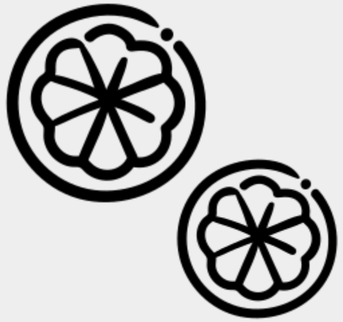


데이터 분석을 위한 3종 패키지

numpy, pandas, matplotlib

9기 교육부 김진영, 김은혜

1. Numpy

01 numpy 배열

02 배열의 생성과 변형

03 배열의 연산

04 기술 통계

05 난수 발생

Numpy란?

파이썬에서 배열을 사용하기 위한 표준 패키지

수치 해석용 파이썬 패키지로, 벡터/행렬 사용하는 선형대수 계산에 주로 사용

Numpy의 사용

import numpy as np

```
import numpy as np # numpy 패키지를 np로 줄여서 사용
```

01 Numpy 배열

1차원 배열 만들기

array() 함수 사용

ar=np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

ar

```
ar = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
ar
```

출력결과)

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

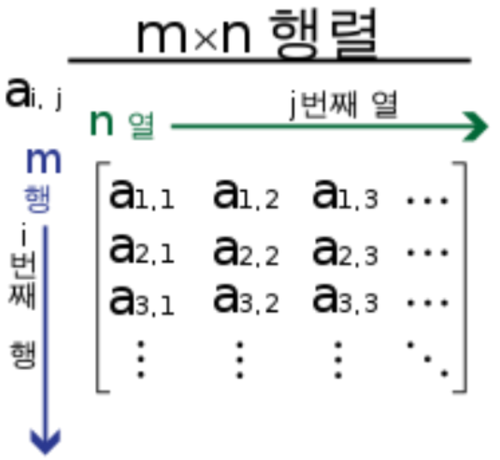
01 Numpy 배열

2차원 배열 만들기

2차원 배열 (matrix)

행(row) 의 개수 : len(c)

열(column) 의 개수 : len(c[0]), len(c[1])



ex)

```
c=np.array([[0,1,2],[3,4,5]])
len(c) # 행의 수, 출력결과 2
c[0] # array([0,1,2])
len(c[0]) # 열의 수, 출력결과 3
```

```
array([0, 1, 2],
      [3, 4, 5]) # c의 출력 결과
```

바깥쪽(전체) 리스트: [[0, 1, 2],[3, 4, 5]]

안쪽 리스트: [0, 1, 2] 와 [3, 4, 5]

1차원 배열 인덱싱

파이썬에서의 리스트 인덱싱과 동일

인덱스 번호가 0부터 시작한다는 것 유의!

(뒤에서부터 거꾸로 셀 때는 -1부터 시작)

```
a=np.array([0, 1, 2, 3, 4, 5])
a[0] # 출력결과 0
a[2] # 출력결과 2
a[-1] # 출력결과 5
```

다차원 배열 인덱싱

콤마(,) 를 사용하여 접근


```
b[0,0] # 첫번째 행의 첫번째 열
b[0,1] # 첫번째 행의 두번째 열
b[-1,-1] # 마지막 행의 마지막 열
```

fancy 인덱싱

정수나 boolean 값을 가지는 다른 numpy 배열로 배열을 인덱싱

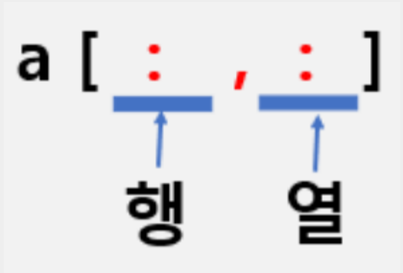
ex)

```
array=np.array([i for i in range(10)]) # array([0,1,2,3,4,5,6,7,8,9])
index=np.array([True, False, True, False, True, False, True, False, True, False])
print(array[index]) # 출력결과: [0,2,4,6,8]
print(array[array%2==0]) # 출력결과: [0,2,4,6,8]
```

배열 슬라이싱

배열의 원소 중 복수개를 접근

ex)




```
a[0,:] # 첫번째 행 전체
a[:,1] # 두번째 열 전체
a[1,1:] # 두번째 행의 두번째 열부터 끝열까지
```

Inf와 NaN

Inf(Infinity) : 무한대

NaN (Not A Number) : 정의할 수 없는 숫자

ex)

np.array([0,1,-1,0]) / np.array([1,0,0,0])

출력결과) array([0. , inf, -inf, nan])

1 나누기 0 : Inf

Log(0) : -Inf

0 나누기 0 : NaN

zeros

크기가 정해져 있고, 모든 값이 0인 배열 생성

ones

크기가 정해져 있고, 모든 값이 1인 배열 생성

```
#크기가 5이고 모든 값이 0인 배열 생성
a = np.zeros(5)
array([0.,0.,0.,0.,0.]) # 출력 결과

#5행 2열의 배열, int 자료형
b=np.zeros((5,2), dtype="i")
array([0,0],
        [0,0],
        [0,0],
        [0,0],
        [0,0], dtype=int32) # 출력 결과
```

zeros_like

다른 배열과 같은 크기,
모든 값이 0인 배열 생성

ones_like

다른 배열과 같은 크기,
모든 값이 1인 배열 생성

```
#크기가 3이고 모든 값이 0인 배열 생성
a = np.zeros(3)
array([0.,0.,0.]) # 출력 결과

#a와 같은 크기, 모든값 0인 배열 생성
b=np.zeros_like(a, dtype="f")
array([0.,0.,0.],  dtype=float32) # 출력 결과

#a와 같은 크기, 모든값 1인 배열 생성
c=np.ones_like(a, dtype="i")
array([1, 1, 1],  dtype=int32) # 출력 결과
```

02 배열의 생성과 변형

배열 생성

empty

배열 생성만 하고, 특정 값으로 초기화 하지 않음

```
a=np.empty(4,3))
array([[6.94820328e-310, 4.67533915e-310, 5.28964691e+180],
       [6.01346953e-154, 4.81809028e+233, 7.86517465e+276],
       [6.01346953e-154, 2.58408173e+161, 2.46600381e-154],
       [2.47379808e-091, 4.47593816e-091, 6.01347002e-154]])
```

arange

numpy 버전의 range 명령

```
np.arange(10) # 0~9
array([0,1,2,3,4,5,6,7,8,9]) # 출력 결과

#시작, 끝(포함X), 간격
np.arange(3,21,2) #간격 2씩 증가
array([3,5,7,9,11,13,15,17,19]) # 출력 결과
```

linspace, logspace

선형 구간 / 로그 구간을 지정한 구간 수 만큼 분할

ex)

```
np.linspace(0,100,5) # 시작, 끝(포함O), 개수
array([0.,25.,50.,75.,100.]) # 출력 결과

np.logspace(0.1,1,10)
array([ 1.25892541,  1.58489319,  1.99526231,  2.51188643,  3.16227766,
        3.98107171,  5.01187234,  6.30957344,  7.94328235, 10.]) # 출력 결과
```

2차원 배열의 전치

전치 : 행과 열을 바꾸는 작업

ex)

```
A=np.array([[1,2,3],[4,5,6]])
array([1,2,3],
      [4,5,6]) # 출력 결과
```

```
A.T
array([1,4],
      [2,5],
      [3,6]) # 출력 결과
```

reshape()

만들어진 배열 내부 데이터 보존한 채로 형태만 변경

flatten(), ravel()

다차원 배열을 1차원으로 변경

```
a=np.arange(4)
array([0,1,2,3]) # 출력 결과

#배열 a를 2행 2열의 다차원 형태로 변경
b=a.reshape(2,2)
array([0,1],
        [2,3]) # 출력 결과

#배열 b를 1차원 형태로 변경
c=b.flatten() #혹은 b.ravel()도 가능
array([0,1,2,3]) # 출력 결과
```

hstack()

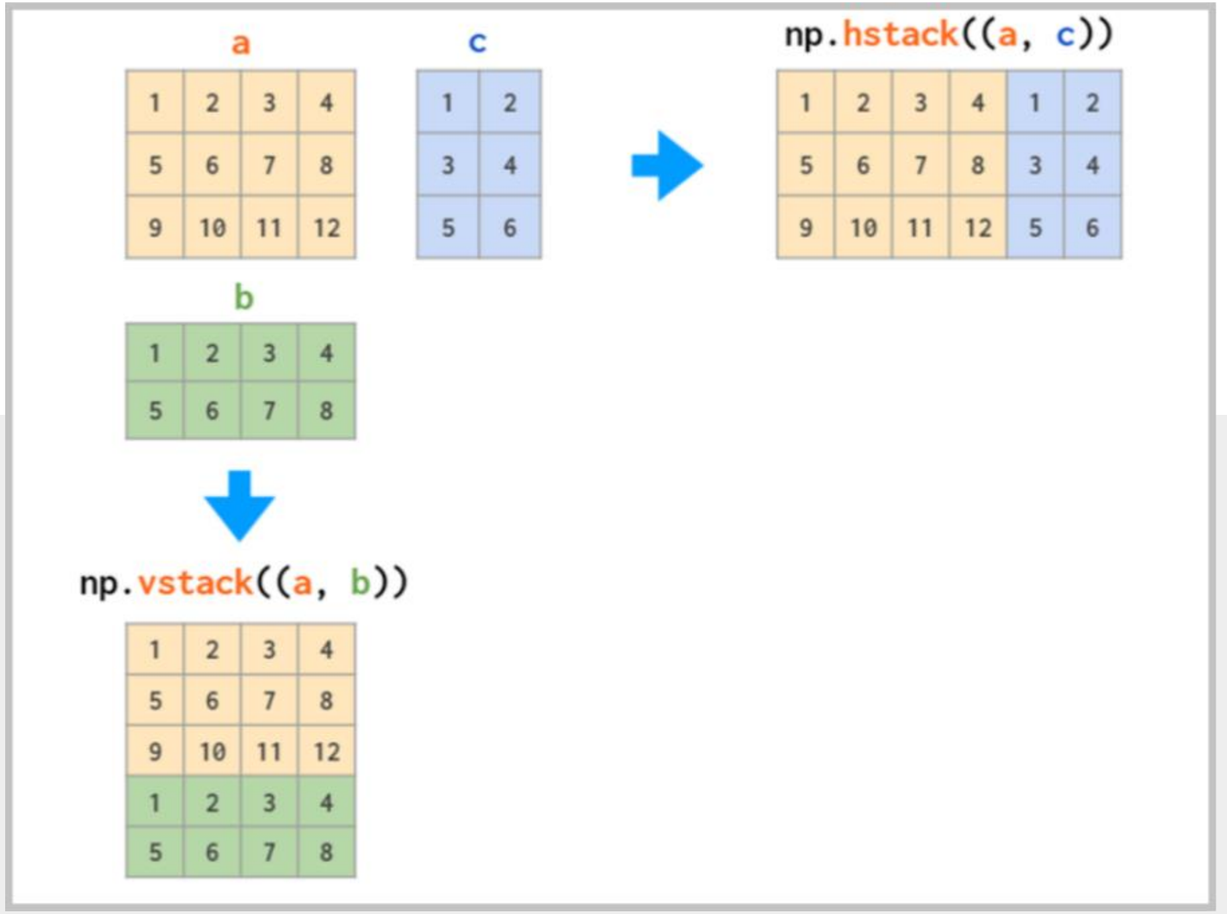
열 연결

행의 수가 같은 두개 이상의 배열 옆으로 연결

vstack()

행 연결

열의 수가 같은 두개 이상의 배열 위아래로 연결



벡터화 연산

$$z = x + y$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 10000 \end{bmatrix} + \begin{bmatrix} 10001 \\ 10002 \\ 10003 \\ \vdots \\ 20000 \end{bmatrix} = \begin{bmatrix} 1+10001 \\ 2+10002 \\ 3+10003 \\ \vdots \\ 10000+20000 \end{bmatrix} = \begin{bmatrix} 10002 \\ 10004 \\ 10006 \\ \vdots \\ 30000 \end{bmatrix}$$

ex)

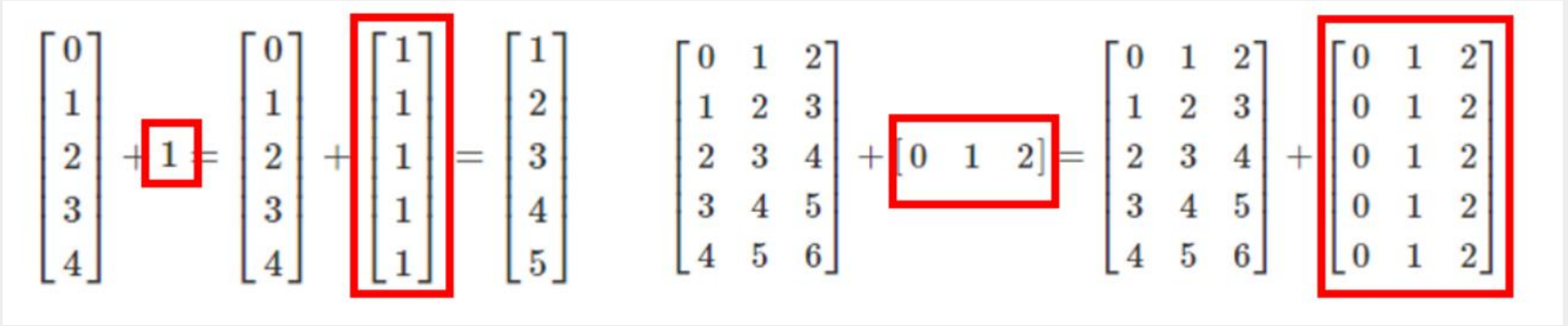
```
a=np.array([1,2,3,4])
b=np.array([4,2,2,4])
print(a+b) # [5, 4, 5, 8]
print(a==b) # [False, True, False, True]
```

브로드캐스팅

서로 다른 크기를 가진 두 배열의 사칙연산 지원

크기가 작은 배열을 자동으로 반복 확장하여 큰 배열에 맞춤

ex)



기술통계

len()	데이터의 개수
mean()	표본 평균
var()	표본 분산
std()	표본 표준 편차
max()	최댓값
min()	최솟값
median()	중앙값
percentile(자료, 분위수)	분위수

* 분위수 숫자 의미) 0: 최솟값, 25: 1분위수, 50: 2분위수, 75: 3분위수, 100: 최댓값

05 난수 발생

시드 설정과 데이터 순서 바꾸기

시드(seed)

컴퓨터가 정해진 알고리즘에 의해, 난수 수열을 생성하는 특정 시작 숫자
일단 생성된 난수는 다음번 난수 생성을 위한 시드값이 된다.
특정 시드값 사용시, 다음에 만드는 난수는 예측 가능

shuffle()

데이터의 순서를 바꿀 때 사용

```
x=np.arange(10)
print(x) # [0,1,2,3,4,5,6,7,8,9]
np.random.shuffle(x)
print(x) # ex) [5,8,0,6,1,3,4,9,2,7]
```

2. pandas

01 pandas 패키지 소개

02 데이터프레임 고급 인덱싱

03 데이터프레임의 데이터 조작

Pandas란?

고수준의 자료 구조와 빠르고 쉬운 데이터 분석 도구를 제공하는 파이썬 라이브러리

pandas의 자료구조에는 series, dataframe이 있다.

1) series : 일련의 객체를 담을 수 있는 1차원 배열 같은 구조
2) dataframe : 2차원 자료 구조

Pandas의 사용

import pandas as pd

```
import pandas as pd
```

01 pandas 패키지 소개

series

series 생성

pd.Series()

```
s = pd.Series([9904312, 3448737, 2890451, 2466052],
              index=["서울", "부산", "인천", "대구"])
s
```

```
서울    9904312
부산    3448737
인천    2890451
대구    2466052
dtype: int64
```

```
pd.Series(range(10, 14))
```

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

series의 index, value, name 속성

index: series 인덱스 접근

values: series 값 접근

name:

series 데이터/ 인덱스 이름 붙이기 가능

```
s.index
```

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

```
s.values
```

```
array([9904312, 3448737, 2890451, 2466052])
```

```
s.name = "인구"
s.index.name = "도시"
s
```

```
도시
서울    9904312
부산    3448737
인천    2890451
대구    2466052
Name: 인구, dtype: int64
```

series 연산

벡터화 연산 가능

연산은 series 값에만 적용, 인덱스 값은 적용X

ex)

```
s = pd.Series([9904312, 3448737, 2890451, 2466052],
              index=["서울", "부산", "인천", "대구"])
s
```

```
서울    9904312
부산    3448737
인천    2890451
대구    2466052
dtype: int64
```

```
s / 1000000
```

```
도시
서울    9.904312
부산    3.448737
인천    2.890451
대구    2.466052
Name: 인구, dtype: float64
```

series 인덱싱

numpy 배열에서 사용한 인덱싱/슬라이싱 방법 사용 가능

인덱스 라벨을 사용하여 인덱싱 슬라이싱 가능 (문자열 라벨을 이용한 슬라이싱은 콜론 뒤 값도 포함)

인덱스 라벨이 문자인 경우, 점(.)을 사용해 값 접근 가능

ex)

```
s = pd.Series([9904312, 3448737, 2890451, 2466052],
              index=["서울", "부산", "인천", "대구"])
s
```

```
서울    9904312
부산    3448737
인천    2890451
대구    2466052
dtype: int64
```

```
s[1:3]  # 두번째(1)부터 세번째(2)까지 (네번째(3) 미포함)
```

```
도시
부산    3448737
인천    2890451
Name: 인구, dtype: int64
```

```
s["부산":"대구"]  # 부산에서 대구까지 (대구도 포함)
```

```
도시
부산    3448737
인천    2890451
대구    2466052
Name: 인구, dtype: int64
```

series와 딕셔너리 자료형

딕셔너리는 순서가 없기 때문에, series 데이터도 순서가 보장되지 않는다.

순서 정하고 싶다면, 인덱스를 리스트로 지정해야 한다.

ex)

```
s2 = pd.Series({"서울": 9631482, "부산": 3393191, "인천": 2632035, "대전": 1490158})
s2
```

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

```
s2 = pd.Series({"서울": 9631482, "부산": 3393191, "인천": 2632035, "대전": 1490158},
               index=["부산", "서울", "인천", "대전"])
s2
```

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

데이터의 갱신, 추가, 삭제

기존

```
rs

부산    1.636984
서울    2.832690
인천    9.818107
dtype: float64
```

rs #기존 데이터
부산 : 1.636984
서울 : 2.832690
인천 : 9.818107

갱신

rs['부산']=1.63
rs

rs #갱신 후, 데이터
부산 : 1.630000
서울 : 2.832690
인천 : 9.818107

추가

rs['대구']=1.41
rs

rs #추가 후, 데이터
부산 : 1.630000
서울 : 2.832690
인천 : 9.818107
대구 : 1.410000

삭제

del rs['서울']
rs

rs #삭제 후, 데이터
부산 : 1.630000
인천 : 9.818107
대구 : 1.410000

dataframe의 생성

step1. 하나의 열이 되는 데이터를 리스트/일차원 배열로 준비

step2. 각 열에 대한 이름(라벨)을 키로 가지는 딕셔너리 생성

step3. dataframe 생성자에 데이터를 넣는다.

(열방향 인덱스는 columns, 행방향 인덱스는 index 인수로 지정)

ex)

```
data = {
    "2015": [9904312, 3448737, 2890451, 2466052],
    "2010": [9631482, 3393191, 2632035, 2431774],
    "2005": [9762546, 3512547, 2517680, 2456016],
    "2000": [9853972, 3655437, 2466338, 2473990],
    "지역": ["수도권", "경상권", "수도권", "경상권"],
    "2010-2015 증가율": [0.0283, 0.0163, 0.0982, 0.0141]
}
columns = ["지역", "2015", "2010", "2005", "2000", "2010-2015 증가율"]
index = ["서울", "부산", "인천", "대구"]
df = pd.DataFrame(data, index=index, columns=columns)
df
```

	지역	2015	2010	2005	2000	2010-2015 증가율
서울	수도권	9904312	9631482	9762546	9853972	0.0283
부산	경상권	3448737	3393191	3512547	3655437	0.0163
인천	수도권	2890451	2632035	2517680	2466338	0.0982
대구	경상권	2466052	2431774	2456016	2473990	0.0141

01 pandas 패키지 소개

dataframe

데이터의 갱신, 추가, 삭제

기존

특성	지역	2015	2010	2005	2000	2010-2015 증가율
도시						
서울	수도권	9904312	9631482	9762546	9853972	0.0283
부산	경상권	3448737	3393191	3512547	3655437	0.0163
인천	수도권	2890451	2632035	2517680	2466338	0.0982
대구	경상권	2466052	2431774	2456016	2473990	0.0141

갱신

```
df["2010-2015 증가율"] = df["2010-2015 증가율"] * 100
df
```

특성	지역	2015	2010	2005	2000	2010-2015 증가율
도시						
서울	수도권	9904312	9631482	9762546	9853972	2.83
부산	경상권	3448737	3393191	3512547	3655437	1.63
인천	수도권	2890451	2632035	2517680	2466338	9.82
대구	경상권	2466052	2431774	2456016	2473990	1.41

추가

```
df["2005-2010 증가율"] = ((df["2010"] - df["2005"]) / df["2005"] * 100).round(2)
df
```

특성	지역	2015	2010	2005	2000	2010-2015 증가율	2005-2010 증가율
도시							
서울	수도권	9904312	9631482	9762546	9853972	2.83	-1.34
부산	경상권	3448737	3393191	3512547	3655437	1.63	-3.40
인천	수도권	2890451	2632035	2517680	2466338	9.82	4.54
대구	경상권	2466052	2431774	2456016	2473990	1.41	-0.99

삭제

```
del df["2010-2015 증가율"]
df
```

특성	지역	2015	2010	2005	2000	2005-2010 증가율
도시						
서울	수도권	9904312	9631482	9762546	9853972	-1.34
부산	경상권	3448737	3393191	3512547	3655437	-3.40
인천	수도권	2890451	2632035	2517680	2466338	4.54
대구	경상권	2466052	2431774	2456016	2473990	-0.99

열 인덱싱

```
# 하나의 열만 인덱싱하면 시리즈가 반환된다.
df["지역"]
```

```
도시
서울      수도권
부산      경상권
인천      수도권
대구      경상권
Name: 지역, dtype: object
```

```
# 여러개의 열을 인덱싱하면 부분적인 데이터프레임이 반환된다.
df[["2010", "2015"]]
```

특성	2010	2015
도시		
서울	9631482	9904312
부산	3393191	3448737
인천	2632035	2890451
대구	2431774	2466052

```
# 2010이라는 열을 반환하면서 데이터프레임 자료형을 유지
df[["2010"]]
```

특성	2010
도시	
서울	9631482
부산	3393191
인천	2632035
대구	2431774

df["지역"]
하나의 열만 인덱싱하면 시리즈 반환

df[["2010","2015"]]
여러개 열 인덱싱하면,
부분적인 데이터프레임 반환

df[["2010"]]
2010이라는 열을 반환하면서,
데이터프레임 자료형 유지

행 인덱싱

series에서의 인덱싱 방법과 동일

인덱스 라벨을 사용하여 인덱싱 슬라이싱 가능 (문자열 라벨을 이용한 슬라이싱은 콜론 뒤 값도 포함)

ex) df[1:2] #1행 출력

df["서울": "부산"] #인덱스 라벨 서울~부산(포함)까지 출력

개별 데이터 인덱싱

df

특성	지역	2015	2010	2005	2000
도시					
서울	수도권	9904312	9631482	9762546	9853972
부산	경상권	3448737	3393191	3512547	3655437
인천	수도권	2890451	2632035	2517680	2466338
대구	경상권	2466052	2431774	2456016	2473990

df["2015"]["서울"]

출력결과) 9904312

loc

라벨 값 기반의 2차원 인덱싱

ex)

df.loc[행 인덱싱 값]

df.loc[행 인덱싱 값, 열 인덱싱 값]

df

	A	B	C	D
a	10	11	12	13
b	14	15	16	17
c	18	19	20	21

	A	B	C	D
c	18	19	20	21

df.loc[df.A>15]

	B	D
a	11	13
b	15	17

df.loc[["a","b"],["B","D"]]

iloc

순서를 나타내는 정수 기반의 2차원 인덱싱

ex)

df.iloc[행 인덱싱 값]

df.iloc[행 인덱싱 값, 열 인덱싱 값]

df

	A	B	C	D
a	10	11	12	13
b	14	15	16	17
c	18	19	20	21

```
A    18
B    19
C    20
D    21
Name: c, dtype: int64
```

df.iloc[-1]

	B	C
c	19	20

df.iloc[2:3,1:3]

03 데이터프레임의 데이터 조작

데이터 개수와 카테고리 값 세기

count()

데이터 개수 셀 때 사용

NaN 값은 세지 않아, 데이터에서 누락된 부분 찾을 때 유용

데이터 프레임에서는 각 열마다 별도로 데이터 개수 센다.

value_counts()

카테고리 값 셀 때 사용

데이터프레임에서는 각 열마다 별도로 적용해야 한다. ex) df[0].value_counts()

sort_index()

인덱스 값을 기준으로 정렬

sort_values()

데이터 값을 기준으로 정렬

NaN 값이 있는 경우, NaN 값이 가장 마지막으로 간다.

3. matplotlib

01 matplotlib 소개

02 matplotlib의 여러가지 plot

matplotlib

파이썬에서 자료를 차트나 플롯으로 시각화하는 패키지

시각화의 장점

많은 양의 데이터를 한눈에 파악 가능하다.
누구나 쉽게 데이터 인사이트를 찾을 수 있다.
정확한 데이터 분석 결과를 도출할 수 있다.
효과적인 데이터 인사이트 공유로 데이터 기반의 의사결정을 할 수 있다.

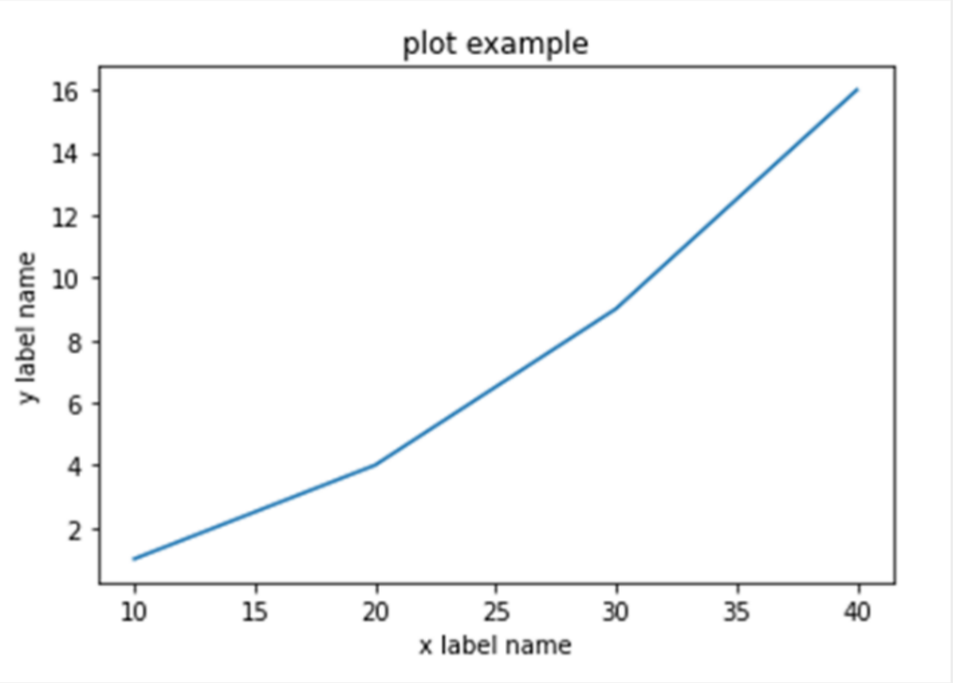
line plot

시간, 순서에 따라

데이터가 어떻게 변화하는지 보여주기 위해 사용

* title : 제목을 표시
* xlabel : x축 이름을 표시
* ylabel : y축 이름을 표시

```
import matplotlib.pyplot as plt
plt.title("plot example")
plt.xlabel("x label name")
plt.ylabel("y label name")
plt.plot([10, 20, 30, 40], [1, 4, 9, 16])
plt.show()
```



import matplotlib.pyplot as plt

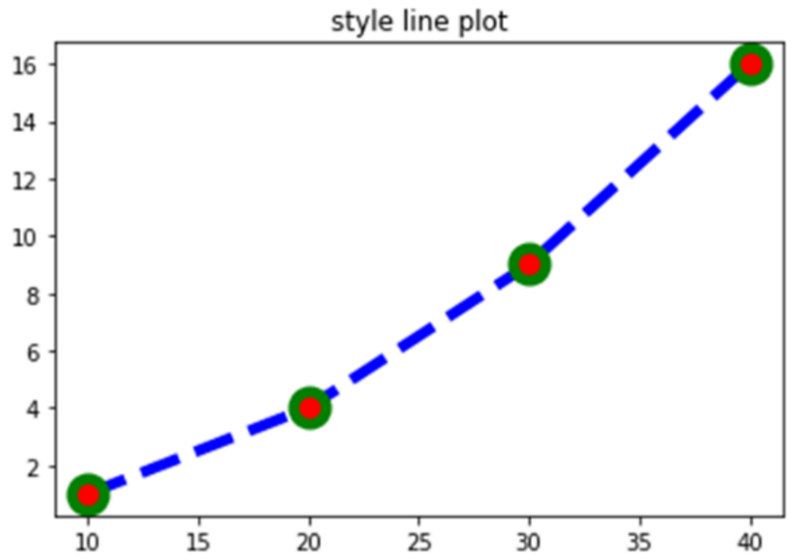
plt.plot([10,20,30,40],[1,4,9,16])

plt.show()

line plot

* c : 색깔지정
* marker : 데이터 위치 나타냄
* ls: 선 스타일 (line style)
* lw: 선 굵기
* ms: 마커 크기 (marker size)
* mec: 마커 선 색깔
* mew: 마커 선 굵기
* mfc: 마커 내부 색깔

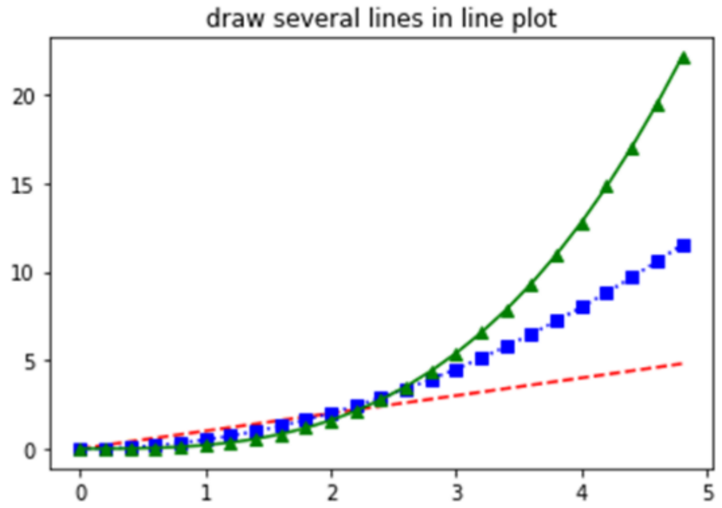
```
import matplotlib.pyplot as plt
plt.plot([10, 20, 30, 40], [1, 4, 9, 16], c="b",
         lw=5, ls="--", marker="o", ms=15, mec="g", mew=5, mfc="r")

plt.title("style line plot")
plt.show()
```



plt.plot([10,20,30,40],[1,4,9,16],c="b",
lw=5,ls="--",marker="o",ms=15,mec="g",
mew=5,mfc="r")

line plot

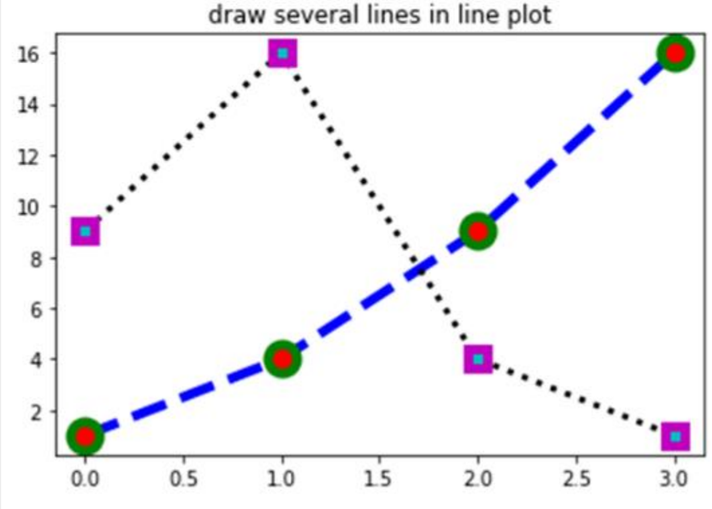
```
import matplotlib.pyplot as plt
import numpy as np
t = np.arange(0., 5., 0.2)
plt.title("draw several lines in line plot")
plt.plot(t, t, 'r--', t, 0.5 * t**2, 'bs:', t, 0.2 * t**3, 'g^-')
plt.show()
```



t=np.arrange(0.,5.,0.2)

plt.plot(t,t,"r--",t,0.5*t**2,"bs:",t,0.2*t**3,"g^-")

```
import matplotlib.pyplot as plt
plt.title("draw several lines in line plot")
plt.plot([1, 4, 9, 16],
         c="b", lw=5, ls="--", marker="o", ms=15, mec="g", mew=5, mfc="r")
plt.plot([9, 16, 4, 1],
         c="k", lw=3, ls=":", marker="s", ms=10, mec="m", mew=5, mfc="c")
plt.show()
```



plt.plot([1, 4, 9, 16],
c="b", lw=5, ls="--", marker="o", ms=15, mec="g", mew=5, mfc="r")
plt.plot([9, 16, 4, 1],
c="k", lw=3, ls=":", marker="s", ms=10, mec="m", mew=5, mfc="c")

02 matplotlib의 여러가지 plot

bar chart

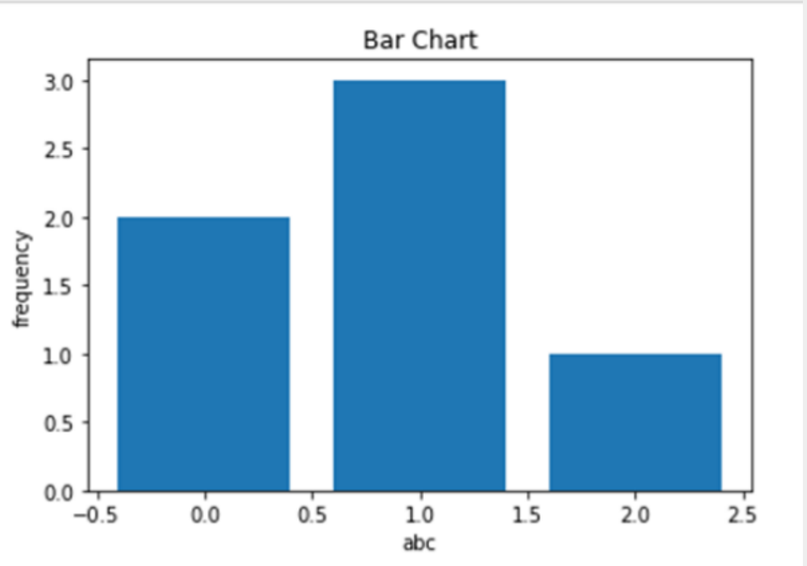
bar chart

x 데이터가 카테고리 값인 경우,

bar이나 barh로 bar chart 시각화

```
import matplotlib as mpl
import matplotlib.pylab as plt

y = [2, 3, 1]
x = np.arange(len(y))
xlabel = ['A', 'B', 'C']
plt.title("Bar Chart")
plt.bar(x, y)
plt.xlabel("abc")
plt.ylabel("frequency")
plt.show()
```

bar



plt.bar(x,y)

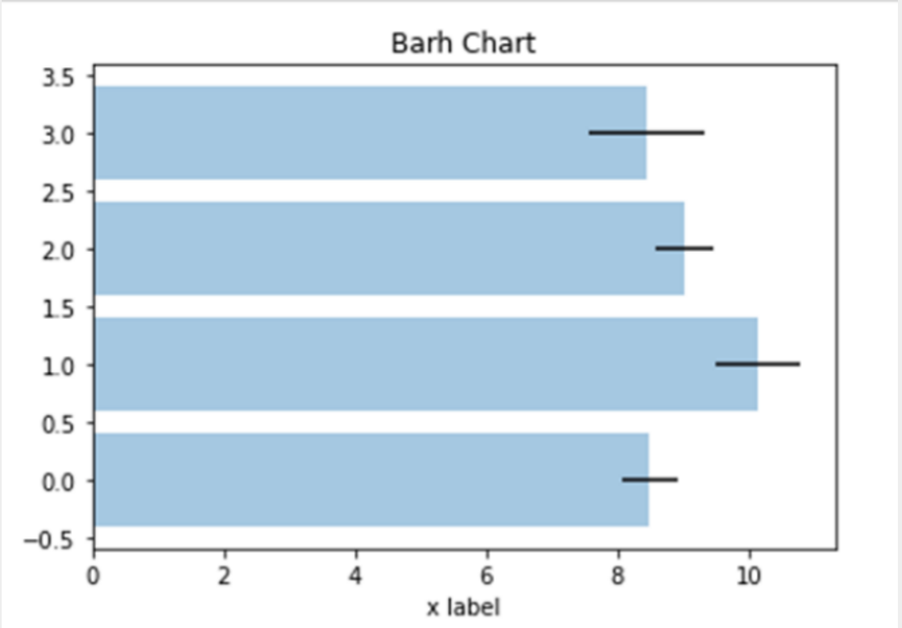
bar chart

x 데이터가 카테고리 값인 경우,

bar이나 barh로 bar chart 시각화

barh

```
people = ['A', 'B', 'C', 'D']
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))
plt.title("Barh Chart")
plt.barh(y_pos, performance, xerr=error, alpha=0.4)
plt.xlabel('x label')
plt.show()
```



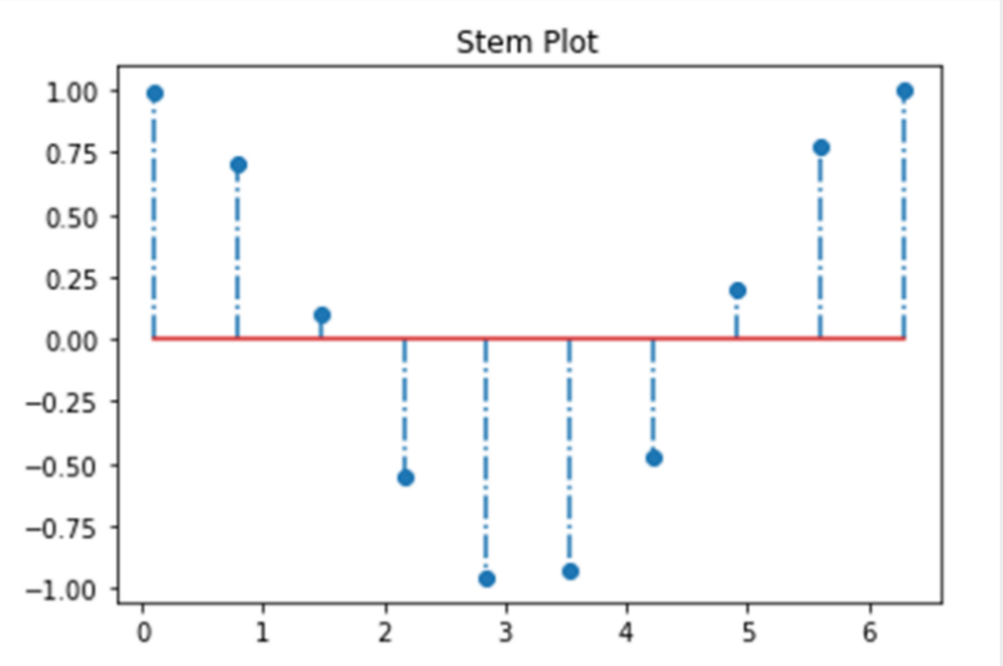
plt.barh(y_pos,performance,xerr=error, alpha=0.4)

stem plot

bar chart와 유사하지만, 폭이 없다.

주로 이산확률 함수나 자기 상관관계 묘사에 사용

```
import matplotlib.pyplot as plt
x = np.linspace(0.1, 2 * np.pi, 10)
plt.title("Stem Plot")
plt.stem(x, np.cos(x), '-.')
plt.show()
```



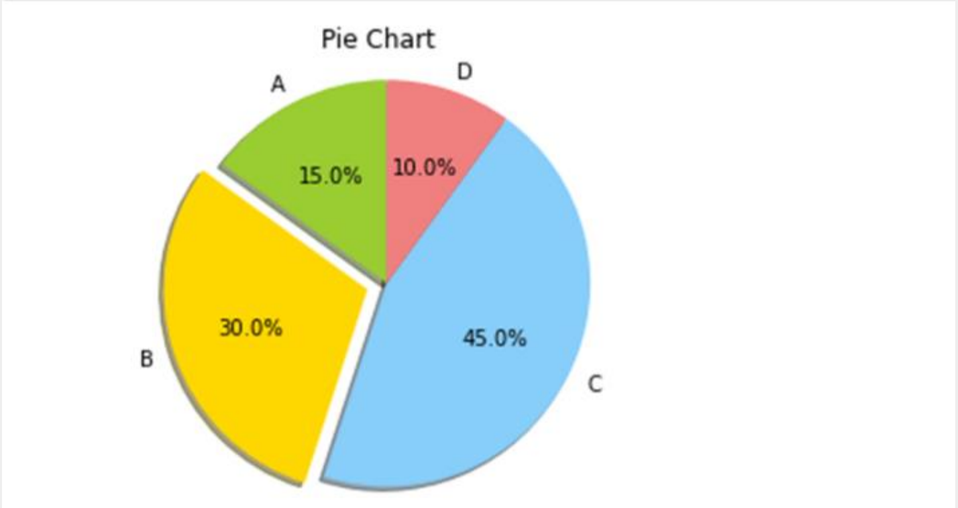
plt.stem(x,np.cos(x),'-.')

pie chart

카테고리 별, 상대적인 비교에 사용

* explode:
  부채꼴이 파이 차트의 중심에서 벗어나는 정도
* autopct : 부채꼴 안에 표시될 숫자 형식 지정
* shadow: true 설정시 파이 차트 그림자 표시
* startangle: 부채꼴 그려지는 시작 각도 설정

```
labels = ['A', 'B', 'C', 'D']
sizes = [15, 30, 45, 10]
colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
explode = (0, 0.1, 0, 0)
plt.title("Pie Chart")
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%1.1f%%', shadow=True, startangle=90)
plt.axis('equal')
plt.show()
```



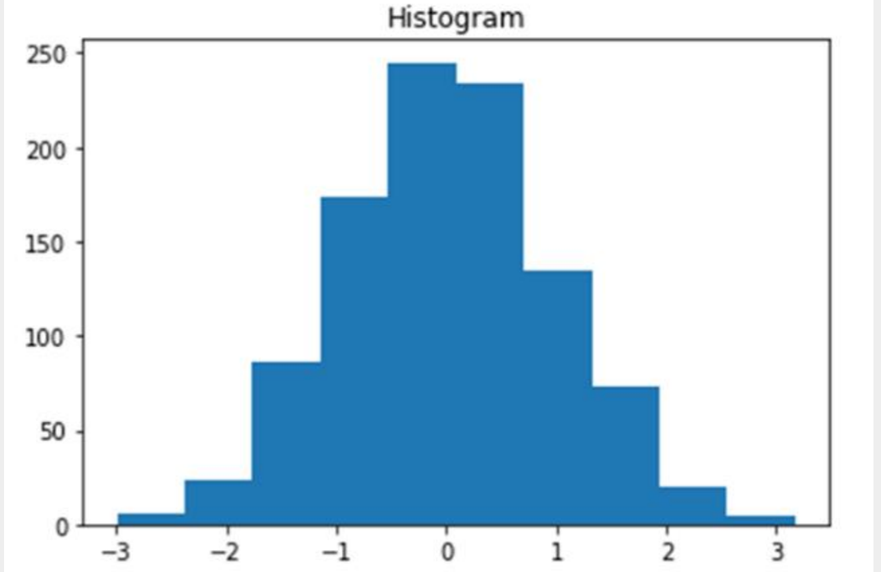
plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=90)

histogram

도수분포표를 그래프로 나타낸 것

주로 연속형 데이터 분포 파악에 사용

```
x = np.random.randn(1000)
plt.title("Histogram")
plt.hist(x, bins=10)
plt.show()
```

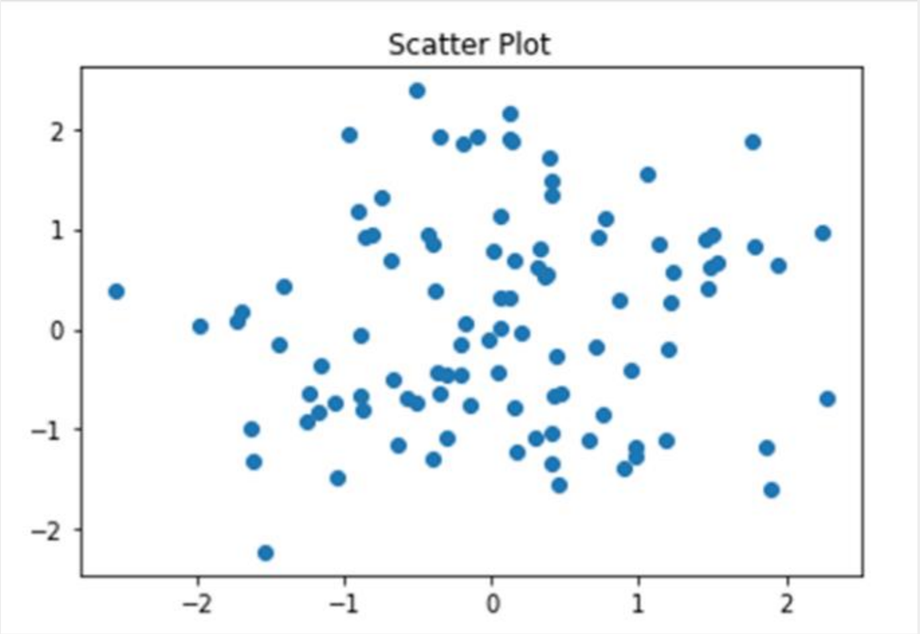


plt.hist(x, bins=10)

scatter plot

두 개의 실수 데이터 집합의 상관관계 살펴볼 때 사용

```
X = np.random.normal(0, 1, 100)
Y = np.random.normal(0, 1, 100)
plt.title("Scatter Plot")
plt.scatter(X, Y)
plt.show()
```



plt.scatter(X,Y)

실습

도미와 빙어 분류 (p57)

목표

도미와 빙어의 생선 특징을 기준으로 자동 분류해주는 머신러닝 프로그램 설계

사용 데이터

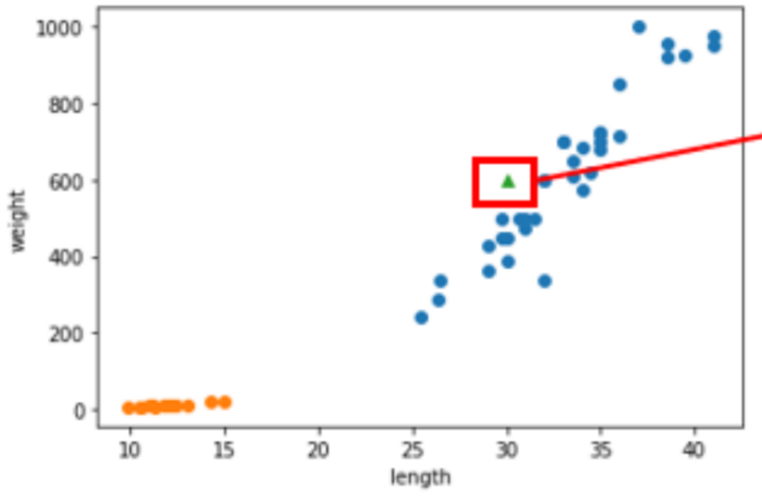
도미, 빙어

(참고: 캐글 공개 데이터셋 : https://www.kaggle.com/aungpyaeap/fish-market)

사용 알고리즘

k-최근접 이웃 알고리즘

Scikit-learn의 k-최근접 이웃 알고리즘은 주변에서 가장 가까운 5개의 데이터를 보고,
다수결의 원칙에 따라 데이터를 예측하는 알고리즘



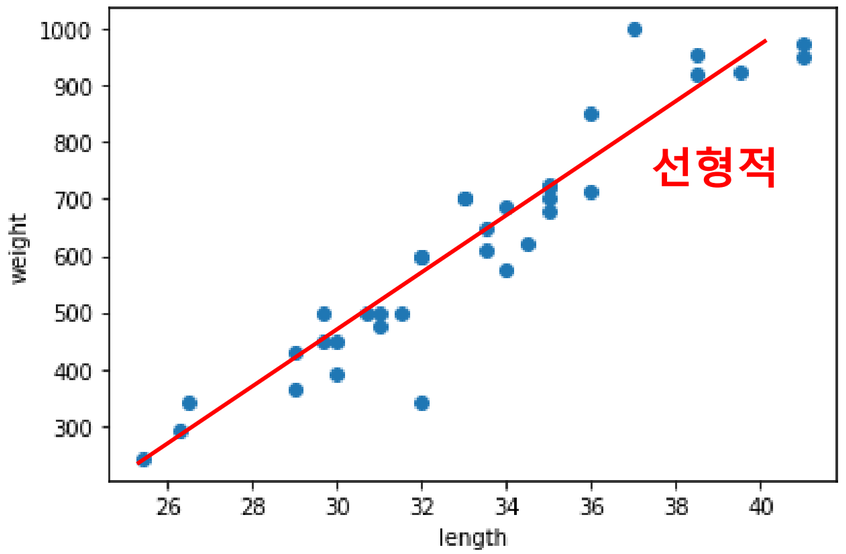
어떤 데이터라고 예측?

파란색 동그라미 vs 주황색 동그라미

02 도미 데이터 준비하기

```
bream_length = [25.4, 26.3, 26.5, 29.0, 29.0,
29.7, 29.7, 30.0, 30.0, 30.7, 31.0, 31.0, 31.5
, 32.0, 32.0, 32.0, 33.0, 33.0, 33.5, 33.5, 34
.0, 34.0, 34.5, 35.0, 35.0, 35.0, 35.0, 36.0,
36.0, 37.0, 38.5, 38.5, 39.5, 41.0, 41.0]

bream_weight = [242.0, 290.0, 340.0, 363.0, 43
0.0, 450.0, 500.0, 390.0, 450.0, 500.0, 475.0,
 500.0, 500.0, 340.0, 600.0, 600.0, 700.0, 700
.0, 610.0, 650.0, 575.0, 685.0, 620.0, 680.0,
700.0, 725.0, 720.0, 714.0, 850.0, 1000.0, 920
.0, 955.0, 925.0, 975.0, 950.0]
```

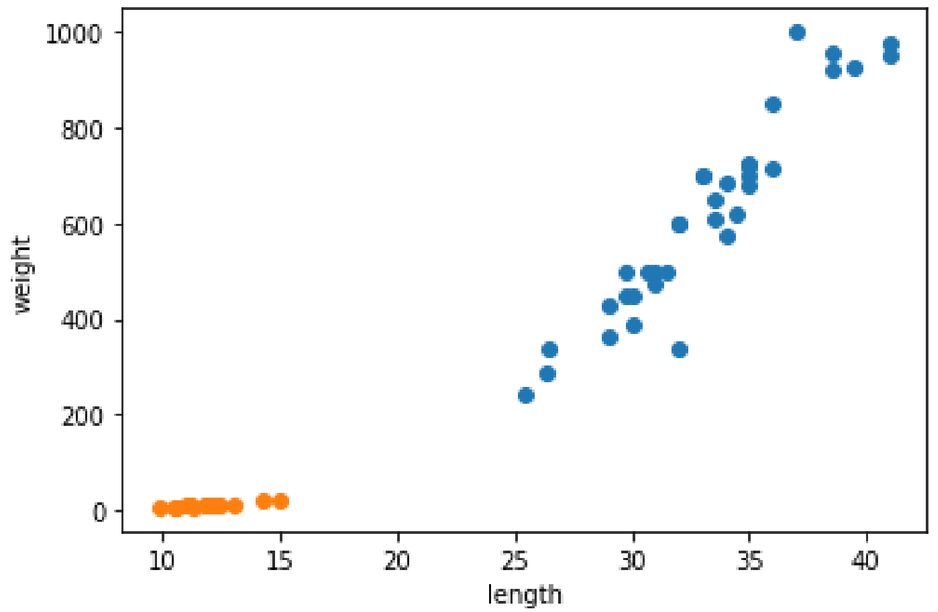
```
import matplotlib.pyplot as plt

plt.scatter(bream_length, bream_weight)
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```



03 빙어 데이터 준비하기

```
smelt_length = [9.8, 10.5, 10.6, 11.0, 11.2,
11.3, 11.8, 11.8, 12.0, 12.2, 12.4, 13.0, 14.
3, 15.0]

smelt_weight = [6.7, 7.5, 7.0, 9.7, 9.8, 8.7,
 10.0, 9.9, 9.8, 12.2, 13.4, 12.2, 19.7, 19.9]
```

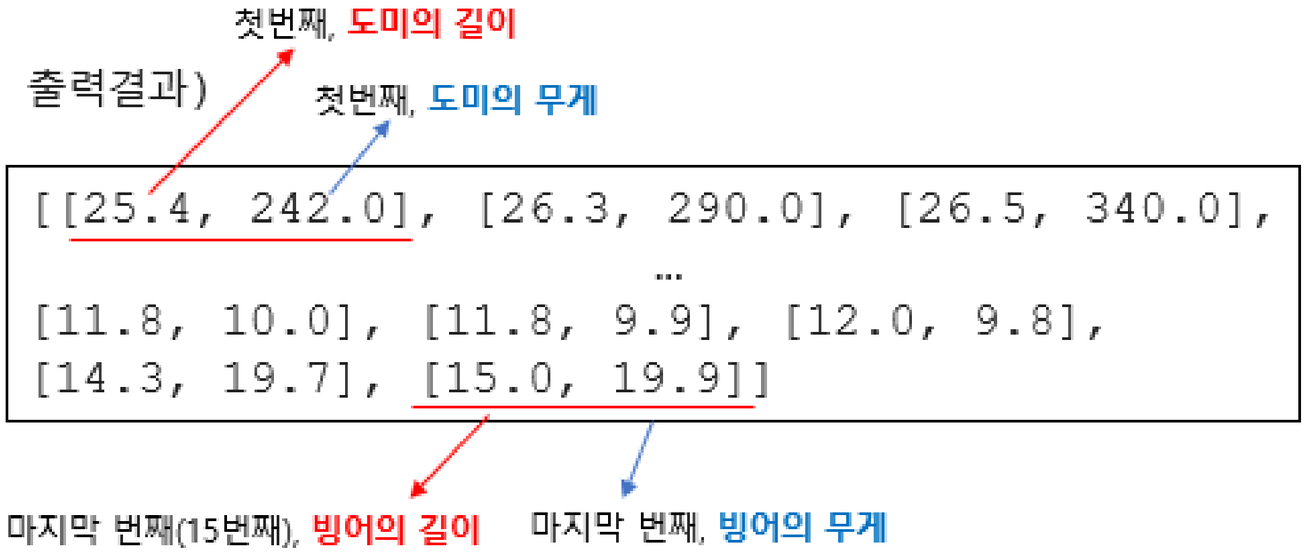
```
import matplotlib.pyplot as plt

plt.scatter(bream_length, bream_weight)
plt.scatter(smelt_length, smelt_weight)
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```



04 정답 데이터 준비하기

```
length = bream_length+smelt_length
weight = bream_weight+smelt_weight

fish_data = [[l, w] for l, w in zip(length, weight)]
```

```
fish_target = [1]*35 + [0]*14
print(fish_target)
```



출력결과)

```
[1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0]
```

04 k-최근접 이웃 알고리즘 추가 설명

가장 가까운 **5개**의 데이터를 보고,
다수결의 원칙에 따라 데이터를 예측

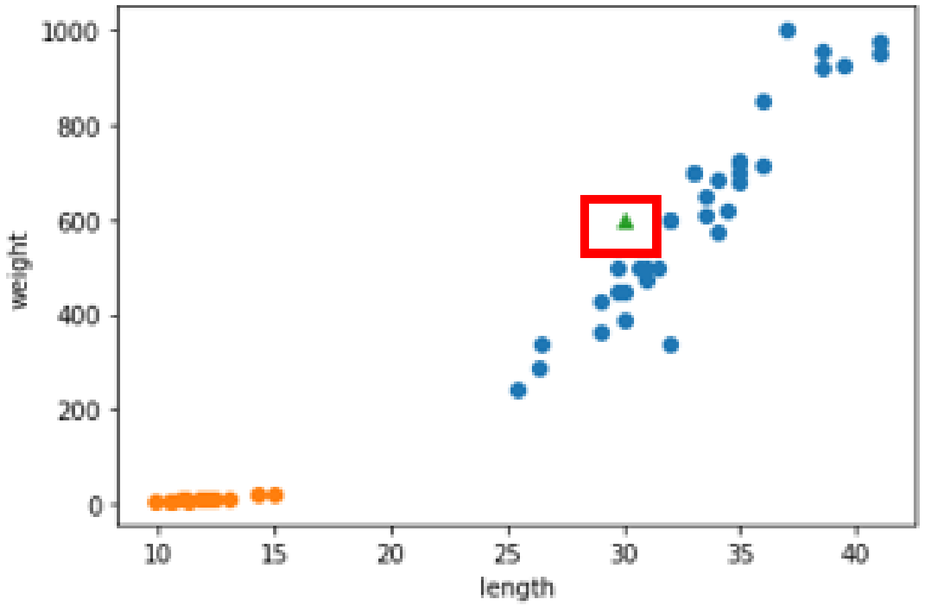
```
from sklearn.neighbors import KNeighborsClassifier

kn = KNeighborsClassifier()
kn.fit(fish_data, fish_target)
kn.score(fish_data, fish_target)
```

가장 가까운 **49개**의 데이터를 보고,
다수결의 원칙에 따라 데이터를 예측

```
from sklearn.neighbors import KNeighborsClassifier

kn = KNeighborsClassifier(n_neighbors=49)
kn.fit(fish_data, fish_target)
kn.score(fish_data, fish_target)
```

05 결론

```
plt.scatter(bream_length, bream_weight)
plt.scatter(smelt_length, smelt_weight)
plt.scatter(30, 600, marker='^')
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```




```
kn.predict([[30, 600]])
```

출력결과)

```
array([1])
```

Q & A

감사합니다!