
▣ Ch17 데이터 분석을 위한 데이터 구조 다루기

Ch16과 Ch17 내용을 함께 다룬다. (이미 Python을 다룰 줄 아는 학생들을 대상으로 진행하기 때문에)

- > Python의 데이터 구조 형식 중에 데이터 분석에 관련된 쓰임새 위주로 살펴본다.
- > 하나의 값 만을 저장하는 스칼라(Scalar) 형 int, float, str, bool 자료형
- > 여러 개의 값을 사용할 수 있는 비스칼라 형 list, tuple, dictionary 자료형
- > 외장 Pandas 자료형인 series, data frame 자료형

■ Python 자료 구조 종류

[내장 자료형]

- >> 스칼라(Scalar) 형
- >> 리스트(list) 형
- >> 튜플(Tuple) 형
- >> 딕셔너리(Dictionary) 형

[외장 자료형] Pandas 자료형

- << 시리즈(Series) 형
- << 데이터 프레임(Data Frame) 형

In []:

▣ 스칼라(Scalar) 형

- > 하나의 값 만으로 구성된 자료 구조
 - > Data Type : int, float, str, bool 등
 - > 관련 함수 : int(), float(), str(), bool()
- <> 기본 자료형

```
In [10]: ## 스칼라(Scalar) 형 ##
## 생성 및 초기화
a = 3          #int형 자료 구조 생성 및 초기화
print(type(a))      #자료구조 확인
b = 4.15        #float형 자료 구조 생성 및 초기화
print(type(b))
c = 'hello, world!'    #str형 자료 구조 생성 및 초기화
print(type(c))

<class 'int'>
<class 'float'>
<class 'str'>
```

○ Python에서는 자료 구조가 모두 **class** 형태로 만들어진다.

○ 따라서 자료형에 대해 메서드들이 제공된다.

<> 데이터 타입 바꾸기

```
In [13]: ## 타입 바꾸기: int(), float(), str()
a = 3.14
print(type(a), a)
b = int(a)          # float > int
print(type(b), b)
c = str(b)          # int > str
print(type(c), c)
d = float(c)        # str > float
print(type(d), d)
e = str(d)          # float > str
print(type(a), e)

<class 'float'> 3.14
<class 'int'> 3
<class 'str'> 3
<class 'float'> 3.0
<class 'float'> 3.0
```

<> 자료형의 메서드 사용하기

```
In [127... c.upper()           # 'HELLO, WORLD!'
Out[127]: 'HELLO, WORLD!'
```

```
In [128... c.capitalize()       # 'Hello, world!'
Out[128]: 'Hello, world!'
```

```
In [129... c.title()            # 'Hello, World!'
Out[129]: 'Hello, World!'
```

```
In [130... c.replace('!', '!!!')  # 'Hello, World!!!'
Out[130]: 'hello, world!!!'
```

```
In [131... c.find(',')          # 5
Out[131]: 5
```

```
In [132]: c.split(',') # ['Hello', 'World!']
```

```
Out[132]: ['Hello', 'World!']
```

```
In [ ]:
```

☒ 비 스칼라 형

- > 여러 개의 값을 사용할 수 있도록 구성된 자료 구조
- > Data Type : list, tuple, dictionary
- > 관련 함수 : list(), tuple(), dict()

```
In [ ]:
```

○ 리스트(List) 형 : []

- > 여러 개의 값을 나열된 자료 구조
- > 기본적으로 순서번호(Index)로 값에 접근
- > 값의 삽입, 삭제가 가능하다.
- > 각 값들은 [와] 안에 ,로 구분하여 초기화 한다.

<> 추출하기

```
In [7]: ## 리스트(List) 형 : [ ] ##
```

```
## 생성 및 검색
a = [1, 2, 'a', 'b']          #리스트 생성 및 초기화
print(a[0])    #리스트는 0번째부터 시작 > a[0]는 한 개 값
print(a[-1])   #거꾸로 첫 번째
print(a[2:3])  #2번째부터 3번째 전까지 > a[2:3]는 한 개 값이지만 범위이므로 결과는 룩
print(a[2:])   #2번째부터 끝까지 > 결과는 룩은 값 list형
print(a[:2])   #처음부터 2번째 전까지 > 결과는 룩은 값 list형
```

```
1
b
['a']
['a', 'b']
[1, 2]
```

<> 수정 조작하기

```
In [6]: ## 값 변경
```

```
a[2] = 'A'
a
```

```
Out[6]: [1, 2, 'A', 'b']
```

관련 메서드

- > append() 항목 값 추가하기

- > **remove()** 항목 값으로 제거하기
- > **insert(index, value)** index로 항목 추가하기
- > **index()** index 값으로 검색하기
- > **sort()** 항목을 정렬하여 바꾸기 (매개변수 **reverse=False**가 기본값)

```
In [13]: ## 생성 및 조작
a.append('c')          # 맨 마지막에 'c'를 추가
a.insert(2, 3)          # 2번째에 값 3을 추가
a.remove(3)            # 3을 찾아 제거
a.index('a')           # 'a'의 위치 값 반환
a
```

Out[13]: [1, 2, 'a', 'b', 'c']

관련 함수

- > **len()** 항목 개수 구하기
- > **sorted()** 항목을 정렬하여 리스트를 반환(원 리스트는 변화 없음)
- > **del()** 특정 항목 제거하기
- > **clear()** 모든 항목 제거하기

```
In [17]: ## 관련 함수: 정렬
a = [5, 3, 7, 8, 2, 1]
asort = sorted(a)    # 값을 정렬
asort
```

Out[17]: [1, 2, 3, 5, 7, 8]

```
In [16]: ## 관련 함수: 제거, 갯 수
del(a[1])          # 1번째 값을 제거
print(a)
len(a)
```

[5, 8, 2, 1]

Out[16]: 4

<> 1, 2차원 리스트 초기화 하기

```
In [20]: ## 1차원 리스트 초기화 방법
arr = [0, 0, 0, 0, 0]
arr = [0]*5;
arr = [0 for i in range(5)];
arr=[];          # 만들고
for i in range(5):
    arr.append(0)      # 추가하고
arr
```

Out[20]: [0, 0, 0, 0, 0]

```
In [21]: ## 2차원 리스트 초기화 방법
arr = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
arr = [[0 for j in range(4)] for i in range(3)];
```

```

arr = [] # 1차원 리스트 생성
for i in range(3):
    temp = [] # 열 생성
    for j in range(4):
        temp.append(0) # 열 값 추가
    arr.append(temp) # 1차원 리스트에 열 추가
arr

```

Out[21]: [[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]]

<> 반복문으로 리스트 읽어내기

```

In [14]: ## 1차원 리스트 읽어내기
a = [1, 2, 'a', 'b'] #리스트 생성 및 초기화
for x in a: #range() 대신에 리스트 자료 Unpacking
    print(x, end=' ')

```

1 2 a b

```

In [15]: ## 2차원 리스트 읽어내기
a = [1, 2, ['a', 'b'], 4] #리스트 생성 및 초기화
for x in a: #range() 대신에 리스트 자료 Unpacking
    if type(x) is list: #x가 list type인지 확인
        for y in x:
            print(y, end=' ')
    else:
        print(x, end=' ')

```

1 2 a b 4

[실습] List로 막대 그래프 그리기

> import seaborn 모듈 사용

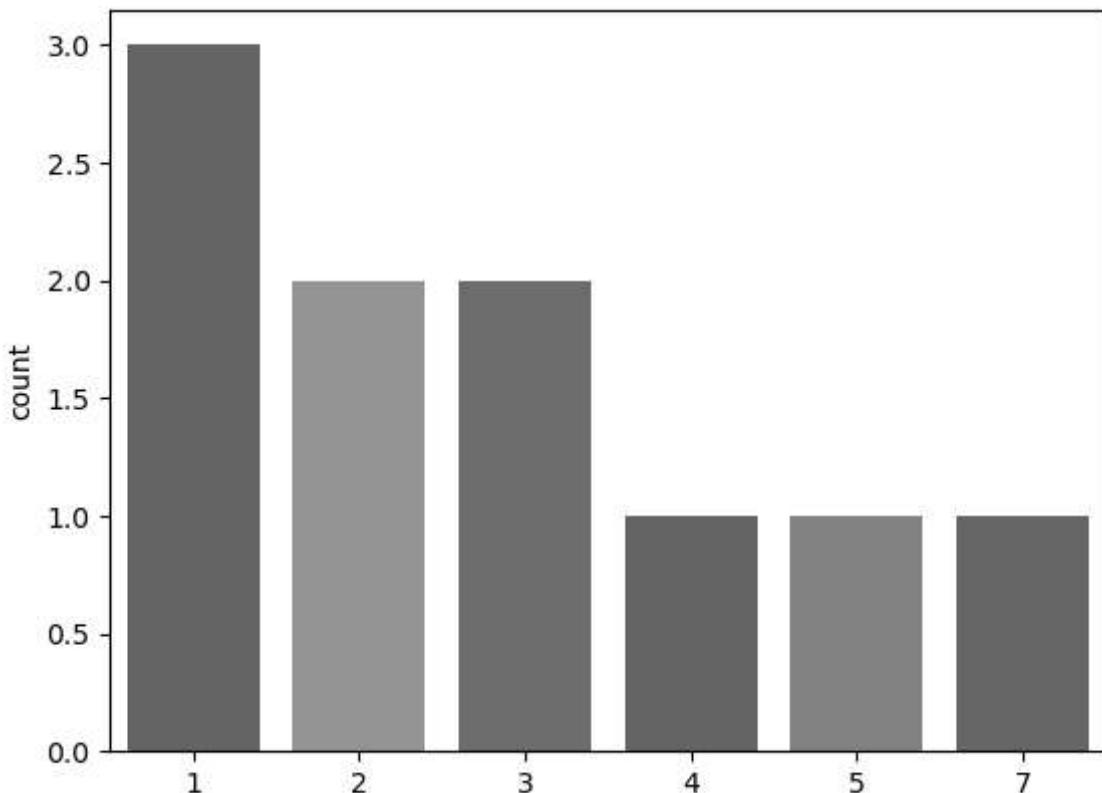
```

In [6]: ## 리스트(List) 형 : []
## List로 막대 그래프 그리기
alist = [1, 4, 5, 3, 2, 1, 7, 3, 1, 2]
print(alist)

import seaborn as sns
sns.countplot(x = alist)

```

[1, 4, 5, 3, 2, 1, 7, 3, 1, 2]
Out[6]: <AxesSubplot:ylabel='count'>



[실습] List로 막대 그래프 그리기 (응용)

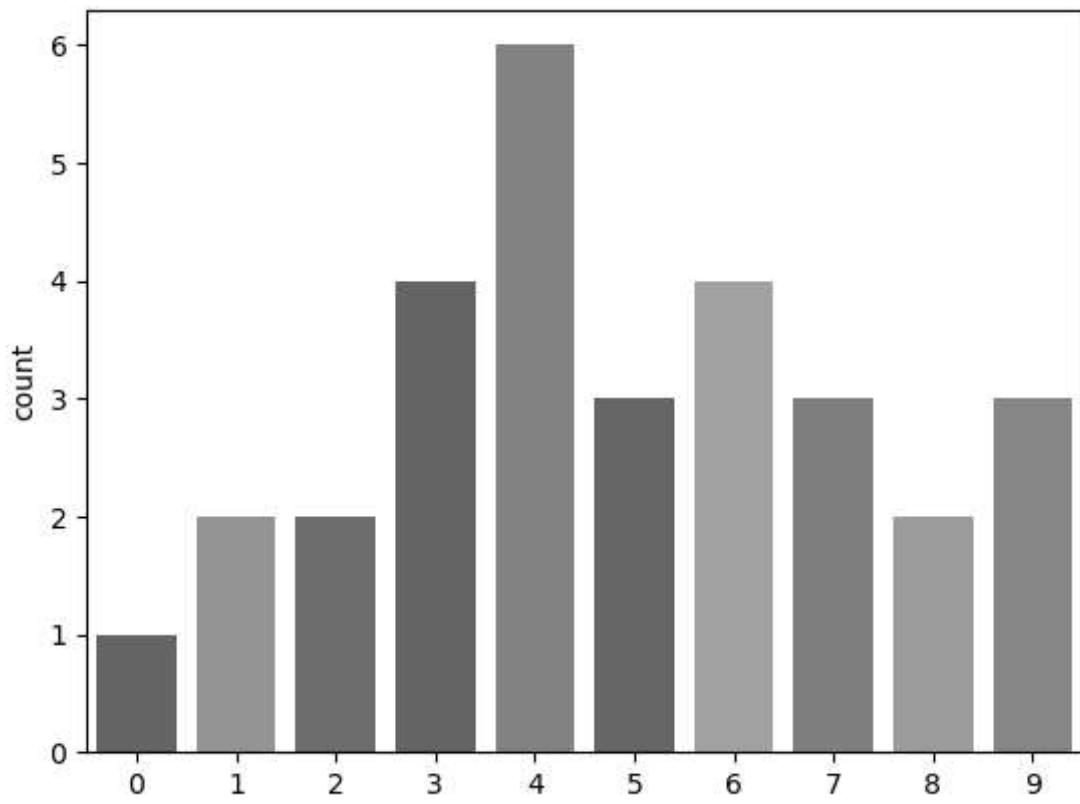
> List를 랜덤 수로 초기화 함. (range는 0~9까지)

> List의 길이는 30으로 함

```
In [75]: ## 리스트(List) 형 : []
## [실습] List를 랜덤 수로 초기화 함
## 리스트 구성
import random
alist = []
for _ in range(30):
    x = random.randrange(0, 10)
    alist.append(x)
print(alist)

## List로 막대 그래프 그리기
import seaborn as sns
sns.countplot(x = alist)

[3, 8, 1, 4, 7, 4, 7, 9, 8, 4, 6, 6, 5, 6, 6, 2, 4, 3, 9, 5, 3, 3, 5, 9, 4, 4, 1, 2,
0, 7]
<AxesSubplot:ylabel='count'>
Out[75]:
```



```
In [1]: ## seaborn 패키지 인터페이스 매개변수 확인  
import seaborn as sb  
sb.countplot?
```

```
Signature:
```

```
sb.countplot(  
    ...  
    *,  
    ... x=None,  
    ... y=None,  
    ... hue=None,  
    ... data=None,  
    ... order=None,  
    ... hue_order=None,  
    ... orient=None,  
    ... color=None,  
    ... palette=None,  
    ... saturation=0.75,  
    ... dodge=True,  
    ... ax=None,  
    ... **kwargs,  
)
```

```
Docstring:
```

```
Show the counts of observations in each categorical bin using bars.
```

A count plot can be thought of as a histogram across a categorical, instead of quantitative, variable. The basic API and options are identical to those for :func:`barplot`, so you can compare counts across nested variables.

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the ``x``, ``y``, and/or ``hue`` parameters.
- A "long-form" DataFrame, in which case the ``x``, ``y``, and ``hue`` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, ... n) on the relevant axis, even when the data has a numeric or date type.

See the :ref:`tutorial <categorical_tutorial>` for more information.

Parameters

x, y, hue : names of variables in ``data`` or vector data, optional
Inputs for plotting long-form data. See examples for interpretation.

data : DataFrame, array, or list of arrays, optional
Dataset for plotting. If ``x`` and ``y`` are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

order, hue_order : lists of strings, optional
Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

orient : "v" | "h", optional
Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguity when both `x` and `y` are numeric or when plotting wide-form data.

color : matplotlib color, optional
Color for all of the elements, or seed for a gradient palette.

palette : palette name, list, or dict
Colors to use for the different levels of the ``hue`` variable. Should

```
.... be something that can be interpreted by :func:`color_palette`, or a
.... dictionary mapping hue levels to matplotlib colors. ....
saturation : float, optional
.... Proportion of the original saturation to draw colors at. Large patches
.... often look better with slightly desaturated colors, but set this to
.... ``1`` if you want the plot colors to perfectly match the input color
.... spec. ....
dodge : bool, optional
.... When hue nesting is used, whether elements should be shifted along the
.... categorical axis. ....
ax : matplotlib Axes, optional
.... Axes object to draw the plot onto, otherwise uses the current Axes. ....
kwargs : key, value mappings
.... Other keyword arguments are passed through to
.... :meth:`matplotlib.axes.Axes.bar`.
```

Returns

```
ax : matplotlib Axes
.... Returns the Axes object with the plot drawn onto it. ....
```

See Also

```
barplot : Show point estimates and confidence intervals using bars. ....
catplot : Combine a categorical plot with a :class:`FacetGrid`. ....
```

Examples

Show value counts for a single categorical variable:

```
.. plot::
.... :context: close-figs

.... >>> import seaborn as sns
.... >>> sns.set_theme(style="darkgrid")
.... >>> titanic = sns.load_dataset("titanic")
.... >>> ax = sns.countplot(x="class", data=titanic)
```

Show value counts for two categorical variables:

```
.. plot::
.... :context: close-figs

.... >>> ax = sns.countplot(x="class", hue="who", data=titanic)
```

Plot the bars horizontally:

```
.. plot::
.... :context: close-figs

.... >>> ax = sns.countplot(y="class", hue="who", data=titanic)
```

Use a different color palette:

```
.. plot::
.... :context: close-figs

.... >>> ax = sns.countplot(x="who", data=titanic, palette="Set3")
```

Use :meth:`matplotlib.axes.Axes.bar` parameters to control the style.

```
.. plot::
.... :context: close-figs
```

```

...     >>> ax = sns.countplot(x="who", data=titanic,
...                         facecolor=(0, 0, 0, 0),
...                         linewidth=5,
...                         edgecolor=sns.color_palette("dark", 3))

Use :func:`catplot` to combine a :func:`countplot` and a
:class:`FacetGrid`. This allows grouping within additional categorical
variables. Using :func:`catplot` is safer than using :class:`FacetGrid`
directly, as it ensures synchronization of variable order across facets:

.. plot::
   :context: close-figs

...     >>> g = sns.catplot(x="class", hue="who", col="survived",
...                         data=titanic, kind="count",
...                         height=4, aspect=.7);
File:    c:\users\admin\anaconda3\lib\site-packages\seaborn\categorical.py
Type:    function

```

In []:

○ 튜플(Tuple) 형 : ()

- > 리스트와 같이 여러 개의 값들을 나열된 자료 구조
- > 리스트와 다르게 한 번 초기화된 자료는 **수정할 수 없다.**
- > 고정된 값을 미리 정렬시켜서 튜플로 만들어 놓고 사용하면 검색 속도 빨라짐.

<> 생성하기

```

In [16]: ## Tuple 생성하기
          a = (1, 2, 3, 4, 5)    #튜플의 생성 및 초기화
          a

```

Out[16]: (1, 2, 3, 4, 5)

```

In [20]: # () 생략하고 튜플 만들기
          a = 1, 2, 3, 4, 5    #튜플의 패킹(Packing)
          a

```

Out[20]: (1, 2, 3, 4, 5)

<> 추출하기

```

In [19]: ## 튜플(Tuple) 형 : ( )
          ## 생성 및 검색
          a = (1, 2, 3, 4, 5)    #튜플의 생성 및 초기화
          print(a[0])
          print(a[1:3])      #범위로 추출하므로 결과는 tuple 형

```

1
(2, 3)

<> 튜플의 패킹(Packing)과 언패킹(Unpacking)

```
In [133]: ## 튜플(Tuple) 형 : ( )
## 패킹(Packing), 언패킹(Unpacking)
b = 10, 20, 30      #튜플의 패킹(Packing)
x, y, z = b        #튜플의 언패킹(Unpacking)
print(x, y, z)
```

```
10 20 30
```

```
In [ ]:
```

○ 딕셔너리(Dictionary) 형 : {}

- > 키(Key)와 대응되는 값(Value)이 짹을 이루어 나열된 자료 구조
- > 값의 삽입, 삭제가 가능하다.
- > 순서번호(Index) 대신에 키 값으로 값에 접근

<> 생성하기

```
In [42]: ## 딕셔너리(Dictionary) 형 : {} ##
## Key : Value로 생성
a = {'A':90, 'B':80, 3:70, 4:60}          #딕셔너리 생성 및 초기화
a
```

```
Out[42]: {'A': 90, 'B': 80, 3: 70, 4: 60}
```

<> 조작 하기

추가하기

```
In [45]: ## 항목 추가하기
a['C'] = [79, 70] #맨 뒤에 추가됨
a
```

```
Out[45]: {'A': 90, 'B': 80, 3: 70, 4: 60, 'C': [79, 70]}
```

삭제하기

```
In [ ]: ## 항목 추가하기
del(a['C']) # id 키:값 쌍 삭제
a
```

```
Out[ ]: {'A': 90, 'B': 80, 3: 70, 4: 60}
```

값 변경하기

```
In [51]: ## 항목 값 변경하기
a['B'] = [89, 80] #Key가 없으면 추가됨
a
```

```
Out[51]: {'A': 90, 'B': [89, 80], 3: 70, 4: 60}
```

<> 추출하기

```
In [52]: ## 딕셔너리(Dictionary) 형 : { } ##
## 추출
a = {'A':90, 'B':80, 3:70, 4:60}          #딕셔너리 생성 및 초기화
print(a['A'])
print(a[3])
```

90

70

```
In [58]: ## 딕셔너리(Dictionary) 형 : { } ##
## Key와 Value 추출
keys = a.keys()      #Key 추출
values = a.values()  #Value 추출
print(keys)
print(values)
```

```
dict_keys(['A', 'B', 3, 4])
```

```
dict_values([90, 80, 70, 60])
```

<> 반복문으로 Dictionary 읽어내기

```
In [62]: ## Dictionary 키 읽어내기
a = {'A':90, 'B':80, 3:70, 4:60}
for x in a.keys():    #range() 대신에 Dictionary 자료 Unpacking
    print(x)
```

A

B

3

4

```
In [60]: ## [Dictionary] 반복문으로 값 읽어내기
for x in a.values():    #range() 대신에 Dictionary 자료 Unpacking
    print(x)
```

90

80

70

60

```
In [61]: ## [Dictionary] 반복문으로 키, 값 읽어내기
for x in a.items():        #range() 대신에 적용
    print(x)
for x, y in a.items():     #range() 대신에 적용
    print(x, y)
```

('A', 90)

('B', 80)

(3, 70)

(4, 60)

A 90

B 80

3 70

4 60

```
In [ ]:
```

▣ 외장 Pandas 자료형

> pandas라는 패키지를 import 해서 사용 가능한 추가 자료 구조

> Data Type : series, dataframe

> 관련 함수 : Series(), DataFrame()

In []:

○ 시리즈(Series) 형

> 1차원 배열로 여러 값을 나열한 자료 구조

> 각 데이터 항목은 index로 식별된다.

> 데이터 프레임의 데이터를 구성하는 열 값들을 시리즈로 추출하여 조작할 수 있다.

> pandas 패키지를 통해 사용할 수 있다.

<> 생성하기

In [63]:

```
## 시리즈(Series) 형 ##
## index 미지정 생성하기
import pandas as pd
a = pd.Series([11, 22, 33, 44]) #index 자동 부여
a
```

Out[63]:

```
0    11
1    22
2    33
3    44
dtype: int64
```

In [67]:

```
## 시리즈(Series) 형 ##
## index 지정 생성하기
b = pd.Series([11, 22, 33, 44], index = [1, 2, 3, 4]) #index 강제 부여
b
```

Out[67]:

```
1    11
2    22
3    33
4    44
dtype: int64
```

<> 조작하기

추가/수정 하기

> index가 없으면 추가, 있으면 수정

In [76]:

```
## index=3에 9 할당
a = pd.Series([11, 22, 33, 44]) #index 자동 부여
a[4] = 9 #index가 없으면 추가, 있으면 수정
a
```

```
Out[76]: 0    11
          1    22
          2    33
          3    44
          4     9
         dtype: int64
```

```
In [79]: ## index=3에 9 할당
b = pd.Series([11, 22, 33, 44], index = [1, 2, 3, 4]) #index 강제 부여
b[4] = 9      #index가 없으면 추가, 있으면 수정
b
```

```
Out[79]: 1    11
          2    22
          3    33
          4     9
         dtype: int64
```

추출 하기

```
In [88]: ## 시리즈(Series) 형 ##
## 생성 및 추출
a = pd.Series([11, 22, 33, 44])
print(a[1])      #값으로 반환
print(a[1:2])    #Series로 반환
```



```
22
1    22
dtype: int64
```

```
In [102...]: ## 생성 및 추출
b = pd.Series([11, 22, 33, 44], index = ['a', 'b', 'c', 'd'])
print(b['a'])
print(b['b'])
print(b[['a', 'b']])  #List로 검색, Series로 반환
```



```
22
22
a    11
b    22
dtype: int64
```

삭제하기

```
In [93]: b = pd.Series([11, 22, 33, 44], index = ['a', 'b', 'c', 'd'])
del(b['b'])
b
```

```
Out[93]: a    11
          c    33
          d    44
         dtype: int64
```

```
In [90]: ## 시리즈 조작
a = pd.Series([11, 22, 33, 44])

a['e'] = 50;           #새로운 값 추가
print(a)
a['e'] = 55;           #새로운 값으로 변경
print(a)
del(a['e']);           #값 제거
print(a)
```

```
0    11  
1    22  
2    33  
3    44  
e    50  
dtype: int64  
0    11  
1    22  
2    33  
3    44  
e    55  
dtype: int64  
0    11  
1    22  
2    33  
3    44  
dtype: int64
```

반복문으로 읽어내기

```
In [95]: ## 시리즈를 for 반복문으로 읽어내기  
a = pd.Series([11, 22, 33, 44]) #index 자동 부여  
for x in a: #range() 대신에 적용  
    print(x)
```

```
11  
22  
33  
44
```

pandas 함수의 출력 결과 활용하기

```
In [100...]: ## pandas 함수 활용  
a = pd.Series([11, 22, 33, 44]) #index 자동 부여  
a.value_counts() #항목별 분포 수  
a.sum() #항목의 합  
a.mean() #항목 평균  
a.max() #최고값 항목  
a.min() #최소값 항목
```

```
Out[100]: 11
```

```
In [ ]:
```

[실습] List로 Series 만들기

- > List를 랜덤 수로 초기화 함. (range는 0~9까지)
- > List의 길이는 30으로 함
- > 각 숫자의 빈도 수를 index로 하는 Series 구성

```
In [103...]: ## 리스트(List) 형 : []  
## [실습] List를 랜덤 수로 초기화 함  
## 리스트 구성  
import random  
alist = []  
for _ in range(30):  
    x = random.randrange(0, 10)
```

```

alist.append(x)
print(alist)

# 빈도 리스트 data 만들기
data = [0 for i in range(10)]
for x in alist:
    data[x] += 1
# 인덱스 리스트 idx 만들기
idx = [i for i in range(10)]
print(data)
print(idx)

## pandas 패키지 활용 : 데이터 시리즈(2차원 자료 구조) 구성
import pandas as pd
ds = pd.Series(data, index = idx)
print(ds)

[6, 1, 1, 4, 0, 1, 9, 1, 2, 1, 7, 8, 4, 8, 9, 5, 9, 5, 9, 2, 3, 8, 7, 5, 9, 0, 7, 2,
5, 9]
[2, 5, 3, 1, 2, 4, 1, 3, 3, 6]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 ... 2
1 ... 5
2 ... 3
3 ... 1
4 ... 2
5 ... 4
6 ... 1
7 ... 3
8 ... 3
9 ... 6
dtype: int64

```

In []:

○ 데이터 프레임(Data Frame) 형

- > 행과 열로 나열되는 **2차원적** 자료 구조
- > 행과 열 값들은 딕셔너리 구조를 활용하여 초기화 한다.
- > 하나의 열 값들은 시리즈형으로 추출하여 사용 가능하다.
- > **pandas** 패키지를 통해 사용할 수 있다.

생성하기

In [105...]

```

## 데이터 프레임(Data Frame) 형 ##
## 생성 및 검색
import pandas as pd
df = pd.DataFrame({'Eng' : [87, 79, 80],
                   'Mat' : [80, 90, 80]}) #index 자동 부여
df

```

Out[105]:

	Eng	Mat
0	87	80
1	79	90
2	80	80

In [149...]

```
## [Data Frame] 생성 및 조작
import pandas as pd
df = pd.DataFrame({'Eng' : [87, 79, 80],
                    'Mat' : [80, 90, 80]},
                    index = ['Kims', 'Lees', 'Parks']) #index 강제 부여
df
```

Out[149]:

	Eng	Mat
Kims	87	80
Lees	79	90
Parks	80	80

추출하기

> Data Frame에서 변수를 추출하면 Series형으로 반환

In [138...]

```
## [Data Frame] 조작 : 열 추출
x = df['Eng']; # 'Eng' 열 값들을 시리즈(Series)로 추출, 결과는 Series
print(type(x))
x
```

```
<class 'pandas.core.series.Series'>
```

Out[138]:

```
Kims    87
Lees    79
Parks   80
Name: Eng, dtype: int64
```

In [150...]

```
## [Data Frame] 조작 : 열 추출
y = df[['Mat', 'Eng']] # ['Mat', 'Eng'], 결과는 Data Frame
print(type(y))
y
```

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

Out[150]:

	Mat	Eng
Kims	80	87
Lees	90	79
Parks	80	80

추가/수정 하기

> index가 없으면 추가, 있으면 수정

In [151...]

```
## [Data Frame] 조작 : 열 추가
df['avg'] = (df['Eng'] + df['Mat']) / 2 # 'avg' 열을 만들어서 평균 값 추가
df
```

Out[151]:

	Eng	Mat	avg
Kims	87	80	83.5
Lees	79	90	84.5
Parks	80	80	80.0

행, 열 제거하기

In [152...]

```
## [Data Frame] 관련 함수 : 행, 열 제거
ddf = df.drop( 'Lees', axis = 'rows' )           # 행 이름으로 열 제거
ddf = ddf.drop( 'avg', axis = 'columns' )         # 컬럼 이름으로 컬럼 제거
ddf
```

Out[152]:

	Eng	Mat
Kims	87	80
Parks	80	80

pandas 함수의 출력 결과 활용하기

In [156...]

```
## [Data Frame] 관련 함수 사용
df.value_counts()          # 'Eng' 열 값들의 개수 구하기
```

Out[156]:

```
Eng Mat avg
79 90 84.5 ... 1
80 80 80.0 ... 1
87 80 83.5 ... 1
dtype: int64
```

In [157...]

```
df.mean()                  # 'Eng' 열 값들의 평균 구하기
```

Out[157]:

```
Eng ... 82.000000
Mat ... 83.333333
avg ... 82.666667
dtype: float64
```

In [154...]

```
## [Data Frame] 관련 함수 사용
df['Mat'].mean()          # 'Mat' 열 값들의 평균 구하기
```

Out[154]:

```
83.33333333333333
```

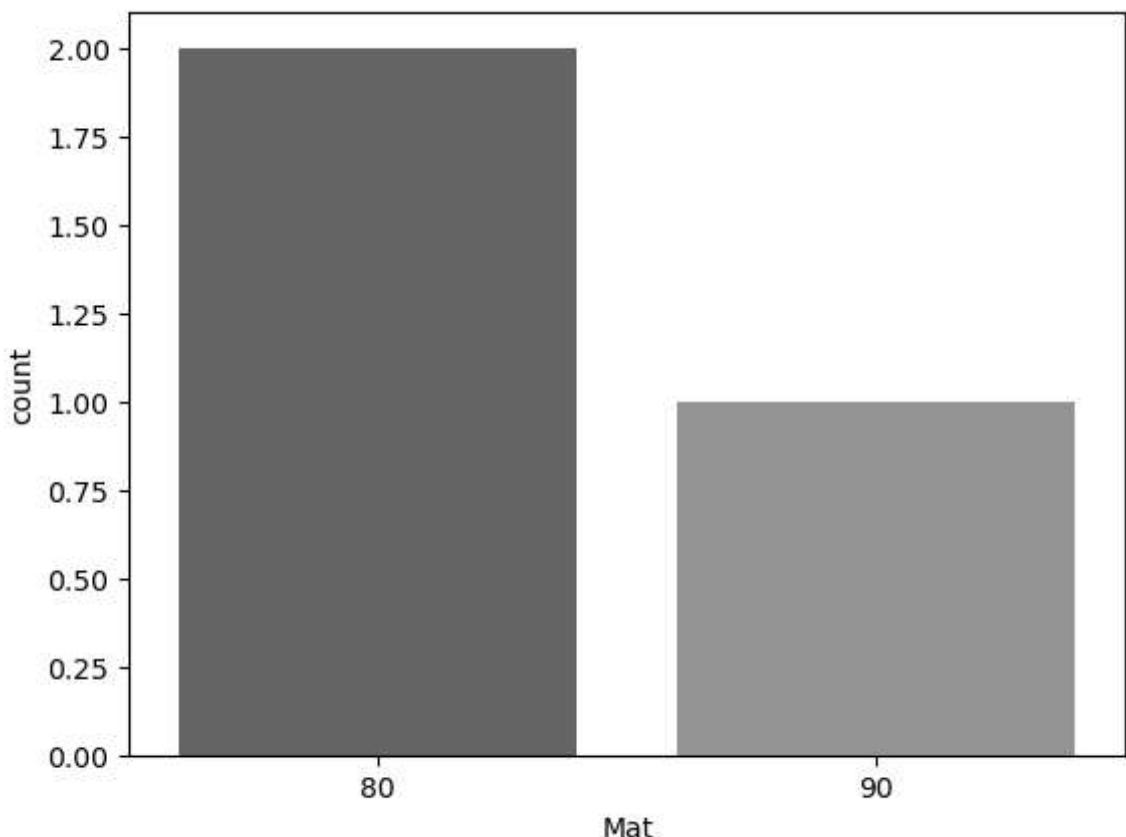
Data Frame으로 그래프 그리기

In [73]:

```
## [Data Frame] 그래프 그리기
import seaborn as sns
sns.countplot(data=df, x='Mat')
```

Out[73]:

```
<AxesSubplot:xlabel='Mat', ylabel='count'>
```



In []: