

Numpy 비교 연산자부터 본 후에,
Pandas 소개, Visualize 목적으로 사용하는 Matplotlib 소개,
하고 마지막으로 Scikit.learn 실습 코드를 보면서 왜 앞에 내용들을 배웠는지 설명.

Numpy 라이브러리 2

All, Any

- All : Array의 데이터가 전부 조건에 만족하면 True
- Any : Array의 데이터 중 하나라도 조건에 만족하면 True

```
a = np.arange(5)
a
```

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

```
np.all(a > 3)
```

```
False
```

```
np.all(a < 5)
```

```
True
```

```
np.any(a > 3)
```

```
True
```

```
np.any(a > 5)
```

```
False
```

all 은 말 그대로 모든 조건 만족하면 **true** 가 나오고,
any 는 하나라도 만족하면 **true** 를 추출해내는 함수이다.

arange()함수를 사용해서 0 에서 4 까지가 들어간 array 를 만들었다.

이제 이걸 하나씩 조건과 비교를 할 것인데,
'a 는 3 보다 크다'를 all 함수에 넣으면,
이 a 안의 요소 하나하나가 3 보다 크다면 true 가 나올 것이다.

그런데 0, 1, 2, 3 은 다 3 보다 작거나 같은 애들이니까 False 가 나온다.
a<5 조건은 0, 1, 2, 3, 4 가 다 5 보다 작기 때문에 True 가 나온다.

any 는 하나라도 3 보다 크면 True 가 나오고,
5 보다 큰 건 하나도 만족하는 게 없으니까 이 때는 False 가 나온다.

- Numpy는 배열의 크기가 동일 할 때 element간 비교의 결과를 Boolean type으로 리턴

```
a = np.array([1, 5, 3], float)
b = np.array([4, 7, 2], float)
a > b
```

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

```
a == b
```

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

```
(a > b).any()
```

```
True
```

```
(a > b).all()
```

```
False
```

numpy 는 배열을 비교할 때, **True** 와 **False** 의 **Boolean type** 으로 리턴 해준다.

배열 간 연산을 하면

각 요소, 각각의 위치마다 비교를 해준다.

a 랑 b 를 비교하면 a 가 b 보다 큰 지 하나씩 비교할 때

1 은 4 보다 작으니까 처음은 **False** 이다.

5 도 7 보다 작으니까 **False**, 3 은 2 보다 커서 **True** 가 나온다.

부등호는 같다라는 뜻이고 a 와 b 에는 같은 게 하나도 없으니까 다 **False** 이다.

`(a>b).any()` 이런 식으로 쓰면 array 요소 중 하나라도 조건을 만족하면 **True** 가 나온다.

`(a>b).all()`을 쓰면 요소 하나하나가 모두 다 만족해야 하는데

만족 안 하는 게 있기 때문에 **False** 가 나오게 된다.

```
a = np.array( [2, 3, 1], float )
np.logical_and( a > 0 , a < 3 ) #and 조건의 비교
```

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

```
b = np.array( [ False, True, True ], bool )
np.logical_not(b) #not 조건의 비교
```

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

```
c = np.array( [ False, False, False ], bool)
np.logical_or( b, c ) #or 조건의 비교
```

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

logical_and 란 함수는 2 가지 조건을 넣을 수 있다.

0 보다 크거나 3 보다 작다라는 두 가지 조건을 써주고,

2 는 맞으니까 **True** 가 나오고, 3 은 **False** 가 나온다.

logical_not 은 한 마디로 원래 나오지 않는 값을 보여준다.

(**True** 면 **False** 로 그냥 반대로 보여준다는 뜻)

logical_and 가 아니라 **or** 이라면 하나라도 **True** 가 있으면 결과가 **True** 가 나온다.

b 랑 c 가 둘 다 **False** 인 첫 번째는 **False** 로 나왔다.

np.where

- where(조건, True, False)

```
a = np.array( [2, 3, 1], float )
np.where( a > 1 , 0, 3 )
array([0, 0, 3])
```

```
a = np.arange( 3, 10)
np.where( a > 6 ) # True 값의 index 값 반환
(array([4, 5, 6], dtype=int64),)
```

```
a = np.array( [ 2, np.NaN, np.Inf ], float )
np.isnan( a ) # null 인 경우 True
array([False, True, False ])
```

```
np.isfinite( a ) # 한정된 수인 경우 True
array([True, False, False])
```

np.where 은 우리가 생각하는 if 문의 역할을 한다.

조건이 있으면 결과값으로 어떤 걸 리턴 하고 아니면 else 써서 다른 거 리턴 해라는 뜻이다.

np.where(조건쓰고, True 일 때 나올 값, False 일 때 나올 값)을 쓰면 된다.

맨 처음 예시에 [2,3,1]이란 array 가 있는데

where 을 써서 1 보다 크면 0 이 리턴 되고 조건에 맞지 않으면 3 이 리턴 되도록 해주었다.

그러면 2 와 3 은 1 보다 크니까 0, 0 이 나오고 1 은 조건에 맞지 않아서 3 이 나온다.

np.arange(3,10)을 하면 3, 4, 5, 6, 7, 8, 9 가 만들어지는데,

이 array 에서 6 보다 크다는 걸 만족하는 건 7, 8, 9 이다.

그러면 앞에서부터 인덱스 값이 차례고 0, 1, 2, 3, 4, 5, 6 이니까

7, 8, 9 에 해당하는 4, 5, 6 이 인덱스 값으로 나온 것이다.

isnan 은 null 값인 경우에만 **true** 가 나온다.

np.NaN 은 **numpy** 의 null 값을 입력하는 함수이고, null 값이니까 **True** 로 나온다.

np.Inf 는 무한대이다.

np.isinfite()는 한정된 수의 경우 **True** 가 나오고

한정되지 않은 **NaN** 이나 **Inf** 의 경우에는 **False** 가 나온다.

⊛ argmax, argmin

- array내 최대값 또는 최소값의 index 반환

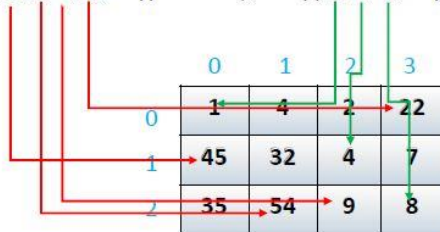
```
a = np.array( [2, 3, 1, 5, 6, 22, 11] )  
np.argmax(a), np.argmin(a)
```

(5, 2)

- axis 기반의 반환

```
a = np.array( [ [1, 4, 2, 22], [45, 32, 4, 7], [34, 54, 9, 8] ] )  
np.argmax(a, axis = 0), np.argmin(a, axis = 1)
```

(array([1, 2, 2, 0], dtype=int64), array([0, 2, 3], dtype=int64))



최대값과 최소값의 인덱스를 반환하는 함수들이다.

맨 처음 예시에서 argmax 값은 가장 큰 값은 22 이니까,

0, 1, 2, 3, 4, 5, 6 중 5 에 있어서 인덱스 5 가 나온다.

가장 min 값은 1 이기 때문에 0, 1, 2, 순으로 또 보면 2 가 나오게 된다.

이것도 axis 기준으로 만들 수 있다.

3 by 4 매트릭스를 axis=0 기준으로 argmax 찾아가라고 하면

row 를 찾아 따라가면서 가장 큰 인덱스 값을 반환한다.

1, 45, 35 중에서 가장 큰 45 의 인덱스 값인 1 이 반환되고,

4, 32, 54 중에서 54 의 인덱스 값인 2 가 나온다.

axis=1 인 경우에는 반대로 컬럼 벡터를 기준으로 한다.

1, 4, 2, 22 중 가장 작은 1 에 대한 인덱스는 0 이 되고,

두 번째 줄인 45, 32, 4, 7 에서 가장 작은 4 에 대한 인덱스는 2 이다.

=> axis 는 많이 쓰이니 꼭 알아둘 것!

⊛ boolean index

- numpy의 배열은 특정 조건에 따른 값을 배열 형태로 추출 가능
- comparison operation 함수들도 모두 사용 가능

```
t_a = np.array( [3, 5, 8, 0, 7, 4], float )  
t_a > 4
```

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

```
t_a[t_a > 4] # 조건이 True 인 index의 요소값만 추출
```

array([5., 8., 7.])

```
t_c = t_a < 4  
t_c
```

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

```
t_a[t_c]
```

array([3., 0.])

Boolean index 는 True 와 False 값이 있으면, 그 중 True 에 해당하는 값을 꺼내준다.

True, False...하나하나가 다 인덱스의 역할을 해주고,

True 에 해당하는 요소 값 즉, value 를 추출해준다.

맨 위의 예시에서

`t_a > 4` 를 하면 4 보다 큰 값에 대해서만 True 가 나오게 된다.

이 결과를 인덱스로 사용한다.

`t_a[t_a>4]`로 넣어주면 index 가 True 인 것들만 뽑겠다는 뜻이다.

🔍 fancy index

- numpy의 array를 index value로 사용하여 값을 추출

```
f_a = np.array([1, 2, 3, 4, 5, 6], float)
f_b = np.array([1, 0, 2, 0, 1, 4], int) # 반드시 integer로 선언
f_a[f_b] # bracket index, b 배열의 값을 index로 하여 a의 값 추출
```

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

```
f_a.take(f_b) # take : bracket index와 같은 효과
```

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

0	1	2	3	4	5
1	2	3	4	5	6

fancy index 라고 해서 아예 array 를 index 로 사용할 수 있다.

index 로 쓰려면 무조건 int 형식으로 써야 한다.

기존의 인덱스를 뽑아주듯 괄호 안에 넣어주면

인덱스 값이 [1,0,2,0,1,4]니까 처음에 1 에 해당하는 2 가 나온다.

그 다음에 인덱스 0 에 해당하는 1 이 나온다.

`f_a.take` 는 위에서 쓰는 거랑 똑같은 말이라 편한 대로 써주면 된다.

```
f_a = np.array([ [1, 3, 7], [3, 4, 5] ], float)
f_b = np.array([1, 0, 1, 0, 1], int)
f_c = np.array([1, 1, 0, 0, 2], int)
f_a[f_b, f_c] # [axis = 0, axis = 1]
```

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

	0	1	2
0	1	3	7
1	3	4	5

매트릭스면 axis 기준으로 row 기준, column 벡터 기준을 다 써줄 수 있다.

맨 처음 값은 `f_b` 도 인덱스 1, `f_c` 도 인덱스 1 이니까 row 1 번 column 1 번의 값인 4 가 나오게 된다.

loadtxt, savetxt

- text type의 데이터를 읽고, 저장

```
a = np.loadtxt( "./price.txt" )
a[5] # python 기본타입 float 형태로 보임
array([[1.00e+00, 2.30e+04, 3.40e+03, 1.96e+04],
       [2.00e+00, 4.20e+04, 6.90e+03, 3.51e+04],
       [3.00e+00, 6.10e+04, 2.70e+03, 5.83e+04],
       [4.00e+00, 1.90e+04, 8.60e+03, 1.04e+04],
       [5.00e+00, 6.90e+04, 3.70e+03, 6.53e+04]])
```

```
a_int = a.astype(int)
a_int[:3]
array([[ 1, 23000, 3400, 19600],
       [ 2, 42000, 6900, 35100],
       [ 3, 61000, 2700, 58300]])
```

데이터를 저장하는 방식을 볼 것인데 어려운 건 없다.

text 파일이면 loadtxt() 로 불러오면 되고,

그런데 결과값을 보면 지금 상태로는 보기 힘든 타입으로 되어있어서 보기가 힘들다...

그 밑에 예제에서 astype()이라는 함수를 사용해서 int로 변환해주었다.

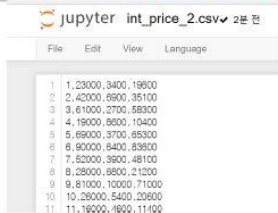
a_int[:3]은 전체 다 안 보고 일부만 살짝 확인하려고 써준 건데,

row 기준으로 :3 이니까 0, 1, 2 인덱스를 뽑아준다. (= 총 3 줄이 출력됨)

```
np.savetxt( "int_price.csv", a_int, delimiter = "," )
```



```
np.savetxt( "int_price_2.csv", a_int, fmt = '%d', delimiter = "," )
```



저장할 때는 text 말고 csv 형태로 저장할 수도 있다.

csv 는 데이터 넣고 뽑으실 때 가장 많이 쓰는 단위이다.

저장하는 각각 데이터들의 나누는 표시, 구분기호가 ","로 되어있다.

그리고 저장할 때도 보기 어려운 지수 형태로 나와있는데,

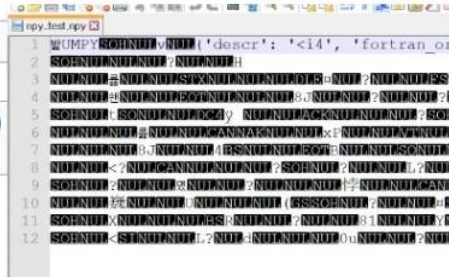
이걸 %d 라는 argument 통해서 간단한 int 형식으로 저장할 수 있다.

- Q: numpy object : npy
- Numpy object (pickle) 형태로 데이터를 저장하고 불러옴
 - Binary 형태로 파일 저장

```
np.save( "np_test.npy", arr = a_int )
```

```
np_array = np.load( file = "np_test.npy" )
np_array[:5]
```

```
array([[ 1, 23000, 3400, 19600],
       [ 2, 42000, 6900, 35100],
       [ 3, 61000, 2700, 58300],
       [ 4, 19000, 8600, 10400],
       [ 5, 69000, 3700, 65300]])
```



위와 같이 쓰면 **바이너리 형태, pickle 형태로 저장된다.**

이렇게 저장해도 되는데 메모장이나 여러 가지 다른 걸 사용해서 열어보면 이상한 글자가 뜰 것이다.
(일종의 암호화)

* 실습코드에 맨 윗줄에 새로 import 해줘야하는데 생략되어 있어서 코드가 안 돌아간다.

import numpy as np 를 맨 윗줄에 직접 써주고 돌릴 것!

Q: logical_and, logical_or 함수들은 반드시 인수가 2 개여야 하나요?

2 개 이상을 비교하고 싶으면 못 하나요? -> 반드시 인수가 2 개이도록 만들어진 함수입니다. 1 개나 3 개는 실습해보면 오류가 뜹니다. 2 개씩 비교하도록 만들어진 것이기 때문에 다른 것과 비교를 하려면 logical_and(logical_and(비교 1),비교 2) 이런 식으로 여러 번 써줘야 합니다.

Q: Nan 과 Null 은 다른 것 아닌가요?

엄밀히 구분하면 둘은 다른 것입니다.

Nan 은 숫자와 연산이 가능하고

Null 은 숫자와 연산이 가능하지 않습니다.

Nan 은 무언가가 있지만 Null 은 아예 데이터가 없는 것입니다.

Q: npy 로 저장하면 장점이 무엇인가요?

암호화가 되니까 개인정보가 중요한데,

그런 걸 자신만 보게 만들 수 있다는 점이 좋습니다.

또 파일의 용량도 줄일 수 있습니다.

npy 1.68KB, csv 2.04KB, txt 는 2.04KB 로 가장 작습니다.

* 이미 **run 된 내용을 지우려면 cell all output clear** 를 선택해준다.

current output clear 나 kernel 에서 restart & clear 하면 깔끔하게 사라진다.

그런데 이렇게 커널의 output 을 다 날리면 저장되어 있던 게 다 날아가기 때문에 처음부터 import 를 다 새로 해줘야 한다.

* 지금 다룬 것이 Numpy 의 전부가 아니라,

numpy 홈페이지에 들어가보면 이것 말고도 많은 것들이 있다.

tutorial 등 이런 것들이 있구나 하면서 보면 된다.

Pandas 라이브러리

pandas 는 파이썬에서 엑셀처럼 데이터 처리하도록 만든 것이다.
numpy 에서 사용하는 함수는 기본으로 들어가 있어서 다 사용할 수 있다.

● pandas 구성

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

serise

DataFrame 중 하나의 Column에 해당
하는 데이터의 모음 Object

DataFrame

Data Table 전체를 포함하는 Object

pandas 는 serise 와 data frame 으로만 이루어져있다.

이 전체를 numpy array 로 보면 matrix 형태일 것이다.

그 형식으로 이루어진 테이블 전체를 데이터 프레임이라고 한다.

R 에서 쓰는 Dataframe 을 파이썬에서 따온 것이다.

serise 는 그 데이터 프레임의 칼럼 하나를 부른다.

● Series

- column vector를 표현하는 object
- index가 포함된 numpy

```
from pandas import Series, DataFrame
import pandas as pd
ex_obj = Series()
```

shift + tab

Init signature: Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)
Docstring:
One-dimensional ndarray with axis labels (including time series).

Dataframe 의 series 를 불러올 때는

from pandas import Series 랑 DataFrame 으로 불러온다.

● Series

- column vector를 표현하는 object
- numpy.ndarray 의 subclass
- 어떤 type의 data라도 다룰 수 있다.
- index labels은 필요하나 순서대로 정렬 될 필요는 없다
- duplicates 가능하다 (성능 안좋아짐)

```
list_data = [ 3, 2, 4, 1, 5]
ex_obj = Series(data = list_data)
ex_obj
```

```
0 3
1 2
2 4
3 1
4 5
dtype: int64
```

index	values
0	3
1	2
2	4
3	1
4	5

type : int 64

series 는 numpy 의 array 랑 거의 같다고 볼 수 있는데 이걸 대신 index 가 붙어있다.

그래서 이 인덱스를 통해서 우리가 원하는 value 를 가져올 수 있다.

지금 위에 예시에서는

이 인덱스 값들이 unique 하게 되어있는데

꼭 unique 하게 안하고 0,1,1,... 이런 식으로 중복으로 쓸 수도 있다.

그러면 인덱스가 2 개가 다 1 로 지정되어있을 때

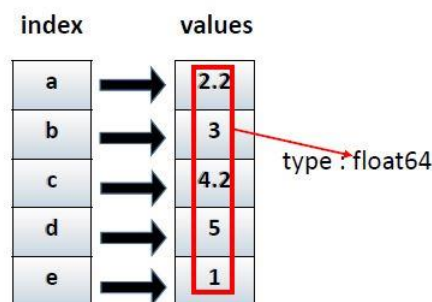
1 에 대한 값을 불러오라고 하면 2 랑 4 가 동시에 둘 다 불러와지게 된다.

(그래서 우리가 인덱스 만들 때 보통 중복되도록 만들지는 않는다.

보통은 다 유니크하게 만든다.)

```
list_data = [ 2.2, 3, 4.2, 5, 1]
list_name = [ "a", "b", "c", "d", "e"]
exp_obj = Series( data = list_data, index = list_name )
exp_obj
```

```
a  2.2
b  3.0
c  4.2
d  5.0
e  1.0
dtype: float64
```



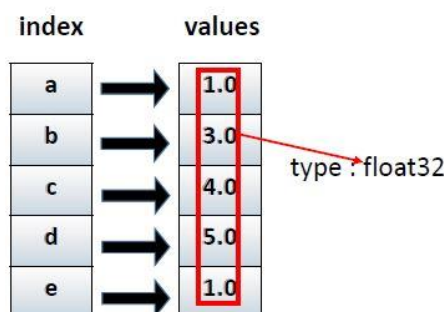
만들 때는 series() 안에 리스트 데이터와 인덱스를 넣어준다.

그러면 이렇게 데이터와 인덱스가 합쳐진 하나의 시리즈를 구성할 수 있다.

인덱스는 꼭 숫자일 필요는 없다. 다른 형식도 가능하다.

```
dict_data = { "a" : 1, "b" : 3, "c" : 4, "d" : 5, "e" : 1 }
exp_obj = Series( dict_data , dtype = np.float32, name = "dict_series" )
exp_obj
```

```
a  1.0
b  3.0
c  4.0
d  5.0
e  1.0
Name: dict_series, dtype: float32
```



딕트 형식으로 넣어주는 것도 가능한데,

key 는 index 로 가고 value 는 value 로 들어가게 된다.

옵션에 데이터 타입과 이름을 설정해줄 수 있는데, 크게 중요하진 않다.

```
exp_obj["a"] # data의 index에 접근
```

```
1.0
```

```
exp_obj["a"] = 1.4 # data index를 통해 값 할당  
exp_obj
```

```
a 1.4  
b 3.0  
c 4.0  
d 5.0  
e 1.0  
Name: dict_series, dtype: float32
```

index	values
a	1.4
b	3.0
c	4.0
d	5.0
e	1.0

인덱스도 마찬가지로 index 통해서 접근할 수 있습니다.

object 에 "a"라는 인덱스의 값을 보고 싶으면 exp_obj["a"] 이렇게 쓰면 되고,
이전에 있던 값을 할당도 할 수 있다.

exp_obj["a"]=1.4 를 하면 1.0 으로 원래 값이 들어있었는데, 1.4 로 바뀐다.

```
exp_obj.values # 값에 해당하는 리스트 반환. numpy 형태
```

```
array([1.4, 3., 4., 5., 1.], dtype=float16)
```

```
exp_obj.index # index 리스트 형태로 반환
```

```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

```
exp_obj.name = "example"
```

```
exp_obj.index.name = "index_a"
```

```
exp_obj
```

```
index_a  
a 1.400391  
b 3.000000  
c 4.000000  
d 5.000000  
e 1.000000  
Name: example, dtype: float16
```

데이터 타입 중 dict 는 키와 value 를 원래 따로 뽑을 수 있었던 것처럼

.values 를 하면 value 만 뽑아낼 수 있고

.index 를 치면 전체 인덱스만 뽑아낼 수 있다. (+ 인덱스 타입도 같이 알려준다!)

index.name 도 지어줄 수 있고, 이름도 할당해줄 수 있다.

```
dict_data = {"a": 1, "b": 3, "c": 4, "d": 5, "e": 1}
```

```
indexes = ["a", "b", "c", "D", "e", "f", "g"]
```

```
series_obj_2 = Series(dict_data, index = indexes)
```

```
series_obj_2
```

```
a 1.0  
b 3.0  
c 4.0  
D NaN  
e 1.0  
f NaN  
g NaN  
dtype: float64
```

index 값 기준으로 series 생성

value 를 dict 로 놓고 index 를 리스트 형태로 넣을 때 ,
 같은 인덱스끼리는 치환이 되는데
 인덱스가 같지 않거나 없는 값이 나타나면 NaN 값이 들어오게 된다.

dict_data 에는 a, b, c, d 인데,
 indexes 에 있는 건 a, b, c, D 라서 같이 없으니까 D 에 NaN 이 들어가게 된다.

exp_obj[exp_obj > 1]

index_a
 a 1.400391
 b 3.000000
 c 4.000000
 d 5.000000
 Name: example, dtype: float16

exp_obj * 3

index_a
 a 4.203125
 b 9.000000
 c 12.000000
 d 15.000000
 e 3.000000
 Name: example, dtype: float16

np.exp(exp_obj)

index_a
 a 4.058594
 b 20.078125
 c 54.593750
 d 148.375000
 e 2.718750
 Name: example, dtype: float16

"x" in ex_obj

False

exp_obj.to_dict()

{'a': 1.400390625, 'b': 3.0, 'c': 4.0,
 'd': 5.0, 'e': 1.0}

numpy 에서 할 수 있는 건 다 할 수 있다.

exp_obj>1 만 하면 True, False 가 나오는데
 이걸 series 안에 인덱스로 넣어주면 True 값에 해당하는 것들만 나온다.

곱하기 연산도 할 수 있고 지수함수도 써서 나오게 할 수 있다.

"x" in ex_obj 는 ex_obj 안에 x 가 있으면 true 가 나오게 된다.

to_dict()는 시리즈가 dict 타입으로 변하는 것이고,
 보통 to 라고 하면 그 형식으로 변환해라 라는 뜻이다.

import pandas as pd

data_url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data'
 df_data = pd.read_csv(data_url, sep = '\s+', header = None)
 # csv 타입 데이터 로드, separate 는 빈공간으로 저장, Column 은 없음

df_data.head()

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

실습 시 데이터 로딩은 이렇게 하는 것도 가능하다.

이 사이트에 있는 데이터를 불러오는 것이다.

<http://archive.ics.uci.edu/ml/index.php>

UCI Machine Learning Repository

Welcome to the UC Irvine Machine Learning Repository! We currently maintain 442 data sets as a service to the machine learning community. You may view all data sets through our searchable interface. Our old web site is still available, for those who prefer the old format. For a general overview archive.ics.uci.edu

우리가 쓰는 iris 데이터도 있고, 학습 모델,

여러 가지 다른 카테고리 기준으로 데이터 나눠져 있고 쉽게 접할 수 없는 데이터도 있다

ex)cancer. 이것 통해서 데이터 분석 연습하기 좋다.

DataFrame

- 여러 개의 series의 합
- 각각의 컬럼은 다른 type으로 구성될 수 있다.
- row와 column index가 존재
- 컬럼 삽입이나 삭제를 통해 사이즈를 유연하게 조절 할 수 있음

columns		foo	bar	baz	qux
index	A	0	x	2.7	True
	B	4	y	6	True
	C	8	z	10	False
	D	-12	w	NA	False
	E	16	a	18	False

이제 **pandas**의 **Dataframe**을 보면 위와 같다.

판다스의 데이터 프레임은 시리즈들을 모아서 만든 테이블이다.

각각의 컬럼은 원하는 타입으로 구성할 수 있다.

숫자 타입 스트링 타입 소수점 타입 불린 타입 ...등등...

```
r_data = {
  "city": [
    "Seoul", "Seoul", "Jeonju", "Busan", "Incheon", "Goyang"
  ],
  "name": [
    "이정수", "문철수", "오영희", "채호야", "문동석", "고나리"
  ],
  "job": [
    '학생', '정치인', '경찰', '소방관', '영업', 'it'
  ],
  "age": [
    22, 53, 26, 33, 43, 34
  ]
}
```

먼저 데이터 프레임을 만들어 본다.

column 에도 인덱스가 있는데, 이 인덱스도 형식을 마음대로 만들 수 있다.

지금 데이터 프레임은 dict 타입으로 만들었고 키는 string, value 는 리스트 형식으로 만들었다.

이걸 만들면 이런 형식의 데이터 프레임이 만들어진다.

DataFrame

```
df = pd.DataFrame( data, columns = [ 'name', 'job', 'city', 'age' ] )  
df
```

	name	job	city	age
0	이정수	학생	Seoul	22
1	문철수	정치인	Seoul	53
2	오영희	경찰	Jeonju	26
3	채호야	소방관	Busan	33
4	문동석	영업	Incheon	43
5	고나리	it	Goyang	34

```
DataFrame(r_data, columns = [ "age", "city" ] )
```

	age	city	column 선택
0	22	Seoul	
1	53	Seoul	
2	26	Jeonju	
3	33	Busan	
4	43	Incheon	
5	34	Goyang	

```
DataFrame(r_data, columns = [ 'name', 'job', 'comp' ] )
```

	name	job	comp
0	이정수	학생	NaN
1	문철수	정치인	NaN
2	오영희	경찰	NaN
3	채호야	소방관	NaN
4	문동석	영업	NaN
5	고나리	it	NaN

새로운 column 추가

data 를 넣은 후에 컬럼명을 적으면

내가 원하는 column 만 뽑아서 데이터 프레임 만들 수 있다.

원하는 컬럼이 2 개뿐이라면 2 개만 뽑아서 만들면 된다.

(아래 두 개 예시는 df= 이부분이 없으니까 어떤 변수에 할당한 건 아니고 그냥 복사본만 보여준 것이다.)

그리고 새로운 컬럼도 추가할 수가 있는데,

새 컬럼 이름을 넣어주면 그 값은 NaN 이 나오게 된다.

```
df = DataFrame(r_data, columns = [ 'name', 'job', 'city', 'age', 'comp' ])
df.name # 속성이름으로 추출
```

```
0 이정수
1 문철수
2 오영희
3 채호야
4 문동석
5 고나리
Name: name, dtype: object
```

column 선택 하는건
DataFrame 에서 series data 추출 하는 것

```
df["name"] # dict type 처럼 변수명과 컬럼명으로 추출
```

```
0 이정수
1 문철수
2 오영희
3 채호야
4 문동석
5 고나리
Name: name, dtype: object
```

속성이름으로 데이터 값을 추출할 수도 있고,
컬럼 인덱스를 통해 추출할 수도 있다. 결과값으로는 시리즈가 나오게 된다.

- loc, iloc

```
ex_ind = pd.Series(np.nan, index = [ 12, 13, 14, 3, 2, 1 ])
ex_ind
```

```
12 NaN
13 NaN
14 NaN
3 NaN
2 NaN
1 NaN
dtype: float64
```

```
ex_ind.loc[:2] # index location 기준, 인덱스 name
```

```
12 NaN
13 NaN
14 NaN
3 NaN
2 NaN
dtype: float64
```

```
ex_ind.iloc[:2] # index position 기준
```

```
12 NaN
13 NaN
dtype: float64
```

loc 는 **index location** 인데 location 이라고 생각하지 말고 인덱스 네임이라고 생각하면 된다.
loc[3]이라고 하면 3 에있는 value 를 가져온다.

iloc 는 **name** 이 아니라 **number** 기준이다.

데이터는 NaN 값으로 다 넣고 인덱스를 12,13,14 형식으로 넣고
여기서 loc[:2]를 쓰면 이 이름을 있는데 까지 뽑겠다고 하는 것이다.
그래서 2 가 나올 때까지 12, 13, 14, 3, 2 데이터가 다 나왔다.

그러나 **iloc** 를 쓰면 넘버 위치 기준을 기준으로 한다.

(앞에서부터 0, 1, 2, 이런 순으로 나오는 것)

그래서 두 번째까지 나오게 되어서 12, 13 까지만 뽑히는 것이다.

- column에 새로운 데이터 할당

```
df.comp = df.age > 30
df
```

	name	job	city	age	comp
0	이정수	학생	Seoul	22	False
1	문철수	정치인	Seoul	53	True
2	오영희	경찰	Jeonju	26	False
3	채호야	소방관	Busan	33	True
4	문동석	영업	Incheon	43	True
5	고나리	it	Goyang	34	True

Boolean index 형태

비교한 것을 새로운 칼럼에 넣어줬는데 30 보다 작은 값들은 다 False 값으로 만들어졌다.
이 True, False 들이 comp 라는 컬럼에 할당되었다.

```
df.T # transpose, numpy 기능 똑같이 쓸 수 있음
```

	0	1	2	3	4	5
name	이정수	문철수	오영희	채호야	문동석	고나리
job	학생	정치인	경찰	소방관	영업	it
city	Seoul	Seoul	Jeonju	Busan	Incheon	Goyang
age	22	53	26	33	43	34
comp	False	True	False	True	True	True

그리고 T 는 그냥 뭐 numpy 기능을 쓸 수 있으니
transpose (전치행렬)를 써서 이렇게 바꿔볼 수 있다.

```
df.values # 값 출력
```

```
array([[ '이정수', '학생', 'Seoul', 22, False],
       [ '문철수', '정치인', 'Seoul', 53, True],
       [ '오영희', '경찰', 'Jeonju', 26, False],
       [ '채호야', '소방관', 'Busan', 33, True],
       [ '문동석', '영업', 'Incheon', 43, True],
       [ '고나리', 'it', 'Goyang', 34, True]], dtype=object)
```

```
df.to_csv() # csv 형태 출력
```

```
'name,job,city,age,comp\n0,이정수,학생,Seoul,22,False\n1,문철수,정치인,Seoul,53,True\n2,오영희,경찰,Jeonju,26,False\n3,채호야,소방관,Busan,33,True\n4,문동석,영업,Incheon,43,True\n5,고나리,it,Goyang,34,True\n'
```

values 를 쓰면 Dataframe 의 value 들이 다 나오게 된다.

여기도 values 대신 index 를 썼다면 인덱스 값 들이 나왔을 것이다.

to_csv 는 저장하는게 아니라 형태를 변환해준다.

csv 는 구분자가 있는 txt 파일인데,

지금 함수 안에 아무것도 안 써줘서 기본 디폴트 값인 ","가 구분자로 들어갔다.

중간에 들어가 있는 \n0 은 엔터라고 보면 된다.

- column 삭제

```
del df["comp"]
df
```

	name	job	city	age
0	이정수	학생	Seoul	22
1	문철수	정치인	Seoul	53
2	오영희	경찰	Jeonju	26
3	채호야	소방관	Busan	33
4	문동석	영업	Incheon	43
5	고나리	it	Goyang	34

column 은 이런 식으로 삭제도 가능하다.

- dict 안에 dict

```
ex_dict = {'seoul': {1: 33, 2: 22}, 'busan': {3: 35, 4: 37}}
ex_dict
{'seoul': {1: 33, 2: 22}, 'busan': {3: 35, 4: 37}}
```

```
DataFrame(ex_dict)
```

	seoul	busan
1	33.0	NaN
2	22.0	NaN
3	NaN	35.0
4	NaN	37.0

만약에 dict 안에 이런 식으로 중복된 딕트가 또 들어가 있는 경우에는
가장 앞에 있는 키는 컬럼의 이름이 된다.

그 안에 포함된 딕트의 앞에 있는 키는 로우 인덱스 이름이 된다.

그래서 seoul 에 있는 1, 2 와 busan 에 있는 3, 4 로 왼쪽에 인덱스가 만들어 졌는데,
1,2 는 서울에만 값이 있고 부산에는 없으니까 NaN 값으로 데이터가 뜨는 걸 볼 수 있다.

🔗 column names와 index number를 이용한 data selection

- 엑셀 데이터이용, xlrd 모듈 설치

```
!conda install --y xlrd # promp 상의 conda 명령어 jupyter에서 사용
```

```
df = pd.read_excel("./excel_sample/sampleddata.xlsx")
df.head()
```

	OrderDate	Region	Rep	Item	Units	Unit Cost	Total
0	2016-01-06	East	Jones	Pencil	95	1.99	189.05
1	2016-01-23	Central	Kivell	Binder	50	19.99	999.50
2	2016-02-09	Central	Jardine	Pencil	36	4.99	179.64
3	2016-02-26	Central	Gill	Pen	27	19.99	539.73
4	2016-03-15	West	Sorvino	Pencil	56	2.99	167.44

가상 데이터인데, 이런 엑셀 파일을 불러오려면 xlrd 라는 모듈을 설치해야 한다.

!는 주피터 노트북 상에서 shell 에서 쓸 수 있는 명령어들 쓰게 만들어주는 방법이다.

원래 기본 프롬프트(shell)에서

conda 나 pip 같은 걸 쓸 수 있었는데 주피터 노트북에서도 쓸 수 있다.

```
df["Region"].head(3) # 한개 column 선택
```

```
0    East
1  Central
2  Central
Name: Region, dtype: object
```

```
df[["Region", "Item", "Unit Cost"]].head(3) #1개 이상의 column 선택
```

	Region	Item	Unit Cost
0	East	Pencil	1.99
1	Central	Binder	19.99
2	Central	Pencil	4.99

인덱스 이름을 통해 가져온다.

df["Region"]을 쓰면 그 컬럼에 해당하는 시리즈 값 들이 다 나오게 된다.

그 뒤에 붙은 .head()는 앞에서 몇 번째 값까지 가져오냐는 것인데,

지금 head(3)이라고 썼으니까 0, 1, 2 번째까지 가져왔다.

그리고 하나 이상의 컬럼을 선택할 때는 대괄호 2 개 이상 쓰는 걸 유의해야 한다!

df[["Region","Item",...]] 이렇게 [[]]2 개씩 쓰는 거 유의!

```
df[:3] # column 이름 없이 사용하는 index number는 row 기준 표시
```

	OrderDate	Region	Rep	Item	Units	Unit Cost	Total
0	2016-01-06	East	Jones	Pencil	95	1.99	189.05
1	2016-01-23	Central	Kivell	Binder	50	19.99	999.50
2	2016-02-09	Central	Jardine	Pencil	36	4.99	179.64

```
df["Region"][:3]
```

```
# column 이름과 함께 row index 사용시, 해당 column만 리턴
```

```
0    East
1  Central
2  Central
Name: Region, dtype: object
```

컬럼 이름 없이 그냥 index number 로 가져올 때는 꼭 가져오시면 된다.

그리고 아래 예제처럼 내가 원하는 column 에 원하는 row 만큼 가져올 수도 있다.

df["Region"][:3]이면 Region 컬럼에 꼭 있는 데이터 중에서 0, 1, 2 까지만 가져오는 것이다.

Index 변경

```
df.index = df["Units"]  
df.head(3)
```

	OrderDate	Region	Rep	Item	Units	Unit Cost	Total
Units							
95	2016-01-06	East	Jones	Pencil	95	1.99	189.05
50	2016-01-23	Central	Kivell	Binder	50	19.99	999.50
36	2016-02-09	Central	Jardine	Pencil	36	4.99	179.64

```
del df['Units']  
df.head(3)
```

	OrderDate	Region	Rep	Item	Unit Cost	Total
Units						
95	2016-01-06	East	Jones	Pencil	1.99	189.05
50	2016-01-23	Central	Kivell	Binder	19.99	999.50
36	2016-02-09	Central	Jardine	Pencil	4.99	179.64

인덱스를 원래 있는 데이터로 변경할 수 있다.

밑에 예제는 인덱스로 이미 Units 데이터가 들어갔으니까 중복된 값을 지워준 것이다.

index 재설정

```
df.index = list(range(0,43))  
df.head()
```

	OrderDate	Region	Rep	Item	Unit Cost	Total
0	2016-01-06	East	Jones	Pencil	1.99	189.05
1	2016-01-23	Central	Kivell	Binder	19.99	999.50
2	2016-02-09	Central	Jardine	Pencil	4.99	179.64
3	2016-02-26	Central	Gill	Pen	19.99	539.73
4	2016-03-15	West	Sorvino	Pencil	2.99	167.44

인덱스가 순서대로 안 되어있으면 range 써서
위에서부터 숫자가 순서대로 들어가도록 만들어줄 수 있다.
(row 개수가 맞지 않으면 에러가 뜨니까 잘 계산해서 넣을 것)

data drop

```
df.drop(1).head(3) # index name로 drop
```

	OrderDate	Region	Rep	Item	Unit Cost	Total
0	2016-01-06	East	Jones	Pencil	1.99	189.05
2	2016-02-09	Central	Jardine	Pencil	4.99	179.64
3	2016-02-26	Central	Gill	Pen	19.99	539.73

```
df.drop([0,1,2,3]).head(3) # 한개 이상의 index name로 drop
```

	OrderDate	Region	Rep	Item	Unit Cost	Total
4	2016-03-15	West	Sorvino	Pencil	2.99	167.44
5	2016-04-01	East	Jones	Binder	4.99	299.40
6	2016-04-18	Central	Andrews	Pencil	1.99	149.25

인덱스 네임 기준으로 데이터를 drop 할 수 있다.

여러 개 하려면 대괄호로 묶어서 넣어주면 된다.

data drop

- axis 지정한 축을 기준으로 drop

```
df.drop("Rep", axis = 1).head(3) # df.drop(["Rep", "Item"], axis = 1)
```

	OrderDate	Region	Item	Unit Cost	Total
0	2016-01-06	East	Pencil	1.99	189.05
1	2016-01-23	Central	Binder	19.99	999.50
2	2016-02-09	Central	Pencil	4.99	179.64

- 실제 drop 되는 것은 아니고, 복사본을 만들어서 보여주는 것
- drop된 dataframe을 생성하고 싶으면 다시 할당 해야함
- 즉시 반영하고 싶으면 inplace = True 사용

```
df.head(3)
```

	OrderDate	Region	Rep	Item	Unit Cost	Total
0	2016-01-06	East	Jones	Pencil	1.99	189.05
1	2016-01-23	Central	Kivell	Binder	19.99	999.50
2	2016-02-09	Central	Jardine	Pencil	4.99	179.64

axis=1 이라고 표시하면 원래 우리가 row 쪽으로 가면서 처리를 하는 건데,
여기서는 해당 컬럼에 대한 series 가 다 지워지게 된다.

다른 함수에서 옵션으로 사용하는 axis 와 달라서 헷갈리게 된다.

argument 중에서 inplace 라는 옵션이 있는데,

변수에 새로 할당하거나 새로운 복사본 만들고 거기에 넣고 싶어하면 True 로 해주면 된다.

= 지금 바로 데이터에 저장해서 이 값을 가지고 싶을 때!

```
s1 = Series( range(1, 6), index = list("abcde"))  
s1
```

```
a 1  
b 2  
c 3  
d 4  
e 5  
dtype: int64
```

```
s2 = Series( range(6, 11), index = list("cdefg"))  
s2
```

```
c 6  
d 7  
e 8  
f 9  
g 10  
dtype: int64
```

판다스도 numpy 와 마찬가지로 operation 을 할 수가 있다.

Series operation

```
s1.add(s2)
```

```
a NaN
b NaN
c 9.0
d 11.0
e 13.0
f NaN
g NaN
dtype: float64
```

```
s1 + s2
```

```
a NaN
b NaN
c 9.0
d 11.0
e 13.0
f NaN
g NaN
dtype: float64
```

```
s1.add(s2, fill_value = 0)
```

```
a 1.0
b 2.0
c 9.0
d 11.0
e 13.0
f 9.0
g 10.0
dtype: float64
```

a, b, f, g 는 인덱스가 없어서 둘을 연산했을 때 NaN 값이 뜨게 된다.
같은 인덱스를 가지고 있는 것들끼리는 다 연산이 되었다.
대신에 옵션에 fill_value = 0 을 넣게 되면
인덱스가 없는 시리즈에는 value 에 0 을 할당해준다.

DataFrame operation

- DataFrame은 column과 index 모두 고려
- add operation을 쓰면 NaN값 변환시킬 수 있음
- Operation types : add, sub, div, mul

```
df1 = DataFrame(np.arange(4).reshape(2, 2), columns = list("ab"))
df1
```

```
   a  b
0  0  1
1  2  3
```

```
df2 = DataFrame(np.arange(9).reshape(3, 3), columns = list("abc"))
df2
```

```
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8
```

2x2 와 3 곱하기 3 의 데이터 프레임이 있다.


```
df1 + df2
```

	a	b	c
0	0.0	2.0	NaN
1	5.0	7.0	NaN
2	NaN	NaN	NaN

```
df1.add(df2, fill_value = 0)
```

	a	b	c
0	0.0	2.0	2.0
1	5.0	7.0	5.0
2	6.0	7.0	8.0

df1 에는 컬럼 인덱스 중 c 가 없고 row 인덱스 중 2 가 없다.
그러면 연산 결과 없는 애들은 NaN 이 나오게 된다.

Series + DataFrame

```
df = DataFrame(np.arange(9).reshape(3, 3),  
               columns = list("abc"))  
df
```

	a	b	c
0	0	1	2
1	3	4	5
2	6	7	8



	a	b	c
0	10	11	12
1	10	11	12
2	10	11	12

column 기준으로
broadcasting 발생

```
s = Series(np.arange(10,13),  
           index = list("abc"))  
s
```

a 10
b 11
c 12
dtype: int32

```
df + s
```

	a	b	c
0	10	12	14
1	13	15	17
2	16	18	20

numpy 에서 브로드캐스팅을 한 것처럼 **pandas 도 브로드캐스팅이 있다.**

이건 컬럼 인덱스 기준으로 해주는데,

+를 해보면 데이터 프레임 안에 a, b, c 와 0, 1, 2 인덱스가 있고

series 에는 a, b, c 인덱스가 있다. 원래 a, b, c 는 row 인덱스인데, 컬럼 인덱스로 변경이 된다.

```
df = DataFrame(np.arange(9).reshape(3, 3),  
               columns = list("abc"))  
df
```

	a	b	c
0	0	1	2
1	3	4	5
2	6	7	8



	0	1	2
0	10	11	12
1	10	11	12
2	10	11	12

column 기준으로
broadcasting 발생한다. 서로
같은 index가 존재하지
않으므로 NaN이 리턴

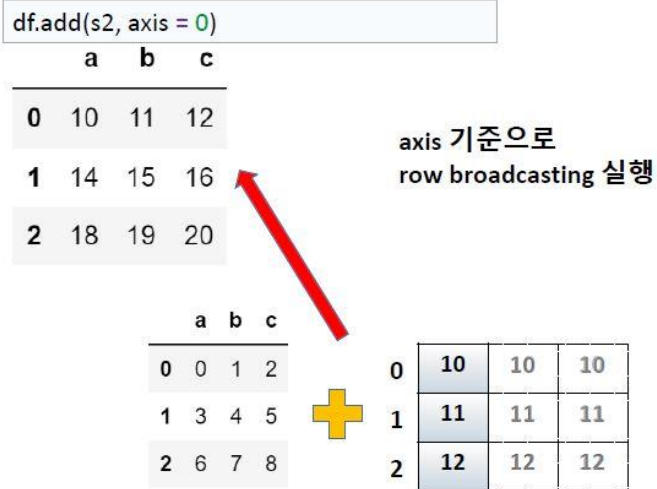
```
s2 = Series(np.arange(10, 13))  
s2
```

0 10
1 11
2 12
dtype: int32

```
df + s2
```

	a	b	c	0	1	2
0	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN

만약에 인덱스를 a, b, c 가 아닌 0,1,2 이런식으로 만들면
컬럼 인덱스가 같지 않기 때문에 전부 다 NaN 값이 나오게 된다.



column 말고 row 쪽으로 하고 싶으면 axis=0 옵션을 쓰면 된다.

Lambda

- 한 줄로 함수를 표현하는 익명 함수 기법
- Lisp 언어에서 시작된 기법으로 오늘날 현대언어에 많이 사용

lambda argument : expression



왼쪽은 파이썬에서 함수를 쓸 때 사용하는 형식이다.

오른쪽에 있는 lambda 는 복잡한 함수를 한 줄로 표현하는 방법이다.

x+5 라는 람다식을 만들고, x 에 3 을 넣으면 3+5 가 되어서 8 이 나온다.

```
f = lambda x: x / 3 # 하나의 인수만 처리
f(6)
2.0
```

```
f = lambda x: x ** 2 # 하나의 인수만 처리
f(4)
16
```

```
(lambda x: x + 4)(2) # 이름 할당하지 않는 lambda
6
```

나누기도 할 수 있고 제곱도 할 수 있고 할당 안하고 계산만 할 수도 있다.

map(function, sequence)

```
seq = [1, 2, 3, 4, 5]
f = lambda x: x * 3
list(map(f, seq))
```

```
[3, 6, 9, 12, 15]
```

```
seq = [3, 2, 6, 4, 5]
f = lambda x: x - 3
list(map(f, seq))
```

```
[0, -1, 3, 1, 2]
```

for 문을 쓰지 않고 map 함수를 써서 각 요소마다 람다에서 정한 연산을 시킬 수 있다.
map(앞에서 function 쓰고, 뒤에 sequence 쓰면) 각 시퀀스마다 연산이 된다.

예를 들어 [1,2,3,4,5]란 시퀀스가 있고
람다에 $x * 3$ 을 만들어서 맵을 쓰게 되면
1 곱하기 3, 2 곱하기 3, 3 곱하기 3.... 이렇게 연산이 되어 나온다.

그런데 지금 표현처럼 list(map(f,seq))이렇게 쓰지 않고
map(f,seq)까지만 쓰면 메모리 주소만 보여주기 때문에
실제 값을 보고 싶다면 list(map()) 혹은 tuple(map()) 이런 식으로 써서 봐야 한다.

* 지금 map 함수는 나중에 series 에 적용할 것이다.

```
f = lambda x, y: x + y
list(map(f, seq, seq)) # 두개 이상의 인수가 있을 때는
# 두개의 sequence 형을 써야함
```

```
[2, 4, 6, 8, 10]
```

```
list(map(lambda x: x * x, seq)) # 익명함수 그대로 사용
# python3 에서 list 함수를 쓰지 않으면 memory 주소가 나옴
```

```
[1, 4, 9, 16, 25]
```

두 개의 시퀀스에도 쓸 수 있다.

[1,2,3,4,5]와 [1,2,3,4,5]에 각각 더하면 [2,4,6,8,10]이 나올 것이다.

그 밑에 곱해주는 함수도 역시 시퀀스에 쓸 수 있다. => **시리즈에도 쓸 수 있다!**

map for series

- Pandas의 series type의 데이터에도 map 함수 사용가능
- function 대신 dict, sequence형 자료등으로 대체 가능

```
s = Series(np.arange(10))
s.head(5)
```

```
0  0
1  1
2  2
3  3
4  4
dtype: int32
```

```
s.map(lambda x: x + 3).head(5)
```

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

객체에다가 맵을 적용해서 쓰게 되면 각 시리즈에 연산을 해준다.

```
dic = {1: 'apple', 2: 'banana', 3: 'carrot'}
```

```
s.map(dic).head(5) # dict type으로 데이터 교체, 값이 없으면 NaN
```

```
0  NaN
1  apple
2  banana
3  carrot
4  NaN
dtype: object
```

```
s2 = Series(np.arange(100, 500, 100))
```

```
s.map(s2).head(5) # 같은 위치의 데이터를 s2로 전환
```

```
0  100.0
1  200.0
2  300.0
3  400.0
4  NaN
dtype: float64
```

map을 dict 타입에도 적용을 할 수가 있다.

원래 0, 1, 2, 3, 4 이고 dict 안에 1, 2, 3 이 들어있는데,

dict를 넣으면 키 값이 인덱스로 들어가기 때문에

0과 4는 NaN으로 나오고

나머지는 dict에 있는 값으로 변형이 된다.

다음 예제에서 시리즈 형태로 [100,200,300,400]을 쓰면 인덱스는 0,1,2,3 일 것이다.

각 0, 1, 2, 3에 대해 이 값으로 치환해주는 것이다.

인덱스가 같지 않은 마지막 값은 NaN이 나오게 된다.

(값이 100~400까지 4개밖에 없으니까 5번째 값인 4는 안 채워져서 NaN이다.)

```
df.sex.unique()
```

```
array(['male', 'female'], dtype = object)
```

```
df["sex_code"] = df.sex.map({"male": 0, "female": 1})  
# 성별 string type -> int type  
df.head()
```

	city	age	sex	job	marriage	height	sex_code
0	seoul	23	male	student	n	67.43	0
1	busan	42	male	salesman	y	76.21	0
2	jeonju	52	female	ceo	y	54.33	1
3	incheon	13	male	student	n	35.22	0
4	daejeon	43	female	professor	n	48.99	1

```
df.sex_code.unique()
```

```
array([0, 1], dtype=int64)
```

map 을 통해서 인덱스 value 도 변경할 수 있다.

```
df.marriage.replace({"y": 1, "n": 0}).head()
```

```
0  0  
1  1  
2  1  
3  0  
4  0
```

```
Name: marriage, dtype: int64
```

```
df.marriage.replace(["y", "n"], [1, 0]).head()
```

```
0  0  
1  1  
2  1  
3  0  
4  0
```

```
Name: marriage, dtype: int64
```

replace function 은 맵이랑 비슷한데 데이터 바로 변환하는 기능으로만 사용할 수 있다.

replace()를 이용해서 y 와 n 의 인덱스에 대해서 1 과 0 으로 변경하라는 말이다.

같은 결과를 내는데 들어갈 때 딕트 형식이라는 것과 리스트 형식이라는 것의 차이밖에 없다.

```
df_2 = df[["age", "height"]]  
df_2.head()
```

	age	height
0	23	67.43
1	42	76.21
2	52	54.33
3	13	35.22
4	43	48.99

```
f = lambda x: x.max() - x.min()
df_2.apply(f) # 각 column 별로 결과값 반환
```

```
age      39.00
height   60.44
dtype: float64
```

- 내장 연산 함수를 사용할 때도 똑같은 효과를 거둘 수 있음
- mean, std 등 사용가능

```
df_2.sum()
```

```
age      299.00
height   531.54
dtype: float64
```

```
df_2.apply(sum)
```

```
age      299.00
height   531.54
dtype: float64
```

apply()는 시리즈 전체 컬럼 전체에 함수를 적용하는 것이다.

데이터 베이스 부분을 배우면 알겠지만,

group by 가 된 것이라고도 표현할 수 있다.

컬럼 전체에 대해서 grouping 되었다...

최대값과 최소값을 먼저 max 와 min 으로 구한다.

그리고 age 에서 제일 큰 값과 제일 작은 값의 차이를 빼서 이런 결과가 나왔다.

맨 아래에는 그냥 sum 을 썼는데, **sum** 은 합을 구해주는 내장 함수다.

```
def f(x):
    return Series([x.min(), x.max(), x.mean()], index = ["min", "max", "mean"])
df_2.apply(f)
```

	age	height
min	13.000000	35.22
max	52.000000	95.66
mean	33.222222	59.06

apply 는 데이터 프레임에도 쓸 수가 있다.

Series 를 이용해서 스칼라가 아닌 시리즈 형태로도 뽑을 수 있다.

age 그룹, height 그룹마다 min, max, mean 값이 나왔다. (그룹바이)

```
f = lambda x: -x
df_2.applymap(f).head(5)
```

	age	height
0	-23	-67.43
1	-42	-76.21
2	-52	-54.33
3	-13	-35.22
4	-43	-48.99

```
f = lambda x: -x
df_2["height"].apply(f).head(5)
```

```
0 -67.43
1 -76.21
2 -54.33
3 -35.22
4 -48.99
Name: height, dtype: float64
```

applymap 은 컬럼 전체에 대한 연산을 해준다.

(apply 는 특정 컬럼에만 해줌) 앞에서 본 map 이랑 applymap 이 비슷한데, map 은 series 에 쓰다면 applymap 은 데이터 프레임에 쓴다고 볼 수 있다.

Pandas 내장함수

```
df = pd.read_csv("./excel_sample/map_1.csv")
df.head()
```

	city	age	sex	job	marriage	height
0	seoul	23	male	student	n	67.43
1	busan	42	male	salesman	y	76.21
2	jeonju	52	female	ceo	y	54.33
3	incheon	13	male	student	n	35.22
4	daejeon	43	female	professor	n	48.99

```
df.describe()
```

	age	height
count	9.000000	9.000000
mean	33.222222	59.060000
std	12.978615	18.133007
min	13.000000	35.220000
25%	24.000000	48.990000
50%	33.000000	54.330000
75%	43.000000	67.430000
max	52.000000	95.660000

우선은 내장함수 중

describe()는 내가 가진 데이터 프레임 안에 있는 numeric 값에 대한 요약 정보를 보여준다.

count = 몇 개, mean = 평균값

unique

- series data의 유일한 값을 list로 리턴

```
df.job.unique() # 유일한 직업의 값 list
```

```
array(['student', 'salesman', 'ceo', 'professor', 'clerk', 'researcher',  
      'police', 'artist'], dtype=object)
```

```
np.array(dict(enumerate(df["job"].unique())))) # dict type으로 index
```

```
array({0: 'student', 1: 'salesman', 2: 'ceo', 3: 'professor', 4: 'clerk',  
      5: 'researcher', 6: 'police', 7: 'artist'}, dtype=object)
```

unique 는 인덱스로 사용하기 좋다.

유니크 각각 값에 대해 enumerate 를 쓰면 unique 한 숫자가 할당이 된다.

```
key = np.array(list(enumerate(df["job"].unique())), dtype=str)[: , 1].tolist()
key # label index 값 추출
```

```
['student',
'salesman',
'ceo',
'professor',
'clerk',
'researcher',
'police',
'artist']
```

```
value =list(map(int, np.array(list(enumerate(df["job"].unique()))[: , 0].tolist()))
value # label 값 추출
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

이걸 dict 나 list 형식으로 변경해서 key 값과 value 값으로 나눠서 담는다.

* 이걸 하는 이유는 원래 "job" 컬럼에 있는 데이터는
"student" "salesman" "ceo" 이런 식으로 되어있는데
분석할 때
글자 타입을 곱하기 나누기는 할 수가 없기 때문에
각각 숫자 value 로 변경해줘서 연산을 하게 된다.

sum

- 기본적인 column 또는 row 값의 연산을 지원
- sub, mean, min, max, count, median, mad, var 등

```
df.sum(axis = 0) # column 합산
```

```
city      36.00
age       299.00
sex        5.00
job       28.00
marriage   4.00
height   531.54
dtype: float64
```

```
df.sum(axis = 1).head() # row 합산
```

```
0    90.43
1   121.21
2   112.33
3    51.22
4    99.99
dtype: float64
```

내장함수로 지원되는 sum 도 가능하고 이걸 또 axis 기준으로 할 수 있다.

isnull

- column 또는 row 값의 NaN(null)값의 index 반환

```
df.isnull().head(3)
```

	city	age	sex	job	marriage	height
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False

```
df.isnull().sum().head(3)
```

```
city    0  
age     0  
sex     0  
dtype: int64
```

isnull 은 데이터가 null 값일 때 True 가 나오는 것이다.

지금 데이터에는 Null 이 없어서 False 만 나왔다.

데이터가 무조건 정형화 되어있는 게 아니기 때문에
숫자 자리에 숫자 아닌 게 들어갈 때
그 값들을 찾아낼 수 있으니까 **데이터 전처리할 때 사용한다.**

sort_values

- column 값을 기준으로 데이터를 sorting

```
df.sort_values(["age", "height"], ascending = True).head(5)  
# ascending = True 는 오름차순 정렬
```

	city	age	sex	job	marriage	height
3	3	13	0	0	0	35.22
0	0	23	0	0	0	67.43
8	8	24	1	7	0	46.15
5	5	24	1	4	1	51.32
6	6	33	0	5	0	95.66

sort_value 는 정렬을 해준다.

ascending = True 를 쓰면 오름차순이고 False 면 내림차순이 된다.

cumsum

- column 값을 기준으로 데이터를 sorting

```
df.cumsum().head(5) # 누적합
```

	city	age	sex	job	marriage	height
0	0.0	23.0	0.0	0.0	0.0	67.43
1	1.0	65.0	0.0	1.0	1.0	143.64
2	3.0	117.0	1.0	3.0	2.0	197.97
3	6.0	130.0	1.0	3.0	2.0	233.19
4	10.0	173.0	2.0	6.0	2.0	282.18

cumsum 은 누적합이다.

```
df.cummax().head(5) # 내려가면서 가장 큰값만 보여줌
```

	city	age	sex	job	marriage	height
0	0.0	23.0	0.0	0.0	0.0	67.43
1	1.0	42.0	0.0	1.0	1.0	76.21
2	2.0	52.0	1.0	2.0	1.0	76.21
3	3.0	52.0	1.0	2.0	1.0	76.21
4	4.0	52.0	1.0	3.0	1.0	76.21

cummax 는 인덱스따라 내려가면서
가장 큰 값이 나오면 그 값만 화면에 출력하는 것이다.

Correlation & Covariance

- 상관계수와 공분산을 구하는 함수
- corr, cov, corrwith

```
df.age.corr(df.height) # df의 col1, col2 상관관계 계산
```

0.27022986518660663

```
df.age.cov(df.height) # 변수간 공분산 계산
```

63.596249999999976

```
df.corrwith(df.height) # 다른시리즈나 데이터프레임과 상관관계 계산  
# 하나의 변수와 나머지 변수 간의 상관관계를 볼 때 좋음
```

이건 상관계수와 공분산을 구하는 함수인데, 많이 쓰게 될 것이다.

예를 들어, 내가 데이터 분석을 하는데 자전거의 수요에 영향을 미치는 요인들을 보고 싶으면,
날씨도 있고, 날짜, 공휴일, 월급날 같은 것들이 있을 텐데
결과적으로 여름과 일요일에 가장 많이 탄다 (높은 상관계수) 를
보고 싶을 때 많이 쓴다.

이걸 돌려서 관계 있는 중요한 변수를 선택해서 필요한 것만 잘라내는 등으로 활용한다.

```
df.corr() # 전체 상관관계
```

	city	age	sex	job	marriage	height
city	1.000000	-0.049236	0.519615	0.940845	-0.086603	-0.116544
age	-0.049236	1.000000	0.399997	0.163983	0.550250	0.270230
sex	0.519615	0.399997	1.000000	0.594442	0.350000	-0.500684
job	0.940845	0.163983	0.594442	1.000000	0.051245	0.059439
marriage	-0.086603	0.550250	0.350000	0.051245	1.000000	0.024197
height	-0.116544	0.270230	-0.500684	0.059439	0.024197	1.000000

전체적인 상관계수를 볼 수도 있다.

job 과 city 의 관계가 높아서 어디 사느냐에 따라 job 이 다르다.

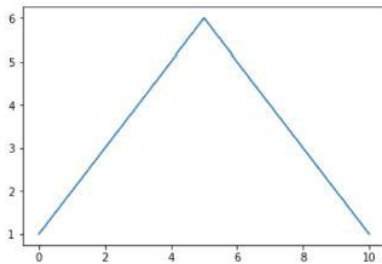
age 와 marriage 는 나이가 많을 수록 결혼한 사람이 많을 테니 양의 상관을 보일 것이다.

Matplotlib 라이브러리

우리가 보고서를 쓰는 발표를 할 때는 ppt 로 많이 할텐데,
중요하게 생각하는 것을 쉽게 보고 이해할 수 있게 비주얼라이즈 하는 것이다.

```
import matplotlib.pyplot as plt
%matplotlib inline # jupyter notebook 상에서 그래프 출력

plt.figure() # 객체 선언
plt.plot([1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1])
plt.show() # 보여주기
```

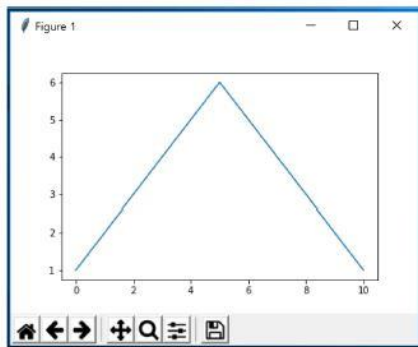


Matplotlib 에서 가장 많이 쓰이는 것이 pyplot 이다.

inline 은 앞으로 주피터 안에서 그림을 출력하고 싶을 때 쓰는 것이다.

```
%matplotlib tk # 주피터노트북 외부에서 출력

plt.figure() # 객체 선언
plt.plot([1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1])
plt.show() # 보여주기
```



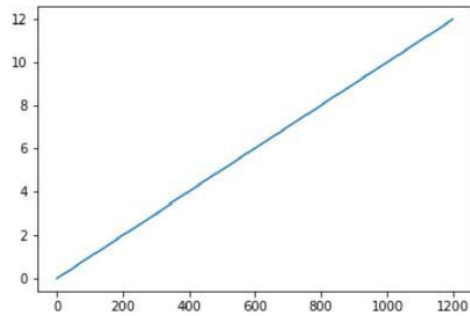
새로운 윈도우 창에서 그림을 보고 싶으면 tk 를 써주면 된다.

어쨌든 기본적인 건 figure 통해서 먼저 객체 선언하고

그 다음에 plot 통해서 값을 넣어주고 show 를 통해 보여주는 것이다.

Numpy 활용

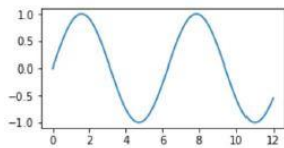
```
import numpy as np
x = np.arange(0, 12, 0.01)
plt.figure
plt.plot(x)
plt.show()
```



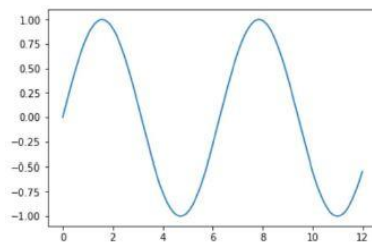
보면 numpy array 를 통해서 0 에서 12 까지 만들었으니까 1200 개가 만들어졌다!
이 1200 개에 대한 plot 을 그리면 위와 같다.

figure 크기조정

```
plt.figure(figsize = (4, 2))
plt.plot(x, y)
plt.show()
```



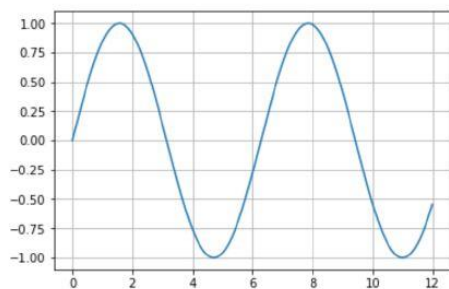
```
plt.figure(figsize = (6, 4))
plt.plot(x, y)
plt.show()
```



이런 식으로 **figure()**안에 **argument** 를 써서 **figure size** 도 만들어줄 수 있다.
shift+tab 이나 matplotlib 사이트 들어가면 argument 설명 있다.

눈금(grid)

```
plt.figure(figsize = (6, 4))
plt.plot(x, y)
plt.grid() # grid(color='r', linestyle='-', linewidth=2)
plt.show()
```



grid 함수는 뒤에 눈금이 나오게 할 수도 있다.

라벨(label)

```
dy = np.diff(y) # 차분함수 적용
```

```
len(x), len(dy)
```

```
(1200, 1199)
```

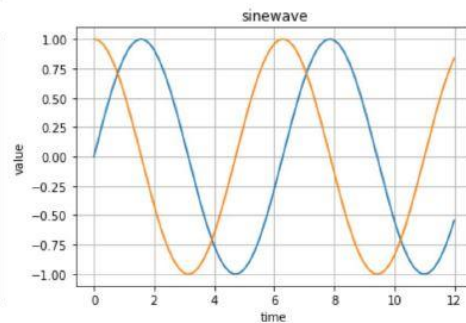
```
%whos
```

Variable	Type	Data/Info
dy	ndarray	1199: 1199 elems, type 'float64', 9592 bytes
np	module	<module 'numpy' from 'C:\<...>ges\numpy__init__.py'>
plt	module	<module 'matplotlib.pyplot' from 'C:\<...>matplotlib\pyplot.py'>
x	ndarray	1200: 1200 elems, type 'float64', 9600 bytes
y	ndarray	1200: 1200 elems, type 'float64', 9600 bytes

맨 위에 있는 **diff** 는 지금 막 중요한 건 아니다. 아까 처음에 있던 코사인데이터에서 사인 데이터 만드느라 쓴 건데, **미분하는 함수**를 쓴 것이다.

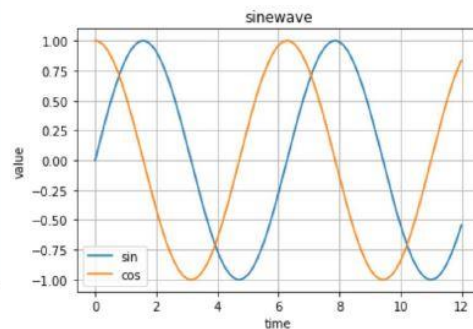
```
dy = np.insert(dy, 0, 0.01)/0.01
```

```
plt.figure(figsize = (6, 4))  
plt.plot(x, y)  
plt.plot(x, dy)  
plt.grid()  
plt.title("sinewave")  
plt.xlabel('time') # x축 라벨 적용  
plt.ylabel('value') # y축 라벨 적용  
plt.show()
```



title 이란 함수로 위에 제목을 만들고, xlabel 과 ylabel 을 각각 만들 수 있다.

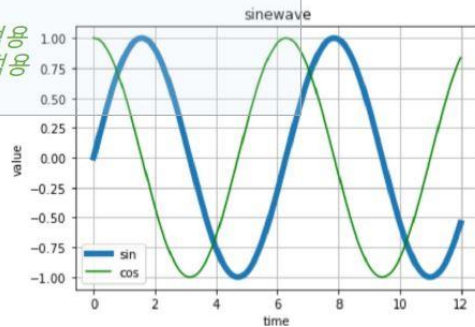
```
plt.figure(figsize = (6, 4))  
plt.plot(x, y, label = 'sin')  
plt.plot(x, dy, label = 'cos')  
plt.grid()  
plt.legend()  
plt.title("sinewave")  
plt.xlabel('time') # x축 라벨 적용  
plt.ylabel('value') # y축 라벨 적용  
plt.show()
```



plt.legend()가 추가되었는데,
그러면 오른쪽 그래프의 왼쪽 하단에 보면 **범례가 생성**되었다!

● 색 및 라인의 크기

```
plt.figure(figsize = (6, 4))
plt.plot(x, y, lw = 5, label = 'sin') # lw 옵션: 라인크기조정
plt.plot(x, dy, 'g', label = 'cos')
plt.grid() # 'g' 색 옵션
plt.legend()
plt.title("sinewave")
plt.xlabel('time') # x 축 라벨 적용
plt.ylabel('value') # y 축 라벨 적용
plt.show()
```

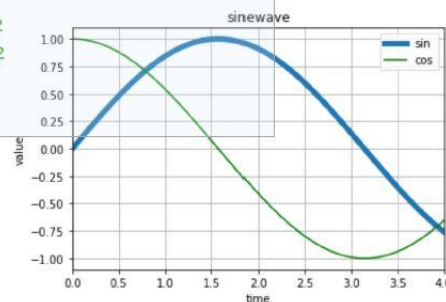


이거는 색깔 옵션 g 는 초록색이란 뜻이고

argument 로 간단히 쓸 수 있는데 color=green 이런식으로 쓸 수도 있다.

● xlim - x, y 축 범위지정

```
plt.figure(figsize = (6, 4))
plt.plot(x, y, lw = 5, label = 'sin') # lw 옵션: 라인크기조정
plt.plot(x, dy, 'g', label = 'cos')
plt.grid() # 'g' 색 옵션
plt.legend()
plt.title("sinewave")
plt.xlabel('time') # x 축 라벨 적용
plt.ylabel('value') # y 축 라벨 적용
plt.xlim(0, 4)
plt.show()
```



xlim 과 ylim 을 통해서 x 축과 y 축의 범위를 조정할 수 있다.

지정 안 해주면 내가 넣은 데이터의 범위까지 다 나오는데,

이렇게 지정하면 원하는 범위까지 자를 수 있다.

● 마커

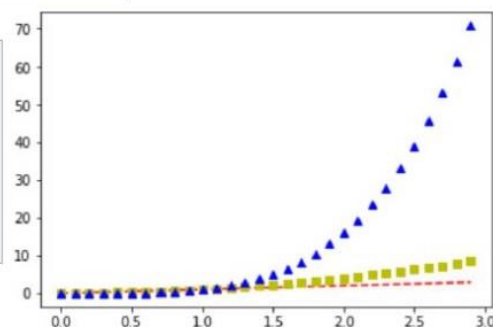
```
a = np.arange(0, 3, 0.1)

plt.figure()
plt.plot(a, a, 'r--', a, a**2, 'ys', a, a**4, 'b^')
plt.show()
```

* plot안에 여러개의 x,y 데이터를
가지면, 다수의 그래프 가능
* r-- : 빨간색 점선
* 'ys' : 노란색 박스
* 'b^' 위로 향한 세모모양의 파란색

```
a = np.arange(0, 3, 0.1)

plt.figure()
plt.plot(a, a, 'r--')
plt.plot(a, a**2, 'ys')
plt.plot(a, a**4, 'b^')
plt.show()
```



마커는 내가 만든 plot에서 어떤 선으로 표현될 건지를 지정해줄 수 있다.

r--는 red에 점선으로 나온다는 것을 의미한다.

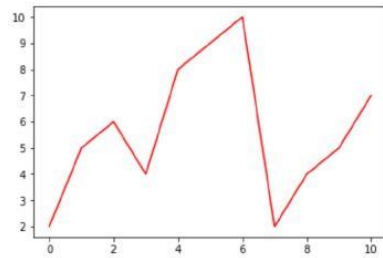
ys는 노란색 박스(square)모양으로 나온다.

b^는 파란색 세모 모양으로 나온다.

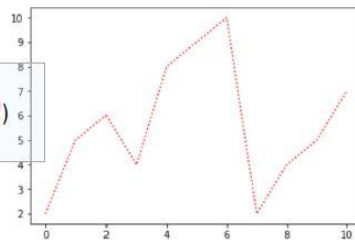
마커

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [2, 5, 6, 4, 8, 9, 10, 2, 4, 5, 7]
```

```
plt.figure()
plt.plot(a, b, color='red')
plt.show()
```

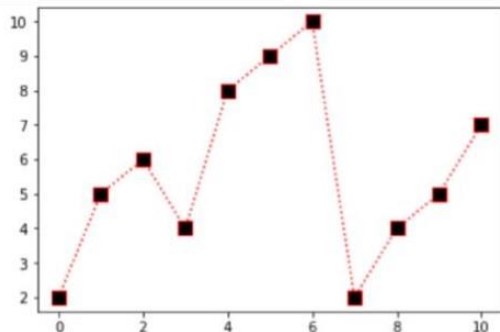


```
plt.figure()
plt.plot(a, b, color='red', linestyle='dotted')
plt.show()
```



색깔과 라인 스타일을 이렇게 옵션으로 써줄 수도 있다.

```
plt.figure()
plt.plot(a, b, color='red', linestyle='dotted', marker='s', markerfacecolor='black',
        markersize=10)
plt.show()
```



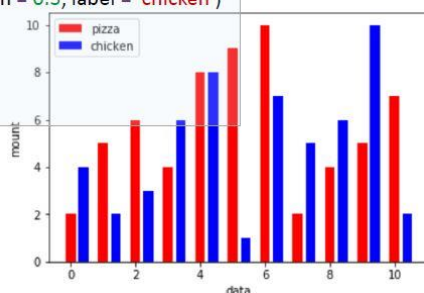
marker 안에 색도 칠할 수 있고, 마커 사이즈도 변경할 수 있다.

여러형태의 그래프(scatter, bar, barh, pie)

- bar : 바형태로 표현하는 데이터

```
y = np.array([4, 2, 3, 6, 8, 1, 7, 5, 6, 10, 2])
```

```
plt.figure()
plt.bar(a, b, color='r', width=0.3, label='pizza')
plt.bar(a + 0.4, y, color='b', width=0.3, label='chicken')
plt.xlabel('data')
plt.ylabel('count')
plt.legend()
plt.show()
```



바형태의 그래프도 그릴 수 있다.

이건 조사한 데이터는 아니고 가짜로 매달 피자과 치킨의 선호도를 가상으로 만들어본 것이다.

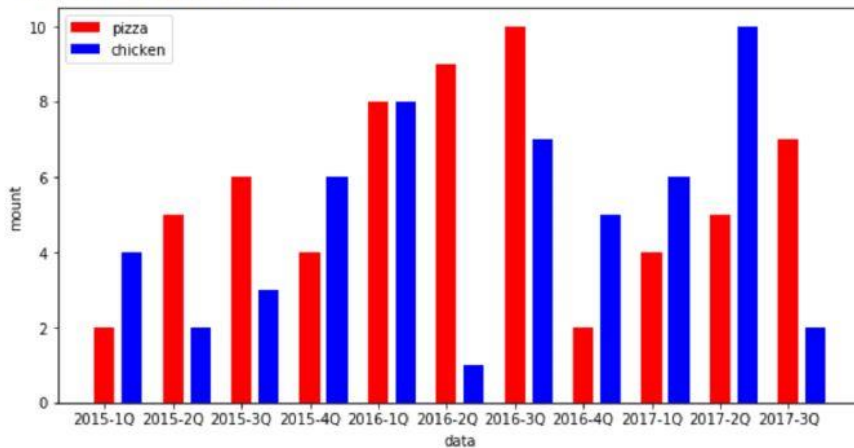
* xticks를 이용한 x축 설정

```
plt.figure(figsize = (20, 10))

plt.bar(a, b, color = 'r', width = 0.3, label = 'pizza')
plt.bar(a + 0.4, y, color = 'b', width = 0.3, label = 'chicken')
plt.xlabel('data')
plt.ylabel('mount')
plt.legend()
plt.xticks(a, ('2015-1Q', '2015-2Q', '2015-3Q', '2015-4Q', '2016-1Q', '2016-2Q', '2016-3Q', '2016-4Q', '2017-1Q', '2017-2Q', '2017-3Q'))

plt.show()
```

* xticks를 이용한 x축 설정



x 축과 y 축은 원래 숫자로 나타나는데,

이거를 xtick 을 이용해서 지정을 해줄 수가 있다.

위와 같이 데이터를 넣어서 2015 년 1 분기에는 치킨이 더 많이 나갔다...

이런식으로 이해하기 쉽게 축의 이름을 설정해줄 수 있다.

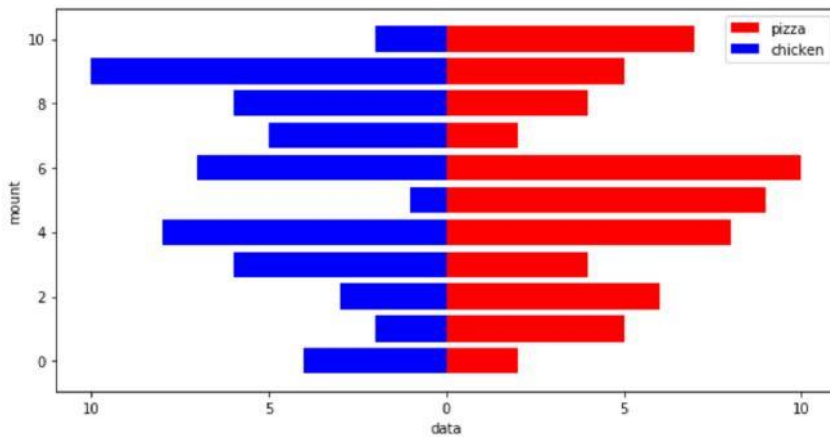
* barh를 이용한 대칭 바그래프

```
plt.figure(figsize = (10, 5))

plt.barh(a, b, color = 'r', width = 0.3, label = 'pizza')
plt.barh(a - y, y, color = 'b', width = 0.3, label = 'chicken')
plt.xlabel('data')
plt.ylabel('mount')
plt.legend()
plt.xticks([-10, -5, 0, 5, 10], ('10', '5', '0', '5', '10'))

plt.show()
```

* barh를 이용한 대칭 바그래프



barh 로 대칭바를 그릴 수 있다.

xticks 의 앞 부분은 실제 데이터 값들이고,

오른쪽이 밖에 보일 라벨 이름들이다.

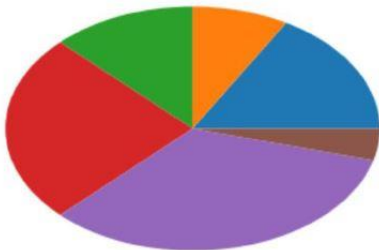
그런데 -10 에서 10 까지 써야 하는 이유는

대칭으로 그리려면 - 값이 들어가야 하기 때문이다.

- pie

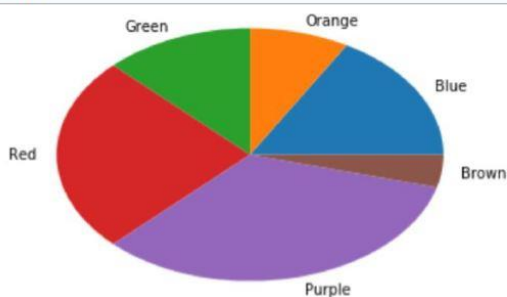
```
y = np.array([4, 2, 3, 6, 8, 1])
```

```
plt.figure()
plt.pie(y)
plt.show()
```



- pie

```
label = ['Blue', 'Orange', 'Green', 'Red', 'Purple', 'Brown']
plt.figure
plt.pie(y, labels = label)
plt.show()
```



파이 그래프도 이름도 쓸수 있고 색도 변경할 수 있다.

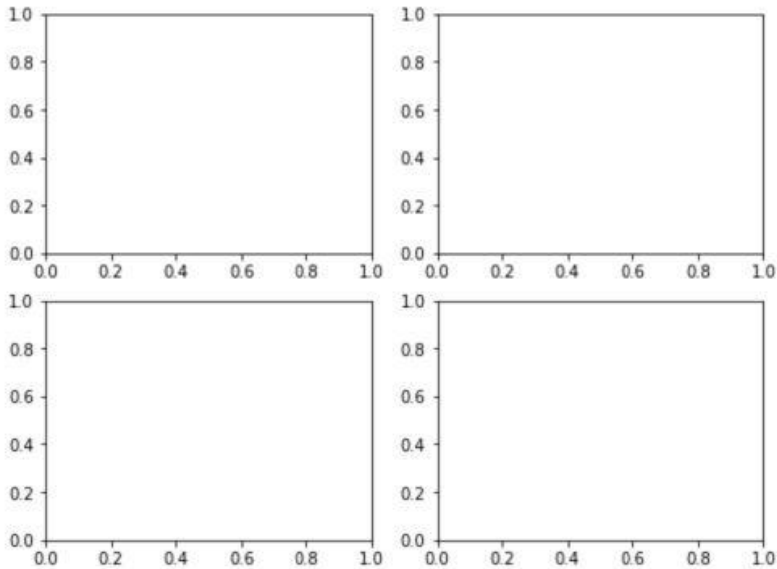
subplot

- 한 화면에 여러 그래프 나타내기

```
plt.figure(figsize = (8, 6))

plt.subplot(221) # 2 by 2 subplot 중 1 번째
plt.subplot(222) # 2 by 2 subplot 중 2 번째
plt.subplot(223) # 2 by 2 subplot 중 3 번째
plt.subplot(224) # 2 by 2 subplot 중 4 번째

plt.show()
```



subplot 이라고 해서 하나의 figure 안에 여러 그래프를 따로 나눠서 그릴 수가 있다.

쓸 때는 2x2 배열로 위와 같이 둘 경우 221 222 223 224 이런 식으로 쓰면 된다.

```
x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.figure(figsize = (10, 8))

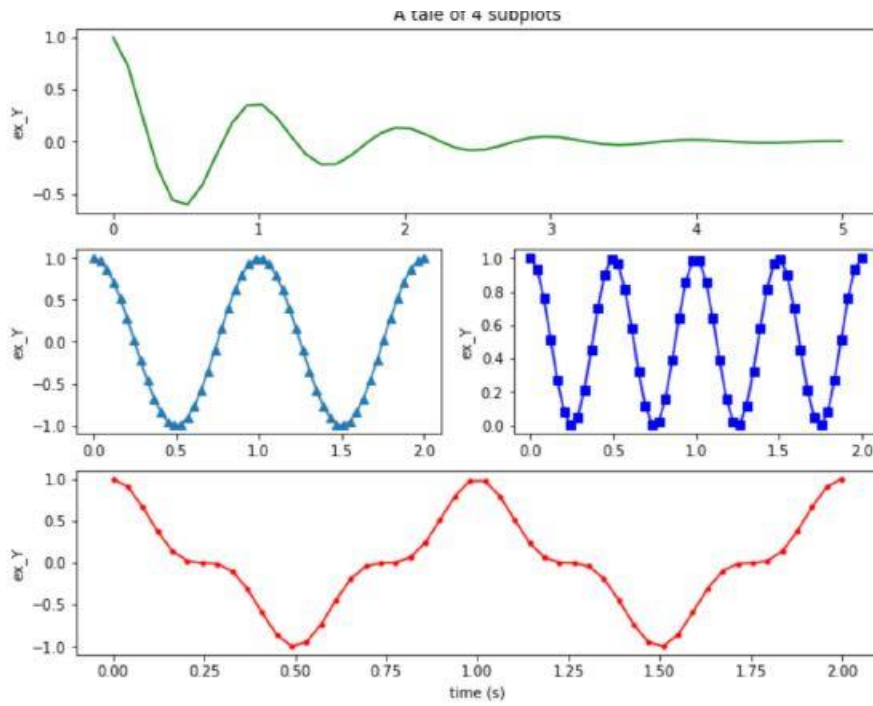
plt.subplot(3, 1, 1)
plt.plot(x1, y1, 'g-')
plt.title('A tale of 4 subplots')
plt.ylabel('ex_Y')
```

```
plt.subplot(3, 2, 3)
plt.plot(x2, y2, '^-')
plt.ylabel('ex_Y')

plt.subplot(3, 2, 4)
plt.plot(x2, y2**2, '-sb')
plt.ylabel('ex_Y')

plt.subplot(3, 1, 3)
plt.plot(x2, y2**3, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('ex_Y')

plt.show()
```

여기에 그래프 하나하나 넣어서 이런 식으로 표현하실 수 있다.

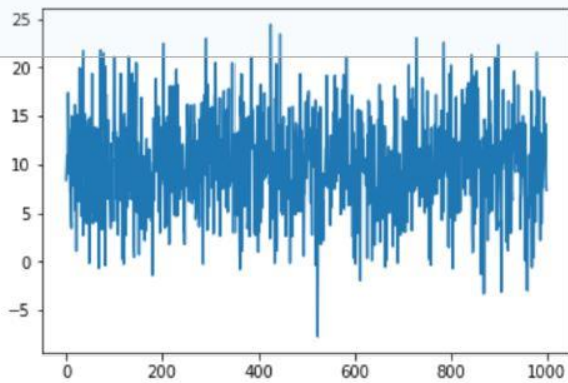


히스토그램 & 박스플롯(boxplot)

- 가우스분포를 가지는 랜덤변수 활용

```
data = np.random.normal(10, 5, 1000)
# 중간값: 10, 크기: 5, 데이터갯수: 500개, 가우스 분포이므로 5~15를 넘을 수 있음
```

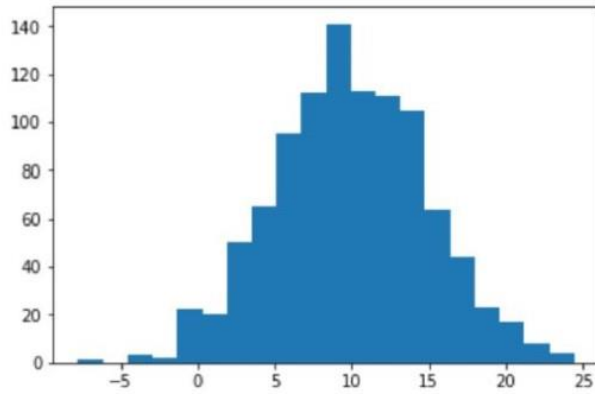
```
plt.figure()
plt.plot(data)
plt.show()
```



히스토그램과 박스플롯을 그리기 위해 가우스 분포를 가지는 데이터를 생성하였다.

그냥 plot 을 해주면 많은 데이터가 들어가서 이런 식으로 나온다.

```
plt.figure()
plt.hist(data, bins = 20)
plt.show()
```



위 그림이 히스토그램이다. (bin 부분은 옵션이라 빼도 된다.)

```
s1 = np.random.normal( loc = 0, scale = 2, size = 1000)
s2 = np.random.normal( loc = 5, scale = 0.5, size = 1000)
s3 = np.random.normal( loc = 10, scale = 1, size = 500)
```

```
plt.figure(figsize = (10,8))
```

```
plt.plot(s1, label = 's1')
```

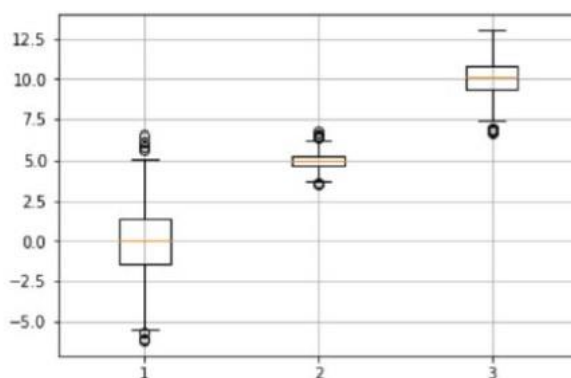
```
plt.plot(s2, label = 's2')
```

```
plt.plot(s3, label = 's3')
```

```
plt.legend()
```

```
plt.show()
```

```
plt.figure()
plt.boxplot((s1, s2, s3))
plt.grid()
plt.show()
```



그리고 랜덤값을 넣어서 박스플롯을 만들었다.

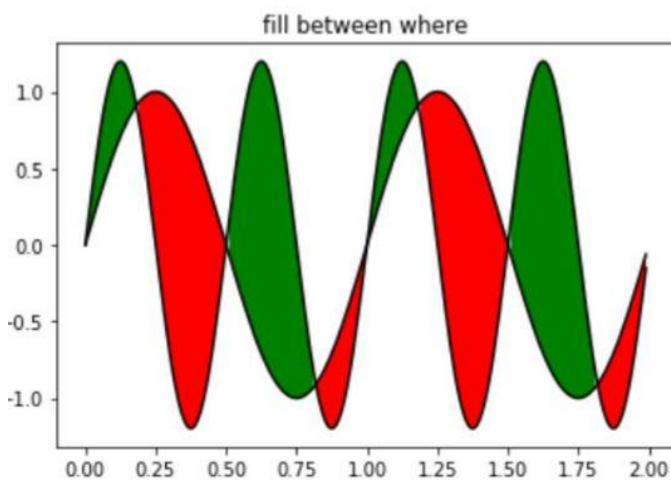
example_1

```
# https://matplotlib.org/gallery/pyplots/whats\_new\_98\_4\_fill\_between.html#sphx-glr-gallery-pyplots-whats-new-98-4-fill-between-py
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig = plt.figure()
ax = fig.subplots()
ax.plot(x, y1, x, y2, color = 'black')
ax.fill_between(x, y1, y2, where = y2 > y1, facecolor = 'green')
ax.fill_between(x, y1, y2, where = y2 <= y1, facecolor = 'red')
ax.set_title('fill between where')

plt.show()
```



Matplotlib 사이트에 가시면 정말 많은 예제가 있다!

위에 데이터를 이렇게 색깔을 칠해볼 수가 있는데,
where 이라는 절을 써서 y_2 가 y_1 보다 클 때
fill_between 을 사용해서 그 사이의 색을 칠하라고 써주었다.

example_2

```
# Fixing random state for reproducibility
np.random.seed(19680801)

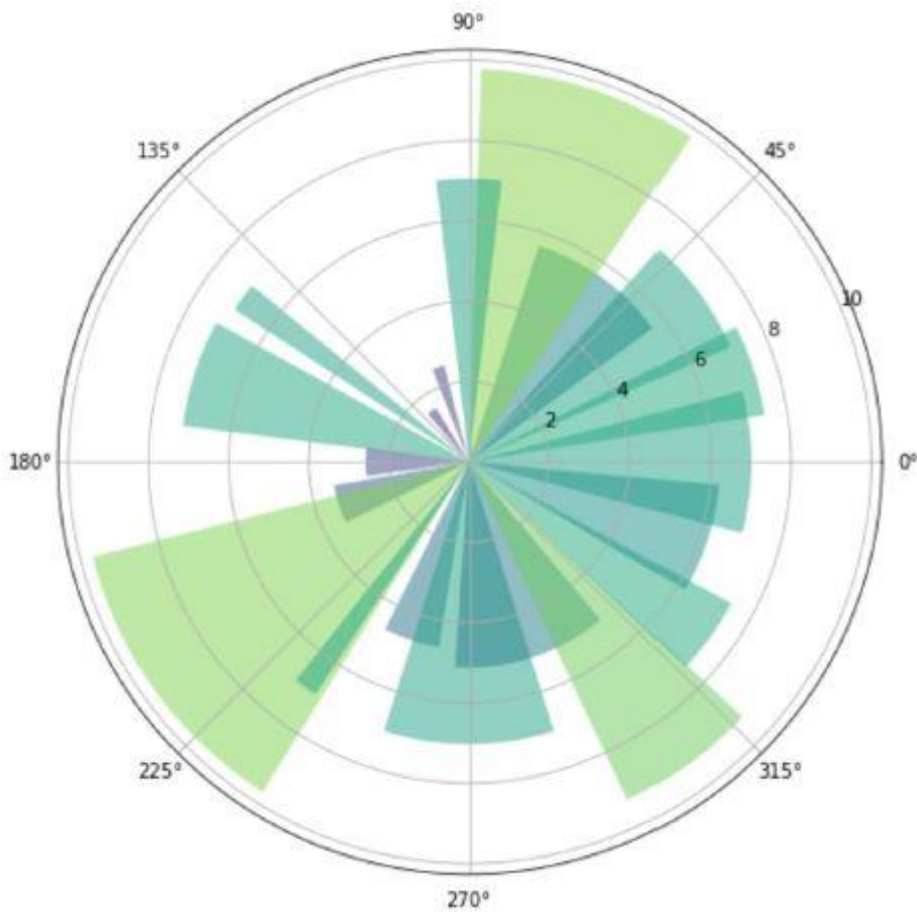
# Compute pie slices
N = 20
theta = np.linspace(0.0, 2 * np.pi, N, endpoint = False) # 0~20 사이를 2*pi로 균등하게 나눈 값
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)

plt.figure(figsize = (10, 8))

ax = plt.subplot(111, projection = 'polar')
bars = ax.bar(theta, radii, width = width, bottom = 0.0)

# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.viridis(r / 12))
    bar.set_alpha(0.5)

plt.show()
```



이건 **polar** 형태인데 **plot** 안에 **bar** 를 통해서 그림을 또 그려준 것이다.
이 **bar** 의 **argument** 가 달라지면 색이 변한다.

example_2

```
plt.figure(figsize = (10, 8))
ax = plt.subplot(111, projection = 'polar')
bars = plt.subplot(111, projection = 'polar').bar(theta, radii, width = width, bottom = 0.0)

for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.viridis(r / 10.)) # plasma, inferno, magma
    bar.set_alpha(0.5)

plt.show()
```

