판다스 설명(pandas)

- series, DataFrame등의 자료구조를 활용한 데이터분석 기능을 제공해주는 라이브러리
- 라이브러리 구성
  - 여러종류의 클래스와 다양한 함수로 구성
  - 시리즈와 데이터 프레임의 자료 구조 제공
  - 시리즈(1차원 배열) 데이터프레임(2차원 행열구조)

판다스의 목적

- 서로 다른 유형의 데이터를 공통된 포맷으로 정리하는 것
- 행과 열로 이루어진 2차원 데이터프레임을 처리 할 수 있는 함수제공 목적
- 실무 사용 형태 : 데이터 프레임

Series

- pandas의 기본 객체 중 하나
- numpy의 ndarray를 기반으로 인덱싱을 기능을 추가하여 1차원 배열을 나타냄
- index를 지정하지 않을 시, 기본적으로 ndarray와 같이 0-based 인덱스 생성, 지정할 경우 명시적으로 지정된 index를 사용
- **같은 타입**의 **0개 이상**의 데이터를 가질 수 있음

1. 자료구조: 시리즈
   - 데이터가 순차적으로 나열된 1차원 배열 형태
   - 인덱스(index)와 데이터 값(value)이 일대일로 대을
   - 딕셔너리와 비슷한 구조 : {key(index):value}
2. 시리즈의 인덱스
   - 데이터 값의 위치를 나타내는 이름표 역할
3. 시리즈 생성 : 판다스 내장함수인 Series()이용
   - **리스트**로 시리즈 만들기
   - 딕셔너리로 시리즈 만들기
   - 튜플로 시리즈 만들기

판다스 모듈 import

- 대부분의 코드에서 pandas 모듈은 pd 라는 별칭을 사용함
- 데이터분석에서 pandas와 numpy 두 패키지는 기본 패키지로 본다
- numpy는 np라는 별칭을 사용

In [1]:

```
import pandas as pd
import numpy as np
```

Series 생성하기

- data로만 생성하기

- index는 기본적으로 0부터 자동적으로 생성

In [2]:

```
# pd.Series(집합적 자료형)
# pd.Series(리스트)
s = pd.Series([1,2,3])
s
# 위 코드는 시리즈 생성 시 인덱스를 명시하지 않았음.  0 base 인덱스 생성
```

Out[2]:

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

In [3]:

```
# pd.Series(튜플)
s = pd.Series((1.0,2.0,3.0))
s
```

Out[3]:

```
0    1.0
1    2.0
2    3.0
dtype: float64
```

In [4]:

```
# pd.Series(1,2,3) # 시리즈 생성시 반드시 집합적 자료형을 이용해야 함
```

In [5]:

```
s2 = pd.Series(['a','a','c']) #dtype: object
s2
```

Out[5]:

```
0    a
1    a
2    c
dtype: object
```

In [6]:

```
# 리스트내에 서로 다른 type의 data가 있으면 형변환 일어남- 문자열로 변환됨
s_1 = pd.Series(['a',1,3.0]) #dtype: object
s_1
```

Out[6]:

```
0      a
1      1
2    3.0
dtype: object
```

- 범위를 시리즈의 value 생성하는 데 사용하기 - range/np.arange 함수 사용

In [7]:

```
s = pd.Series(range(10,14)) # index 인수는 생략됨
s
```

Out[7]:

```
0    10
1    11
2    12
3    13
dtype: int64
```

In [8]:

```
range(10,14)
```

Out[8]:

```
range(10, 14)
```

In [9]:

```
np.arange(200)
```

Out[9]:

```
array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
        26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,
        39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
        52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,
        65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,
        78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,
        91,  92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103,
       104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116,
       117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129,
       130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142,
       143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155,
       156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168,
       169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181,
       182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194,
       195, 196, 197, 198, 199])
```

In [10]:

```
s3 = pd.Series(np.arange(200))
s3
```

Out[10]:

```
0        0
1        1
2        2
3        3
4        4
      ...
195    195
196    196
197    197
198    198
199    199
Length: 200, dtype: int32
```

- 결측값을 포함해서 시리즈 만들기
  - 결측값 NaN - numpy 라는 모듈에서 생성할 수 있음
  - 결측값 생성 위해서는 numpy 모듈 import

In [11]:

```
# NaN은 np.nan 속성을 이용해서 생성
s=pd.Series([1,2,3,np.nan,6,8])
s
# dtype: float64
# 판다스가 처리하는 자료구조인 시리즈와 데이터프레임에서 결측치가 있는 경우는 datatype이 float으로 변
```

Out[11]:

```
0    1.0
1    2.0
2    3.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

- 인덱스 명시해서 시리즈 만들기
  - 숫자 인덱스 지정
  - s = pd.Series([값1,값2,값3],index=[1,2,3])

In [12]:

```
s=pd.Series([10,20,30],index=[1,2,3])
s
```

Out[12]:

```
1    10
2    20
3    30
dtype: int64
```

- 문자 인덱스 지정

In [13]:

```
s= pd.Series([95,100,88], index = ['홍길동','이몽룡','성춘향'])
s
```

Out[13]:

```
홍길동     95
이몽룡    100
성춘향     88
dtype: int64
```

- 인덱스 활용 : 시리즈의 index
  - 시리즈의 index는 index 속성으로 접근

In [14]:

```
s0=pd.Series([10,20,30],index=[1,2,3])
s0
```

Out[14]:

```
1    10
2    20
3    30
dtype: int64
```

In [15]:

```
s0.index #Int64Index([1, 2, 3], dtype='int64')
```

Out[15]:

```
Int64Index([1, 2, 3], dtype='int64')
```

In [16]:

```
s00 = pd.Series([1,2,3]) # index를 명시하지 않음
s00.index # 범위 인덱스가 생성
```

Out[16]:

```
RangeIndex(start=0, stop=3, step=1)
```

In [17]:

```
s= pd.Series([9904312,3448737,289045,2466052],
             index=["서울","부산","인천","대구"])
s.index #Index(['서울', '부산', '인천', '대구'], dtype='object')
```

Out[17]:

```
Index(['서울', '부산', '인천', '대구'], dtype='object')
```

- 시리즈.index.name 속성
  - 시리즈의 인덱스에 이름을 붙일 수 있음

In [18]:

```
s.index.name = '광역시'
s
```

Out[18]:

```
광역시
서울     9904312
부산     3448737
인천      289045
대구     2466052
dtype: int64
```

In [19]:

```
s.index
```

Out[19]:

```
Index(['서울', '부산', '인천', '대구'], dtype='object', name='광역시')
```

- 시리즈의 값: numpy 자료구조 - 1차원 배열
  - values 속성으로 접근
  - 시리즈.values

In [20]:

```
s.values
# 시리즈의 값의  전체 형태는  array(numpy의 자료구조) 형태
```

Out[20]:

```
array([9904312, 3448737,  289045, 2466052], dtype=int64)
```

- 시리즈.name 속성
  - 시리즈 데이터에 이름을 붙일 수 있다.
  - name 속성은 값의 의미 전달에 사용

In [21]:

```
s.name = '인구'
s
```

Out[21]:

```
광역시
서울     9904312
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

인덱싱:

- 데이터에서 특정한 데이터를 골라내는 것

시리즈의 인덱싱 종류

1. 정수형 위치 인덱스(integer position)
2. 인덱스 이름(index name) 또는 인덱스 라벨(index label)

    - 인덱스 별도 지정하지 않으면 0부터 시작하는 정수형 인덱스가 지정됨

원소접근

- 정수형 인덱스 : 숫자 s[0]
- 문자형 인덱스 : 문자 s['인천']

In [22]:

```
print(s.index) # 문자열형 인덱스
s['인천'] # 문자형 인덱스로 접근
s[2] # 위치 인덱스 사용 가능
```

```
Index(['서울', '부산', '인천', '대구'], dtype='object', name='광역시')
```

Out[22]:

289045

In [23]:

```
# 정수형 인덱스인 경우
s03 = pd.Series([1,2,3], index=[1,2,3])
s03
s03[1] # 명시적 인덱스(정수인덱스임) 사용
# s03[0] # 위치인덱스 접근 - KeyError
# 정수인덱스인 경우 위치인덱스는 사용 불가
```

Out[23]:

1

In [24]:

```
# 문자형 인덱스(부산 데이터 추출)
s['부산']
```

Out[24]:

3448737

In [25]:

```
# 두개 이상의 인덱싱 코드를 나열하면 - 튜플형태로 반환
s[3], s['대구']
```

Out[25]:

(2466052, 2466052)

In [26]:

```
# 노트북 팁
# 데이터에 두번 접근
s
s # 마지막 연산에 대해서만 접근 결과를 보여줌
```

Out[26]:

```
광역시
서울     9904312
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

리스트 이용 인덱싱

- 자료의 순서를 바꾸거나 특정자료 여러개를 선택할 수 있다.
- 인덱스값 여러개를 이용해 접근시 []안에 넣는다

In [27]:

```
print(s)
# s[0,3,1] #KeyError: 'key of type tuple not found and not a MultiIndex'
s[0],s[3],s[1] #(9904312, 2466052, 3448737)
# 시리즈명[[인덱스리스트]] - 시리즈형태로 반환
s[[0,3,1]] #- 인덱스 리스트 내의 해당 인덱스의 item을 추출 후 시리즈 형태로 반환
```

```
광역시
서울     9904312
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

Out[27]:

```
광역시
서울     9904312
대구     2466052
부산     3448737
Name: 인구, dtype: int64
```

시리즈 슬라이싱

- 정수형 위치 인덱스를 사용한 슬라이싱
  - 시리즈[start:stop+1]
- 문자(라벨)인덱스 이용 슬라이싱
  - 시리즈['시작라벨':'끝라벨'] : 표시된 라벨 범위 모두 추출

In [28]:

```
print(s)
s[[1,2]]
s[['부산','인천']]
s[1:3] # 시리즈 슬라이싱을 사용하면 시리즈로 반환
```

```
광역시
서울     9904312
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

Out[28]:

```
광역시
부산     3448737
인천      289045
Name: 인구, dtype: int64
```

In [29]:

```
# 문자인덱스를 이용한 슬라이싱 가능
# 표시된 문자인덱스 범위 모두 추출
s["부산":"대구"]
```

Out[29]:

```
광역시
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

In [30]:

```
# 정수형 인덱스를 명시했을 경우
s_01 = pd.Series([100,200,300,400], index=[1,2,3,4])
print(s_01)
s_01[[2,3,4]]

# 슬라이싱 사용 - # 0-base 슬라이싱을 사용
s_01[2:4]

# 시리즈의 인덱스를 명시할때는 가급적이면 문자형으로 명시하는 것이 좋다
```

```
1    100
2    200
3    300
4    400
dtype: int64

C:₩Users₩bbobbee₩AppData₩Local₩Temp₩ipykernel_428₩1582668601.py:7: FutureWarning: Th
e behavior of `series[i:j]` with an integer-dtype index is deprecated. In a future v
ersion, this will be treated as *label-based* indexing, consistent with e.g. `series
[i]` lookups. To retain the old behavior, use `series.iloc[i:j]`. To get the future
behavior, use `series.loc[i:j]`.
  s_01[2:4]
```

Out[30]:

```
3    300
4    400
dtype: int64
```

문자 인덱스

- [.]연산자를 이용하여 접근가능

In [31]:

```
# 인덱스를 문자값으로 지정한 시리즈
s0 = pd.Series(range(3),index=['a','b','c'])
s0
```

Out[31]:

```
a    0
b    1
c    2
dtype: int64
```

In [32]:

```
s0['a']
s0.a
```

Out[32]:

```
0
```

In [33]:

```
print(s) # 한글문자 인덱스
s['서울']
s.서울
```

```
광역시
서울     9904312
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

Out[33]:

9904312

인덱스 통한 데이터 업데이트

In [34]:

```
s['서울'] = 10000000
s['서울']
```

Out[34]:

10000000

In [35]:

```
s
```

Out[35]:

```
광역시
서울    10000000
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

인덱스 재 사용 하기

In [36]:

```
print(s.index)
s1 = pd.Series(np.arange(4),s.index)
s1
```

```
Index(['서울', '부산', '인천', '대구'], dtype='object', name='광역시')
```

Out[36]:

```
광역시
서울    0
부산    1
인천    2
대구    3
dtype: int32
```

시리즈 연산

In [37]:

```
# 문자 인덱스의 시리즈  s 확인 후 연산 실습
s
# 도시
# 서울    9904312
# 부산    3448737
# 인천     289045
# 대구    2466052
# Name: 인구, dtype: int64
```

Out[37]:

```
광역시
서울    10000000
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

벡터화 연산

- numpy 배열처럼 pandas의 시리즈도 벡터화 연산 가능
- 벡터화 연산이란 집합적 자료형의 원소 각각을 독립적으로 계산을 진행하는 방법
  - 단, 연산은 시리즈의 값에만 적용되며 인덱스 값은 변경 불가

In [38]:

```
# 시리즈 원소 각각에 대하여 + 4 연산을 진행 - 벡터화 연산
pd.Series([1,2,3]) + 4
```

Out[38]:

```
0    5
1    6
2    7
dtype: int64
```

In [39]:

```
# s 시리즈의 단위가 커서 단위를 변경하고 자 함 1/1000000
# 시리즈 자체를 1000000으로 나누면 됨, 벡터화연산을 진행 함
print(s)
s/1000000 # 대입하지 않았음
```

```
광역시
서울     10000000
부산      3448737
인천       289045
대구      2466052
Name: 인구, dtype: int64
```

Out[39]:

```
광역시
서울     10.000000
부산      3.448737
인천      0.289045
대구      2.466052
Name: 인구, dtype: float64
```

In [40]:

```
s
```

Out[40]:

```
광역시
서울     10000000
부산      3448737
인천       289045
대구      2466052
Name: 인구, dtype: int64
```

In [41]:

```
# 벡터화 인덱싱도 가능
# 시리즈[조건]
# s시리즈 값 중 2500000 보다 크고 5000000보다 작은 원소를 추출
s[(s>250e4) & (s<500e4)]
# s 시리즈 각 원소값 각각에 대해서 조건식을 확인해서 결과가 True인 원소를 반환
```

Out[41]:

```
광역시
부산     3448737
Name: 인구, dtype: int64
```

Boolean selection

- boolean Series가 []와 함께 사용되면 True 값에 해당하는 값만 새로 반환되는 Series객체에 포함됨
- 다중조건의 경우, &(and), |(or)를 사용하여 연결 가능

In [42]:

```
s0 = pd.Series(np.arange(10), np.arange(10)+1)
s0
```

Out[42]:

```
1     0
2     1
3     2
4     3
5     4
6     5
7     6
8     7
9     8
10    9
dtype: int32
```

In [43]:

```
s0 > 5
```

Out[43]:

```
1     False
2     False
3     False
4     False
5     False
6     False
7      True
8      True
9      True
10     True
dtype: bool
```

In [44]:

```
s0[s0>5]
```

Out[44]:

```
7     6
8     7
9     8
10    9
dtype: int32
```

In [45]:

```
# s0 에서 짝수 값만 추출
s0[s0%2 == 0]
```

Out[45]:

```
1    0
3    2
5    4
7    6
9    8
dtype: int32
```

In [46]:

```
s0
```

Out[46]:

```
1     0
2     1
3     2
4     3
5     4
6     5
7     6
8     7
9     8
10    9
dtype: int32
```

In [47]:

```
# 인덱스에도 관계연산이 가능
s0.index > 5
```

Out[47]:

```
array([False, False, False, False, False,  True,  True,  True,  True,
        True])
```

In [48]:

```
s0[s0.index>5]
```

Out[48]:

```
6     5
7     6
8     7
9     8
10    9
dtype: int32
```

In [49]:

```
# s0의 value가 5를 초과하고 8미만인 아이템(원소)만 추출하시오
s0[(s0>5) & (s0<8)]
```

Out[49]:

```
7    6
8    7
dtype: int32
```

In [50]:

```
(s0 >= 7).sum() # True의 개수 총 합
```

Out[50]:

```
3
```

In [51]:

```
 (s0[s0 >= 7]).sum() # 조건의 결과가 True인 원소들의 합
```

Out[51]:

```
24
```

- 두 시리즈간의 연산

In [52]:

```
num_s1=pd.Series([1,2,3,4],index=['a','b','c','d'])
num_s1
```

Out[52]:

```
a    1
b    2
c    3
d    4
dtype: int64
```

In [53]:

```
num_s2=pd.Series([5,6,7,8],index=['b','c','d','a'])
num_s2
```

Out[53]:

```
b    5
c    6
d    7
a    8
dtype: int64
```

In [54]:

```
num_s1 + num_s2 # 시리즈간의 연산은 같은 인덱스를 찾아 연산을 진행
```

Out[54]:

```
a     9
b     7
c     9
d    11
dtype: int64
```

In [55]:

```
num_s3=pd.Series([5,6,7,8],index=['e','b','f','g'])
num_s4=pd.Series([1,2,3,4],index=['a','b','c','d'])
```

In [56]:

```
# 동일한 인덱스는 연산을 진행하고 나머지 인덱스는 연산처리가 불가능 해서 NaN값으로 처리
num_s3 - num_s4
```

Out[56]:

```
a    NaN
b    4.0
c    NaN
d    NaN
e    NaN
f    NaN
g    NaN
dtype: float64
```

In [57]:

```
num_s3.values - num_s4.values
# values 속성을 사용해 값만을 추출해 연산을 진행하게 되면 시리즈의 형태가 사라지므로
# 동일 위치 원소들끼리 연산을 진행
# 시리즈.values는 array 형태 반환
```

Out[57]:

```
array([4, 4, 4, 4], dtype=int64)
```

딕셔너리 와 시리즈의 관계

- 시리즈 객체는 라벨(문자)에 의해 인덱싱이 가능
- 실질적으로는 라벨을 key로 가지는 딕셔너리 형과 같다고 볼 수 있음
- 딕셔너리에서 제공하는 대부분의 연산자 사용 가능
  - in 연산자 : T/F
  - for 루프를 통해 각 원소의 key와 value에 접근 할수 있다.

- in 연산자/ for 반복문 사용

In [58]:

```
s
```

Out[58]:

```
광역시
서울    10000000
부산     3448737
인천      289045
대구     2466052
Name: 인구, dtype: int64
```

In [59]:

```
# 인덱스가 서울인 원소가 시리즈에 있는지 확인( in)
'서울' in s
```

Out[59]:

```
True
```

In [60]:

```
# 인덱스가 대전인 원소가 시리즈에 있는지 확인( in)
'대전' in s
```

Out[60]:

```
False
```

In [61]:

```
# 인덱스가 대전인 원소가 시리즈에 없는지 확인(not in)
'대전' not in s
```

Out[61]:

```
True
```

In [62]:

```
# 딕셔너리의 items() 함수 시리즈에 사용 가능
s.items() # zip 객체
```

Out[62]:

```
<zip at 0x169531ac880>
```

In [63]:

```
list(s.items())
```

Out[63]:

```
[('서울', 10000000), ('부산', 3448737), ('인천', 289045), ('대구', 2466052)]
```

In [64]:

```
# 시리즈 각 원소 출력
for k, v in s.items() :
    print('%s=%d' % (k,v))
```

서울=10000000
부산=3448737
인천=289045
대구=2466052

딕셔너리로 시리즈 만들기

- Series({key:value,key1:value1....})
- 인덱스 -> key
- 값 -> value

In [65]:

```
scores = {'홍길동':96, '이몽룡':100, '성춘향':88}
s=pd.Series(scores)
s
```

Out[65]:

```
홍길동     96
이몽룡    100
성춘향     88
dtype: int64
```

In [66]:

```
city = {'서울':9631482,'부산':3393191,'인천':2632035,'대전':1490158}
s=pd.Series(city)
s
```

Out[66]:

```
서울    9631482
부산    3393191
인천    2632035
대전    1490158
dtype: int64
```

- 딕셔너리의 원소는 순서를 갖지 않는다.
  - 딕셔너리로 생성된 시리즈의 원소도 순서가 보장되지 않는다.
  - 만약 순서를 보장하고 싶으면 인덱스를 리스트로 지정해야 한다.

In [67]:

```
city = {'서울':9631482,'부산':3393191,'인천':2632035,'대전':1490158}
s=pd.Series(city,index=city.keys())
s
```

Out[67]:

```
서울    9631482
부산    3393191
인천    2632035
대전    1490158
dtype: int64
```

In [68]:

```
s=pd.Series(city, index=['부산','인천','서울','대전'])
s
```

Out[68]:

```
부산    3393191
인천    2632035
서울    9631482
대전    1490158
dtype: int64
```

시리즈 데이터의 갱신,추가, 삭제

- 인덱싱을 이용하면 딕셔너리 처럼 갱신, 추가 가능

In [69]:

```
s
```

Out[69]:

```
부산    3393191
인천    2632035
서울    9631482
대전    1490158
dtype: int64
```

In [70]:

```
# s 시리즈의 부산의 인구 값을 1630000으로 변경
s['부산'] = 1630000
s
```

Out[70]:

```
부산    1630000
인천    2632035
서울    9631482
대전    1490158
dtype: int64
```

In [71]:

```
# 시리즈내의 원소 삭제 - del 명령을 사용
del s['서울']
s
```

Out[71]:

```
부산    1630000
인천    2632035
대전    1490158
dtype: int64
```

In [72]:

```
# 시리즈에 새로운 원소 추가
s['대구'] = 1875000
s
```

Out[72]:

```
부산    1630000
인천    2632035
대전    1490158
대구    1875000
dtype: int64
```

Series 함수

Series size, shape, unique, count, value_counts 함수

- size(속성) : 개수 반환
- shape(속성) : 튜플형태로 shape반환
- unique: 유일한 값만 ndarray로 반환
- count : NaN을 제외한 개수를 반환
- mean: NaN을 제외한 평균
- value_counts: NaN을 제외하고 각 값들의 빈도를 반환

In [73]:

```
s1 = pd.Series([1, 1, 2, 1, 2, 2, 2, 1, 1, 3, 3, 4, 5, 5, 7, np.NaN])
s1
```

Out[73]:

```
0     1.0
1     1.0
2     2.0
3     1.0
4     2.0
5     2.0
6     2.0
7     1.0
8     1.0
9     3.0
10    3.0
11    4.0
12    5.0
13    5.0
14    7.0
15    NaN
dtype: float64
```

In [74]:

```
len(s1)
```

Out[74]:

```
16
```

In [75]:

```
s1.size
```

Out[75]:

```
16
```

In [76]:

```
s1.shape # 차원으로 표현하기 때문에 튜플형태로 출력
```

Out[76]:

```
(16,)
```

In [77]:

```
s1.unique() # nan도 하나의 값으로 보고 표현되어짐
```

Out[77]:

```
array([ 1.,  2.,  3.,  4.,  5.,  7., nan])
```

In [78]:

```
s1.count() # nan을 제외한 원소의 개수
```

Out[78]:

```
15
```

In [79]:

```
a = np.array([2,2,2,2,np.NaN]) # array 타입
print(a.mean()) # array에 대해 mean() 적용하면 nan이 포함된 계산을 진행 - nan이 반환

b=pd.Series(a) # 배열을 시리즈로 변경
print(b)
b.mean() # NaN 빼고 계산
```

```
nan
0    2.0
1    2.0
2    2.0
3    2.0
4    NaN
dtype: float64
```

Out[79]:

```
2.0
```

In [80]:

```
s1
```

Out[80]:

```
0     1.0
1     1.0
2     2.0
3     1.0
4     2.0
5     2.0
6     2.0
7     1.0
8     1.0
9     3.0
10    3.0
11    4.0
12    5.0
13    5.0
14    7.0
15    NaN
dtype: float64
```

In [81]:

```
s1.mean()
```

Out[81]:

```
2.6666666666666665
```

In [82]:

```
s1.value_counts()
# 각 원소들에 대해 동일값의 원소끼리 그룹핑하여 개수를 세서 반환하는 함수
```

Out[82]:

```
1.0    5
2.0    4
3.0    2
5.0    2
4.0    1
7.0    1
dtype: int64
```

날짜 자동 생성 : date_range

In [83]:

```
# 날짜 인덱스를 이용하여 시리즈 만들기
# 날짜 표시 : '년-월-일' 형태의 문자열로 표시
index_date = ['2018-10-07','2018-10-08','2018-10-09','2018-10-10']
s4 = pd.Series([200,195,np.nan,205],index = index_date)
s4
```

Out[83]:

```
2018-10-07    200.0
2018-10-08    195.0
2018-10-09      NaN
2018-10-10    205.0
dtype: float64
```

In [84]:

```
type(s4.index[0]) # str
```

Out[84]:

```
str
```

- 판다스 패키지의 date_range 함수 (날짜생성)
  - pd.date_range(start=None, end=None, periods=None, freq='D')
  - start : 시작날짜/ end= 끝날짜 / periods = 날짜 생성기간/ fref = 날짜 생성 주기
  - start는 필수 옵션/end나 periods는 둘 중 하나가 있어야 함/ freq는 기본 Day로 설정

B	비즈니스 데이
C	커스텀 비즈니스 데이
D	일별
W	주별
M	월별 말일
BM	비즈니스 월별
MS	월별 시작일
BMS	비즈니스 월별 시작일
Q	분기별 말일
BQ	비즈니스 분기별
QS	쿼터 시작일
BQS	비지니스 분기 시작일
A	연도별 말일
BA	비즈니스 연도별 말일
AS	연도별 시작일
BAS	비즈니스 연도별 시작일
H	시간별
T	분별
S	초별
L	밀리초(Milliseconds)
U	마이크로초(Microseconds)

In [85]:

```
pd.date_range(start='2018-10-01', end='2018-10-20')
# DatetimeIndex 반환
# dtype='datetime64[ns]'
```

Out[85]:

```
DatetimeIndex(['2018-10-01', '2018-10-02', '2018-10-03', '2018-10-04',
               '2018-10-05', '2018-10-06', '2018-10-07', '2018-10-08',
               '2018-10-09', '2018-10-10', '2018-10-11', '2018-10-12',
               '2018-10-13', '2018-10-14', '2018-10-15', '2018-10-16',
               '2018-10-17', '2018-10-18', '2018-10-19', '2018-10-20'],
              dtype='datetime64[ns]', freq='D')
```

In [86]:

```
pd.date_range(start='2018-10-01', end='2018-10-20', freq='d')
#freq='d' 기본 값
```

Out[86]:

```
DatetimeIndex(['2018-10-01', '2018-10-02', '2018-10-03', '2018-10-04',
               '2018-10-05', '2018-10-06', '2018-10-07', '2018-10-08',
               '2018-10-09', '2018-10-10', '2018-10-11', '2018-10-12',
               '2018-10-13', '2018-10-14', '2018-10-15', '2018-10-16',
               '2018-10-17', '2018-10-18', '2018-10-19', '2018-10-20'],
              dtype='datetime64[ns]', freq='D')
```

In [87]:

```
pd.date_range(start='2018-10-01', end='2018-10-20', freq='3D') # 3일씩 증가
```

Out[87]:

```
DatetimeIndex(['2018-10-01', '2018-10-04', '2018-10-07', '2018-10-10',
               '2018-10-13', '2018-10-16', '2018-10-19'],
              dtype='datetime64[ns]', freq='3D')
```

In [88]:

```
pd.date_range(start='2018-10-01', end='2018-10-20', freq='w')
# 2018-10-01일을 기준으로 1주일씩 증가하는 날짜
# 1주 시작일 일요일을 표시
# 2018년 10월 1일 - 월요일
```

Out[88]:

```
DatetimeIndex(['2018-10-07', '2018-10-14'], dtype='datetime64[ns]', freq='W-SUN')
```

In [89]:

```
# 2018년 10월 1일 이후 일요일 날짜 4개 생성
pd.date_range(start='2018-10-01', periods=4, freq='w')
```

Out[89]:

```
DatetimeIndex(['2018-10-07', '2018-10-14', '2018-10-21', '2018-10-28'], dtype='datet
ime64[ns]', freq='W-SUN')
```

In [90]:

```
# 2018-10-01일 이후 월의 마지막 날짜 4개 생성
pd.date_range(start='2018-10-01', periods=4, freq='m')
```

Out[90]:

```
DatetimeIndex(['2018-10-31', '2018-11-30', '2018-12-31', '2019-01-31'], dtype='datet
ime64[ns]', freq='M')
```

In [91]:

```
pd.date_range(start='2018-10-01', periods=4, freq='M')
```

Out[91]:

```
DatetimeIndex(['2018-10-31', '2018-11-30', '2018-12-31', '2019-01-31'], dtype='datet
ime64[ns]', freq='M')
```

In [92]:

```
pd.date_range(start='2018-10-01', periods=4, freq='MS')
```

Out[92]:

```
DatetimeIndex(['2018-10-01', '2018-11-01', '2018-12-01', '2019-01-01'], dtype='datet
ime64[ns]', freq='MS')
```

In [93]:

```
pd.date_range(start='2018-10-01', periods=12, freq='2BM')
# '2BM' 업무일 기준 2개월 간격 월말 주기
```

Out[93]:

```
DatetimeIndex(['2018-10-31', '2018-12-31', '2019-02-28', '2019-04-30',
               '2019-06-28', '2019-08-30', '2019-10-31', '2019-12-31',
               '2020-02-28', '2020-04-30', '2020-06-30', '2020-08-31'],
              dtype='datetime64[ns]', freq='2BM')
```

In [94]:

```
pd.date_range(start='2018-10-01', periods=4, freq='QS')
# 분기 시작일 기준
```

Out[94]:

```
DatetimeIndex(['2018-10-01', '2019-01-01', '2019-04-01', '2019-07-01'], dtype='datet
ime64[ns]', freq='QS-JAN')
```

In [95]:

```
pd.date_range(start='2018-10-01', periods=4, freq='AS')
# 2018년 10월 1일 이후 연도 시작일 4개 생성
```

Out[95]:

```
DatetimeIndex(['2019-01-01', '2020-01-01', '2021-01-01', '2022-01-01'], dtype='datet
ime64[ns]', freq='AS-JAN')
```

In [96]:

```
pd.date_range(start='2018-01-01', periods=4, freq='AS')
```

Out[96]:

```
DatetimeIndex(['2018-01-01', '2019-01-01', '2020-01-01', '2021-01-01'], dtype='datet
ime64[ns]', freq='AS-JAN')
```

- 판다스 패키지의 date_range 함수 (시간생성)

In [97]:

```
pd.date_range(start='2018-1-20 08:00', periods=10, freq='H')
```

Out[97]:

```
DatetimeIndex(['2018-01-20 08:00:00', '2018-01-20 09:00:00',
               '2018-01-20 10:00:00', '2018-01-20 11:00:00',
               '2018-01-20 12:00:00', '2018-01-20 13:00:00',
               '2018-01-20 14:00:00', '2018-01-20 15:00:00',
               '2018-01-20 16:00:00', '2018-01-20 17:00:00'],
              dtype='datetime64[ns]', freq='H')
```

In [98]:

```
pd.date_range(start='2018-1-20 08:00', periods=10, freq='BH')
# 업무시간 기준 9 to 5로 설정
```

Out[98]:

```
DatetimeIndex(['2018-01-22 09:00:00', '2018-01-22 10:00:00',
               '2018-01-22 11:00:00', '2018-01-22 12:00:00',
               '2018-01-22 13:00:00', '2018-01-22 14:00:00',
               '2018-01-22 15:00:00', '2018-01-22 16:00:00',
               '2018-01-23 09:00:00', '2018-01-23 10:00:00'],
              dtype='datetime64[ns]', freq='BH')
```

In [99]:

```
pd.date_range(start='2018-1-20 08:00', periods=10, freq='30min')
```

Out[99]:

```
DatetimeIndex(['2018-01-20 08:00:00', '2018-01-20 08:30:00',
               '2018-01-20 09:00:00', '2018-01-20 09:30:00',
               '2018-01-20 10:00:00', '2018-01-20 10:30:00',
               '2018-01-20 11:00:00', '2018-01-20 11:30:00',
               '2018-01-20 12:00:00', '2018-01-20 12:30:00'],
              dtype='datetime64[ns]', freq='30T')
```

In [100]:

```
pd.date_range(start='2018-1-20 08:00', periods=10, freq='10S')
```

Out[100]:

```
DatetimeIndex(['2018-01-20 08:00:00', '2018-01-20 08:00:10',
               '2018-01-20 08:00:20', '2018-01-20 08:00:30',
               '2018-01-20 08:00:40', '2018-01-20 08:00:50',
               '2018-01-20 08:01:00', '2018-01-20 08:01:10',
               '2018-01-20 08:01:20', '2018-01-20 08:01:30'],
              dtype='datetime64[ns]', freq='10S')
```

Series 끝

In [ ]:

```

```