```
# 구글 드라이브 사용 권한 설정
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# 작업할 폴더 위치 설정 (본인 폴더 경로에 맞게 수정필요)
colab_path = "/content/drive/MyDrive/MachineLearning/4wk"
```

Python에서 다차원 배열 작업을 위한 NumPy 라이브러리에 대한 실습

NumPy: Numerical Python의 줄임말로 NumPy는 다차원 배열 데이터 구조를 효율적으로 사용할 수 있는 편리한 Python 인터페이스를 제공

N-dimensional Arrays(ndarray)

- 고정된 크기의 동형 다차원 배열
- 고정된 크기: 배열을 생성할 때, 크기 결정
- 동형: 같은 type의 원소로 구성된 배열

```
# numpy 라이브러리 import
import numpy as np

# list로 numpy array 생성
lst = [[1,2,3],[4,5,6]]
ary2d = np.array(lst)
print(ary2d)
# tuple로 numpy array 생성
tup = ((0.1,0.2,0.3))
ary2d_2 = np.array(tup)
print(ary2d_2)
```

```
[[1 2 3]
 [4 5 6]]
[0.1 0.2 0.3]
```

기본적으로 NumPy는 ndarray를 생성할 때, 배열의 유형을 추론
Python 정수를 배열에 전달했으므로 `ndarray` 객체 `ary2d`는 64비트 시스템에서 `int64` 유형이어야 하고, 이는 `dtype` 속성을 출력하여 확인 가능

```
# 생성한 배열의 data type 확인
ary2d.dtype
```

```
dtype('int64')
```

다양한 유형의 NumPy 배열을 구성하려면 `array` 함수의 `dtype` 매개변수에 인수를 전달할 수 있음
예를 들어 `np.int32`를 사용하면 32비트 배열을 생성 가능
지원되는 데이터 유형의 전체 목록은 공식 NumPy 문서를 참조
배열이 생성되면 다음 예와 같이 `astype` 메서드를 통해 해당 유형을 다운캐스트하거나 다시 캐스팅할 수 있음

```
# 생성한 배열의 type을 float32로 변경 및 출력
float32_ary = ary2d. astype(np. float32)
float32_ary
```

```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

```
# 배열의 타입 확인
float32_ary.dtype
```

```
dtype('float32')
```

```
# 배열 요소의 갯수 확인
ary2d.size
```

```
6
```

```
# 배열의 차원 수 확인
ary2d.ndim
```

```
2
```

```
# 각 배열 차원의 요소 수(*)
ary2d.shape
```

```
(2, 3)
```

Array Construction Routines

이 섹션에서는 배열 구성 함수를 소개

- 1이나 0의 배열을 생성하는 것은 계산에 초기 값을 사용하지 않고 즉시 다른 값으로 채우려는 경우 자리 표시자 배열로 유용할 수 있음

```
# 배열의 요소를 모두 1로 초기화
np.ones((3,3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
# 배열의 요소를 모두 0로 초기화
np.zeros((3,3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

```
# np.eye(n) -> nxn차원의 대각행렬 생성
np.eye(3)
```

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

np.diag은 두가지 기능을 제공

- 대각행렬 생성 기능 : v가 1차원 배열일때, v를 대각행렬로 하는 2차원 배열을 만들어줌
- 대각요소 추출 기능 : v가 2차원 배열일때, v의 대각선 요소들을 1차원으로 뽑아줌

```
# [0 1 2] 값을 가지는 list arr1d 생성
arrid = np.arange (3)
print(arrid)
# arr1d의 값으로 대칭행렬로 만들어 출력
print(np.diag(arrid))
```

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

```
# [0, 1, 2, ..., 8]의 값을 가지는 3x3형태의 2차원 list arr2d 생성
arr2d = np.arange(9).reshape((3,3))
print(arr2d)
# arr2d의 대각선 요소를 출력
print(f"np.diag(arr2d): {np.diag(arr2d)}")
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
np.diag(arr2d): [0 4 8]
```

```
# (1, 2, 3)로 대각행렬 생성
np.diag((1,2,3))
```

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

지정된 범위 내에서 일련의 숫자를 생성하는 함수: `arange`, `linspace`
NumPy의 `arange` 함수는 Python의 `range` 객체와 동일한 구문형식을 가짐

- 두 개의 인수가 제공되면 첫 번째 인수는 시작 값을 나타내고 두 번째 값은 중지 값을 정의
- Python의 `range` 와 유사하게 세 번째 인수를 제공하여 단계를 정의할 수 있음

```
# numpy의 arange 함수를 사용하여 4에서 10까지의 값을 가지는 배열 생성
np.arange(4., 10.)
```

```
array([4., 5., 6., 7., 8., 9.])
```

```
# numpy의 arange 함수를 사용하여 1에서 11까지 2씩 더해지는 배열을 생성

np.arange (1., 11., 2)
```

```
array([1., 3., 5., 7., 9.])
```

linspace 함수는 지정된 범위에서 지정한 갯수의 균일한 값을 가지는 배열을 생성가능

```
# numpy의 linspace 함수를 사용하여 0에서 1사이에 균일한 간격을 가지는 5개 요소를 가지는 배열 생성
np.linspace (0.,1., num=5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Array Indexing

이 섹션에서는 다양한 인덱싱 방법을 통해 NumPy 배열 요소를 검색하는 기본 사항을 살펴보겠습니다. 간단한 NumPy 인덱싱 및 슬라이싱은 Python 목록과 유사하게 작동하며, 1차원 배열의 첫 번째 요소를 검색하는 다음 코드 조각에서 이를 보여줍니다.

```
# numpy array ary를 생성하여 첫번째 요소를 출력
ary = np.array([1, 2, 3])
ary[0]
```

```
1
```

```
# ary의 처음 두개의 값을 출력

print(ary[:2])
```

```
[1 2]
```

둘 이상의 차원이나 축이 있는 배열로 작업하는 경우 아래 일련의 예에 표시된 대로 인덱싱 또는 슬라이싱 작업을 쉼표로 구분합니다.

```
# 2차원 array를 생성하여 출력
ary = np.array([[1,2,3],
                [4,5,6]])
ary
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
# ary의 우하단 값을 출력
ary[-1,- 1]# 가장끝값에대한접근
```

```
6
```

```
# ary의 첫번째 행, 두번째 열의 값

ary[0,1]
```

```
2
```

```
# ary의 첫번째 행 출력

ary[0]
```

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

```
# ary의 첫번째 열 출력

ary[:,0]
```

```
array([1, 4])
```

```
# ary의 모든 행에 대해 2번째 열까지 출력

ary[:,:2]
```

```
array([[1, 2],
       [4, 5]])
```

```
# ary의 첫번째 요소 출력

ary[0,0]
```

```
1
```

Array Math and Universal Functions

- ndarray에 수행하는 다양한 연산

NumPy는 수학 연산자(`+`, `-`, `/`, `*` 및 `**`)를 직접 사용하여 연산을 수행

```
# 수학연산자 '+'를 사용하여 ary의 모든 요소에 1을 더한 배열을 출력
ary + 1
```

```
array([[2, 3, 4],
       [5, 6, 7]])
```

```
# 수학연산자 '**'를 사용하여 ary의 모든 요소의 제곱값을 가지는 배열을 출력

ary**2
```

```
array([[ 1,  4,  9],
       [16, 25, 36]])
```

```
# 2차원 배열 ary 선언
ary = np.array([[1,2,3],
                [4,5,6]])
```

```
# numpy의 add.reduce 함수를 사용하여 ary의 각 열의 모든 행의 합에 해당하는 1차원 배열을 출력
print(np.add.reduce(ary))
```

```
[5 7 9]
```

```
# sum 함수를 사용하여 ary의 각 열의 모든 행의 합에 해당하는 1차원 배열을 출력
print(ary.sum(axis=0))
```

```
[5 7 9]
```

```
# numpy의 add.reduce 함수를 사용하여 ary의 각 행의 모든 열의 합에 해당하는 1차원 배열을 출력

print(np.add.reduce(ary,axis=1))
```

```
[ 6 15]
```

```
# sum 함수를 사용하여 ary의 각 열의 모든 행의 합에 해당하는 1차원 배열을 출력
print(ary.sum(axis=0))
```

```
[5 7 9]
```

```
# sum 함수를 사용하여 ary의 각 행의 모든 열의 합에 해당하는 1차원 배열을 출력
print(ary.sum(axis=1))
```

```
[ 6 15]
```

```
# sum 함수를 사용하여 ary 모든 요소의 합을 출력
ary.sum()
```

```
21
```

Broadcasting

```
ary1 = np.array([1, 2, 3])
ary2 = np.array([4, 5, 6])

# '+' 연산자로 ary1과 ary2의 합을 출력
ary1+ary2
```

```
array([5, 7, 9])
```

```
ary3 = np.array([[4, 5, 6],
                 [7, 8, 9]])

# '+'연산자로 ary3과 ary1의 합을 출력
ary3 + ary1
```

```
array([[ 5,  7,  9],
       [ 8, 10, 12]])
```

Advanced Indexing -- Memory Views and Copies

기본적인 정수 기반 인덱싱 및 슬라이싱은 메모리에 NumPy 배열의 *view*를 생성 (얕은 복사)
view: 메모리 자원을 절약하기 위해 불필요한 배열 복사본을 만드는 작업 방지

```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

# ary의 첫번째 행을 대입연산자'='을 사용하여 first_row에 입력
first_row=ary[0]

# first_row의 각 요소에 99를 더하여 출력
first_row +=99
ary
```

```
array([[100, 101, 102],
       [  4,   5,   6]])
```

위의 예제에 나타나는 것과 같이 `first_row` 값을 변경하면 원래 배열에도 영향을 미침.
그 이유는 `ary[0]` 이 `ary` 의 첫 번째 행에 대한 view를 생성한 후 해당 요소가 99만큼 증가했기 때문

```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

# ary의 첫번째 생을 슬라이싱(:)으로 first_row에 입력
first_row= ary[:1] # 슬라이싱의경우에도얕은복사가실행됨

# first_row의 각 요송 99를 더하여 출력
first_row +=99
ary
```

```
array([[100, 101, 102],
       [  4,   5,   6]])
```

copy 함수, pancy 인덱싱 예제 (깊은 복사)

```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

# copy()함수를 사용하여  ary의 두번째 열을 second_row에 입력
second_row = ary[1].copy()

# second_row에 99를 더하여 출력
second_row += 90
ary
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

pancy 인덱싱을 통해 연속되지 않은 정수 인덱스의 튜플 또는 리스트 객체를 사용하여 원하는 배열 요소를 반환할 수 있음.
연속되지 않은 시퀀스로 수행될 수 있으므로 메모리에서 연속적인 슬라이스인 뷰를 반환할 수 없어, 항상 배열의 복사본을 반환

```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

# 연속되지 않는 정수 인덱스의 배열 요소 반환 예제
ary[:,[0,2]]
```

```
array([[1, 3],
       [4, 6]])
```

```
# pancy 인덱싱 예제
this_is_a_copy = ary[:,[0,2]] # 깊 은 복 사

this_is_a_copy += 99

ary
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

인덱싱을 위해 True 및 False 값의 bool 배열 마스크 활용

```
# mask를 사용한 인덱싱
ary = np.array([[1,2,3],
                [4,5,6]])


# ary의 요소 중 3보다 큰 값에 대한 mask 생성
greater3_mask = ary > 3
greater3_mask
```

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

```
# mask를 사용하여 ary의 요소중 3보다 큰 값만 출력

ary[greater3_mask]
```

```
array([4, 5, 6])
```

논리연산자 and, &, or, | 또한 활용 가능

```
# 다양한 논리 연산자를 활용하여 출력
ary[(ary > 3) & ( ary % 2 ==0 ) ]
```

```
array([4, 6])
```

Random Number Generators

머신러닝과 딥러닝에서는 난수 배열을 생성해야 하는 활용하는 경우가 많음
ex) 파라매터 초기화 등

```
# seed값 고정 하기, random 한 값 생성
np.random.seed(123)#전역적, 모든 NumPy의 난수 생성 함수가 동일한 시드를 사용하여 랜덤값 출력
np.random.rand(3)
```

```
array([0.69646919, 0.28613933, 0.22685145])
```

```
rng1 = np.random.RandomState(seed=123) # 개별적인 인스턴스를 생성할 수 있으므로, 각 인스턴스는 독립적으로  시드를 설정하고 난수를 생성
rng1.rand(3)
```

```
array([0.69646919, 0.28613933, 0.22685145])
```

Reshaping Arrays

배열이 특정 계산을 수행할 때, *올바른* 모양을 갖지 않는 상황을 자주 직면하게 됨
NumPy 배열의 크기는 고정되어 있지만, 모양을 변경할 수 있음
NumPy는 다른 모양의 배열 보기를 얻을 수 있는 `reshape` 메소드를 제공

예를 들어, 다음과 같이 `reshape` 를 사용하여 1차원 배열을 2차원 배열로 재구성할 수 있음

```
# 1에서 6의 값을 가지는 1차원 배열 arr1d를  생성
ary1d= np.array([1, 2, 3, 4, 5, 6])
```

```
# reshape 함수로 2x3의 이차원 배열 arry2_view 생성
ary2d_view = ary1d.reshape(2,3)
ary2d_view
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
# numpy의 may_share_memory 함수를 사용하여 ary2d_view와 arr2d가 같은 메모리를 공유하는 확인(True -> 얕은 복사)
np.may_share_memory(ary2d_view, ary1d ) # 얕 은 복 사. 두 배 열 이 같 은 메 모 리 를 가 르 킴
```

```
True
```

하나의 축만 지정하고 하나의 축은 `-1` 로 지정하면 numpy에서 알맞은 축의 갯수를 계산하여 배열의 모양을 변경

```
# ary1d의 reshape의 하나의 축을 -1로 지정하여 2차원 배열 출력
ary1d.reshape(2,-1)
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

`reshape` 함수를 사용하면 2차원 배열을 1차원으로 펼 수도 있음

```
# numpy 함수를 사용하여 2차원 배열을 1차원으로 펼치기
ary = np.array([[[1, 2, 3],
                 [4, 5, 6]]])

ary.reshape(-1)
```

```
array([1, 2, 3, 4, 5, 6])
```

두개의 배열 값을 연결하고 싶을 때, NumPy 배열의 크기는 고정되어 있으므로 두 배열을 합친 **새 배열**을 만들어야 함
두 개 이상의 배열 객체를 결합하려면 다음 예와 같이 NumPy의 `concatenate` 함수를 사용할 수 있음

```
ary = np.array([1, 2, 3])

# 배열의 값을 뒤로 연결
np.concatenate((ary, ary))
```

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

```
ary = np.array([[1, 2, 3]])

# 첫번째 축에 대해 배열을 연결 axis=0
np.concatenate((ary, ary), axis=0)
```

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

```
# 두번째 축에 대해 배열을 연결 axis=1
np.concatenate((ary, ary), axis=1)
```

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

Comparison Operators and Masks

비교 연산자(예: `<`, `>`, `<=` 및 `>=`)를 사용하여 해당 배열의 부울 마스크를 만들 수 있음
이 마스크는 해당 배열의 값에 따라 `True` 및 `False` 요소로 구성

```
# 비교연산자 예제
ary = np.array([1,2, 3, 4])

mask = ary >2
mask
```

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

부울 마스크는 특정 조건을 만족하는 요소의 갯수를 확인할 때도 활용 가능

```
# 생성한 mask를 사용하여 ary에서 2보다 큰 요소의 갯수를 출력
mask.sum()
```

```
2
```

`np.where` : 배열의 특정 요소에 값을 할당하는데 유용한 함수

```
# numpy의 where 함수를 사용하여 배열에서 2보다 큰 모든 값에 1을 할당하고, 그렇지 않으면 0을 할당
np.where(ary>2,1,0)
```

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

Linear Algebra with NumPy Arrays

1차원 NumPy 배열을 행 벡터를 나타내는 데이터 구조, 2차원 NumPy 배열로 열 벡터를 나타내는 데이터 구조 생성 가능

```
# 배열 연산을 위해 row_vector와 column_vector 생성
row_vector = np.array([1,2,3])
print(row_vector)
column_vector = np.array([[1, 2, 3]]).reshape(-1,1) #열벡터
print(column_vector)
```

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

1차원 배열을 2차원 배열로 바꾸는 것은 아래의 방법으로 간단하게 새 축을 추가할 수 있음

```
# 1차원 배열을 2차원 배열로 변경 1
row_vector[:,np.newaxis]
```

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

```
# 1차원 배열을 2차원 배열로 변경 2
row_vector[:,None]
```

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

행렬 간의 행렬 곱셈을 수행하려면 왼쪽 행렬의 열 수가 오른쪽 행렬의 행 수와 일치해야 함
NumPy에서는 `matmul` 함수를 통해 행렬 곱셈을 수행

```
# 배열 연산을 위한 matrix 생성
matrix= np.array([[1, 2, 3],
                  [4,5,6]])
matrix
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
# numpy의 matmul 함수를 사용하여 matrix와 column_vector의 행렬곱의 결과를 출력
np.matmul(matrix, column_vector)
```

```
array([[14],
       [32]])
```

```
#  numpy의 matmul 함수를 사용하여 row_vector와 row_vector의 곱을 출력
np.matmul(row_vector, row_vector)
```

```
14
```

```
#  numpy의 dot 함수를 사용하여 matrix와 row_vector의 곱을 출력
np.dot(matrix,row_vector )
```

```
array([14, 32])
```

더블클릭 또는 Enter 키를 눌러 수정

```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])

# transpose함수를 사용하여 2차원 배열 matrix의 전치행렬 출력
matrix.transpose()
```

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```
# `transpose`의 약칭 `T`로도 전치행렬 출력 가능

matrix.T
```

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

SciPy

- 과학 계산용 함수를 모아 놓은 파이썬 패키지
- Numpy가 지원하지 않는 고성능 선형대수, 함수 최적화, 신호 처리, 특수한 수학 함수와 통계 분포 등을 포함한 많은 기능을 제공
- scikit-learn은 알고리즘을 구현할 때 SciPy의 여러 함수를 사용
  SciPy document link

Matplotlib

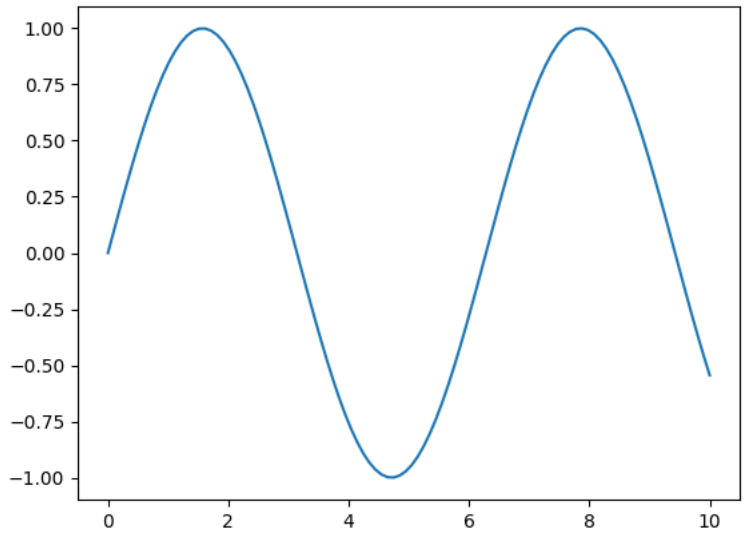
- 데이터 차트나 그래프로 시각화하는 패키지
- 데이터 분석 이전에 데이터 이해를 위한 시각화
- 데이터 분석 이후에 결과를 시각화 공식 matplot 갤러리

```
# 코드 셀에서 그래프를 생성하고 실행한 셀 아래에 이미지 형태로 바로 출력하도록 설정
%matplotlib inline

import matplotlib.pyplot as plt
```

Plotting Functions and Lines

```
# 그래프 예제 1
# linspace함수로 0에서 10사이의 100개의 값을 요소로 가지는 배열 x를 생성
x = np.linspace(0,10,100)

# x에 대한 sin(x)를 그래프로 출력
plt.plot(x, np.sin(x))
```

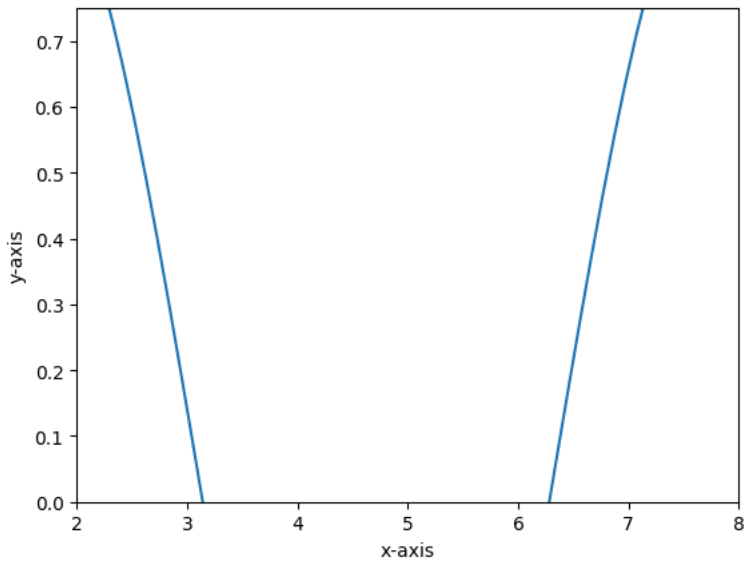
```
[<matplotlib.lines.Line2D at 0x7d50de7054b0>]
```



Add axis ranges and labels:

```
# 그래프 예제 2
# linspace함수로 0에서 10사이의 100개의 값을 요소로 가지는 배열 x를 생성, x에 대한 sin(x)를 그래프 생성
x = np.linspace(0,10,100)
plt.plot(x, np.sin(x))


# xlim, ylim함수를 사용하여 x축의 값을 2에서 8, y축의 값을 0에서 0.75까지 출력
plt.xlim([2,8])
plt.ylim([0,0.75])
# xlabel과 ylabel을 'x-axis', 'y-axis'로 설정하여 표를 출력
plt.xlabel('x-axis')
plt.ylabel('y-axis')
```

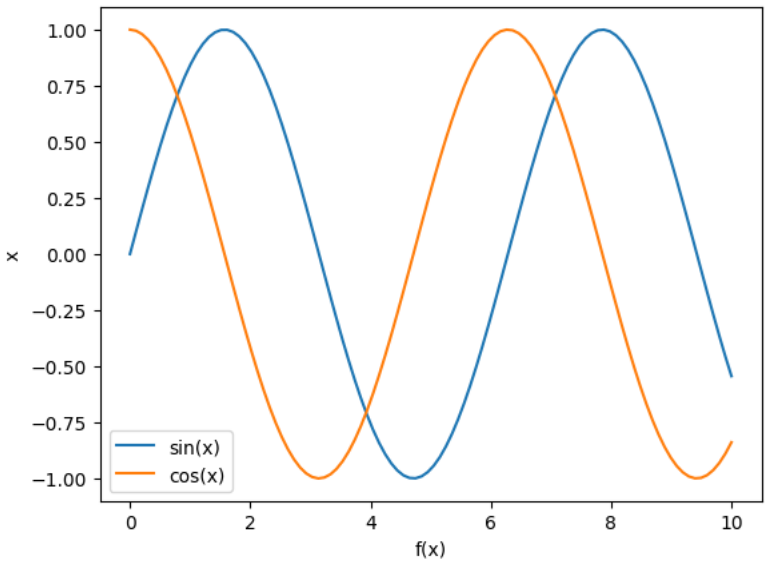
```
Text(0, 0.5, 'y-axis')
```



`plt.legend`: 그림에 범례(legend)를 추가

loc = ['best': 최적의 위치를 자동으로 선택, 'upper rignt': 오른쪽 상단, 'upper left': 왼쪽 상단, 'lower right': 오른쪽 하단, 'loser left': 왼쪽 하단]

```
# 그래프 예제 3

# linspace함수로 0에서 10사이의 100개의 값을 요소로 가지는 배열 x를 생성
x = np.linspace(0,10,100)


# x 값에 대한 sin, cos함수를 생성, option 'label'을 사용하여 범례를 지정
plt.plot(x, np.sin(x), label=('sin(x)'))
plt.plot(x, np.cos(x), label=('cos(x)'))


# ylabel과 xlabel을 'f(x)', 'x'로 설정
plt.xlabel('f(x)')
plt.ylabel('x')
# 범례의 위치를 좌하단으로 설정하여 표를 출력
plt.legend(loc='lower left')
plt.show()
```



Scatter Plots(산점도)

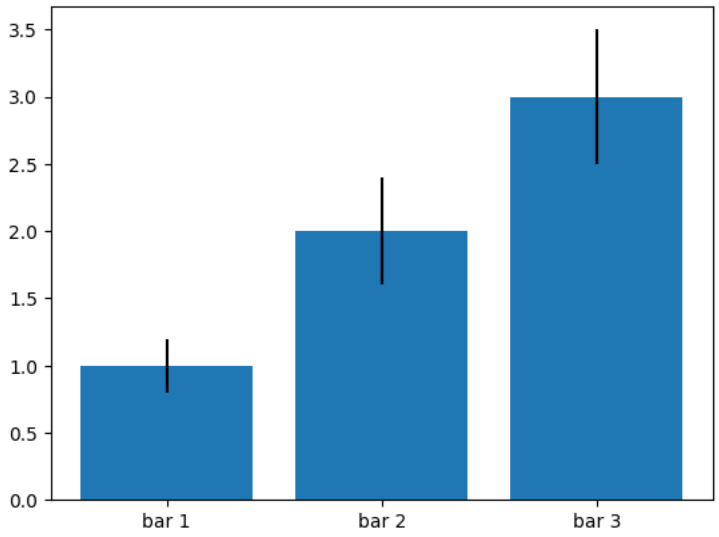
- 산점도(Scatter plot)를 생성, 주로 데이터의 점들을 시각화할 때 사용

```
# 난수 500개를 생성하여 산점도 그래프를 출력
rng = np.random.RandomState(123)

# 정규분포(평균0,표준편차1)를 따르는 난수 50개를 생성
x = rng.normal(size=500)
y = rng.normal(size=500)

plt.scatter(x,y)
plt.show()
```

Bar Plots(막대그래프)

- 막대 그래프(Bar chart)를 생성
- 범주형 데이터(category)의 빈도, 비교, 분포(values)등을 시각화 하는 데 효과적

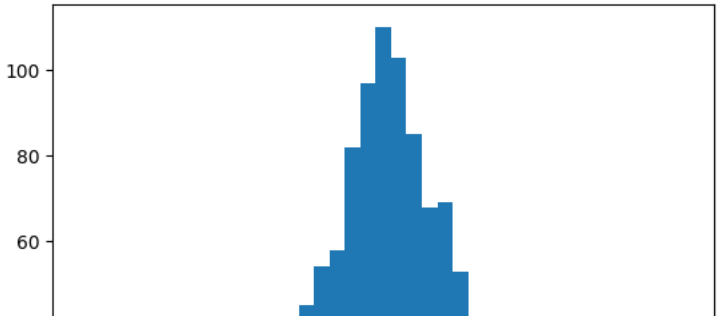
```
# 막대 그래프 예제
means = [1,2,3]
stddevs = [0.2,0.4,0.5]
bar_labels = ['bar 1','bar 2', 'bar 3']

# 막대그래프 출력
x_pos = list(range(len(bar_labels)))
plt.bar(bar_labels, means, yerr=stddevs)

plt.show()
```



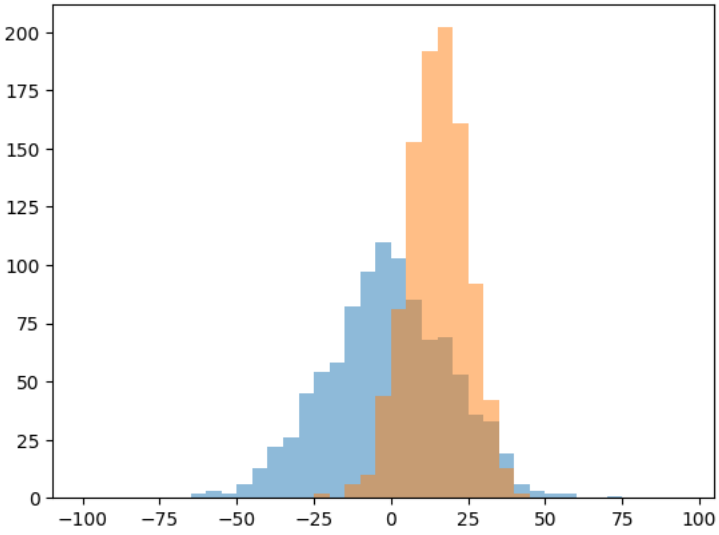
Histograms

- 히스토그램을 생성, 데이터의 분포를 시각화하는 데 효과적
- bin: 데이터를 나누는 구간의 수

```
# histogram 그래프 예제 1

means = [1,2,3]
stddevs = [0.2,0.4,0.5]

rng = np.random.RandomState(123)
x = rng.normal(0,20,1000)

# fixed bin size
bins = np.arange(-100,100,5)

plt.hist(x,bins=bins)
plt.show()
```



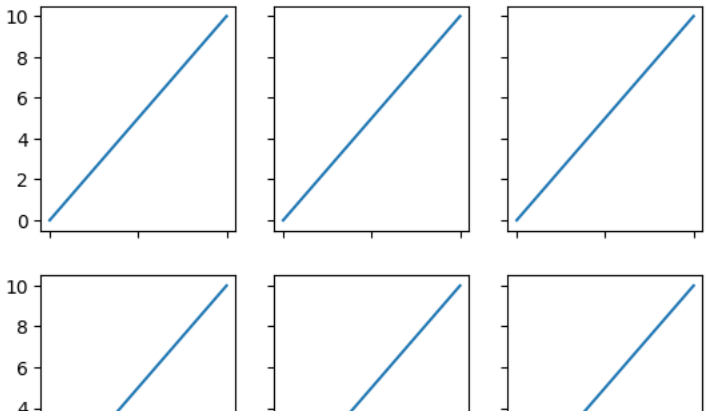

```
# histogram 그래프 예제 2

rng = np.random.RandomState(123)
x1 = rng.normal(0,20,1000)
x2 = rng.normal(15,10,1000)


# fixed bin size
bins = np.arange(-100,100,5)

plt.hist(x1,bins=bins, alpha=0.5)
plt.hist(x2,bins=bins, alpha=0.5)

plt.show()
```



Subplots(하위 그래프)

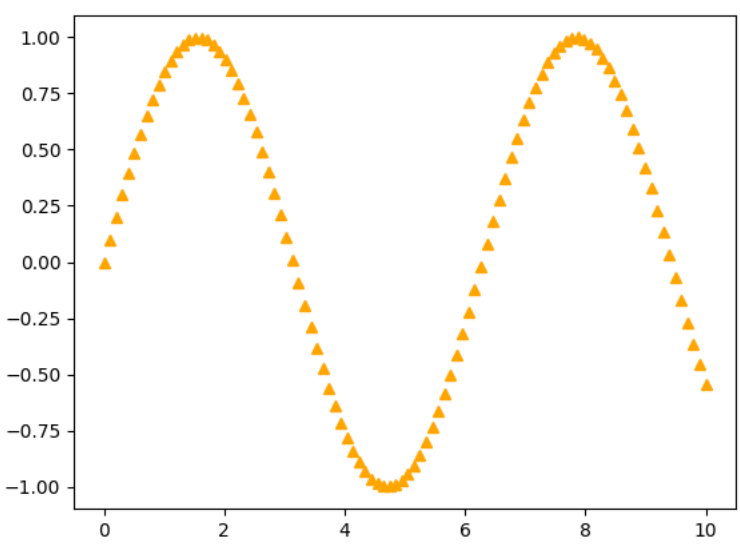
- 여러 개의 하위 그래프(subplot)를 포함하는 Figure 객체와 그래프 객체를 생성
- 하위 그래프에 접근할 때, ax[행, 열] 형식으로 인덱스를 사용

```
import matplotlib.pyplot as plt
# Subplots 그래프 예제
x = range(11)
y = range(11)

fig, ax = plt.subplots(nrows=2, ncols=3, sharex=True, sharey=True)

for row in ax:
  for col in row:
    col.plot(x,y)

plt.show()
```



Colors and Markers

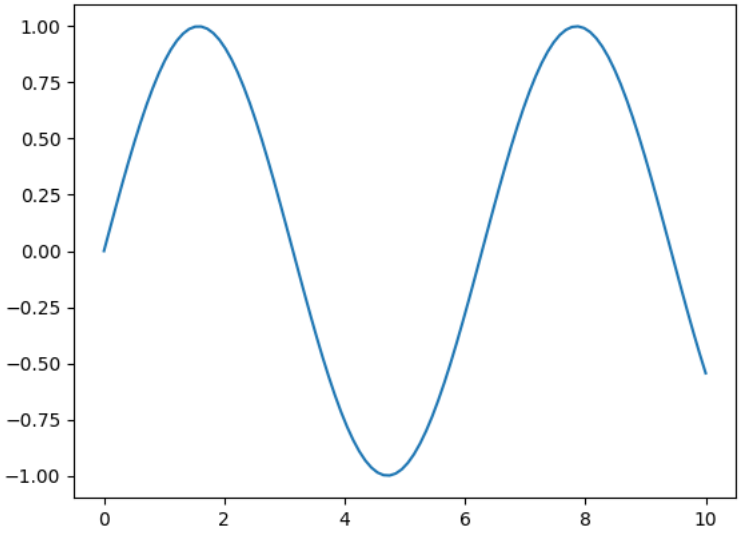
```
# 그래프 디자인 변경 예제

# linspace함수로 0에서 10사이의 100개의 값을 요소로 가지는 배열 x를 생성
x = np.linspace(0,10,100)

# x에 대한 sin(x)를 그래프로 출력
plt.plot(x, np.sin(x), color = 'orange', marker='^', linestyle='')
plt.show()
```



Saving Plots

- 다양한 확장자로 Figure 저장 지원 (.eps, .svg, .jpg, .png, .pdf, .tiff, etc.)

```
# 생성한 그래프 저장 예제
import os

# linspace함수로 0에서 10사이의 100개의 값을 요소로 가지는 배열 x를 생성
x = np.linspace(0,10,100)

# x에 대한 sin(x)를 그래프로 출력
plt.plot(x, np.sin(x))

# 다양한 확장자로 Figure 저장 지원
plt.savefig('myplot.png', dpi=300)
plt.savefig('myplot.pdf')

plt.show()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10