ch.4-1 Introduction to Numpy

Introduction to Numpy

- Numpy는 numberal Python의 약자로, 수치 계산을 지원하기 위한 Python 라이브러리 패키지이다
- Numpy에서 기본적인 데이터 구조는 ndarray라는 다차원 배열 객체이다.
- Numpy는 ndarray의 요소들을 효율적으로 조작할 수 있는 일련의 함수들을 제공한다.
- 참조 설명서를 보려면
  - 도움말 $\Rightarrow$ Numpy 참조
  - 또는 https://numpy.org/doc/stable/index.html #
- Numpy의 기본 구성 요소를 소개합니다.
- 특정 기능을 사용하려면 항상 구글링을 시도하거나 help()를 사용합니다.

Creating ndarray

- ndarray는 list 또는 tuple 개체에서 작성할 수 있습니다.
- numpy.array는 생성 및 배열을 위한 편의 기능일 뿐입니다
- numpy.ndarray는 클래스입니다

> 💡 참고:
>
> - 텐서는 다차원 배열로 표현할 수 있는 "어떤 것"입니다.
> - 텐서플로우는 딥러닝을 위한 구글 제품입니다.
> - 텐서플로우 코딩의 경우, numpy 차원 배열에 매우 능숙해야 합니다.

```
import numpy
```

```
import numpy as np

oneDim = np.array([1.0,2,3,4,5])   # a 1-dimensional array (vector)
print(oneDim)
print("#Dimensions =", oneDim.ndim)
print("Dimension =", oneDim.shape)
print("Size =", oneDim.size)
print("Array type =", oneDim.dtype)
```

```
[1. 2. 3. 4. 5.]
#Dimensions = 1
Dimension = (5,)
Size = 5
Array type = float64
```

- Numpy Element에 C 배열과 같은 유형이 하나만 있습니다.

```
twoDim = np.array([[1,2],[3,4],[5,6],[7,8]])  # a two-dimensional array (matrix)
print(twoDim)
```

```
print("#Dimensions =", twoDim.ndim)
print("Dimension =", twoDim.shape)
print("Size =", twoDim.size)
print("Array type =", twoDim.dtype)
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
#Dimensions = 2
Dimension = (4, 2)
Size = 8
Array type = int32
```

```
arrFromTuple = np.array([(1,'a',3.0),(2,'b',3.5)])  # create ndarray from tuple
print(arrFromTuple)
print("#Dimensions =", arrFromTuple.ndim)
print("Dimension =", arrFromTuple.shape)
print("Size =", arrFromTuple.size)
print("Array type =", arrFromTuple.dtype)
```

```
[['1' 'a' '3.0']
 ['2' 'b' '3.5']]
#Dimensions = 2
Dimension = (2, 3)
Size = 6
Array type = <U11
```

```
# Guess what is printed
print(np.array([1]).shape)
print(np.array([1,2]).shape)
print(np.array([[1],[2]]).shape)
print(np.array([[[1,2,3],[1,2,3]]]).shape)
print(np.array([[[[]]]]).shape)
```

💡 설명

1. `np.array([1]).shape`:
   - 배열

`[1]`은 요소가 하나인 1차원 배열입니다. 따라서 shape는 `(1,)`입니다.

2. `np.array([1,2]).shape`:
   - 배열

`[1,2]`은 요소가 두 개인 1차원 배열입니다. 따라서 shape는 `(2,)`입니다.

3. `np.array([[1],[2]]).shape`:
   - 배열

`[[1], [2]]`은 두 개의 요소가 각각 하나씩 들어있는 2차원 배열입니다. 따라서 shape는 `(2,1)`입니다.

4. `np.array([[[1,2,3],[1,2,3]]]).shape`:
   - 배열

`[[[1,2,3], [1,2,3]]]`은 1차원 요소(리스트)가 2개의 하위 리스트를 가지고 있으며, 각 하위 리스트는 3개의 요소를 포함합니다. 따라서 shape는 `(1,2,3)`입니다.

5. `np.array([[[[[]]]]]).shape`:
   - 배열

`[[[[]]]]`은 4차원 배열입니다. 1개의 요소(리스트)가 있고, 그 안에 다시 1개의 리스트가 포함되며, 그 안에 1개의 빈 리스트가 포함됩니다. 따라서 shape는 가장 깊은 차원까지 각 차원에서 하나의 요소를 가지므로 `(1,1,1,0)`입니다.

```
(1,)
(2,)
(2, 1)
(1, 2, 3)
(1, 1, 1, 0)
```

- Numpy에는 배열을 만드는 데 사용할 수 있는 여러 기능이 내장되어 있습니다.

```
print(np.random.rand(5))      # [0,1] 사이의 균등 분포에서 무작위 숫자 생성
print(np.random.randn(5))     # 평균 0, 표준 편차 1의 정규 분포에서 무작위 숫자 생성
print(np.arange(-10,10,2))    # range와 비슷하지만 리스트 대신 ndarray 반환 (범위 지정)
print(np.arange(12).reshape(3,4))  # 배열을 3행 4열 행렬로 재구성
print(np.linspace(0,1,10))    # [0,1] 구간을 10개의 균등하게 나눈 값으로 분할
print(np.logspace(-3,3,7))    # 10^-3에서 10^3까지 로그 스케일로 균등하게 분할된 숫자 생성
# logspace는 로그 스케일에서 균등하게 간격을 둔 숫자를 반환합니다.
```

```
[0.39549071 0.83200035 0.21630632 0.92803293 0.71053543]
[0.89132678 0.0393841  0.36250086 0.04464502 1.31660902]
[-10  -8  -6  -4  -2   0   2   4   6   8]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
[1.e-03 1.e-02 1.e-01 1.e+00 1.e+01 1.e+02 1.e+03]
```

```
print(np.zeros((2,3)))        # 2 x 3 크기의 영행렬
print(np.ones((3,2)))         # 3 x 2 크기의 일행렬
print(np.eye(3))              # 3 x 3 단위행렬
```

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

요소별 작업

- ndarray의 각 요소에 덧셈, 곱셈 등의 표준 연산자를 적용할 수 있습니다.

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

print(x + 1)      # 덧셈
print(x - 1)      # 뺄셈
print(x * 2)      # 곱셈
print(x // 2)     # 정수 나눗셈
print(x ** 2)     # 제곱
print(x % 2)      # 나머지
print(1 / x)      # 나눗셈
```

```
[2 3 4 5 6]
[0 1 2 3 4]
[ 2  4  6  8 10]
[0 1 1 2 2]
[ 1  4  9 16 25]
[1 0 1 0 1]
[1.         0.5        0.33333333 0.25       0.2       ]
```

Why Numpy?

```
import time
start = time.time()  # 시작 시간 측정

# 반복적인 합 계산
total = 0
# 1.5백만 개의 숫자를 반복하여 합 계산
for item in range(0, 1500000):
    total = total + item

print('sum is: ' + str(total))  # 합의 결과 출력
end = time.time()  # 종료 시간 측정
print(end - start)  # 실행 시간 출력
```

```
sum is: 1124999250000
0.1862163543701172
```

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

print(x + y)  # 배열 x와 배열 y의 각 요소에 대한 덧셈
print(x - y)  # 배열 x와 배열 y의 각 요소에 대한 뺄셈
print(x * y)  # 배열 x와 배열 y의 각 요소에 대한 곱셈
print(x / y)  # 배열 x와 배열 y의 각 요소에 대한 나눗셈
print(x // y)  # 배열 x와 배열 y의 각 요소에 대한 정수 나눗셈
print(x ** y)  # 배열 x와 배열 y의 각 요소에 대한 거듭제곱 계산
```

```
[ 3  6  9 12 15]
[1 2 3 4 5]
[ 2  8 18 32 50]
[2. 2. 2. 2. 2.]
[2 2 2 2 2]
[     2     16    216   4096 100000]
```

```
import numpy as np  # numpy 모듈 임포트

start = time.time()  # 시작 시간 측정

# 벡터화된 합 계산 - 벡터화를 위해 numpy 사용
# np.arange는 0부터 1499999까지의 숫자 시퀀스를 생성
print(np.sum(np.arange(1500000)))

end = time.time()  # 종료 시간 측정
print(end - start)  # 실행 시간 출력
```

```
-282181552
0.005002021789550781
```

```
import numpy as np  # numpy 모듈 임포트
import pandas as pd  # pandas 모듈 임포트

# 0에서 50 사이의 정수로 이루어진 5,000,000 x 4 크기의 배열 생성
# 배열을 데이터 프레임으로 변환하고 열 이름을 'a', 'b', 'c', 'd'로 설정
df = pd.DataFrame(np.random.randint(0, 50, size=(5000000, 4)), columns=('a','b', 'c', 'd'))

# df.shape: 데이터 프레임의 크기 (형태)를 확인
# 주석에는 df.shape를 실행하여 데이터 프레임의 모양을 확인할 수 있다는 암시가 있지만,
# 코드는 df.shape를 호출하지 않습니다.

# 데이터 프레임의 처음 몇 행을 출력
df.head()
```

	a	b	c	d
0	33	31	6	23
1	40	30	35	17
2	42	25	10	45
3	10	36	15	45
4	5	28	23	39

```
import time
start = time.time()  # 시작 시간 측정

# DataFrame을 iterrows()를 사용하여 반복
for idx, row in df.iterrows():
    # 새로운 열 생성
    # 각 행에 대해 'd' 열을 'c' 열로 나누고 100을 곱하여 'ratio' 열에 저장
    df.at[idx, 'ratio'] = 100 * (row["d"] / row["c"])

end = time.time()  # 종료 시간 측정
print(end - start)  # 실행 시간 출력
```

```
<ipython-input-3-ebe2cacf263b>:7: RuntimeWarning: divide by zero encountered in scalar divide
  df.at[idx,'ratio'] = 100 * (row["d"] / row["c"])
<ipython-input-3-ebe2cacf263b>:7: RuntimeWarning: invalid value encountered in scalar divide
  df.at[idx,'ratio'] = 100 * (row["d"] / row["c"])
387.2491133213043
```

```
import time

start = time.time()  # 시작 시간 측정

# 벡터화된 연산을 사용하여 'ratio' 열 생성
# 'd' 열을 'c' 열로 나누고 100을 곱하여 'ratio' 열에 저장
df['ratio'] = 100 * (df['d'] / df['c'])

end = time.time()  # 종료 시간 측정
print(end - start)  # 실행 시간 출력
```

```
0.12962055206298828
```

Indexing & Slicing

- 배열이 있는 특정 요소를 선택하는 방법은 다양합니다.

```
x = np.arange(-5, 5)  # -5부터 4까지의 숫자를 포함하는 배열 생성
print(x)  # 배열 x 출력

y = x[3:5]  # y는 배열 x의 부분 배열에 대한 슬라이스(포인터)입니다.
print(y)  # y 출력
y[:] = 1000  # y의 값을 수정하면 배열 x도 변경됩니다.
print(y)  # 수정된 y 출력
```

```
print(x)  # y의 수정을 통해 변경된 배열 x 출력

z = x[3:5].copy()  # 부분 배열을 복사하여 z를 만듭니다.
print(z)  # z 출력
z[:] = 500  # z의 값을 수정해도 배열 x에는 영향을 주지 않습니다.
print(z)  # 수정된 z 출력
print(x)  # 배열 x 출력 (y의 수정으로 변경된 후의 상태를 확인할 수 있습니다)
```

💡 주목할 점은 `y`는 `x`의 슬라이스로, `y`의 변경이 `x`에도 영향을 준다는 점입니다. 반면, `z`는 `x`의 슬라이스를 복사한 것이므로 `z`의 변경은 `x`에 영향을 주지 않습니다.

```
[-5 -4 -3 -2 -1  0  1  2  3  4]
[-2 -1]
[1000 1000]
[  -5   -4   -3 1000 1000    0    1    2    3    4]
[1000 1000]
[500 500]
[  -5   -4   -3 1000 1000    0    1    2    3    4]
```

- 비고: List를 슬라이싱하면 하위 List(sublist)의 복사본이 생성되지만 Numpy 배열을 슬라이싱하면 복사본이 생성되지 않습니다.

```
x = list(range(-5,5))
print(x)

y = x[3:5]     # y is a slice, i.e., not a pointer to a list in x
print(y)
y[1] = 1000    # modifying the value of y does not change x
print(y)
print(x)
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
[-2, -1]
[-2, 1000]
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

```
my2dlist = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]  # 2차원 리스트
print(my2dlist)  # 2차원 리스트 출력
print(my2dlist[2])  # 세 번째 하위 리스트에 접근

# `my2dlist[:][2]`는 각 하위 리스트의 세 번째 요소에 접근하려고 하지만, 제대로 된 작동을 하지 않습니다.
# 대신 my2dlist[2][2]를 사용하여 세 번째 하위 리스트의 세 번째 요소에 접근할 수 있습니다.

my2darr = np.array(my2dlist)  # 2차원 리스트를 NumPy 배열로 변환
print(my2darr)  # NumPy 배열 출력
print(my2darr[2][:])  # 세 번째 행에 접근
print(my2darr[2, :])  # 세 번째 행에 접근 (위와 동일한 방법)
print(my2darr[:][2])  # 잘못된 방식; 각 하위 리스트의 세 번째 요소가 아니라 세 번째 행에 접근함
print(my2darr[:, 2])  # 세 번째 열에 접근
print(my2darr[:2, 2:])  # 첫 두 행과 마지막 두 열에 접근
print(my2darr[::2, 2:])  # 두 행씩 건너뛰어 마지막 두 열에 접근
```

💡 `my2darr[2][:]`은 `my2darr` 배열의 세 번째 행에 접근하는 연산.

- `my2darr[2]`: `my2darr` 배열에서 세 번째 행(인덱스 2)을 선택합니다.
  - 인덱싱은 0부터 시작하므로 `my2darr[2]`는 세 번째 행을 선택하는 것입니다.
- `[:]`: 이 부분은 슬라이스 연산자로, 해당 행의 모든 열을 선택합니다.
  - 예를 들어, `my2darr[2][:]`은 세 번째 행의 모든 열을 포함하는 배열을 반환합니다.

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
[9, 10, 11, 12]
[9, 10, 11, 12]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[ 9 10 11 12]
[ 9 10 11 12]
[ 3  7 11]
[[3 4]
 [7 8]]
[[ 3  4]
 [11 12]]
```

- **Remark again: It's indexing, not copying (복사가 아니라 인덱싱입니다)**

```
my2darr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])  # 2차원 리스트를 NumPy 배열로 변환
print(my2darr)  # NumPy 배열 출력

print()
sliced = my2darr[::2, 2:]  # 두 행씩 건너뛰어 마지막 두 열을 슬라이스
print(sliced)  # 슬라이스된 배열 출력
print(type(sliced))  # 슬라이스의 타입 출력 (NumPy 배열임)

print()
sliced[:,:] = 1000  # 슬라이스된 배열의 모든 요소를 1000으로 변경
print(my2darr)  # 원래 배열이 변경됨을 확인 (슬라이스는 원본 배열의 뷰이므로)

print()
sliced[0,0] = 2000  # 슬라이스된 배열의 첫 번째 행 첫 번째 열 요소를 2000으로 변경
print(my2darr)  # 원래 배열이 변경됨을 확인 (슬라이스는 원본 배열의 뷰이므로)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[[ 3  4]
 [11 12]]
<class 'numpy.ndarray'>

[[   1    2 1000 1000]
 [   5    6    7    8]
 [   9   10 1000 1000]]
```

```
[[   1    2 2000 1000]
 [   5    6    7    8]
 [   9   10 1000 1000]]
```

- 비고: list을 슬라이싱하면 하위 목록의 복사본이 생성되지만 Numpy array를 슬라이싱하면 복사본이 생성되지 않습니다.

```
# 주: 리스트 슬라이싱은 하위 리스트의 복사본을 만듭니다.
x = list(range(-5, 5))  # -5부터 4까지의 정수로 이루어진 리스트 생성
print(x)  # 리스트 x 출력

y = x[3:5]  # y는 리스트 x의 하위 리스트에 대한 슬라이스(복사본)입니다.
print(y)  # y 출력

y[1] = 1000  # y의 두 번째 요소를 1000으로 변경
print(y)  # 수정된 y 출력

print(x)  # 리스트 x 출력 (y의 수정을 통해 변경되지 않음)
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
[-2, -1]
[-2, 1000]
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

```
my2dlist = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]  # 2차원 리스트
print(my2dlist)  # 2차원 리스트 출력
print(my2dlist[2])  # 세 번째 하위 리스트에 접근

# `my2dlist[:][2]`는 하위 리스트를 반환하지만, 하위 리스트의 세 번째 요소에 직접적으로 접근하지 못합니다.
# 대신, `my2dlist[2][2]`를 사용하여 세 번째 하위 리스트의 세 번째 요소에 접근할 수 있습니다.
# `my2dlist[:, 2]`는 파이썬의 리스트에서는 사용할 수 없는 구문입니다.

my2darr = np.array(my2dlist)  # 2차원 리스트를 NumPy 배열로 변환
print(my2darr)  # NumPy 배열 출력
print(my2darr[2][:])  # 세 번째 행에 접근 (모든 열 선택)
print(my2darr[2, :])  # 세 번째 행에 접근 (위와 동일한 방법)
print(my2darr[:][2])  # 이 코드는 NumPy 배열에서는 세 번째 행에 접근합니다.
                      # 하지만 파이썬의 2차원 리스트에서는 권장되지 않는 방식입니다.
print(my2darr[:, 2])  # 세 번째 열에 접근
print(my2darr[:2, 2:])  # 첫 두 행과 마지막 두 열에 접근
print(my2darr[::2, 2:])  # 두 행씩 건너뛰어 마지막 두 열에 접근
```

💡 [, ,] → 이런 형식으로 사용

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
[9, 10, 11, 12]
[9, 10, 11, 12]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[ 9 10 11 12]
```

```
[ 9 10 11 12]
[ 9 10 11 12]
[ 3  7 11]
[[3 4]
 [7 8]]
[[ 3  4]
 [11 12]]
```

- **Remark again: It's indexing, not copying**

```
my2darr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])  # 2차원 리스트를 NumPy 배열로 변환
print(my2darr)  # NumPy 배열 출력

print()
sliced = my2darr[::2, 2:]  # 행을 두 행씩 건너뛰어 마지막 두 열을 슬라이스
print(sliced)  # 슬라이스된 배열 출력
print(type(sliced))  # 슬라이스된 배열의 타입 출력 (NumPy 배열임)

print()
sliced[:,:] = 1000  # 슬라이스된 배열의 모든 요소를 1000으로 변경
print(my2darr)  # 원본 배열 출력 (슬라이스는 원본 배열의 뷰이므로 변경됨)

print()
sliced[0,0] = 2000  # 슬라이스된 배열의 첫 번째 행 첫 번째 열 요소를 2000으로 변경
print(my2darr)  # 원본 배열 출력 (슬라이스의 변경이 원본 배열에도 영향을 미침)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[[ 3  4]
 [11 12]]
<class 'numpy.ndarray'>

[[   1    2 1000 1000]
 [   5    6    7    8]
 [   9   10 1000 1000]]

[[   1    2 2000 1000]
 [   5    6    7    8]
 [   9   10 1000 1000]]
```

- ndarray는 부울 인덱싱(마스킹이라고도 함)도 지원합니다.

```
x = np.array([1, 2, 3])  # 1, 2, 3으로 이루어진 NumPy 배열
print(x[1:])  # 슬라이싱: 인덱스 1부터 배열의 끝까지 반환
print(x[1:][0])  # 슬라이싱 후 첫 번째 요소 반환

print()
print(x[[True, False, True]])  # 불리언 마스킹: True 값이 있는 인덱스의 요소만 반환

print()
```

```
print(x[[2, 1]])  # 정수 배열 인덱싱: 인덱스 2와 1의 요소를 반환
print(x[[2, 1, 1, 1, 0]])  # 정수 배열 인덱싱: 인덱스 2, 1, 1, 1, 0의 요소를 순차적으로 반환

print()
x[[2, 1, 1, 1, 0]] = 0  # 정수 배열 인덱싱을 사용하여 인덱스 2, 1, 1, 1, 0의 요소를 0으로 변경
print(x)  # 수정된 NumPy 배열 출력
```

💡 결과적으로 `x` 배열은 인덱싱을 통해 일부 요소를 0으로 변경하게 됩니다.

```
[2 3]
2

[1 3]

[3 2]
[3 2 2 2 1]

[0 0 0]
```

```
y = np.arange(35).reshape(5, 7)  # 0부터 34까지의 숫자를 5x7 형태의 NumPy 배열로 생성
b = y > 20  # y 배열의 각 요소가 20보다 큰지 비교하여 불리언 배열 생성
print(b)  # 불리언 배열 출력

print()
t = y[b]  # 불리언 마스킹을 사용하여 y 배열에서 20보다 큰 요소를 필터링
          # 필터링 결과는 항상 1차원 배열이며, 복사본이 생성됩니다. 인덱싱은 아닙니다.
print(t)  # 필터링 결과 출력

print()
t[:3] = 1000  # 필터링된 배열의 첫 세 요소를 1000으로 변경
print(t)  # 수정된 필터링 결과 출력
print(y)  # 원본 배열 y 출력
```

💡 필터링된 배열 `t`는 `y`의 복사본이므로, `t`의 수정은 `y`에 영향을 미치지 않습니다.

```
[[False False False False False False False]
 [False False False False False False False]
 [False False False False False False False]
 [ True  True  True  True  True  True  True]
 [ True  True  True  True  True  True  True]]

[21 22 23 24 25 26 27 28 29 30 31 32 33 34]

[1000 1000 1000   24   25   26   27   28   29   30   31   32   33   34]
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]]
```

- LAB: 값이 2-d 배열의 짝수인 경우 0으로 설정합니다

```
import numpy as np

M = np.arange(35).reshape(5,7)
M
```

```
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
```

```
M[M % 2 == 0] = 0
M
```

```
array([[ 0,  1,  0,  3,  0,  5,  0],
       [ 7,  0,  9,  0, 11,  0, 13],
       [ 0, 15,  0, 17,  0, 19,  0],
       [21,  0, 23,  0, 25,  0, 27],
       [ 0, 29,  0, 31,  0, 33,  0]])
```

- LAB: 값이 2-d 배열의 짝수인 경우 음수를 사용합니다

```
M[M % 2 == 0] = -M[M % 2 == 0]
M
```

```
M[M % 2 == 0] = -M[M % 2 == 0]
M
array([[  0,   1,  -2,   3,  -4,   5,  -6],
       [  7,  -8,   9, -10,  11, -12,  13],
       [-14,  15, -16,  17, -18,  19, -20],
       [ 21, -22,  23, -24,  25, -26,  27],
       [-28,  29, -30,  31, -32,  33, -34]])
```

```
# First Boolean masking

np.where(M % 2 == 0, M, -M)
```

```
array([[  0,  -1,   0,  -3,   0,  -5,   0],
       [ -7,   0,  -9,   0, -11,   0, -13],
       [  0, -15,   0, -17,   0, -19,   0],
       [-21,   0, -23,   0, -25,   0, -27],
       [  0, -29,   0, -31,   0, -33,   0]])
```

- More indexing examples: **Integer array indexing**

```
my2darr = np.arange(1, 13, 1).reshape(4, 3)  # 1부터 12까지의 숫자를 4x3 형태의 NumPy 배열로 생성
print(my2darr)  # 배열 출력

indices = [2, 1, 0, 3]  # 선택한 행 인덱스
print(my2darr[indices, :])  # 선택한 행 인덱스에 따라 전체 열을 포함하는 배열을 반환

rowIndex = [0, 0, 1, 2, 3]  # my2darr에 대한 행 인덱스 배열
columnIndex = [0, 2, 0, 1, 2]  # my2darr에 대한 열 인덱스 배열
print(my2darr[rowIndex, columnIndex])  # 요소별로 인덱싱하여 배열 요소 출력
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[ 7  8  9]
 [ 4  5  6]
 [ 1  2  3]
 [10 11 12]]
[ 1  3  4  8 12]
```

Numpy 산술 및 통계 함수

- nd-array의 요소를 조작하는 데 사용할 수 있는 많은 수학 기능이 내장되어 있습니다.

```
y = np.array([-1.4, 0.4, -3.2, 2.5, 3.4])  # 무작위로 생성된 벡터
print(y)  # 배열 출력
print()

print(np.abs(y))  # 각 요소를 절대값으로 변환
print(np.sqrt(np.abs(y)))  # 각 요소의 절대값에 대한 제곱근 적용
print(np.sign(y))  # 각 요소의 부호 구하기
print(np.exp(y))  # 각 요소에 대한 지수함수(e^y) 적용
print(np.sort(y))  # 배열 정렬; 배열의 정렬된 복사본 반환
print(y)  # 원본 배열 y 출력; 원본 배열은 변하지 않음
```

```
[-1.4  0.4 -3.2  2.5  3.4]

[1.4 0.4 3.2 2.5 3.4]
[1.18321596 0.63245553 1.78885438 1.58113883 1.84390889]
[-1.  1. -1.  1.  1.]
[ 0.24659696  1.4918247   0.0407622  12.18249396 29.96410005]
[-3.2 -1.4  0.4  2.5  3.4]
[-1.4  0.4 -3.2  2.5  3.4]
```

```
x = np.arange(-2, 3)  # -2부터 2까지의 정수 배열 생성
y = np.random.randn(5)  # 5개의 임의의 표준 정규분포 값 배열 생성
print(x)  # 배열 x 출력
print(y)  # 배열 y 출력

print(np.add(x, y))  # 요소별 덧셈: x + y
print(np.subtract(x, y))  # 요소별 뺄셈: x - y
print(np.multiply(x, y))  # 요소별 곱셈: x * y
```

```
print(np.divide(x, y))  # 요소별 나눗셈: x / y
print(np.maximum(x, y))  # 요소별 최대값: max(x, y)
```

```
[-2 -1  0  1  2]
[-0.5915882   0.86652443 -0.76206247 -0.18749133 -0.65986152]
[-2.5915882  -0.13347557 -0.76206247  0.81250867  1.34013848]
[-1.4084118  -1.86652443  0.76206247  1.18749133  2.65986152]
[ 1.1831764  -0.86652443 -0.         -0.18749133 -1.31972304]
[ 3.38073005 -1.15403555 -0.         -5.33357984 -3.03093897]
[-0.5915882   0.86652443  0.          1.          2.        ]
```

```
y = np.array([-3.2, -1.4, 0.4, 2.5, 3.4])  # 무작위로 생성된 벡터
print(y)  # 배열 y 출력

print("Min =", np.min(y))  # 배열의 최소값
print("Max =", np.max(y))  # 배열의 최대값
print("Average =", np.mean(y))  # 배열의 평균/평균값
print("Std deviation =", np.std(y))  # 배열의 표준편차
print("Sum =", np.sum(y))  # 배열의 합계
```

```
[-3.2 -1.4  0.4  2.5  3.4]
Min = -3.2
Max = 3.4
Average = 0.34000000000000014
Std deviation = 2.432776191925595
Sum = 1.7000000000000006
```

More on filtering

```
M = np.arange(25).reshape(5, 5)  # 0부터 24까지의 숫자를 5x5 형태의 배열로 생성
print(M)  # 배열 M 출력

print(M[M % 2 == 1])  # 일반적인 필터링: M 배열의 요소 중 홀수(조건을 충족하는 요소)만 필터링하여 반환
print(np.argwhere(M >= 20))  # 조건을 충족하는 요소의 인덱스 반환
print(np.where(M % 2 == 1, M, 0))  # 조건에 따라 M 요소는 그대로, 조건에 맞지 않으면 0으로 대체
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
[ 1  3  5  7  9 11 13 15 17 19 21 23]
[[4 0]
 [4 1]
 [4 2]
 [4 3]
 [4 4]]
[[ 0  1  0  3  0]
 [ 5  0  7  0  9]
 [ 0 11  0 13  0]
```

```
 [15  0 17  0 19]
 [ 0 21  0 23  0]]
```

New axis

- 기존 배열의 차원을 한 차원 더 늘리는 데 사용됩니다
- 모양 변경 예: n x (새 축) x m ⇒ n x 1 x m

```
t = np.array([1,2,3])  # 1차원 배열 생성
print(t.shape)  # 배열 t의 형상(차원) 출력

x = t[:, np.newaxis]  # np.newaxis를 사용하여 차원 확장 (x 1)
print(x.shape)  # 배열 x의 형상 출력

y = t[np.newaxis, :]  # np.newaxis를 사용하여 차원 확장
print(y.shape)  # 배열 y의 형상 출력
```

💡 설명

- `t.shape`는 배열 `t`의 형상을 출력합니다. `t`는 1차원 배열이므로 형상은 `(3,)`입니다.
- `x = t[:, np.newaxis]`는 `np.newaxis`를 사용하여 `t` 배열의 차원을 확장합니다. 기존의 1차원 배열 `t`가 `(3,)`에서 `(3,1)`의 2차원 배열로 확장됩니다. 이 때 `:`는 전체 배열을 선택하고, `np.newaxis`는 새로운 차원을 추가합니다.
- `y = t[np.newaxis, :]`는 `np.newaxis`를 사용하여 `t` 배열의 차원을 확장합니다. 기존의 1차원 배열 `t`가 `(3,)`에서 `(1,3)`의 2차원 배열로 확장됩니다. 이 때 `np.newaxis`는 새로운 차원을 추가하고, `:`는 전체 배열을 선택합니다.

```
(3,)
(3, 1)
(1, 3)
```

```
print(t)
print(x)
print(y)
```

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

💡 출력 결과 이유

- `t` 배열은 1차원 배열로, `[1, 2, 3]`의 값을 갖습니다.
- `x` 배열은 `t` 배열에 `np.newaxis`를 사용하여 열 차원을 추가한 결과로, `(3, 1)`의 2차원 배열이 됩니다. 따라서 출력 결과는 각 요소가 개별적으로 한 줄에 위치한 열 형태의 2차원 배열 `[[1], [2], [3]]`이 됩니다.
- `y` 배열은 `t` 배열에 `np.newaxis`를 사용하여 행 차원을 추가한 결과로, `(1, 3)`의 2차원 배열이 됩니다. 따라서 출력 결과는 `t` 배열의 요소들이 한 행에 위치한 형태의 2차원 배열 `[[1, 2, 3]]`이 됩니다.

- **x and y are broadcasted** (it is explained below)

```
x + y

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

- 위치에 따라 Dimension 위치가 달라지는지?

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

# 1. 차원을 확장하여 결과의 형상을 확인합니다.
print(t[:, :, np.newaxis].shape)  # (2, 3, 1) 모양의 3차원 배열
print(t[:, :, np.newaxis])  # 차원을 확장한 배열 출력
print()

# 2. 차원을 확장하여 결과의 형상을 확인합니다.
print(t[np.newaxis, :, :].shape)  # (1, 2, 3) 모양의 3차원 배열
print(t[np.newaxis, :, :])  # 차원을 확장한 배열 출력
print()

# 3. 행 차원을 확장하여 결과를 확인합니다.
print(t[:, np.newaxis, :])
```

💡 설명

1. `t[:, :, np.newaxis]`:
   - `:`를 사용하여 전체 행과 열을 선택합니다.
   - `np.newaxis`는 열 차원을 확장합니다.
   - 결과는 `(2, 3, 1)` 형상의 3차원 배열이 됩니다.
2. `t[np.newaxis, :, :]`:
   - `np.newaxis`를 사용하여 행 차원을 확장합니다.
   - `:`를 사용하여 전체 배열을 선택합니다.
   - 결과는 `(1, 2, 3)` 형상의 3차원 배열이 됩니다.
3. `t[:, np.newaxis, :]`:
   - `:`를 사용하여 전체 행을 선택합니다.
   - `np.newaxis`를 사용하여 새로운 축을 추가합니다.
   - `:`를 사용하여 전체 열을 선택합니다.
   - 결과는 `np.array([[1,2,3],[4,5,6]])`의 3차원 배열이 됩니다.

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

 [[4]
```

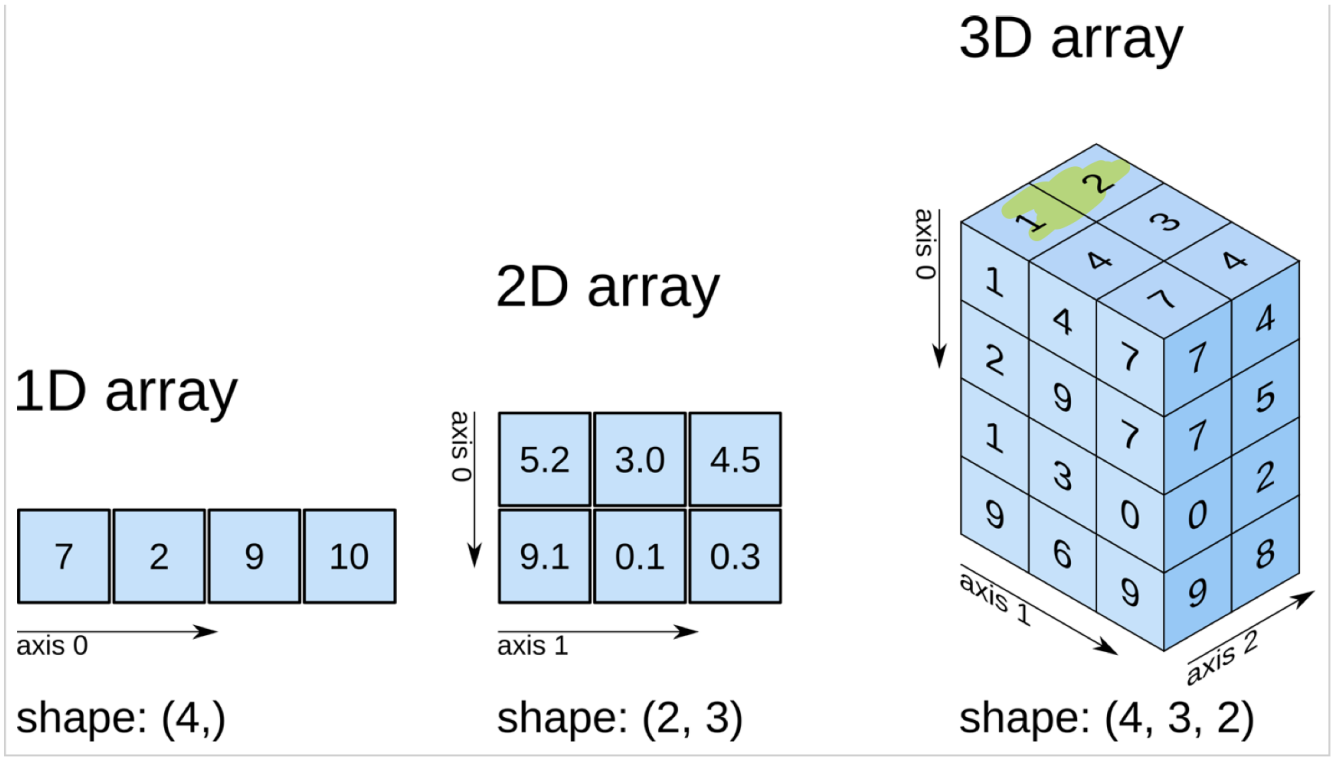
```
  [5]
  [6]]]

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

[[[1 2 3]]

 [[4 5 6]]]
```

N-d array axis view



Broadcasting

- 브로드캐스트는 산술 연산 중에 numpy가 다양한 모양을 가진 배열을 어떻게 처리하는지 설명합니다.
- 특정 제약 조건에 따라 더 작은 배열은 더 큰 배열에 걸쳐 "브로드캐스트"되어 호환 가능한 모양을 갖습니다.

Examples:

```
A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  5 x 4

A      (2d array):  5 x 4
B      (1d array):      4
Result (2d array):  5 x 4

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
```

```
B      (2d array):       3 x 1
Result (3d array):  15 x 3 x 5
```

```
np.array([[1,2],[3,4]]) + np.array([[10]])
```

```
array([[11, 12],
       [13, 14]])
```

```
np.array([[1,2],[3,4]]) + np.array([[10,100]])
```

```
array([[ 11, 102],
       [ 13, 104]])
```

```
A = np.array([[1,2]])
B = np.array([[10],[100]])
print(A.shape, B.shape)
C = A + B
C
```

```
(1, 2) (2, 1)
array([[ 11,  12],
       [101, 102]])
```

```
X = np.array([[1]]*3) + np.array([[0]*10]) # 3 * 1, 1 * 10
X
```

```


X = np.array([[1]]*3) + np.array([[0]*10])
X
array([[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]])
```

💡 `np.array([[1]]*3)`는 `[[1]]` 배열을 3번 반복하여 2차원 배열로 만드는 연산입니다. 이 경우, 형상은 `(3, 1)`이 되며 결과는 `[[1], [1], [1]]`입니다.

`np.array([[0]*10])`는 `[0]`을 10번 반복하여 길이가 10인 2차원 배열을 만듭니다. 이 경우, 형상은 `(1, 10)`이 되며 결과는 `[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]`입니다.

- `[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]`가 3번 반복된 `3 x 10` 배열입니다.

```
# 배열 a 생성 (3x1 크기)
a = np.array([[1], [2], [3]])
```

```
# 배열 a의 전치(행렬의 전치)를 계산하여 배열 b에 저장 (1x3 크기)
b = a.T

# a와 b를 더한 결과를 계산 (broadcasting 기능 사용)
result = a + b

# 결과 배열 출력
result
```

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

Meshgrid

- 벡터화된 평가를 위해 D x N 메쉬 그리드 만들기

```
v = np.array([10,20,30])   # N
w = np.array([5,6])        # D
X, Y = np.meshgrid(v, w)
X + Y
```

```
array([[15, 25, 35],
       [16, 26, 36]])
```

Axis ordering

- 정의상 차원의 축 번호는 배열의 모양 안에서 해당 차원의 인덱스입니다.
  - 인덱싱하는 동안 해당 차원에 액세스하는 데 사용되는 위치이기도 합니다.
- 예를 들어, 2D 배열 a의 모양이 (5,6)이면 a[4,5]까지 a[0,0]에 접근할 수 있습니다.
  - 따라서 축 0은 첫 번째 차원("행")이고, 축 1은 두 번째 차원("열")입니다.
  - "행"과 "열"이 의미가 없는 고차원에서는 축을 관련된 모양과 지수로 생각해 보십시오.
- 예를 들어 np.sum(axis=n)을 하면 차원 n이 축소되고 삭제되며 새 행렬의 각 값은 축소된 값의 합과 같습니다.
  - 예를 들어 b의 모양이 (5,6,7,8)이고 c = b.sum(axis=2)이면 축 2(크기 7의 dimension)가 축소되고 결과는 모양이 (5,6,8)됩니다.
  - 또한 c[x,y,z]는 모든 원소 b[x,y,:,z]의 합과 같습니다.

```
X = np.array([[0,0,0], [1,1,1]])
X.shape

# axis 0 is row; axis 1 is column

# Result: (2, 3)
```

```
X.sum(axis=0) # 차원 0이 축소 및 삭제되거나 차원 0에 대해 집계됩니다

# Result: array([1, 1, 1])
```

```
X.sum(axis=1) # 차원 1이 축소 및 삭제되거나 차원 0에 대해 집계됩니다
```

```
# Result: array([0, 3])
```

```
# 1부터 24까지의 정수로 구성된 1차원 배열을 생성합니다.
X = np.array(range(1, 24 + 1))

# 배열 X를 (2, 3, 4) 형상으로 재구조화하여 3차원 배열로 변환합니다.
# 이때, 배열은 2개의 3x4 행렬로 구성됩니다.
X = X.reshape(2, 3, 4)

# 재구조화된 3차원 배열 X를 출력합니다.
X
```

```
X.shape

# (2, 3, 4)
```

```
X.sum(axis=0)
```

```
array([[14, 16, 18, 20],
       [22, 24, 26, 28],
       [30, 32, 34, 36]])
```

💡 `X`는 `np.arange(24).reshape(2, 3, 4)`을 통해 만들어진 3차원 배열

- `X[0]`: 첫 번째 3x4 행렬

```
[[[ 1,  2,  3,  4],
  [ 5,  6,  7,  8],
  [ 9, 10, 11, 12]],
```

- `X[1]`: 두 번째 3x4 행렬

```
[[13, 14, 15, 16],
  [17, 18, 19, 20],
  [21, 22, 23, 24]]]
```

- `axis=0`을 따라 합계를 계산.
- 첫 번째 열: `[1+13, 5+17, 9+21]` = `[14, 22, 30]`
- 두 번째 열: `[2+14, 6+18, 10+22]` = `[16, 24, 32]`
- 세 번째 열: `[3+15, 7+19, 11+23]` = `[18, 26, 34]`
- 네 번째 열: `[4+16, 8+20, 12+24]` = `[20, 28, 36]`

💡 `axis=0`는 배열에서 첫 번째 축을 나타냅니다. 배열의 축(axis)은 각 배열의 차원을 나타내며, `axis`는 축의 인덱스를 가리킵니다.

2차원 배열인 경우:

- `axis=0`는 **행**을 의미합니다. `axis=0`을 따라 합산한다는 것은 각 **열**을 따라 값들을 합산하는 것을 의미합니다. 따라서, `axis=0`으로 합산하면 결과로 각 열의 값들을 합산한 값들이 반환됩니다
- `axis=0`을 따라 합산할 때는 각 열의 값들을 합산하여 열별로 결과를 반환합니다.

```
X.sum(axis=1)
```

```
array([[15, 18, 21, 24],
       [51, 54, 57, 60]])
```

💡 설명

- `X.sum(axis=1)`을 실행하면 각 '층'에서 동일한 열에 위치한 요소들의 합을 구하게 됩니다. 따라서 각 '층'의 열별 합은 다음과 같습니다:
- **첫 번째 '층':**
  - **첫 번째 열의 합: 1 + 5 + 9 = 15**
  - **두 번째 열의 합: 2 + 6 + 10 = 18**
  - **세 번째 열의 합: 3 + 7 + 11 = 21**
  - **네 번째 열의 합: 4 + 8 + 12 = 24**
- **두 번째 '층':**
  - **첫 번째 열의 합: 13 + 17 + 21 = 51**
  - **두 번째 열의 합: 14 + 18 + 22 = 54**
  - **세 번째 열의 합: 15 + 19 + 23 = 57**
  - **네 번째 열의 합: 16 + 20 + 24 = 60**

```
X.sum(axis=2)
```

```
array([[10, 26, 42],
       [58, 74, 90]])
```

💡 설명

- `X.sum(axis=2)`을 실행하면, 각 '층'의 각 행에 있는 요소들의 합을 구합니다:
- 첫 번째 '층':
  - 첫 번째 행의 합: 1 + 2 + 3 + 4 = 10
  - 두 번째 행의 합: 5 + 6 + 7 + 8 = 26
  - 세 번째 행의 합: 9 + 10 + 11 + 12 = 42
- 두 번째 '층':
  - 첫 번째 행의 합: 13 + 14 + 15 + 16 = 58
  - 두 번째 행의 합: 17 + 18 + 19 + 20 = 74
  - 세 번째 행의 합: 21 + 22 + 23 + 24 = 90

```
X.sum(axis=(1,2))
```

```
array([ 78, 222])
```

💡 설명

`X.sum(axis=(1,2))`을 실행하면, 각 '층'에서 모든 행과 열에 있는 요소들의 총합을 구합니다:

첫 번째 '층'의 합:

(1 + 2 + 3 + 4) + (5 + 6 + 7 + 8) + (9 + 10 + 11 + 12) = 10 + 26 + 42 = 78

두 번째 '층'의 합:

(13 + 14 + 15 + 16) + (17 + 18 + 19 + 20) + (21 + 22 + 23 + 24) = 58 + 74 + 90 = 222

결과적으로, `X.sum(axis=(1,2))`의 결과는 다음과 같은 배열이 됩니다:

```
X.sum(axis=(0,1,2))

# Result: 300
```

💡 설명

`X.sum(axis=(0,1,2))`을 실행하면, 배열의 모든 요소들의 총합을 계산합니다:

첫 번째 '층': (1 + 2 + 3 + 4) + (5 + 6 + 7 + 8) + (9 + 10 + 11 + 12) = 78

두 번째 '층': (13 + 14 + 15 + 16) + (17 + 18 + 19 + 20) + (21 + 22 + 23 + 24) = 222

모든 '층'의 합계:

78 + 222 = 300

따라서, `X.sum(axis=(0,1,2))`의 결과는 300이 됩니다. 이는 배열 내 모든 요소의 총합을 나타내는 스칼라 값입니다.

```
# 2개의 3차원 점 X와 Y를 선언
X = np.array([0, 0, 0])  # 첫 번째 3차원 점 X
Y = np.array([1, 1, 1])  # 두 번째 3차원 점 Y
```

```
# 두 점 X와 Y 사이의 유클리드 거리 계산
distance = np.sqrt(np.sum((X - Y)**2))  # 차이 벡터(X - Y)의 제곱을 구하고, 합을 계산한 후 제곱근을 구함

print(distance)  # 계산된 거리를 출력

# Result: 1.7320508075688772
```

```
import numpy as np

# 2x3 배열을 역순으로 생성
X = np.array(np.arange(2 * 3, 0, -1).reshape(2, 3))
print(X)  # 생성된 배열 출력

print()  # 줄 바꿈

# axis=0에 대해 배열을 정렬
print("axis=0\n", np.sort(X, axis=0))

print()  # 줄 바꿈

# axis=-1(또는 axis=1)에 대해 배열을 정렬
print("axis=-1\n", np.sort(X, axis=-1))

print()  # 줄 바꿈

# 기본 축은 axis=-1(또는 axis=1)이므로 동일한 결과를 출력
print("default is -1\n", np.sort(X))

print()  # 줄 바꿈

# axis=None을 사용하여 전체 배열을 1차원으로 정렬
print("axis=None\n", np.sort(X, axis=None))
```

```
[[6 5 4]
 [3 2 1]]

axis=0
 [[3 2 1]
 [6 5 4]]

axis=-1
 [[4 5 6]
 [1 2 3]]

default is -1
 [[4 5 6]
 [1 2 3]]

axis=None
 [1 2 3 4 5 6]
```

💡 설명

- **코드**:
  - `X`: 2×3 형태의 배열로, 2개의 행과 3개의 열을 가진 배열입니다. 이 배열은 `np.arange(2*3, 0, -1).reshape(2,3)`를 사용하여 6부터 1까지의 숫자를 2x3 배열 형태로 정렬한 것입니다.
- **정렬 결과**:
  - `axis=0`: 이 옵션을 사용하면 각 열을 따라 배열이 정렬됩니다. `X` 배열의 각 열을 정렬하면 다음과 같습니다:
    - 첫 번째 열: [3, 6]
    - 두 번째 열: [2, 5]
    - 세 번째 열: [1, 4]
  - `axis=-1` 또는 `axis=1`: 이 옵션을 사용하면 각 행을 따라 배열이 정렬됩니다. `X` 배열의 각 행을 정렬하면 다음과 같습니다:
    - 첫 번째 행: [1, 2, 3]
    - 두 번째 행: [4, 5, 6]
  - `axis=None`: 이 옵션을 사용하면 배열이 1차원으로 펼쳐진 다음 정렬됩니다. `X` 배열을 1차원으로 펼친 후 정렬하면 `[1, 2, 3, 4, 5, 6]`가 됩니다.
- **요약**:
  - `axis=0`: 각 열을 따라 정렬합니다.
  - `axis=1` 또는 `axis=-1`: 각 행을 따라 정렬합니다.
  - `axis=None`: 배열을 1차원으로 펼친 후 전체를 정렬합니다.

sort vs argsort vs partition vs argpartition

- argmin, argmax, ...

```
# 배열 `X` 초기화
X = np.array([4,10,1,20,45,100,2,1])
print('X =\n', X)

# 배열 `X`의 요소를 오름차순으로 정렬
print('sorted =\n', np.sort(X))

# 배열 `X`를 정렬했을 때 요소의 원래 인덱스를 반환
print('argsorted =\n', np.argsort(X))

# 배열 `X`에서 `3`번째 작은 값이 위치해야 할 자리를 기준으로 부분적으로 정렬
# 첫 `3`개의 요소는 작은 값들로 구성되고, 나머지 요소들은 아직 정렬되지 않은 상태로 배열됨
print('partitioned first 3 =\n', np.partition(X, 3))

# 배열 `X`에서 `3`번째 작은 값이 위치해야 할 자리를 기준으로 부분 정렬했을 때의 요소의 인덱스를 반환
print('argpartitioned first 3 =\n', np.argpartition(X, 3))

# 배열 `X`에서 `-3`번째 (마지막 세 번째) 큰 값이 위치해야 할 자리를 기준으로 배열을 부분적으로 정렬
# 마지막 `3`개의 요소는 큰 값들로 구성되고, 나머지 요소들은 아직 정렬되지 않은 상태로 배열됨
print('partitioned last 3=\n', np.partition(X, -3))

# 배열 `X`에서 `-3`번째 (마지막 세 번째) 큰 값이 위치해야 할 자리를 기준으로 부분 정렬했을 때의 요소의 인덱스를 반
print('argpartitioned last 3=\n', np.argpartition(X, -3))
```

```
X =
 [  4  10   1  20  45 100   2   1]
sorted =
 [  1   1   2   4  10  20  45 100]
argsorted =
 [2 7 6 0 1 3 4 5]
partitioned first 3 =
 [  2   1   1   4  45 100  10  20]
argpartitioned first 3 =
 [6 7 2 0 4 5 1 3]
partitioned last 3=
 [  2   1   1   4  10  20  45 100]
argpartitioned last 3=
 [6 7 2 0 1 3 4 5]
```

Lab.

- 축 0을 따라 2-d 배열 T를 정렬하고, 정렬 키는 축 1을 따라 원소의 합입니다.

```
T = np.array([[2,2],[-1,10],[0,1]])  # 2차원 배열 T를 초기화
I = np.argsort(np.sum(T, axis=1))    # 축 1을 따라 각 행의 합계를 구한 후, 그 합계의 인덱스를 오름차순으로 
T[I, :]                             # 정렬된 인덱스를 사용하여 배열 T의 행을 재정렬
```

💡 설명.

`T = np.array([[2,2],[-1,10],[0,1]])` 은 2차원 배열 `T` 를 초기화합니다. 이 배열은 다음과 같습니다:

```
[[ 2,  2],
 [-1, 10],
 [ 0,  1]]
```

`I = np.argsort(np.sum(T, axis=1))` 명령어는 두 부분으로 나뉩니다:

`np.sum(T, axis=1)` 는 배열 `T` 의 각 행에 대한 합계를 계산합니다.

따라서, 각 행의 합계는 `[4, 9, 1]` 이 됩니다.

`np.argsort(...)` 는 주어진 배열의 요소를 오름차순으로 정렬했을 때의 인덱스를 반환합니다.

`[4, 9, 1]` 의 요소를 오름차순으로 정렬하면 `[1, 4, 9]` 가 되고, 이에 해당하는 원래 배열의 인덱스는 `[2, 0, 1]` 입니다. `T[I, :]` 는 배열 `T` 의 행을 `I` 배열에 따라 재정렬합니다.

`I` 는 `[2, 0, 1]` 이므로, `T` 의 행도 이 인덱스 순서대로 재배치됩니다.

이는 다음과 같은 순서를 의미합니다:

`T` 의 2번째 행이 첫 번째 위치로 이동합니다.

(`[0, 1]`) `T` 의 0번째 행이 두 번째 위치로 이동합니다.

(`[2, 2]`) `T` 의 1번째 행이 세 번째 위치로 이동합니다. (`[-1, 10]`)

결과적으로, `T[I, :]` 를 실행하면 다음과 같은 배열이 생성됩니다:

```
[[ 0,  1],
 [ 2,  2],
 [-1, 10]]
```

이 배열은 원래 배열 `T` 의 행을 각 행의 합계가 작은 순서대로 재정렬한 것입니다.

첫 번째 행의 합계가 가장 작고(`[0, 1]` 의 합계는 1), 다음으로 `[2, 2]` 의 합계는 4,

마지막으로 `[-1, 10]` 의 합계는 9로, 오름차순으로 정렬된 순서를 반영합니다.

```
array([[ 0,  1],
       [ 2,  2],
       [-1, 10]])
```

Vectorized function

- map 함수와 유사

```
import math

# 주어진 값 중 절대값을 기준으로 가장 큰 값을 찾습니다.
# 주어진 값: 1, 2, 3, 4, 5, -100
# key 매개변수로 lambda 함수를 사용하여 각 값의 절대값을 비교합니다.
max_value = max(1, 2, 3, 4, 5, -100, key=lambda x: math.fabs(x))

# 가장 큰 절대값을 가진 값(이 경우 -100)을 반환합니다.
print(max_value)

# Result: -100
```

```
from functools import partial
import math

# functools 모듈에서 partial 함수를 불러옵니다.

# max 함수의 key 매개변수에 lambda 함수를 사용하여 각 입력값의 절대값을 기준으로 최대 값을 찾도록 부분 적용합니다.
mymax = partial(max, key=lambda x: math.fabs(x))

# 이제 mymax 함수를 사용할 때마다 자동으로 key 매개변수에 lambda 함수가 적용됩니다.
```

```
# 두 리스트 [10, 2, 3]와 [4, 5, 6]을 비교하여 각 요소에서 더 큰 값을 반환합니다.
result = list(map(max, [10,2,3], [4,5,6]))

# 결과는 [10, 5, 6]입니다. 이는 각 인덱스에서 더 큰 값을 반환한 것입니다.
print(result)  # [10, 5, 6]
```

💡 설명

`map` 함수는 주어진 두 리스트를 `max` 함수에 각각의 요소가 대응하도록 매핑합니다. 즉, `max` 함수는 각 인덱스에 해당하는 요소를 비교하여 더 큰 값을 반환합니다.

- 첫 번째 요소 비교: `max(10, 4)`는 10과 4를 비교하여 더 큰 값인 10을 반환합니다.
- 두 번째 요소 비교: `max(2, 5)`는 2와 5를 비교하여 더 큰 값인 5를 반환합니다.
- 세 번째 요소 비교: `max(3, 6)`는 3과 6을 비교하여 더 큰 값인 6을 반환합니다.

```
list(map(mymax, [-10,2,3], [4,5,-6]))

# Result: [-10, 5, -6]
```

```
u = np.array([100,2,3,4])
v = np.array([1,2,3,4])
w = np.array([4,3,2,1])
np.vectorize(max)(u, v, w)

# array([100,   3,   3,   4])
```

```
dist = np.vectorize(lambda x, y: np.sqrt(x**2 + y**2))
dist(v, w)

# array([4.12310563, 3.60555128, 3.60555128, 4.12310563])
```

💡 설명

위와 같이 `np.vectorize` 는 람다 함수를 벡터화하여 각 요소에 적용하고, 결과를 배열로 반환합니다. 계산된 값은 다음과 같습니다:

- 첫 번째 요소 쌍에 대한 거리: `sqrt(v[0]**2 + w[0]**2)`
- 두 번째 요소 쌍에 대한 거리: `sqrt(v[1]**2 + w[1]**2)`
- 세 번째 요소 쌍에 대한 거리: `sqrt(v[2]**2 + w[2]**2)`
- 네 번째 요소 쌍에 대한 거리: `sqrt(v[3]**2 + w[3]**2)`

결과적으로 `[4.12310563, 3.60555128, 3.60555128, 4.12310563]` 가 반환됩니다.

- # 3D 포인트 0에서 계산된 벡터화된 거리를 계산합니다.

```
import numpy as np

def calculate_euclidean_distances(points):
    # 이 함수는 입력된 3D 점들의 배열에서 원점으로부터의 유클리드 거리를 계산합니다.
    # points: 각 행이 [x, y, z]로 표현된 3D 점을 나타내는 2차원 배열.

    # points 배열의 각 요소의 제곱을 계산하고, 각 점의 x^2 + y^2 + z^2를 계산하기 위해 축 1을 따라 합을 구합니
    squared_sum = np.sum(np.square(points), axis=1)

    # 제곱의 합의 제곱근을 계산하여 각 점에 대한 유클리드 거리를 구합니다.
    distances = np.sqrt(squared_sum)

    return distances

# 예시 사용법:
# 3D 점의 배열을 정의합니다.
points = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [-1, -2, -3]
])

# 원점(0, 0, 0)에서 유클리드 거리를 계산합니다.
distances = calculate_euclidean_distances(points)


# 거리를 출력합니다.
print(distances)
```

💡 설명.

- `calculate_euclidean_distances` 함수는 각 행이 3D 점 `[x, y, z]` 를 나타내는 2차원 배열 `points` 를 받습니다.
- 함수 내부에서 `points` 배열의 각 요소의 제곱을 계산한 후, 축 1을 따라 합을 구합니다(각 점의 *x***2+ *y***2 + *z***2를 합산).
- 제곱의 합의 제곱근을 계산하여 각 점에 대한 유클리드 거리를 구합니다.
- 함수는 원점에서 각 점까지의 거리를 반환합니다.

Numpy linear algebra (Numpy 선형 대수)

- Numpy는 선형 대수 연산을 지원하기 위해 많은 함수를 제공합니다.

```
import numpy as np

# 2 x 3 랜덤 행렬 생성
X = np.random.randn(2, 3)
print(X)  # 행렬 X 출력

# 행렬 X의 전치(transpose) 계산
print(X.T)  # 행렬 X의 전치 출력

# 크기가 3인 랜덤 벡터 생성
y = np.random.randn(3)
print(y)  # 벡터 y 출력

# 행렬 X와 벡터 y의 행렬-벡터 곱셈
print(X.dot(y))  # 행렬-벡터 곱셈 결과 출력

# np.dot() 함수를 사용하여 행렬 X와 벡터 y의 곱셈
print(np.dot(X, y))  # 행렬-벡터 곱셈 결과 출력 (X.dot(y)와 동일)

# 행렬 X와 X의 전치(X.T)의 행렬-행렬 곱셈
print(X.dot(X.T))  # 행렬-행렬 곱셈 결과 출력

# 행렬 X의 전치(X.T)와 행렬 X의 행렬-행렬 곱셈
print(X.T.dot(X))  # 행렬-행렬 곱셈 결과 출력
```

```
[[-0.67521745 -0.25112232 -0.53902013]
 [-0.31444559  0.26792464 -0.91960302]]
[[-0.67521745 -0.31444559]
 [-0.25112232  0.26792464]
 [-0.53902013 -0.91960302]]
[-0.27857094  0.11301957 -0.0460988 ]
[0.1845624  0.16026873]
[0.1845624  0.16026873]
[[0.80952372 0.64072183]
 [0.64072183 1.01632936]]
[[ 0.55479463  0.08531445  0.65312091]
 [ 0.08531445  0.13484603 -0.11102433]
 [ 0.65312091 -0.11102433  1.13621241]]
```

```
y.dot(y) # 벡터 y와 자신 사이의 내적(dot product)을 계산
# 벡터 y와 자신 사이의 내적(dot product)을 계산

0.09250029216474008
```

```
import numpy as np

# 5x3 차원의 랜덤 행렬 X 생성
X = np.random.randn(5, 3)
print(X)

# X^T * X를 계산하여 C에 할당합니다. 이 결과는 3x3 크기의 정사각행렬입니다.
C = X.T.dot(X)
```

```
print("C = X^T * X:\n", C)

# C의 역행렬을 계산합니다. -> X * X**-1 = I
invC = np.linalg.inv(C)
print("C의 역행렬:\n", invC)

# C의 행렬식(determinant)을 계산합니다.
detC = np.linalg.det(C)
print("C의 행렬식:", detC)

# C의 고유값(eigenvalue) S와 고유벡터(eigenvector) U를 계산합니다.
S, U = np.linalg.eig(C)
print("C의 고유값 S:\n", S)
print("C의 고유벡터 U:\n", U)
```

💡 설명.

- `X`: 5×3 크기의 랜덤 행렬을 생성합니다.
- `C`: `X^T * X` 연산을 수행하여 정사각 행렬 C를 계산합니다.
- `invC`: `np.linalg.inv(C)`를 사용하여 C의 역행렬을 계산합니다.
- `detC`: `np.linalg.det(C)`를 사용하여 C의 행렬식을 계산합니다.
- `S`, `U`: `np.linalg.eig(C)`를 사용하여 C의 고유값 S와 고유벡터 U를 계산합니다.

```
[[ 1.00517715 -0.29554381 -1.29674166]
 [ 1.28813155  0.05589876 -0.22072513]
 [ 0.46327488  0.5101119  -1.30901555]
 [-0.68836097  0.50845609 -0.06891248]
 [-0.79042016  1.3979979  -1.01016794]]
C = X^T * X:
 [[ 3.98289246 -1.44375394 -1.34831834]
 [-1.44375394  2.56361068 -1.74409033]
 [-1.34831834 -1.74409033  4.46896842]]
C의 역행렬:
 [[0.6600184  0.69051673 0.46861787]
 [0.69051673 1.25350588 0.69753544]
 [0.46861787 0.69753544 0.63737548]]
C의 행렬식: 12.749408257332224
C의 고유값 S:
 [0.46098261 4.83874418 5.71574477]
C의 고유벡터 U:
 [[ 0.48384751  0.75707829 -0.43900347]
 [ 0.73115789 -0.6253672  -0.27262429]
 [ 0.4809363   0.18907227  0.85612613]]
```

```
import numpy as np

# 2x2 크기의 행렬 L을 초기화합니다.
L = np.array([[2, 0], [0, 1]])

# 행렬 L의 고유값(eigenvalue) S와 고유벡터(eigenvector) U를 계산합니다.
S, U = np.linalg.eig(L)
```

```
# 계산된 고유값 S와 고유벡터 U를 출력합니다.
print("고유값 S:", S)
print("고유벡터 U:\n", U)
```

```
고유값 S: [2. 1.]
고유벡터 U:
 [[1. 0.]
 [0. 1.]]
```

```
v = np.array([1,1])
```

```
v = L.dot(v)
v

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

💡 `L.dot(v)`의 결과는 다음과 같습니다: `L.dot(v) = np.array([2*1 + 0*1, 0*1 + 1*1])`, 즉 `np.array([2,1])`입니다

The Frobenius norm (프로베니우스 규범)

```
# 1차원 배열을 생성합니다.
X = np.array([1, 2])

# 2-노름(Euclidean norm)을 계산합니다.
# 유클리드 거리로, 배열의 모든 요소의 제곱을 더하고 그 제곱근을 취합니다.
print(np.linalg.norm(X))  # 결과: sqrt(1^2 + 2^2) = sqrt(5)

# 1-노름(Manhattan norm)을 계산합니다.
# 배열의 모든 요소의 절댓값을 합산합니다.
print(np.linalg.norm(X, ord=1))  # 결과: abs(1) + abs(2) = 3

# 무한대 노름(Infinity norm)을 계산합니다.
# 배열의 모든 요소 중 가장 큰 절댓값을 반환합니다. -> 최대값
print(np.linalg.norm(X, ord=np.inf))  # 결과: max(abs(1), abs(2)) = 2

# -무한대 노름을 계산합니다.
# 배열의 모든 요소 중 가장 작은 절댓값을 반환합니다. -> 최소값
print(np.linalg.norm(X, ord=-np.inf))  # 결과: min(abs(1), abs(2)) = 1
```

```
2.23606797749979
3.0
2.0
1.0
```

```
import math
import numpy as np

# 두 개의 벡터 x와 y를 정의합니다.
x = np.array([1, 0])  # 벡터 x는 [1, 0]입니다.
```

```
y = np.array([0, 1])  # 벡터 y는 [0, 1]입니다.

# 첫 번째 코사인 유사도 계산:
# 벡터 x와 y의 내적을 계산하고, 각 벡터의 크기를 계산한 다음, 둘을 나눕니다.
print("cosine =", x.dot(y) / (math.sqrt(x.dot(x)) * math.sqrt(y.dot(y))))

# 두 번째 코사인 유사도 계산:
# 벡터 x와 y의 내적을 계산하고, numpy의 np.linalg.norm 함수를 사용하여 벡터 크기를 계산한 다음, 둘을 나눕니다.
print("cosine =", x.dot(y) / (np.linalg.norm(x) * np.linalg.norm(y)))
```

```
cosine = 0.0
cosine = 0.0
```

LAB: distance matrix

- **In case of 1-d points**

```
import numpy as np

pts = np.array([1., 2, 3, 4, 5])  # 1차원 배열을 생성합니다.

# np.newaxis: 축 하나 더 만듬
u = pts[:, np.newaxis]  # 열 방향으로 1차원 배열을 확장합니다. 결과는 (5, 1) 크기의 2차원 배열입니다.
v = pts.T[np.newaxis, :]  # 행 방향으로 1차원 배열을 확장합니다. 결과는 (1, 5) 크기의 2차원 배열입니다.

# `u`와 `v`의 차이의 절대값을 계산합니다.
result = np.abs(u - v)

print(result)  # `u`와 `v`의 차이의 절대값으로 이루어진 5x5 행렬이 출력됩니다.
```

```
# 대칭 행렬
array([[0., 1., 2., 3., 4.],
       [1., 0., 1., 2., 3.],
       [2., 1., 0., 1., 2.],
       [3., 2., 1., 0., 1.],
       [4., 3., 2., 1., 0.]])
```

💡 `pts.T`는 `pts` 배열의 전치(transpose)를 의미합니다. 전치는 배열의 축(axis)을 바꾸는 연산으로, 배열의 행과 열을 바꾸는 역할을 합니다.

```
# pts = np.array([1., 2, 3, 4, 5])
pts.T

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

- n-d 포인트인 경우

```
import numpy as np

# 2차원 배열 `pts`를 초기화합니다. (3, 2)의 모양으로 2차원 점들을 나타냅니다.
pts = np.array([[1, 0], [1, 1], [0, 1]])
```

```
# `pts`의 모양을 출력합니다.
print(pts.shape)

# 새로운 차원을 추가하여 3차원 배열 `u`를 생성합니다.
# `u`의 모양은 (3, 2, 1)이며, `pts`의 각 행을 새로운 세 번째 차원에 따라 1차원 배열로 확장합니다.
u = pts[:, :, np.newaxis]

# `pts`를 전치하고 새로운 차원을 추가하여 3차원 배열 `v`를 생성합니다.
# `v`의 모양은 (1, 2, 3)이며, `pts`의 각 열을 새로운 첫 번째 차원에 따라 1차원 배열로 확장합니다.
v = pts.T[np.newaxis, :, :]

# Result: (3,2)
```

```
import numpy as np

# 2차원 점들의 배열 `pts`를 초기화합니다. `pts`는 (3, 2) 모양의 배열입니다.
pts = np.array([[0, 0], [1, 1], [0, 0]])

# `pts`의 모양을 출력합니다.
print(pts.shape)

# `pts`의 각 행을 새로운 차원에 따라 확장하여 3차원 배열 `u`를 생성합니다.
# `u`의 모양은 (3, 2, 1)입니다.
u = pts[:, :, np.newaxis]

# `pts`를 전치한 후, 새로운 차원을 추가하여 3차원 배열 `v`를 생성합니다.
# `v`의 모양은 (1, 2, 3)이며, `pts`의 각 열을 새로운 차원에 따라 확장한 것입니다.
v = pts.T[np.newaxis, :, :]

# Result: (3,2)
```

```
# np.linalg.norm(pts)는 주어진 배열 pts의 노름(norm)을 계산하는 함수.
# 노름은 주어진 벡터의 크기를 나타내는 척도

np.linalg.norm(pts)

# 1.4142135623730951
```

💡 설명

- `u = pts[:, :, np.newaxis]`은 입력 배열 `pts`를 새로운 차원으로 확장하여 `(3, 2, 1)` 형상의 배열로 변환합니다.
- `v = pts.T[np.newaxis, :, :]`은 `pts`의 전치 배열을 새로운 차원으로 확장하여 `(1, 2, 3)` 형상의 배열로 변환합니다.

```
print(v.shape)
print(u.shape)

(1, 2, 3)
(3, 2, 1)
```

```
np.sqrt(np.sum((u - v)**2, axis=1))
```

💡 설명

1. **배열 `u` 와 `v`**:
    - `u`와 `v`는 각각 `(3, 2, 1)` 및 `(1, 2, 3)` 형상의 3차원 배열입니다.
    - `u`는 `pts` 배열의 각 점을 `z` 축(마지막 차원)에 배열하여 3차원 배열로 만든 것입니다.
    - `v`는 `pts` 배열의 전치(transpose) 후 새로운 차원을 추가하여 `x` 축(첫 번째 차원)에 배열한 것입니다.
2. `u - v`:
    - `u`와 `v`의 형상이 각각 `(3, 2, 1)` 및 `(1, 2, 3)`이므로, 두 배열은 브로드캐스팅을 통해 형상이 맞춰집니다.

    → (3 ,2, 1) (1, 2, 3) ⇒ (3, 2, 3)
    - `u - v`는 각각의 `u` 요소와 `v` 요소 간의 차이를 계산합니다. 이 연산은 각 점의 좌표 차이를 나타냅니다.
    - 결과는 `(3, 2, 3)` 형상의 배열이 됩니다. 이는 `3`개의 점 (`u`)과 `3`개의 점 (`v`) 사이의 차이 값을 나타냅니다.
3. `np.sum((u - v)**2, axis=1)`:
    - `axis=1`을 지정하면 `u`와 `v` 간의 차이 제곱을 `y` 축(두 번째 차원)에서 합산합니다.
    - `axis=1`의 합산 결과는 `(3, 3)` 형상의 배열로, 각 점 간의 거리를 나타냅니다.
4. `np.sqrt(np.sum((u - v)**2, axis=1))`:
    - `u`와 `v`의 각 점 간 차이의 제곱을 합산한 값을 `sqrt`를 사용하여 제곱근을 계산합니다.
    - 이 연산은 `3`개의 점 간의 유클리드 거리를 계산하여 `(3, 3)` 형상의 배열을 반환합니다.

```
array([[0.        , 1.        , 1.41421356],
       [1.        , 0.        , 1.        ],
       [1.41421356, 1.        , 0.        ]])
```

💡 설명

`np.linalg.norm(u - v, axis=1)`는 `u`와 `v` 사이의 차이 벡터의 유클리드 거리(Euclidean distance)를 계산하는 코드입니다.

- `u`와 `v`는 각각 (3, 2, 1)과 (1, 2, 3)의 형상을 가진 NumPy 배열로, 각각 2차원 포인트를 표현합니다. `u`와 `v`는 서로 다른 형상이지만, 브로드캐스팅을 통해 두 배열을 연산할 수 있습니다.
- `u - v`는 `u` 배열과 `v` 배열의 차이 벡터를 계산합니다. 두 배열은 브로드캐스팅을 통해 (3, 2, 3) 형상으로 확장됩니다. 이는 `u`와 `v` 사이의 차이를 나타내는 배열입니다.
- `np.linalg.norm(u - v, axis=1)`은 차이 벡터의 유클리드 거리를 계산합니다. `axis=1`은 행 단위로 연산을 수행하도록 지시합니다. 즉, 각 포인트 `u`와 `v` 사이의 차이 벡터의 유클리드 거리를 계산합니다.

```
np.linalg.norm(u - v, axis=1)
```

```
np.linalg.norm(u - v, axis=1)
array([[0.        , 1.        , 1.41421356],
       [1.        , 0.        , 1.        ],
       [1.41421356, 1.        , 0.        ]])
```

- 실제 응용 프로그램에서 sklearn.metrics.pairwise를 사용하여 쌍별 거리를 계산합니다

💡 설명

- `euclidean_distances(pts)`: `pts` 배열의 각 요소 간의 유클리드 거리를 계산합니다. 유클리드 거리는 두 점 사이의 직선 거리를 의미합니다. 반환되는 값은 `pts` 배열의 각 쌍에 대한 거리 값으로 구성된 행렬입니다.
- `manhattan_distances(pts)`: `pts` 배열의 각 요소 간의 맨해튼 거리를 계산합니다. 맨해튼 거리는 두 점 사이의 좌표 차이의 절대값을 합한 값으로, 택시 거리라고도 불립니다. 반환되는 값은 `pts` 배열의 각 쌍에 대한 거리 값으로 구성된 행렬입니다.
- `cosine_similarity(pts)`: `pts` 배열의 각 요소 간의 코사인 유사도를 계산합니다. 코사인 유사도는 두 벡터 사이의 각도를 나타내며, 1에 가까울수록 유사한 방향을 가지고 있음을 의미합니다. 반환되는 값은 `pts` 배열의 각 쌍에 대한 유사도 값으로 구성된 행렬입니다.

```
from sklearn.metrics.pairwise import euclidean_distances, manhattan_distances, cosine_similarity

# 2차원 포인트 배열 `pts`의 요소들 간의 유클리드 거리를 계산합니다.
print(euclidean_distances(pts))

# 2차원 포인트 배열 `pts`의 요소들 간의 맨해튼 거리를 계산합니다.
print(manhattan_distances(pts))

# 2차원 포인트 배열 `pts`의 요소들 간의 코사인 유사도를 계산합니다.
print(cosine_similarity(pts))
```

```
[[0.         1.         1.41421356]
 [1.         0.         1.        ]
 [1.41421356 1.         0.        ]]
[[0. 1. 2.]
 [1. 0. 1.]
 [2. 1. 0.]]
[[1.         0.70710678 0.        ]
 [0.70710678 1.         0.70710678]
 [0.         0.70710678 1.        ]]
```

- 자신의 거리를 정의하면

💡 이 코드는 `pairwise_distances` 함수를 사용하여 `pts` 배열의 각 요소 간의 무한대 거리를 계산합니다.

`inf_dist` 함수는 두 벡터 `x`와 `y` 사이의 각 요소 간 절대 차이의 최대값을 반환하는 람다 함수입니다.

`pairwise_distances(pts, metric=inf_dist)`는 주어진 `pts` 배열의 각 쌍에 대한 무한대 거리를 계산하여 행렬로 반환합니다.

```
from sklearn.metrics.pairwise import pairwise_distances

# 사용자 정의 거리 함수 inf_dist는 두 점 x와 y 사이의 무한대 거리를 계산합니다.
# 무한대 거리(inf_dist)는 두 벡터 x와 y의 각 요소 간 절대 차이의 최대값을 의미합니다.
inf_dist = lambda x, y : np.max(np.abs(x - y))

# pairwise_distances 함수를 사용하여 주어진 2차원 포인트 배열 pts의 각 쌍에 대한 무한대 거리를 계산합니다.
# metric 매개변수로 사용자 정의 거리 함수 inf_dist를 사용하여 무한대 거리를 계산합니다.
print(pairwise_distances(pts, metric=inf_dist))
```

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

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