NUMPY, PANDAS, MATPLOTLIB

Mathematics for Artificial Intelligence

School of IT Convergence Eng.
Intelligent Syst.

Heeseok Oh

(ohhs@hansung.ac.kr)

References:

- 김도형의 데이터 사이언스 스쿨 - 한빛미디어
  https://datascienceschool.net/

Module

Module

❖ 모듈(Module) (1/2)

- 변수, 함수, 클래스 등을 모아둔 것
- 다른 프로그램에 호출될 변수, 함수, 클래스 등을 별도 파일로 제작

- 모듈을 불러오는 법
  - 모듈을 불러올 스크립트에 다음과 같은 코드를 추가한다

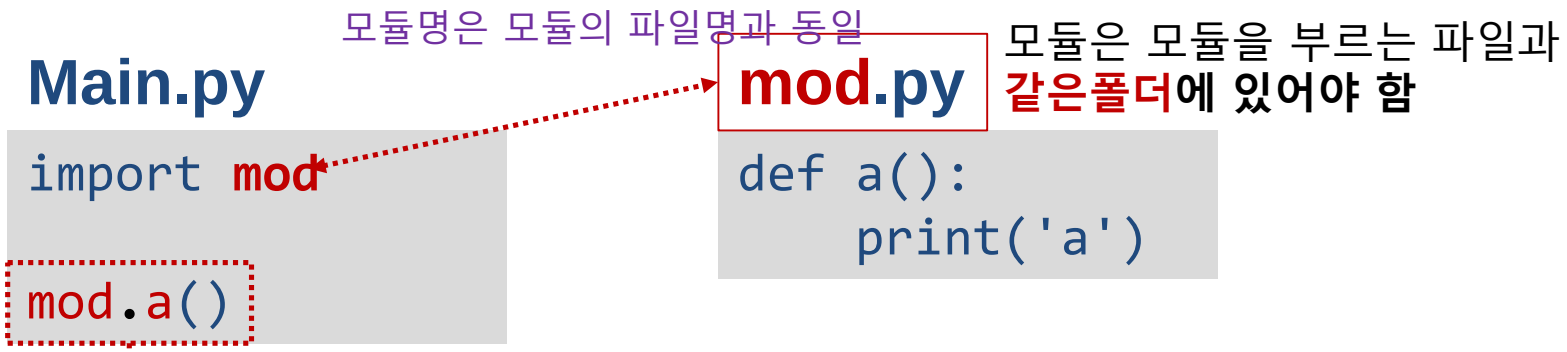
```
import 모듈이름
```

```
import 모듈이름 as 바꿀이름
```

다른 객체와 이름이 겹칠 경우
다른 이름으로 바꿔 불러올 수 있음

❖ 모듈(Module) (2/2)

- 모듈 예제



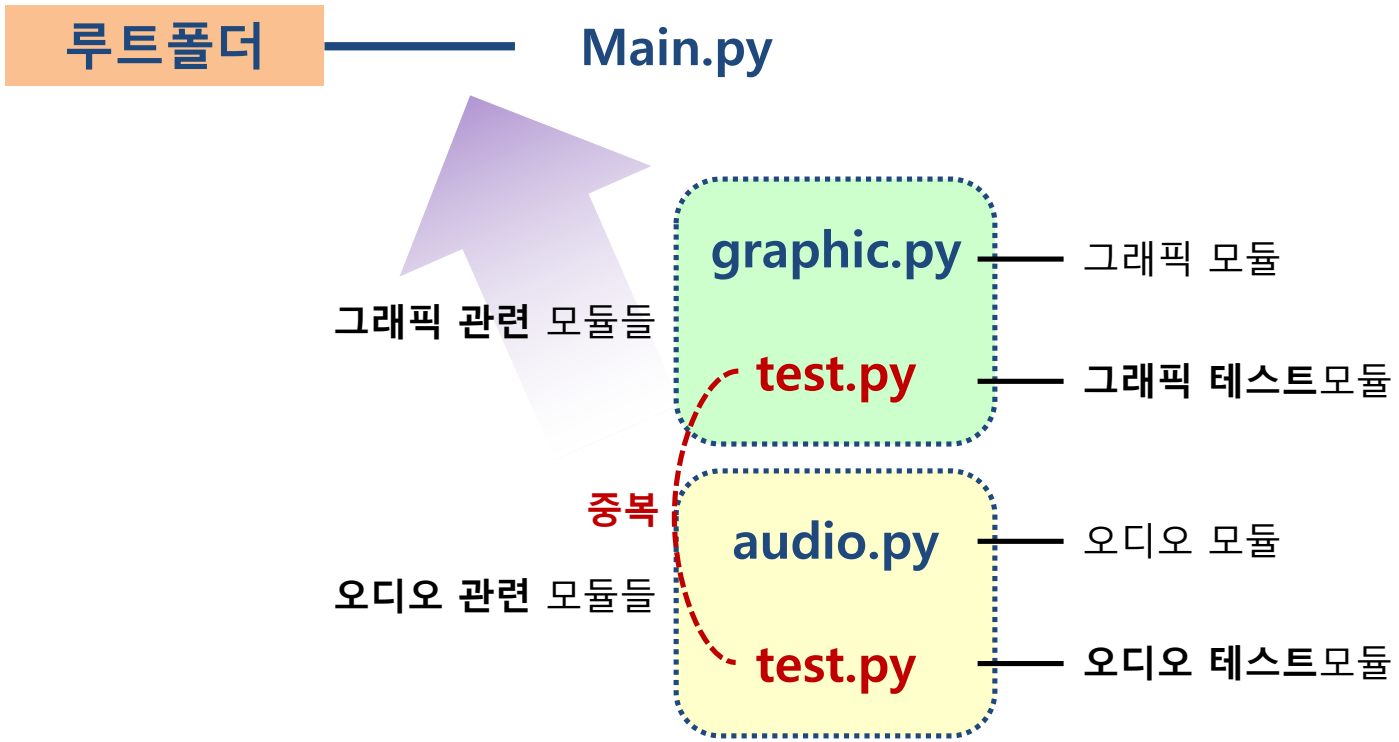
- 결과

```
$ python main.py

 a
```

Package

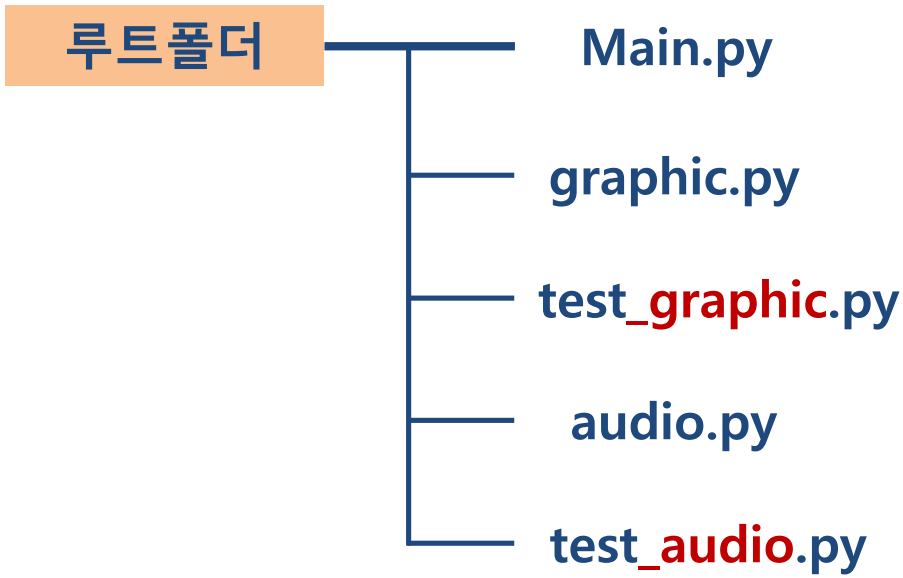
❖ 패키지(Package) (1/5)

- 여러 모듈을 불러쓰다가 모듈이름이 겹치면?



❖ 패키지(Package) (2/5)

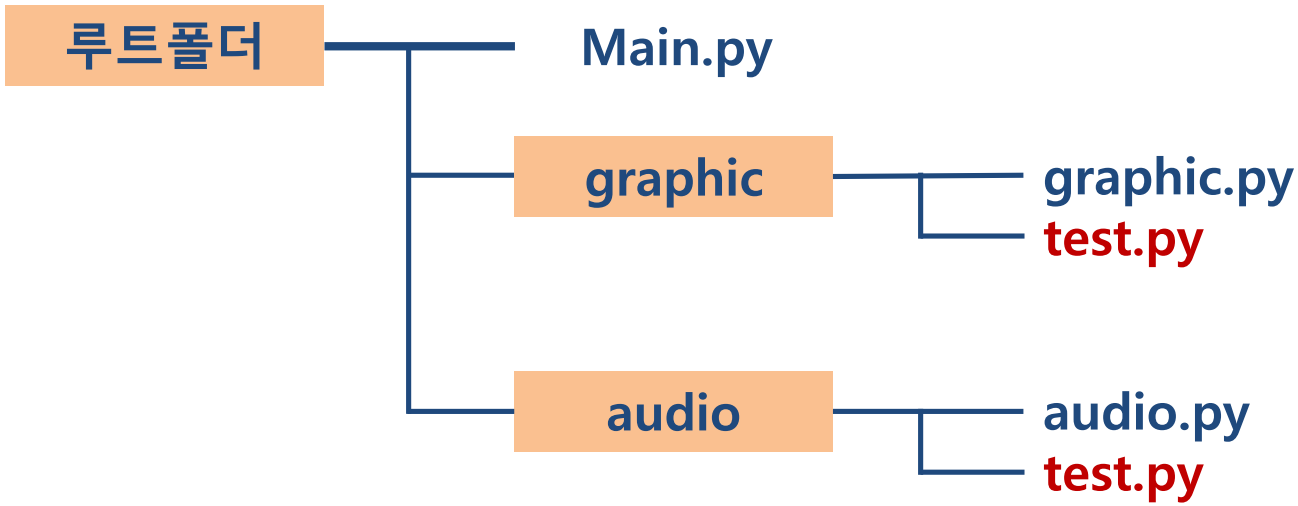
- 여러 모듈을 불러쓰다가 이름이 겹치면?
  - **파일명 바꾸기**



하지만 모든 모듈의 파일명과 내부코드('import 모듈명')를
전부 바꾸는 것은 **번거롭고 비효율적임**

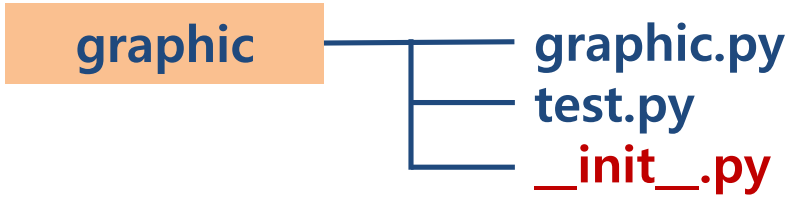
❖ 패키지(Package) (3/5)

- 여러 모듈을 불러쓰다가 이름이 겹치면?
  - **폴더로 구조화하기**



❖ 패키지(Package) (4/5)

- __init__.py의 역할



- 패키지 불러오기(import)가 일어날 때 실행될 내용을 담아둠
  - 예시

graphic₩ __init__.py

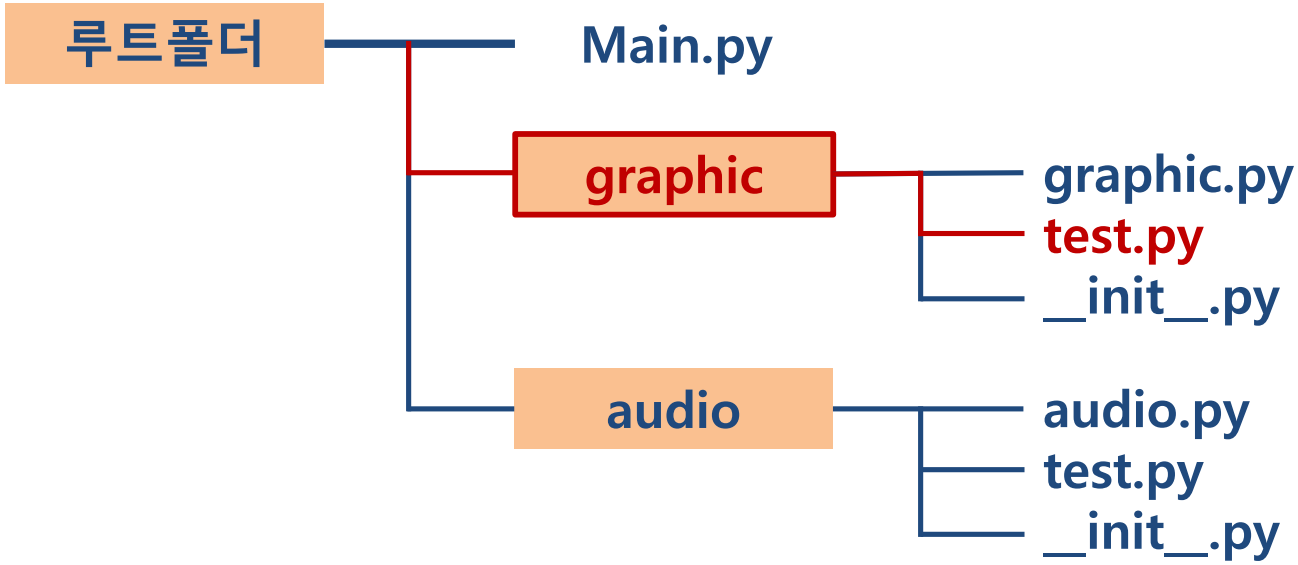
```
print 'import!'
```

```
>>> import graphic
import!
```

import가 **일어나는 순간** __init__.py에 적힌 코드가 **실행**됨

❖ 패키지(Package) (5/5)

- 패키지 예시(패키지 전체 불러오기)



Main.py

```
import graphic
graphic.test.a()
```

graphic 패키지를 불러와

패키지.모듈.함수 형태로 사용

graphic₩ test.py

```
def a():
    print('a')
```

❖ 모듈과 패키지 일부만 불러오기 (1/5)

- 모듈에서 일부 함수 (또는 변수, 클래스)만 필요할 때
- 패키지에서 일부 모듈만 필요할 때

❖ 모듈과 패키지 일부만 불러오기 (2/5)

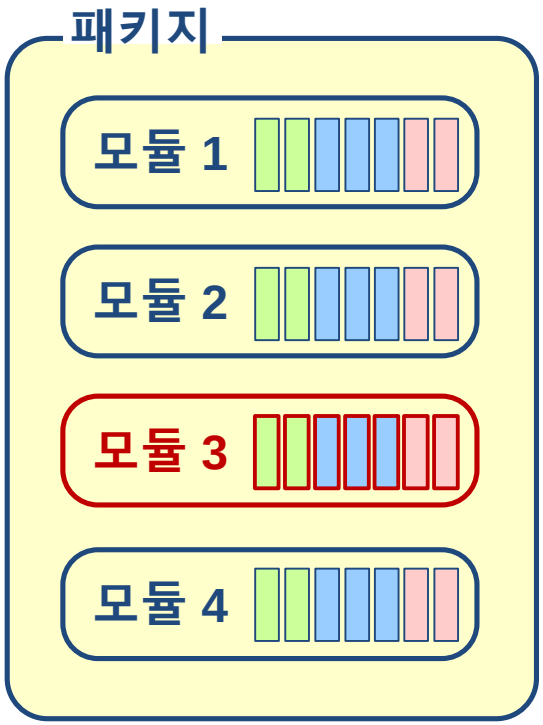
- 모듈에서 일부 함수(또는 변수, 클래스)만 필요할 때

from 모듈 import 함수2

from 모듈 import 함수2 as 바꿀함수명

❖ 모듈과 패키지 일부만 불러오기 (3/5)

- 패키지에서 일부 모듈만 필요할 때

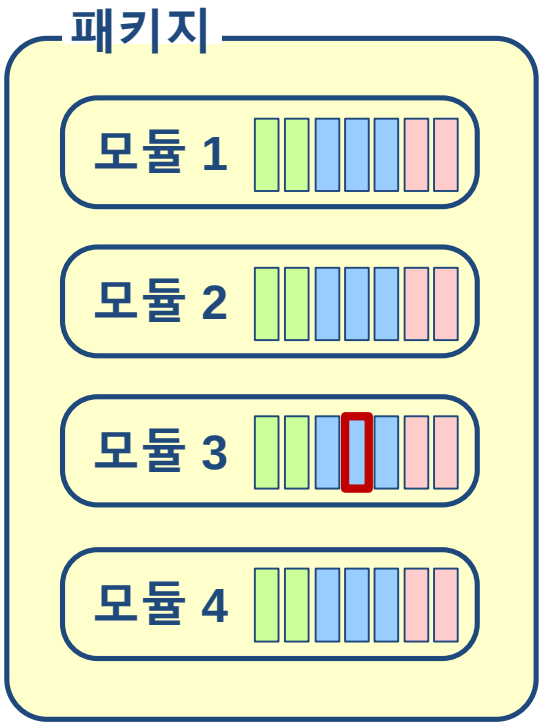



```
from 패키지 import 모듈3
```

```
from 패키지 import 모듈3 as 바꿀모듈명
```

❖ 모듈과 패키지 일부만 불러오기 (4/5)

- 패키지에서 일부 모듈에서 일부 함수(또는 변수, 클래스)만 필요할 때




```
from 패키지.모듈3 import 함수2
```

```
from 패키지.모듈3 import 함수2 as 바꿀함수명
```

❖ 모듈과 패키지 일부만 불러오기 (5/5)

- 모듈을 불러오는 것과 일부 함수만 불러오는 것의 차이점
  - 모듈을 불러와 함수를 호출할 경우

Main.py

```
import mod

mod.a()
```

mod.py

```
def a():
    print('a')
```

  - 모듈에서 함수만 불러와 호출할 경우

Main.py

```
from mod import a

a()
```

함수명 만으로 호출

Package

❖ Asterisk(*)

- from을 이용하여 복수개의 모듈(혹은 함수 등)을 불러옴

```
from 모듈 import *
```

모듈 내의 **모든 변수, 함수, 클래스**를 불러옴

```
from 패키지 import *
```

패키지 내의 **모든 모듈**을 불러옴

import numpy as np

NUMPY

ndarray

❖ 리스트(List)와의 차이

- List는 vector처럼 생겼으나 각종 수학 연산자가 선형 대수의 연산과 같지 않음

```
In [8]: x1 = [0, 1, 2]
        x2 = [3, 4, 5]
        x1 + x2
Out[8]: [0, 1, 2, 3, 4, 5]
```

- 어레이 (Array)
  - 선형대수에서의 벡터 혹은 매트릭스를 대표함
  - +, *, /, -, ** 는 elementwise (원소 단위) 연산을 수행함
  - dot 은 스칼라곱 연산을 수행함
- **Numpy 라이브러리**
  - Python에서의 과학 연산을 위한 기초 패키지
  - N-차원 어레이 객체 연산을 손쉽게 가능

ndarray

❖ 1차원 배열

```
import numpy as np

arr = np.array([0, 2, 3, 4, 5, 6, 7, 8, 9])
print(type(arr))
```

❖ 2차원 배열

```
arr2 = np.array([[0, 1, 2], [3, 4, 5]])  # 2 x 3 array
print(len(arr2))  # row
print(len(arr2[0]))  # column
print(arr2.shape)  # shape
```

❖ 3차원 배열

```
arr3 = np.array([[[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]],
                 [[11, 12, 13, 14],
                  [15, 16, 17, 18],
                  [19, 20, 21, 22]]])   # 2 x 3 x 4 array
print(len(arr3))  # row
print(len(arr3[0]))  # column
print(len(arr3[0][0]))  # depth
print(arr3.shape)  # shape
```

Index

Index

```
arr = np.array([0, 1, 2, 3, 4])
print(arr[2])
print(arr[-1])
print(arr[1] * arr[3])
```

```
arr = np.array([[0, 1, 2], [3, 4, 5]])
print(arr[0, 0])
print(arr[0, 1])
```

Slicing

```
arr = np.array([[0, 1, 2, 3], [4, 5, 6, 7]])
print(arr[0, :])  # [0 1 2 3]
print(arr[:, 1])  # [1 5]
print(arr[1, 1:]) # [5 6 7]
print(arr[:2, :2])
# [[0 1]
# [4 5]]
```

Index를 array로 받을 수 있음

```
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
idx = np.array([True, False, True, False, True,
                False, True, False, True, False])
print(a[idx])  # [0 2 4 6 8]
print(a[a % 2 == 0])  # [0 2 4 6 8]
```

Numpy array 생성

❖ 배열 생성 명령어

- zeros, ones

```
a = np.zeros(5)
b = np.zeros((2, 3))
print(a)  # [0. 0. 0. 0. 0.]
print(b)
# [[0. 0. 0.]
#  [0. 0. 0.]]
```

```
c = np.zeros((2, 5), dtype=int)
print(c)
# [[0 0 0 0 0]
#  [0 0 0 0 0]]
```

```
d = np.ones((2, 5), dtype=int)
print(d)
# [[1 1 1 1 1]
#  [1 1 1 1 1]]
```

- zeros_like, ones_like
  - 크기를 튜플로 명시하지 않고 대상 배열과 같은 크기로 생성

```
d = np.ones((2, 5), dtype=int)
e = np.zeros_like(d)
print(e)
# [[0 0 0 0 0]
#  [0 0 0 0 0]]
```

- arange
  - Numpy에서의 range

```
a = np.arange(10)
b = np.arange(3, 21, 2)
print(a)  # [0 1 2 3 4 5 6 7 8 9]
print(b)  # [ 3  5  7  9 11 13 15 17 19]
```

Numpy array 변형

배열 생성 명령어

- linspace
  - 선형 구간을 지정한 구간의 수만큼 분할

```
a = np.linspace(0, 100, 5)  # 시작, 끝(포함), 갯수
print(a)  # [  0.  25.  50.  75. 100.]
```

배열 크기 변형

- reshape

```
a = np.arange(12)
b = a.reshape(3, 4)
c = a.reshape(3, -1)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
→
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

- flatten
  - 다차원 배열을 무조건 1차원으로 변형

```
a = np.arange(12)
b = a.reshape(3, 4)
c = b.flatten()
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
→
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Numpy array 연결

❖ 배열의 연결 (concatenate)

- hstack: 행의 수가 같은 배열을 옆으로 연결

```
a1 = np.ones((2, 3))
a2 = np.zeros((2, 2))
a = np.hstack([a1, a2])
```

```
[[1. 1. 1.]
 [1. 1. 1.]]

[[0. 0.]
 [0. 0.]]
```

⇨

```
[[1. 1. 1. 0. 0.]
 [1. 1. 1. 0. 0.]]
```

- vstack: 열의 수가 같은 배열을 위아래로 연결

```
b1 = np.ones((2, 3))
b2 = np.zeros((3, 3))
b = np.vstack([b1, b2])
```

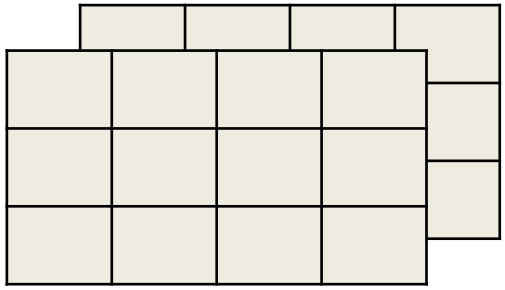
```
[[1. 1. 1.]
 [1. 1. 1.]]

[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

⇨

```
[[1. 1. 1.]
 [1. 1. 1.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

- dstack: 깊이 방향으로 배열을 연결

```
c1 = np.ones((3, 4))
c2 = np.zeros((3, 4))
c = np.dstack([c1, c2])
print(c.shape)  # (3, 4, 2)
```



Numpy array 연결

❖ 배열의 연결 (concatenate)

- tile: 동일한 배열을 반복하여 연결

```
a = np.array([[0, 1, 2], [3, 4, 5]])
b = np.tile(a, (3, 2))
```

```
[[0 1 2]
 [3 4 5]]
```

```
[[0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [0 1 2 0 1 2]
 [3 4 5 3 4 5]]
```

- meshgrid: 그리드 포인트 생성

```
x = np.arange(3)
y = np.arange(5)
X, Y = np.meshgrid(x, y)
grid_points = [list(zip(x, y)) for x, y in zip(X, Y)]
```

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

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

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

Numpy 연산 이용의 장점?

❖ Numpy 연산 vs 반복문 연산

```python
import numpy as np
import time
size_of_vec = 10000000

def pure_python_version():
    t = time.time()
    x = range(size_of_vec)
    y = range(size_of_vec)
    z = []
    for i in range(len(x)):
        z.append(x[i] + y[i])
    return time.time() - t

def numpy_version():
    t = time.time()
    x = np.arange(size_of_vec)
    y = np.arange(size_of_vec)
    z = x + y
    return  time.time() - t

t1 = pure_python_version()
t2 = numpy_version()
print(t1, t2)
print("numpy is in this example " + str(t1/t2) + " faster!")

# numpy is in this example 54.98689637434207 faster!
```

Numpy 연산

❖ 최대, 최소 및 기술통계값

```
x = np.array([1, 2, 3, 4])
print(np.sum(x))
print(x.sum())
print(x.min())
print(x.max())
print(x.argmin())
print(x.argmax())
print(x.mean())
print(np.var(x))
print(np.std(x))
```

```
10
10
1
4
0
3
2.5
2.5
1.25
1.118033988749895
```

```
x = np.array([[1, 1], [2, 2]])
print(x.sum(axis = 0))
print(x.sum(axis = 1))
```

```
[3  3]
[2  4]
```

❖ 정렬

```
a = np.array([[4,  3,  5,  7],
              [1, 12, 11,  9],
              [2, 15,  1, 14]])  # 3x4
print(np.sort(a, axis=0))  # size 3
print(np.sort(a, axis=1))  # size 4
```

```
[[ 1  3  1  7]
 [ 2 12  5  9]
 [ 4 15 11 14]]

[[ 3  4  5  7]
 [ 1  9 11 12]
 [ 1  2 14 15]]
```

```
a = np.array([42, 38, 12, 25])
j = np.argsort(a)
print(j)
print(a[j])
```

```
[2 3 1 0]
[12 25 38 42]
```

Numpy 난수

난수발생

- random.seed

```
print(np.random.rand(5))
print(np.random.rand(5))

np.random.seed(0)
print(np.random.rand(5))
np.random.seed(0)
print(np.random.rand(5))
```

```
[0.56360819 0.5284769  0.7711577  0.30399053 0.39976369]
[0.52194278 0.66054028 0.6439753  0.72283036 0.10043661]

[0.5488135  0.71518937 0.60276338 0.54488318 0.4236548 ]
[0.5488135  0.71518937 0.60276338 0.54488318 0.4236548 ]
```

- rand: 0~1 사이의 균일 분포
- randn: 가우시안 표준 정규 분포
- randint: 균일 분포의 정수 난수

```
print(np.random.rand(5))
print(np.random.randn(5))
print(np.random.randint(low=10, high=20, size=5))
```

```
[0.71096883 0.95499331 0.26769973 0.851815   0.98347745]
[-0.79501454  0.32367464  0.61021112 -1.20333419  1.45846193]
[18 17 19 10 12]
```

Numpy 데이터 카운팅

❖ 데이터 카운팅

- unique

```
index, count = np.unique([11, 11, 2, 2, 34, 34], return_counts=True)
print(index)
print(count)
```

```
[ 2 11 34]
[2 2 2]
```

- bincount
  - 데이터에 존재하지 않아도, 특정 범위 안의 수에서 카운트

```
print(np.bincount([1, 1, 2, 2, 2, 3], minlength=6))
```

```
[0 2 3 1 0 0]
```

import pandas as pd

PANDAS

Pandas

데이터 분석

- 표 데이터를 다루기 위한 Series와 DataFrame 클래스를 제공

Series 클래스

- Numpy 1차원 배열과 비슷하나, 각 데이터의 의미를 나타내는 인덱스가 추가된 형태
- Series = 인덱스 (index) + 값 (value)
- 생성

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

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

- 접근
  - index와 value 속성으로 접근

```
print(s.index)
print(s.values)
```

```
Index(['서울', '부산', '인천', '대구'], dtype='object')
[9904312 3448737 2890451 2466052]
```

Pandas

❖ Series 연산

- Numpy와 같은 벡터 연산이 가능
- Value에만 적용되며 Index 값은 변하지 않음

```
print(s / 1000000)
```

```
서울    9.904312
부산    3.448737
인천    2.890451
대구    2.466052
dtype: float64
```

❖ Series 인덱싱

- Numpy와 같은 인덱싱 + 인덱스 라벨을 이용한 인덱싱

```
print(s[3], s["대구"])
print(s[(250e4 < s) & (s < 500e4)])
```

```
2466052 2466052

부산    3448737
인천    2890451
dtype: int64
```

- 딕셔너리와 유사하게 처리 가능

```
for k, v in s.items():
    print("%s = %d" % (k, v))
```

```
서울 = 9904312
부산 = 3448737
인천 = 2890451
대구 = 2466052
```

Pandas

❖ Series 인덱싱

- 딕셔너리의 경우 원소가 순서를 가지지 않음
- pandas에서는 인덱스를 리스트로 순서를 지정

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

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

❖ Index 기반 연산

- 시리즈 연산을 하는 경우, 같은 인덱스에 대해 연산을 수행

```
s = pd.Series([9904312, 3448737, 2890451, 2466052],
              index=["서울", "부산", "인천", "대구"])
s2 = pd.Series({"서울": 9631482, "부산": 3393191, "인천": 2632035, "대전": 1490158},
               index=["부산", "서울", "인천", "대전"])
print(s - s2)
print(s.values - s2.values)
```

```
대구         NaN : s2에 데이터가 없어 NaN
대전         NaN : s1에 데이터가 없어 Nan
부산     55546.0
서울    272830.0
인천    258416.0
dtype: float64

[ 6511121 -6182745   258416   975894]
```

Pandas

❖ DataFrame 클래스

- 표(2차원 배열)를 생각

```
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)
print(df)
print(df.values)
print(df.columns)
print(df.index)
```

	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

```
[['수도권' 9904312 9631482 9762546 9853972 0.0283]
 ['경상권' 3448737 3393191 3512547 3655437 0.0163]
 ['수도권' 2890451 2632035 2517680 2466338 0.0982]
 ['경상권' 2466052 2431774 2456016 2473990 0.0141]]

Index(['지역', '2015', '2010', '2005', '2000', '2010-2015 증
가율'], dtype='object')

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

Pandas

❖ DataFrame 클래스

- 표(2차원 배열)를 생각
  - Column과 index에 이름을 붙이는 것도 가능

```
df.index.name = "도시"
df.columns.name = "특성“
```

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

- Numpy 배열에서 가능한 대부분의 속성이나 메소드 지원

```
print(df.T)  # transpose
```

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

Pandas

❖ DataFrame 클래스

- 열 인덱싱

```
print(df["지역"])
print(df[["2010", "2015"]])
```

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

dataframe 유지

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

- 행 인덱싱
  - 항상 slicing을 하여 접근

```
print(df[1:2])
print(df["서울":"부산"])
```

특성	지역	2015	2010	2005	2000	2005-2010 증가율
도시						
부산	경상권	3448737	3393191	3512547	3655437	-3.4

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

- 개별 데이터 인덱싱

```
print(df["2015"]["서울"])
```

9904312

Pandas

❖ DataFrame의 numpy식 인덱싱

- loc 인덱서

```
import numpy as np
import pandas as pd

df = pd.DataFrame(np.arange(10, 22).reshape(3, 4),
                  index=["a", "b", "c"],
                  columns=["A", "B", "C", "D"])
print(df.loc["a"])
print(df.loc["b":"c"])
print(df.loc[["b", "c"]])
```

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

- **df.loc[행 인덱스, 열 인덱스]** 형태로 이용 가능

```
print(df.loc["a", "A"])
print(df.loc["b":, "A"])
```

```
b    14
c    18
Name: A, dtype: int32
```

Pandas

❖ DataFrame의 numpy식 인덱싱

- iloc 인덱서
  - loc 인덱서와는 달리, 라벨이 아닌 순서 (정수) 인덱스 이용

```
df = pd.DataFrame(np.arange(10, 22).reshape(3, 4),
                  index=["a", "b", "c"],
                  columns=["A", "B", "C", "D"])
print(df.iloc[0, 1])
print(df.iloc[:2, 2])
```

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

```
11

a    12
b    16
Name: C, dtype: int32
```

```
df.iloc[-1] = df.iloc[-1] * 2  # 인덱스가 하나일 경우 "행" 기준
print(df)
```

	A	B	C	D
a	10	11	12	13
b	14	15	16	17
c	36	38	40	42

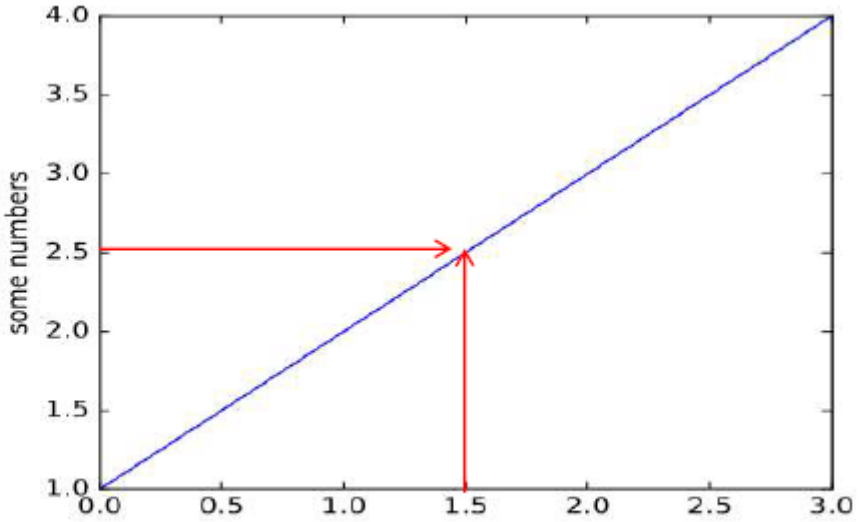
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Matplotlib

Matplotlib.pyplot

- **데이터 시각화**
  - 데이터의 탐색과 전달을 위한 목적
- **Jupyter notebook 이용 시**
  - %matplotlib inline 명령을 실행해야 그래프가 보임
- **좌표의 이해**

Y축



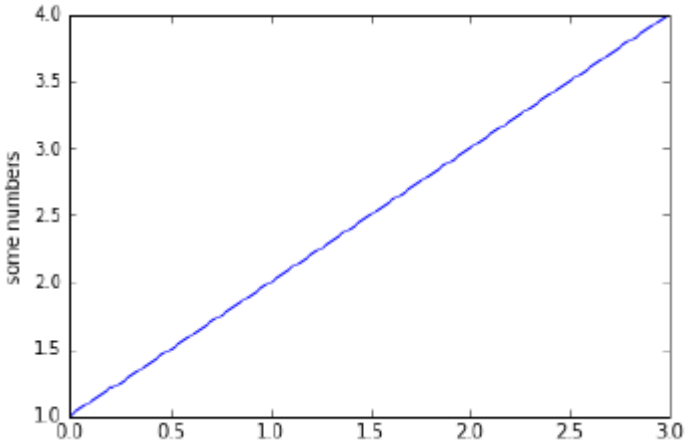
X축

plot

❖ list 입력

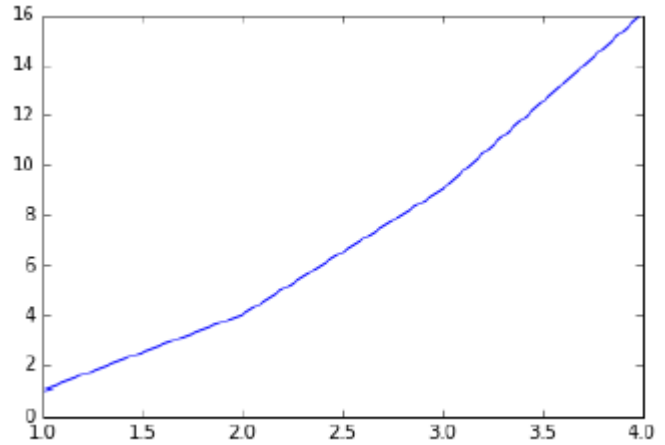
- Index가 x축, 값을 y축으로 인식

```
: import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```



- X축과 y축을 같이 입력

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16])

plt.show()
```

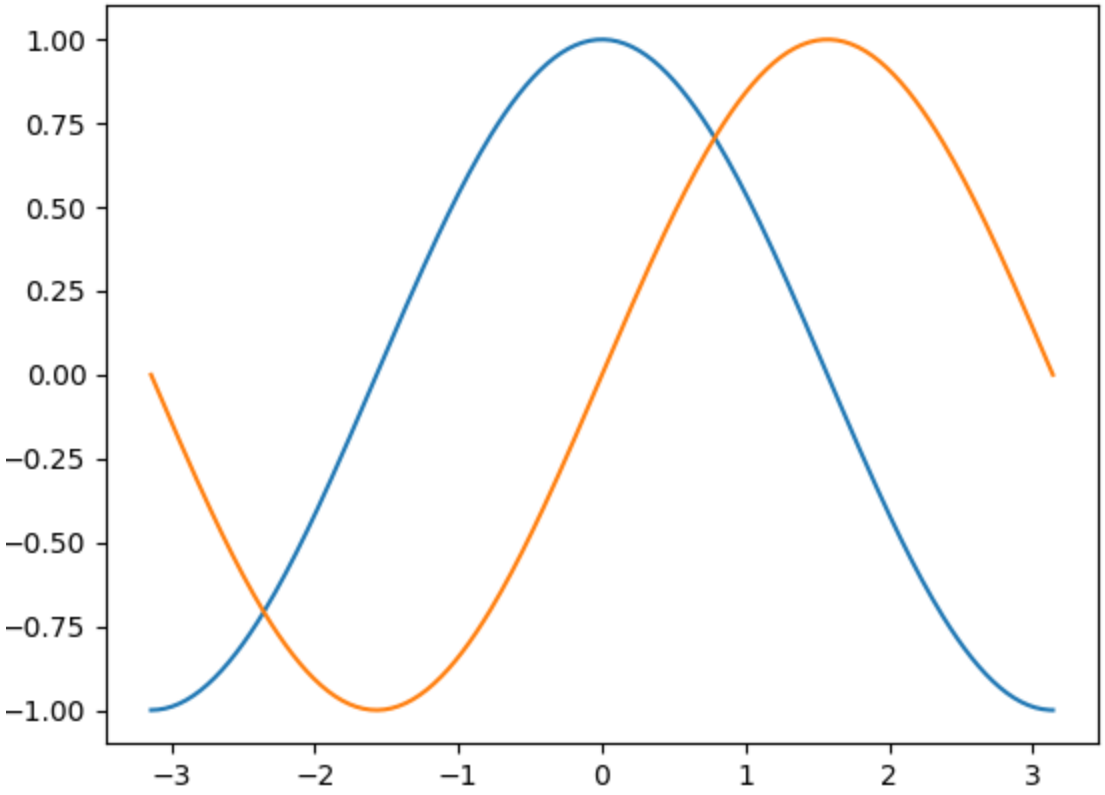


- show: 시각화 명령을 렌더링하고, 입력 이벤트를 기다리라는 지시

plot

- 하나의 figure에 두 개의 그래프 처리

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(x), np.sin(x)
plt.plot(x, C)
plt.plot(x, S)
plt.show()
```



plot

❖ 스타일 지정

- color

문자열	약자
`blue`	`b`
`green`	`g`
`red`	`r`
`cyan`	`c`
`magenta`	`m`
`yellow`	`y`
`black`	`k`
`white`	`w`

plot

❖ 스타일 지정

- marker

`.`	point marker
`,`	pixel marker
`o`	circle marker
`v`	triangle_down marker
`^`	triangle_up marker
`<`	triangle_left marker
`>`	triangle_right marker
`1`	tri_down marker
`2`	tri_up marker
`3`	tri_left marker
`4`	tri_right marker
`s`	square marker
`p`	pentagon marker
`*`	star marker
`h`	hexagon1 marker
`H`	hexagon2 marker
`+`	plus marker
`x`	x marker
`D`	diamond marker
`d`	thin_diamond marker

plot

❖ 스타일 지정

- Line style

선 스타일 문자열	의미
`-`	solid line style
`--`	dashed line style
`-.`	dash-dot line style
`:`	dotted line style

plot

❖ 스타일 지정

- 자주 사용되는 스타일

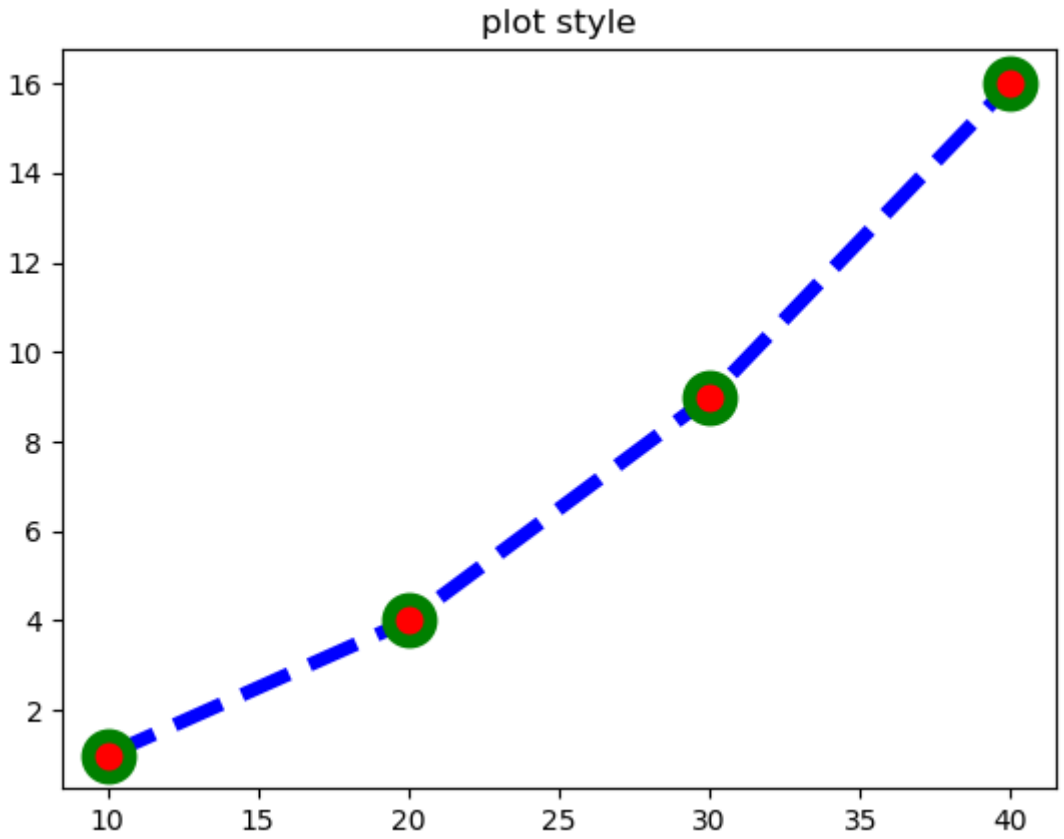
스타일 문자열	약자	의미
`color`	`c`	선 색깔
`linewidth`	`lw`	선 굵기
`linestyle`	`ls`	선 스타일
`marker`		마커 종류
`markersize`	`ms`	마커 크기
`markeredgecolor`	`mec`	마커 선 색깔
`markeredgewidth`	`mew`	마커 선 굵기
`markerfacecolor`	`mfc`	마커 내부 색깔

plot

❖ 스타일 지정

- 자주 사용되는 스타일

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

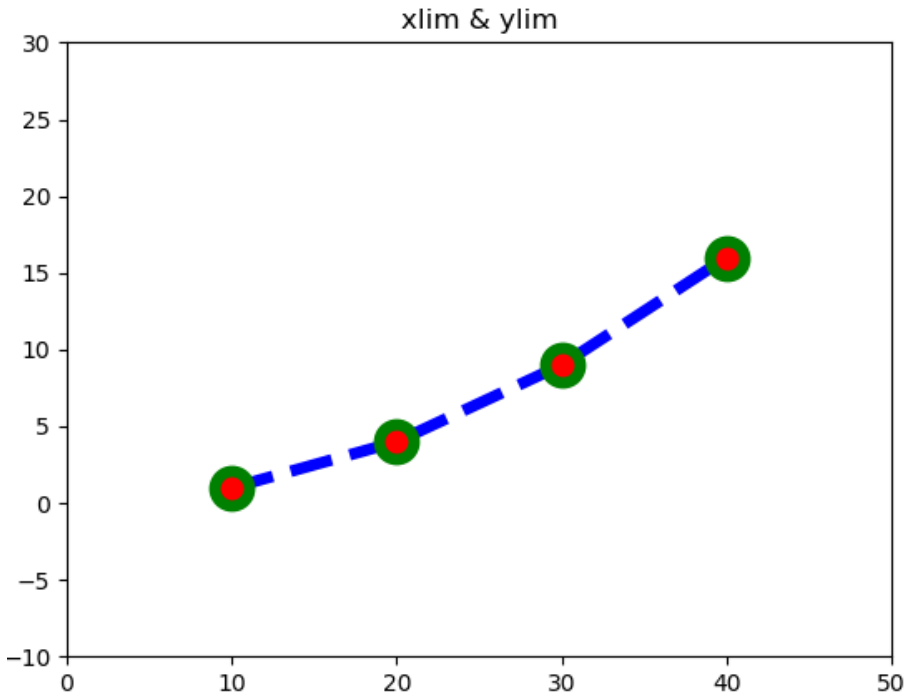


plot

❖ 범위 지정

- X축, y축의 최소값과 최대값을 지정

```
plt.title("xlim & ylim")
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.xlim(0, 50)
plt.ylim(-10, 30)
plt.show()
```

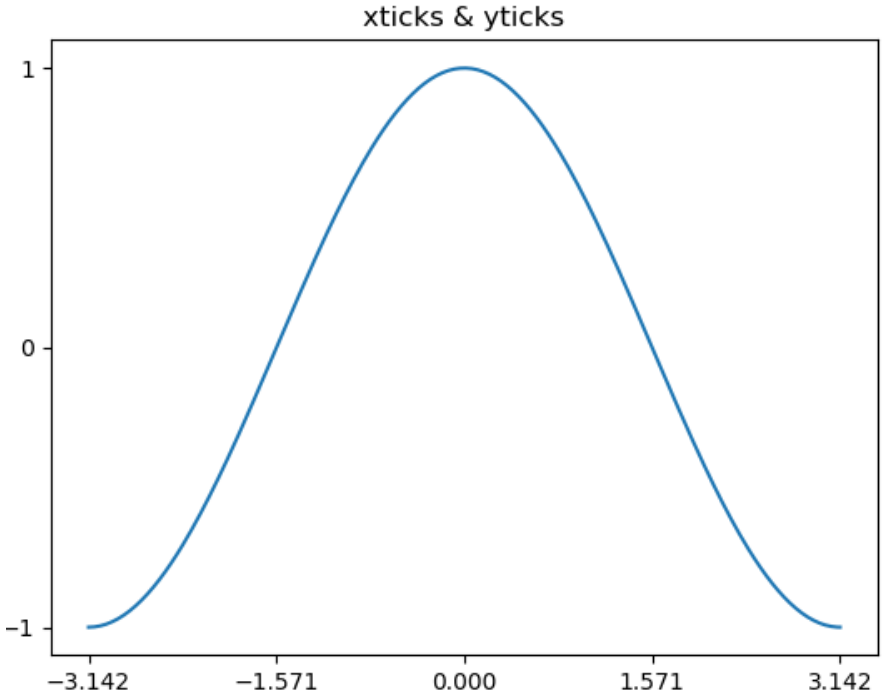


plot

❖ 틱 (tick) 설정

- 축상의 위치 표시 지점 표시

```
X = np.linspace(-np.pi, np.pi, 256)
C = np.cos(X)
plt.title("xticks & yticks")
plt.plot(X, C)
plt.xticks([-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi])
plt.yticks([-1, 0, +1])
plt.show()
```

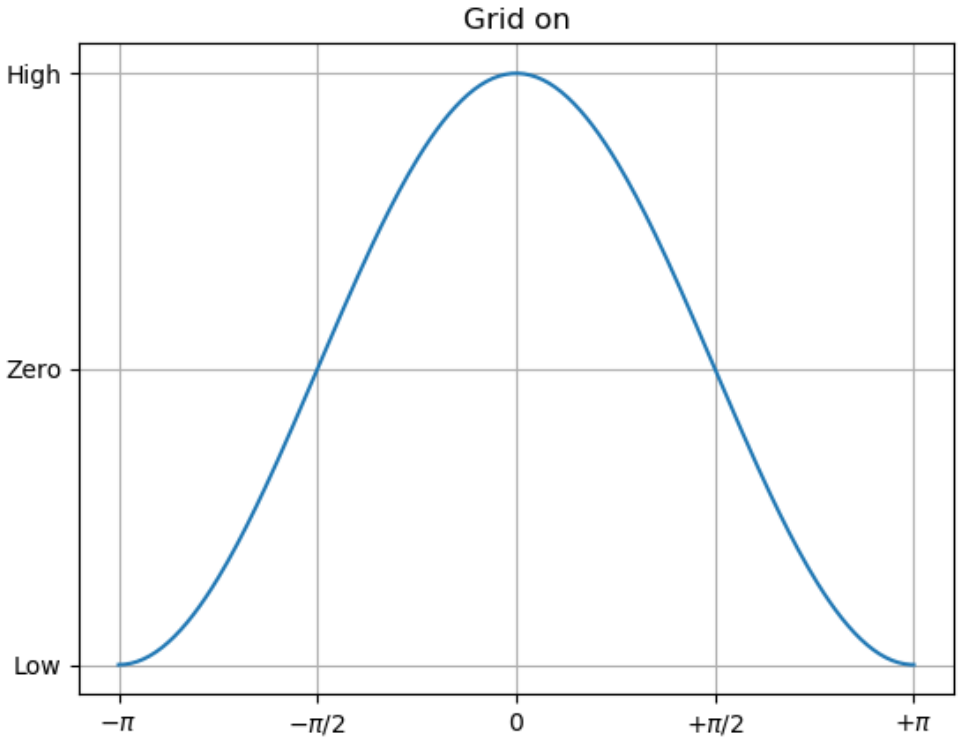


plot

❖ 그리드 (grid) 설정

- Grid line 설정

```
X = np.linspace(-np.pi, np.pi, 256)
C = np.cos(X)
plt.title("Grid on")
plt.plot(X, C)
plt.xticks([-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi], [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
plt.yticks([-1, 0, 1], ["Low", "Zero", "High"])
plt.grid(True)
plt.show()
```

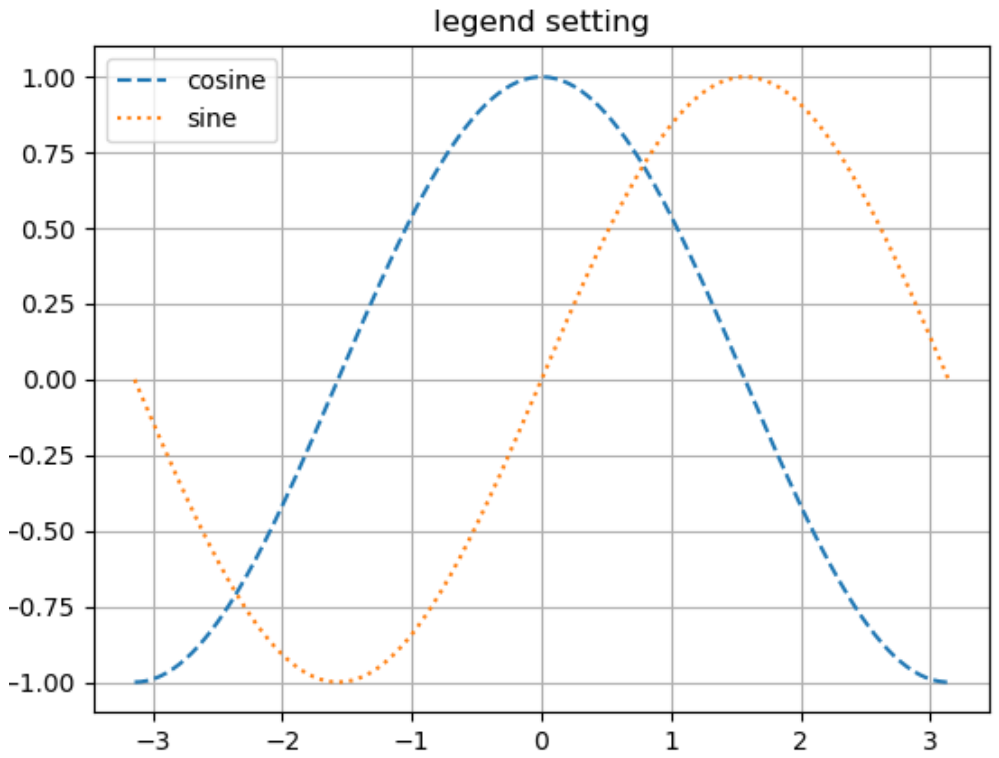


plot

❖ 범례 (legend) 설정

loc 문자열	숫자
best	0
upper right	1
upper left	2
lower left	3
lower right	4
right	5
center left	6
center right	7
lower center	8
upper center	9
center	10

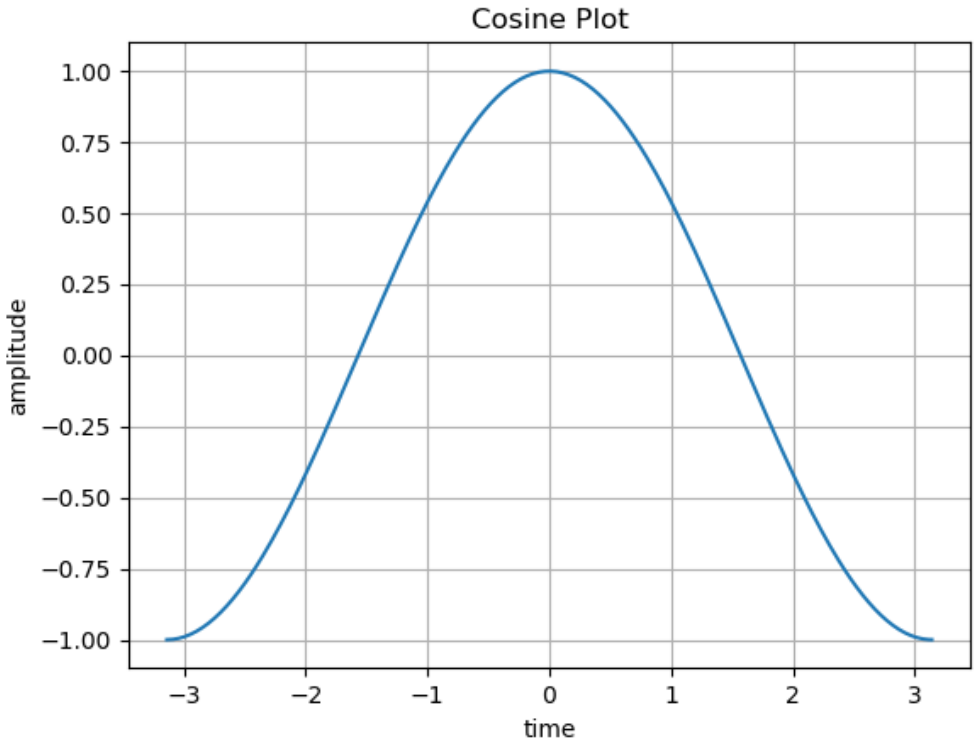
```
X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)
plt.title("legend setting")
plt.plot(X, C, ls="--", label="cosine")
plt.plot(X, S, ls=":", label="sine")
plt.legend(loc=2)
plt.grid(True)
plt.show()
```



plot

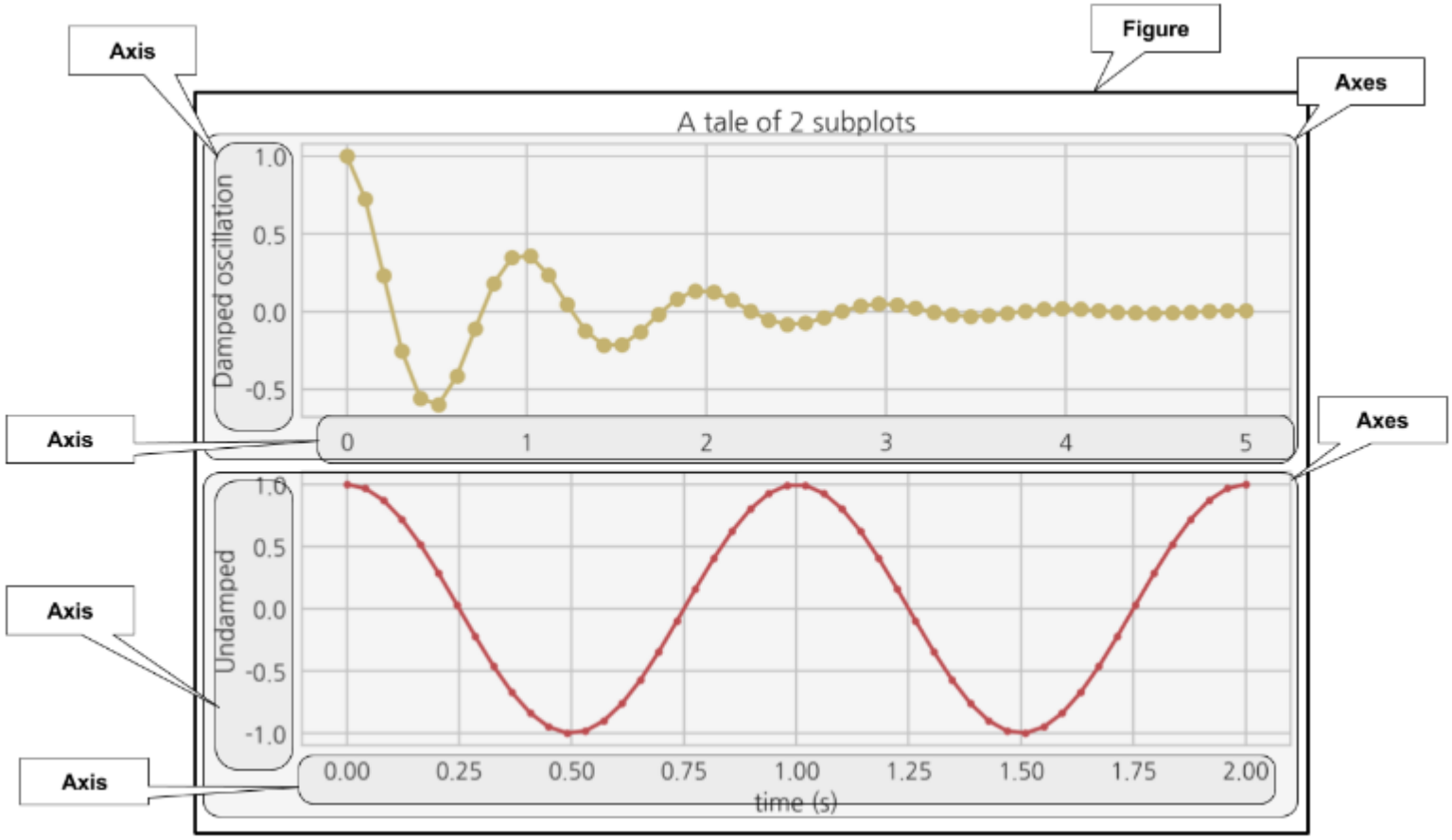
❖ Label과 title

```
X = np.linspace(-np.pi, np.pi, 256)
C, S = np.cos(X), np.sin(X)
plt.plot(X, C, label="cosine")
plt.xlabel("time")
plt.ylabel("amplitude")
plt.title("Cosine Plot")
plt.grid(True)
plt.show()
```



Matplotlib의 객체

- Figure 객체는 한 개 이상의 Axes 객체 포함
- Axes 객체는 두 개 이상의 Axis 객체 포함

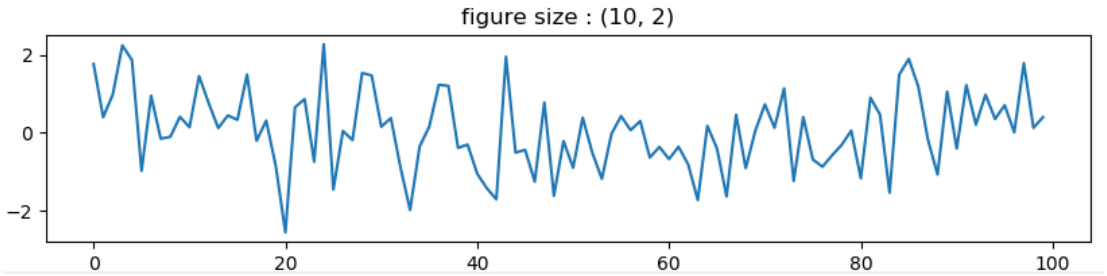


Matplotlib의 객체

❖ Figure 객체와 figure

- 일반적으로는 자동으로 Figure 객체를 생성
- 그래프 윈도우의 설정이 필요할 경우 figure 명령을 명시
  - Figure size 조정

```
np.random.seed(0)
f1 = plt.figure(figsize=(10, 2))
plt.title("figure size : (10, 2)")
plt.plot(np.random.randn(100))
plt.show()
```



❖ Axes 객체와 subplot

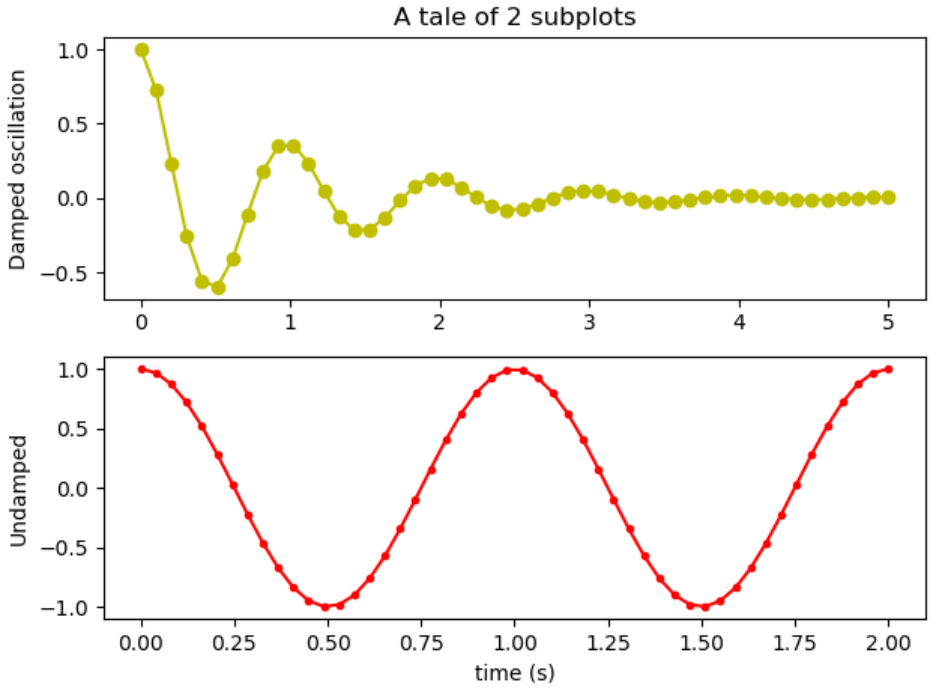
- 일반적으로 자동으로 Axes 객체 생성
- Subplot 명령을 명시하여 Axes 객체 생성
- Figure가 행렬이고, Axes가 행렬의 원소로 생각

```
subplot(2, 1, 1)
# 여기에서 윗부분에 그릴 플롯 명령 실행
subplot(2, 1, 2)
# 여기에서 아랫부분에 그릴 플롯 명령 실행
```

Matplotlib의 객체

❖ Axes 객체와 subplot

```
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)

ax1 = plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'yo-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

ax2 = plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.tight_layout()  # 플롯 간격 자동 조절
plt.show()
```



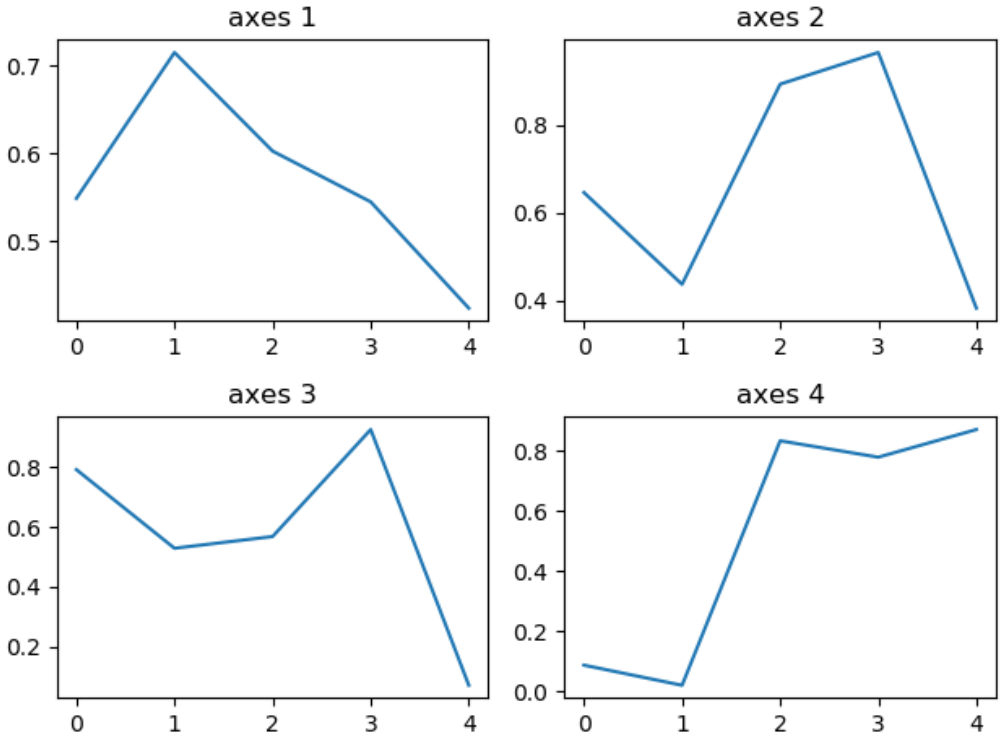
- subplot의 인수를 (2, 2, 1) → (221) 로도 표시 가능 (콤마 생략)
- 플롯 번호의 인덱스가 0이 아닌 1부터 시작하는 것에 유의
  - Matlab의 관행을 따름

Matplotlib의 객체

❖ Axes 객체와 subplots

- 복수의 Axes 객체를 동시에 생성

```
fig, axes = plt.subplots(2, 2)

np.random.seed(0)
axes[0, 0].plot(np.random.rand(5))
axes[0, 0].set_title("axes 1")
axes[0, 1].plot(np.random.rand(5))
axes[0, 1].set_title("axes 2")
axes[1, 0].plot(np.random.rand(5))
axes[1, 0].set_title("axes 3")
axes[1, 1].plot(np.random.rand(5))
axes[1, 1].set_title("axes 4")

plt.tight_layout()
plt.show()
```



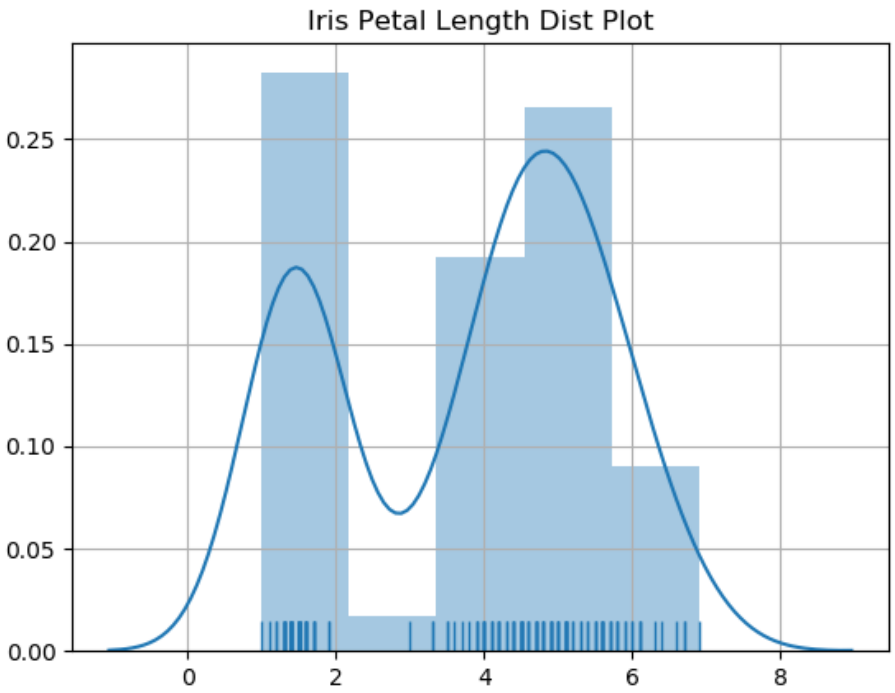
Seaborn 패키지

Seaborn

- Matplotlib 기반 다양한 색상 테마와 통계용 차트 등의 기능을 추가
- import seaborn as sns

1차원 분포 플롯

```
iris = sns.load_dataset("iris")     # 붓꽃 데이터
x = iris.petal_length.values

sns.distplot(x, kde=True, rug=True)
plt.title("Iris Petal Length Dist Plot")
plt.grid(True)
plt.show()
```



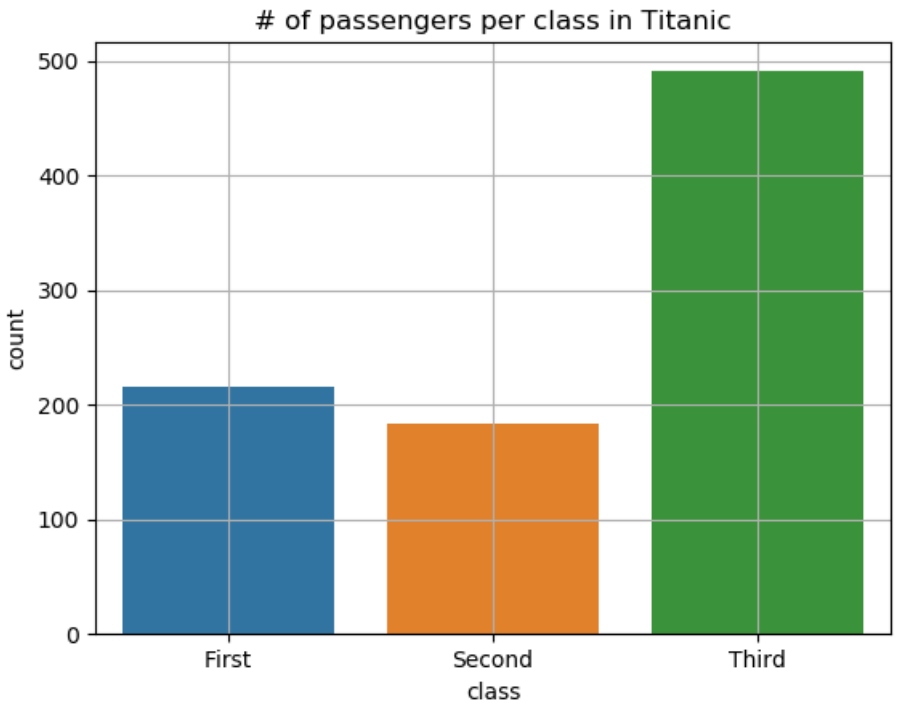
- Rug
  - 데이터 위치를 x축에 표시
- Kde
  - Kernel density를 추정해 히스토그램을 곡선으로 표시

Seaborn 패키지

❖ 카운트 플롯

- countplot: 카테고리 별 데이터 개수 시각화
  - Dataframe에만 이용 가능

```
titanic = sns.load_dataset("titanic")    # 타이타닉호 데이터
sns.countplot(x="class", data=titanic)
plt.title("# of passengers per class in Titanic")
plt.grid(True)
plt.show()
```

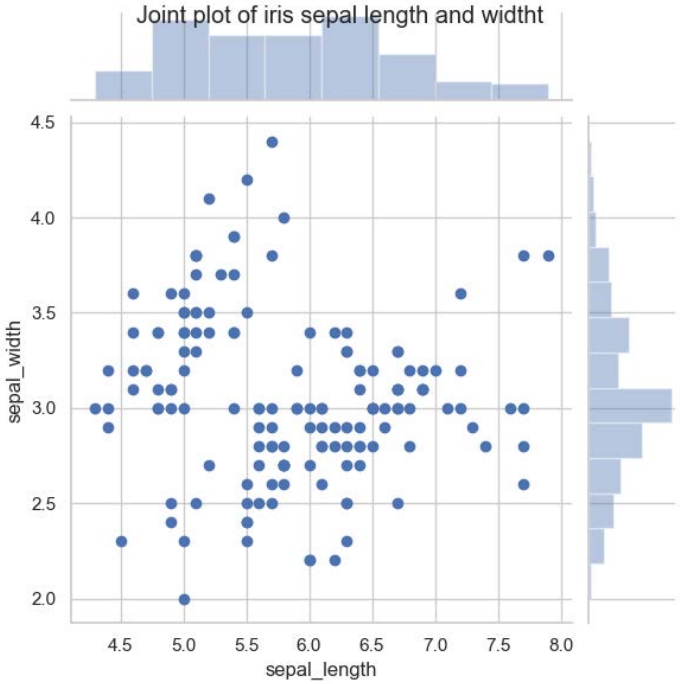


Seaborn 패키지

❖ 2차원 실수 데이터

- jointplot: 스캐터 플롯 + 변수별 히스토그램
  - Dataframe에만 이용 가능

```
iris = sns.load_dataset("iris")     # 붓꽃 데이터
sns.set()
sns.set_style('whitegrid')
sns.set_color_codes()
sns.jointplot(x="sepal_length", y="sepal_width", data=iris)
plt.suptitle("Joint plot of iris sepal length and widtht")
plt.show()
```

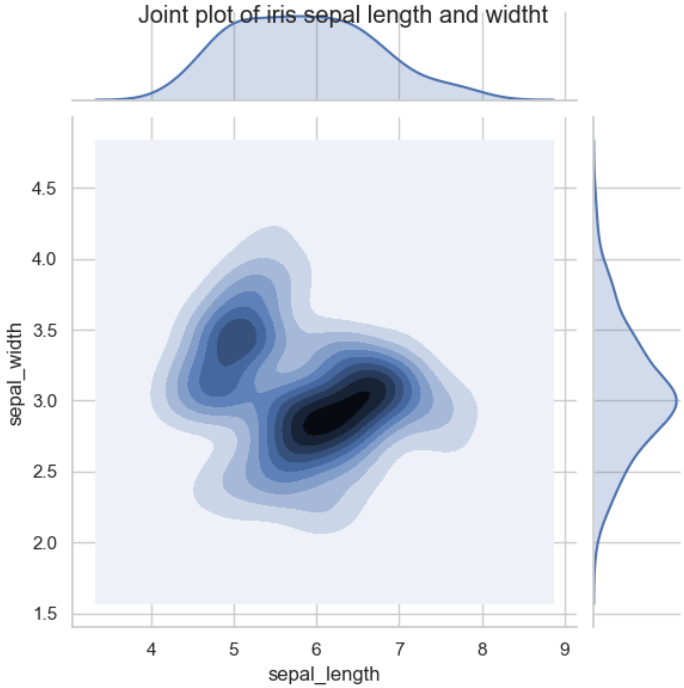


Seaborn 패키지

2차원 실수 데이터

- jointplot: 스캐터 플롯 + 변수별 히스토그램
  - Kind='kde' 일 경우, 커널 밀도 히스토그램

```
iris = sns.load_dataset("iris")     # 붓꽃 데이터
sns.set()
sns.set_style('whitegrid')
sns.set_color_codes()
sns.jointplot(x="sepal_length", y="sepal_width", data=iris, kind='kde')
plt.suptitle("Joint plot of iris sepal length and widtht")
plt.show()
```

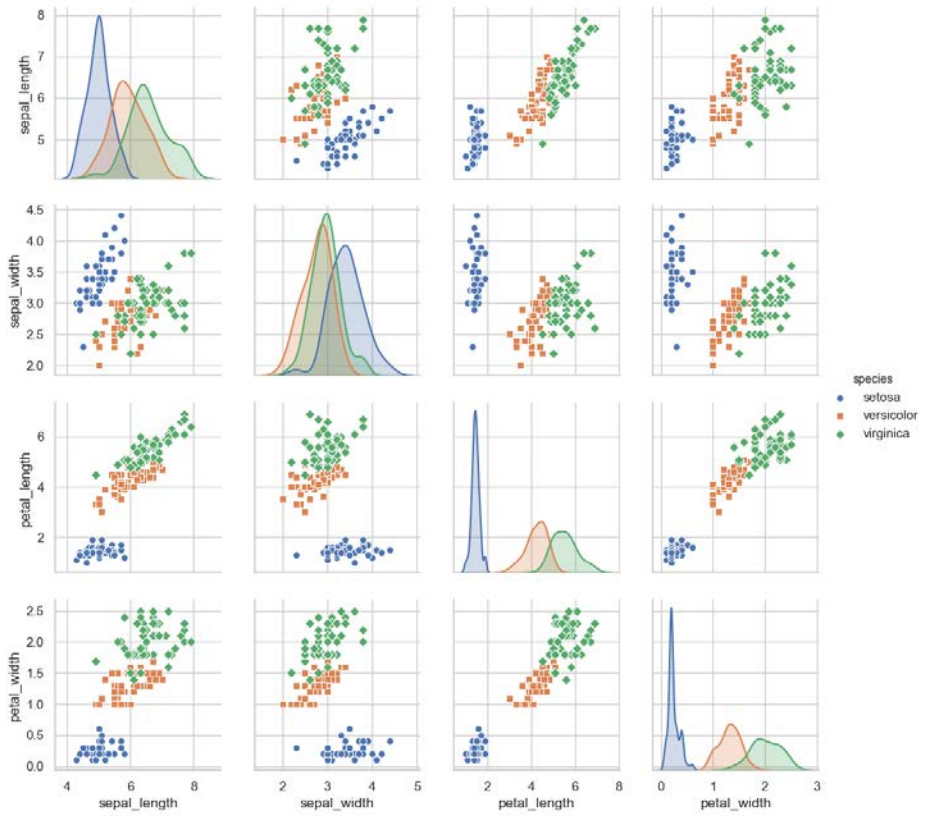


Seaborn 패키지

❖ 다차원 실수 데이터

- pairplot: 조합에 대한 스캐터 플롯 + 히스토그램

```
iris = sns.load_dataset("iris")     # 붓꽃 데이터
sns.set()
sns.set_style('whitegrid')
sns.pairplot(iris, hue="species", markers=["o", "s", "D"])
plt.show()
```

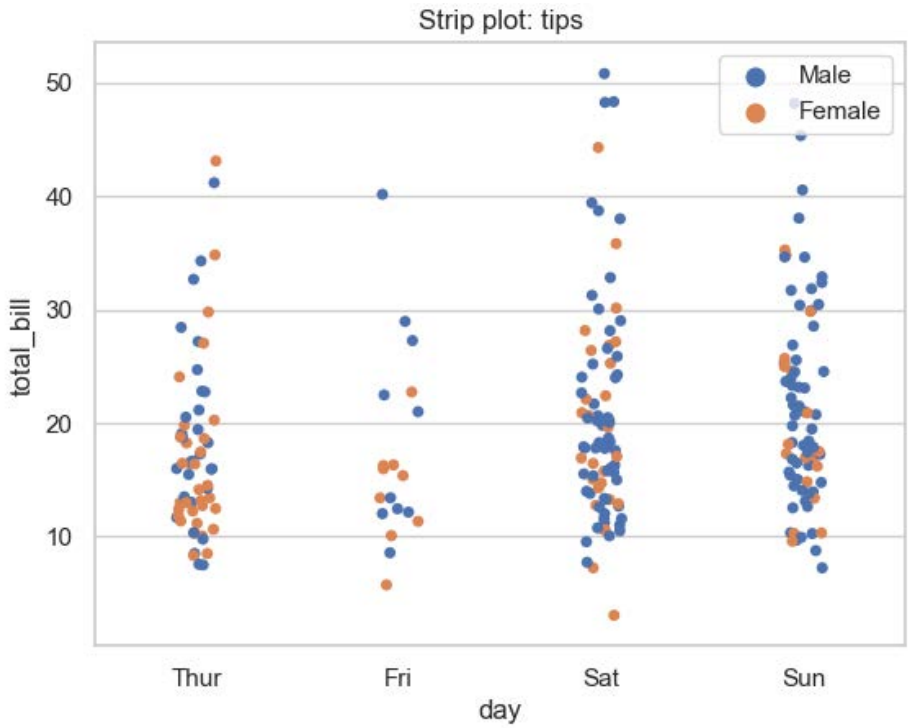


Seaborn 패키지

❖ 다차원 복합 데이터

- stripplot: 복합 데이터에 대한 스캐터 플롯

```
tips = sns.load_dataset("tips")     # 팁 데이터
sns.set()
sns.set_style('whitegrid')
np.random.seed(0)
sns.stripplot(x="day", y="total_bill", hue="sex", data=tips, jitter=True)
plt.title("Strip plot: tips")
plt.legend(loc=1)
plt.show()
```

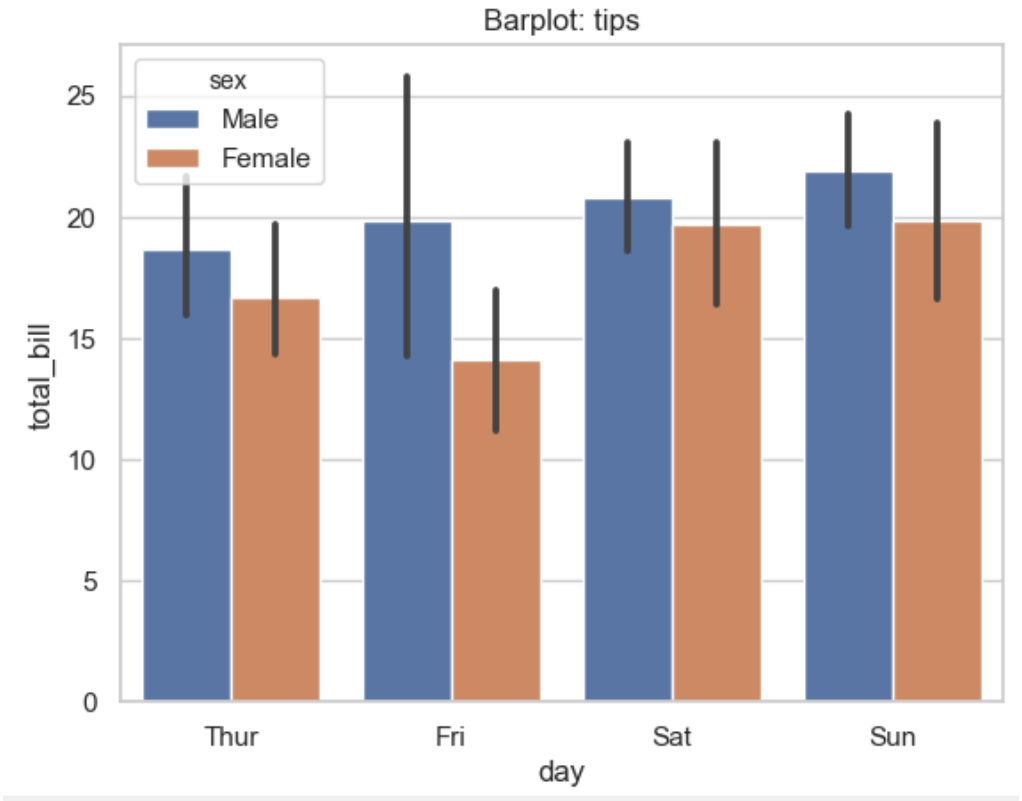


Seaborn 패키지

❖ 다차원 복합 데이터

- barplot: 평균과 표준편차를 시각화

```
tips = sns.load_dataset("tips")     # 팁 데이터
sns.set()
sns.set_style('whitegrid')
sns.barplot(x="day", y="total_bill", hue="sex", data=tips)
plt.title("Barplot: tips")
plt.show()
```

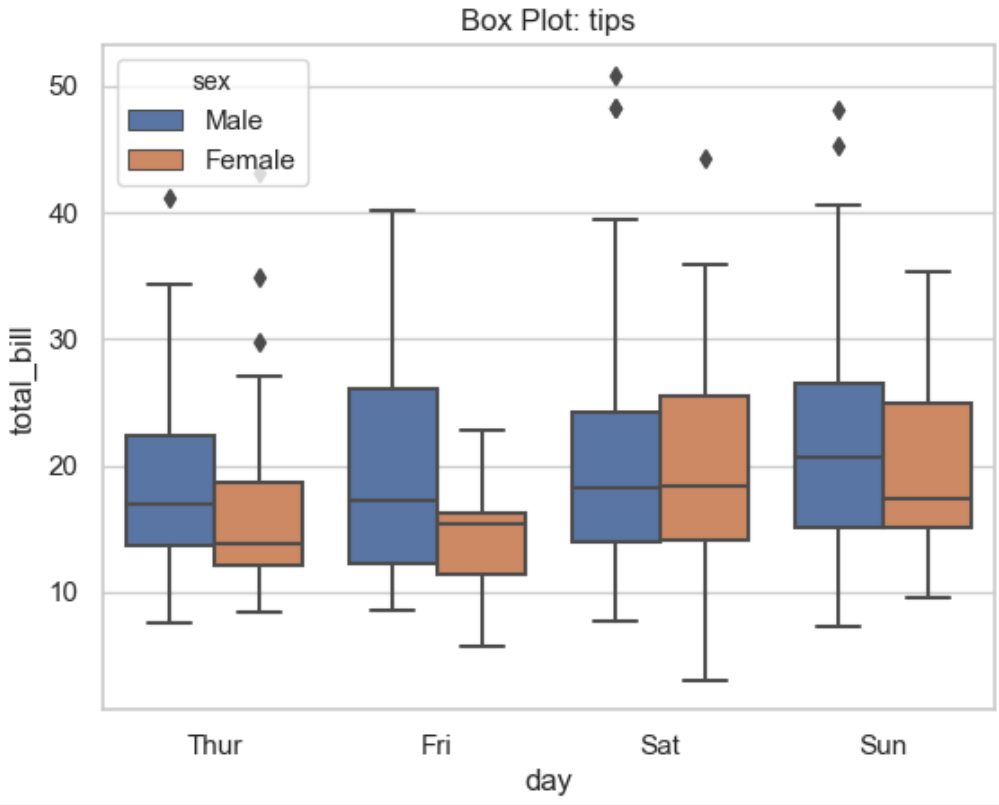


Seaborn 패키지

❖ 다차원 복합 데이터

- boxplot: 중간값 + interquartile range + 아웃라이어 시각화

```
tips = sns.load_dataset("tips")     # 팁 데이터
sns.set()
sns.set_style('whitegrid')
sns.boxplot(x="day", y="total_bill", hue="sex", data=tips)
plt.title("Box Plot: tips")
plt.show()
```

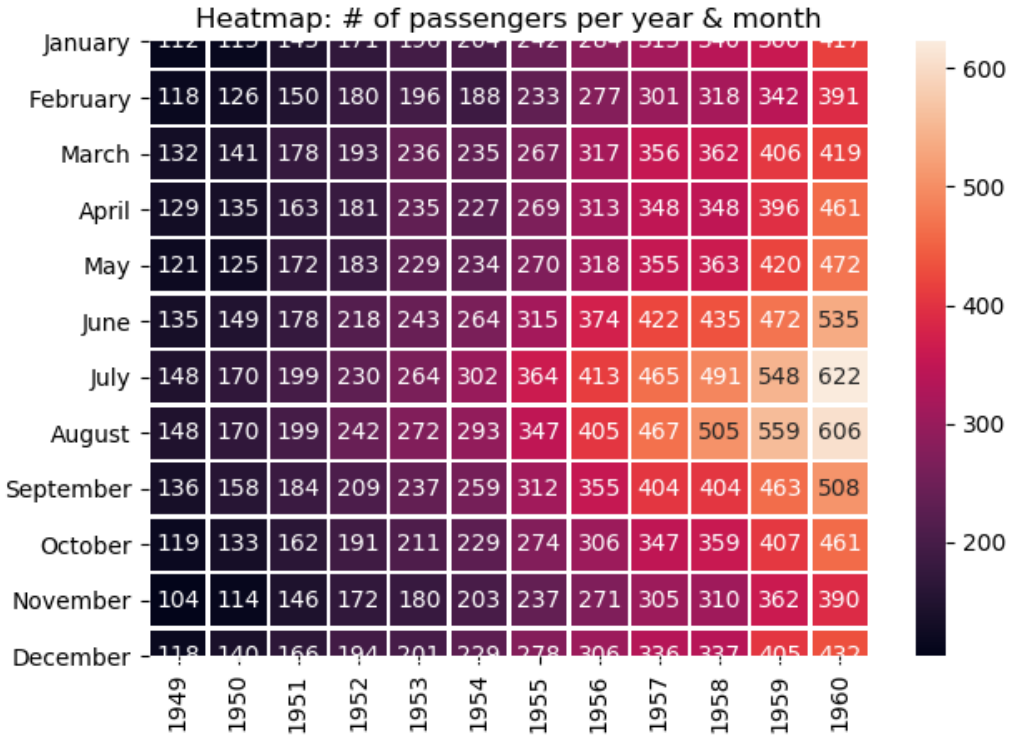


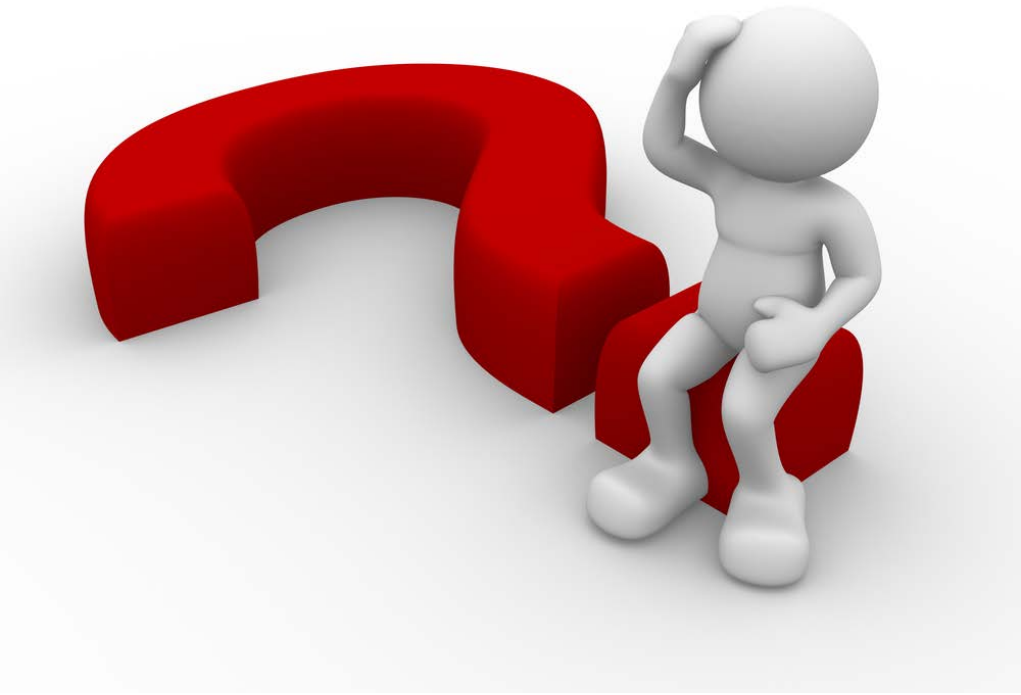
Seaborn 패키지

❖ 다차원 복합 데이터

- heatmap: 두 카테고리의 실수 값

```
titanic = sns.load_dataset("titanic")     # 타이타닉호 데이터
flights_passengers = flights.pivot("month", "year", "passengers")
plt.title("Heatmap: # of passengers per year & month")
sns.heatmap(flights_passengers, annot=True, fmt="d", linewidths=1)
plt.show()
```





Thank You