0	0.0	9.0	5.0	0.0	0.0	1.0	7.0
1	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [59]:

```
df2.fillna(0)[1].astype(int)
```

Out[59]:

```
0           0
1           9
2           7
3           9
max_data    9
Name: 1, dtype: int32
```

In [60]:

```
test_df = df2.fillna(0).astype(float)
test_df
```

Out[60]:

	1	2	3	4	5	6	7
0	0.0	9.0	5.0	0.0	0.0	1.0	7.0
1	9.0	2.0	4.0	5.0	2.0	4.0	2.0
2	7.0	7.0	9.0	1.0	7.0	0.0	6.0
3	9.0	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9.0	9.0	9.0	9.0	7.0	4.0	7.0

In [61]:

```
test_df[1]=test_df[1].astype(int)
test_df
```

Out[61]:

	1	2	3	4	5	6	7
0	0	9.0	5.0	0.0	0.0	1.0	7.0
1	9	2.0	4.0	5.0	2.0	4.0	2.0
2	7	7.0	9.0	1.0	7.0	0.0	6.0
3	9	7.0	6.0	9.0	1.0	0.0	1.0
max_data	9	9.0	9.0	9.0	7.0	4.0	7.0

In [62]:

```
# int(test_df)
# TypeError: int() argument must be a string, a bytes-like object or a number, not 'DataFrame'
```

열 또는 행에 동일한 연산 반복 적용할 때 : apply() 함수

- apply() 함수는 DataFrame의 행이나 열에 복잡한 연산을 vectorizing할 수 있게 해주는 함수로 매우 많이 활용되는 함수임
- 동일한 연산(함수화 되어있어야함)을 모든열에 혹은 모든 행에 반복 적용하고자 할때 사용
- apply(반복적용할 함수, axis=0/1)
  - 0 : 열마다 반복
  - 1 : 행마다 반복
  - 생략시 기본값 : 0

In [63]:

```
#예제 df 생성
df3 = pd.DataFrame({
    'a':[1,3,4,3,4],
    'b':[2,3,1,4,5],
    'c':[1,5,2,4,4]
})
df3
```

Out[63]:

	a	b	c
0	1	2	1
1	3	3	5
2	4	1	2
3	3	4	4
4	4	5	4

In [64]:

```
# np.sum() 함수는 전달된 첫번째 인수 데이터들의 합산 결과를 반환하는 함수
np.sum([1,2,3])
```

Out[64]:

```
6
```

In [65]:

```
# df3의 각 열에 대해서 np.sum() 함수를 반복 적용
np.sum(df3['a'])
np.sum(df3['b'])
np.sum(df3['c'])

for i in ['a','b','c'] :
    print(np.sum(df3[i]))

df3.apply(np.sum,0) # df3의 각 열에 대해서 np.sum() 적용한 결과를 반환
```

```
15
15
16
```

Out[65]:

```
a     15
b     15
c     16
dtype: int64
```

In [66]:

```
# seaborn 패키지의 titanic 데이터셋을 load 하시오
import seaborn as sns

titanic = sns.load_dataset('titanic')
titanic.head() # 처음 5행 출력
titanic.tail() # 마지막 5행 출력
#itanic.head(2) # 처음 2행 출력
```

Out[66]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_mal
886	0	2	male	27.0	0	0	13.00	S	Second	man	Tru
887	1	1	female	19.0	0	0	30.00	S	First	woman	Fals
888	0	3	female	NaN	1	2	23.45	S	Third	woman	Fals
889	1	1	male	26.0	0	0	30.00	C	First	man	Tru
890	0	3	male	32.0	0	0	7.75	Q	Third	man	Tru

In [67]:

```
#  titanic df의 alive, sex, class 열에 대해서 value_counts() 함수를 적용하여 결과를
# 확인하시오.apply()  함수 사용할 것
titanic[['alive','sex','class']].apply(pd.value_counts,0)
```

Out[67]:

	alive	sex	class
First	NaN	NaN	216.0
Second	NaN	NaN	184.0
Third	NaN	NaN	491.0
female	NaN	314.0	NaN
male	NaN	577.0	NaN
no	549.0	NaN	NaN
yes	342.0	NaN	NaN

- 데이터프레임의 기본 집계함수(sum, min, max, mean 등)들은 행/열 단위 벡터화 연산을 수행함
  - apply() 함수를 사용할 필요가 없음

- 일반적으로 apply() 함수 사용은 복잡한 연산을 해결하기 위한 lambda 함수나 사용자 정의 함수를 각 열 또는 행에 일괄 적용시키기 위해 사용

사용자 정의 함수를 apply()에 사용하는 예제

In [68]:

```
# 사용자 정의 함수 생성 :
# 사용자로부터 시리즈를 전달받아(매개변수) 해당시리즈의 최대값과 최소값 차이를
# 구해 반환하는 함수
# diff 를 정의

def diff(x) : # x는 시리즈
    return x.max() - x.min()
```

In [69]:

```
df3
```

Out[69]:

	a	b	c
0	1	2	1
1	3	3	5
2	4	1	2
3	3	4	4
4	4	5	4

In [70]:

```
# 함수 호출
diff(df3['a'])
diff(df3['b'])
diff(df3['c'])
```

Out[70]:

```
4
```

In [71]:

```
# apply 함수 적용
df3.apply(diff,0)
```

Out[71]:

```
a    3
b    4
c    4
dtype: int64
```

In [72]:

```
pd.Series(3)
pd.Series(3).max()
```

Out[72]:

```
3
```

1회성함수 lambda 함수를 apply()에 사용하는 예제

In [73]:

```
(lambda x : x.max()-x.min())(df3['a'])
```

Out[73]:

3

In [74]:

```
df3.apply(lambda x : x.max()-x.min(),0)
```

Out[74]:

```
a    3
b    4
c    4
dtype: int64
```

df3 각 열에 대해 최대값과 최소값의 차이를 구하시오

- apply 함수 사용하지 말것

In [75]:

```
df3.max(axis=0)- df3.min(axis=0)
```

Out[75]:

```
a    3
b    4
c    4
dtype: int64
```

In [76]:

```
# apply()로 df의 행이나 열에 적용되어지는 함수가 행/열 자체가 아닌 요소에 적용되어지는
# 함수라면 각 요소마다 함수를 적용시켜준다.

# np.square 함수는 행/열 별 각 원소에 대해 제곱승 연산을 진행하므로
# 행적용이나 열적용 모두 동일한 결과가 나타남
np.square([1,2,3])
df3
df3.apply(np.square,0)
df3.apply(np.square,1)
```

Out[76]:

	a	b	c
0	1	4	1
1	9	9	25
2	16	1	4
3	9	16	16
4	16	25	16

데이터값을 카테고리 값으로 변환

- 값의 크기를 기준으로하여 카테고리 값으로 변환하고 싶을때
  - cut(data,bins,labels)
    - data : 구간 나눌 실제 값,
    - bins : 구간 경계값
    - labels: 카테고리값
  - qcut(data,구간수,labels)

In [77]:

```
#구간을 나눌 실제 값 : 관측 데이터
ages=[0,0.5,4,6,4,5,2,10,21,23,37,15,38,31,61,20,41,31,100]
```

In [78]:

```
# data : 구간 나눌 실제 값, bins : 구간 경계값, label: 카테고리값
data = ages

# 구간 경계값
# 구간 최소값 < 구간 <= 구간 최대값
bins = [0,4,18,25,35,60,100]
# 0~4 구간 : 영유아 0살 < 영유아 <=4

# 각 구간의 이름 : labels - 카테고리명
# 순서는 구간(bins)의 순서와 동일해야 함
labels = ['영유아','미성년자','청년','중년','장년','노년']
```

In [79]:

```
len(data)
cats = pd.cut(data,bins,labels=labels)
cats
```

Out[79]:

```
[NaN, '영유아', '영유아', '미성년자', '영유아', ..., '노년', '청년', '장년', '중년',
'노년']
Length: 19
Categories (6, object): ['영유아' < '미성년자' < '청년' < '중년' < '장년' < '노년']
```

In [80]:

```
type(cats) #categorical.Categorical
```

Out[80]:

```
pandas.core.arrays.categorical.Categorical
```

In [81]:

```
cat_list = list(cats)
cat_list
```

Out[81]:

```
[nan,
 '영유아',
 '영유아',
 '미성년자',
 '영유아',
 '미성년자',
 '영유아',
 '미성년자',
 '청년',
 '청년',
 '장년',
 '미성년자',
 '장년',
 '중년',
 '노년',
 '청년',
 '장년',
 '중년',
 '노년']
```

In [82]:

```
test = pd.DataFrame({'나이':ages,'연령대':cat_list})
test
test['연령대'].value_counts()
```

Out[82]:

```
영유아     4
미성년자    4
청년      3
장년      3
중년      2
노년      2
Name: 연령대, dtype: int64
```

Categorical 클래스 객체

- 카테고리명 속성 : Categorical.categories
  - 카테고리명 저장
- 코드 속성 : Categorical.codes
  - 인코딩한 카테고리 값을 정수로 갖는다.

In [83]:

```
type(cats)
```

Out[83]:

```
pandas.core.arrays.categorical.Categorical
```

In [84]:

```
cats.categories # cats 변수내의 범주 index를 저장
```

Out[84]:

```
Index(['영유아', '미성년자', '청년', '중년', '장년', '노년'], dtype='object')
```

In [85]:

```
cats.codes
# 인덱스 위치 정수값이 -1이면 NaN으로 결정되어 결측치가 됨
```

Out[85]:

```
array([-1,  0,  0,  1,  0,  1,  0,  1,  2,  2,  4,  1,  4,  3,  5,  2,  4,
        3,  5], dtype=int8)
```

구간 경계선을 지정하지 않고 데이터 개수가 같도록 지정한 수의 구간으로 분할하기 : qcut()

- 형식 : pd.qcut(data,구간수,labels=[d1,d2....])

  - 예)1000개의 데이터를 4구간으로 나누려고 한다면
    - qcut 명령어를 사용 한 구간마다 250개씩 나누게 된다.
    - 예외)같은 숫자인 경우에는 같은 구간으로 처리한다.

In [86]:

```
# 랜덤정수 20개를 생성하고 생성된 정수를 4개의 구간으로 나누시오.

# 각 구간의 label은 Q1,Q2,Q3,Q4 로 설정하시오.

#랜덤정수 생성 : 범위 0-19, size =20
#seed 설정해서 재 실행해도 랜덤정수가 변하지 않도록 생성
np.random.seed(2)
data = np.random.randint(20,size=20)
data

qcat = pd.qcut(data,4,labels=['Q1','Q2','Q3','Q4'])
qcat
```

Out[86]:

```
['Q2', 'Q4', 'Q4', 'Q2', 'Q3', ..., 'Q1', 'Q1', 'Q1', 'Q3', 'Q3']
Length: 20
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
```

In [87]:

```
np.sort(data)
```

Out[87]:

```
array([ 2,  3,  4,  5,  6,  7,  7,  8,  8,  8, 10, 11, 11, 11, 11, 13, 15,
       15, 17, 18])
```

In [88]:

```
pd.value_counts(qcat)
```

Out[88]:

```
Q1    5
Q2    5
Q3    5
Q4    5
dtype: int64
```

In [89]:

```
df0 = pd.DataFrame(data, columns=['관측수'])
# df0
df0['범주']=qcat
df0
```

Out[89]:

	관측수	범주
0	8	Q2
1	15	Q4
2	13	Q4
3	8	Q2
4	11	Q3
5	18	Q4
6	11	Q3
7	8	Q2
8	7	Q2
9	2	Q1
10	17	Q4
11	11	Q3
12	15	Q4
13	5	Q1
14	7	Q2
15	3	Q1
16	6	Q1
17	4	Q1
18	10	Q3
19	11	Q3

인덱스 설정 함수

데이터 프레임 인덱스 설정을 위해 set_index(), reset_index()

- set_index() : 기존 행 인덱스를 제거하고 데이터 열 중 하나를 인덱스로 설정해주는 함수

- reset_index() : 기존 행인덱스를 제거하고 기본인덱스로 변경
- 기본인덱스 : 0부터 1씩 증가하는 정수 인덱스
  - 따로 설정하지 않으면 기존 인덱스는 데이터열로 추가 됨

In [90]:

```
#예제 데이터프레임 생성
df3 = pd.DataFrame({
    'a':[1,3,4,3,4],
    'b':[2,3,1,4,5],
    'c':[1,5,2,4,4]
})
df3
```

Out[90]:

	a	b	c
0	1	2	1
1	3	3	5
2	4	1	2
3	3	4	4
4	4	5	4

In [91]:

```
# df3의 'a'열을 인덱스로 설정
df3.set_index('a') # 원본반영되지 않음
df3
df3.set_index('a',inplace=True) # 원본에 저장
```

In [92]:

```
df3
df3.loc[3]
```

Out[92]:

	b	c
a		
3	3	5
3	4	4

In [93]:

```
# 기본 인덱스(제로베이스)로 인덱스 수정
df3.reset_index() # 인덱스 값은 df의 열 데이터로 추가
df3 #원본반영안됨
```

Out[93]:

	b	c
a		
1	2	1
3	3	5
4	1	2
3	4	4
4	5	4

In [94]:

```
# 인덱스값이 관측값이 아니거나 불필요한 경우는 df의 열데이터로 추가되지 않게 해야함
# drop=True (기존 인덱스 삭제)
df3.reset_index(drop=True)
```

Out[94]:

	b	c
0	2	1
1	3	5
2	1	2
3	4	4
4	5	4

In [95]:

```
df3.reset_index(drop=True,inplace=True)
df3
```

Out[95]:

	b	c
0	2	1
1	3	5
2	1	2
3	4	4
4	5	4

index 이름 변경

- rename(index={현재 index:바꿀 index}) 사용,행인덱스
- rename(columns={현재 index:바꿀 index}) 사용,열인덱스

In [96]:

```
# 행인덱스 값 변경
# df3의 인덱스 0 -> 1반으로 변경
# df3.rename(index={0:'1반'}) # 원본반영 안됨
# df3
df3.rename(index={0:'1반',1:'2반'})
```

Out[96]:

	b	c
1반	2	1
2반	3	5
2	1	2
3	4	4
4	5	4

In [97]:

```
# 컬럼명 변경
df3.rename(columns={'b':'학생'},inplace=True)
```

In [98]:

```
df3
```

Out[98]:

	학생	c
0	2	1
1	3	5
2	1	2
3	4	4
4	5	4

리스트 내포 연산

리스트 내포 for문의 일반 문법

- [표현식(연산식) for 항목 in 반복가능객체 if 조건문]
- if 조건문은 생략 가능하다.
- 반복가능객체 : 리스트, 튜플,딕셔너리,range()등

In [99]:

```
### 리스트 내포 for 문
a=[1,2,3,4]
# 위 a 리스트의 각 원소에 2배한 원소값을 만들고 리스트로 저장하시오. 반복문을 사용할 것
# result =[2,4,6,8]
```

In [100]:

```
a=[1,2,3,4]
result = [] # 빈리스트 생성

for num in a :
    result.append(num*2)

result
```

Out[100]:

```
[2, 4, 6, 8]
```

In [101]:

```
result = [num * 2 for num in a]
result
```

Out[101]:

```
[2, 4, 6, 8]
```

In [108]:

```
# ['id_1','id_2','id_3','id_4'] 와 같은 리스트 요소를
# 내포 for문을 이용해서 생성후 리스트로 추가하시오.
```

In [103]:

```
# 반복문 이용
test = []
for i in range(1,5) :
    test.append('id_'+str(i))
test
```

Out[103]:

```
['id_1', 'id_2', 'id_3', 'id_4']
```

In [104]:

```
test = ['id_'+str(i) for i in range(1,5) ]
test
```

Out[104]:

```
['id_1', 'id_2', 'id_3', 'id_4']
```

In [ ]:

```
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```