

으뜸 파이썬



11장 넘파이

학습목표

- 파이썬에는 풍부하고도 유용한 외부 라이브러리가 있음을 이해한다.
- 과학 계산에 필수적인 넘파이 라이브러리의 용도와 사용법에 대해 이해한다.
- 넘파이의 핵심적인 요소인 ndarray를 사용해 본다.
- ndarray와 리스트의 차이를 이해하고, 그 특성에 맞는 활용을 할 수 있다.
- 넘파이가 과학기술 분야와 머신 러닝 분야에 적합한 이유를 설명할 수 있다.
- ndarray와 선형대수 문제의 연관을 이해한다.
- 넘파이를 이용하여 다양한 행렬, 벡터 문제를 해결할 수 있다.
- 넘파이의 linalg 패키지를 이용하여 다양한 선형대수 문제를 풀 수 있다.
- 파이썬의 리스트와 넘파이의 ndarray를 연동하여 계산할 수 있는 기초 지식을 습득한다.

11.1 넘파이 라이브러리

- 넘파이NumPy

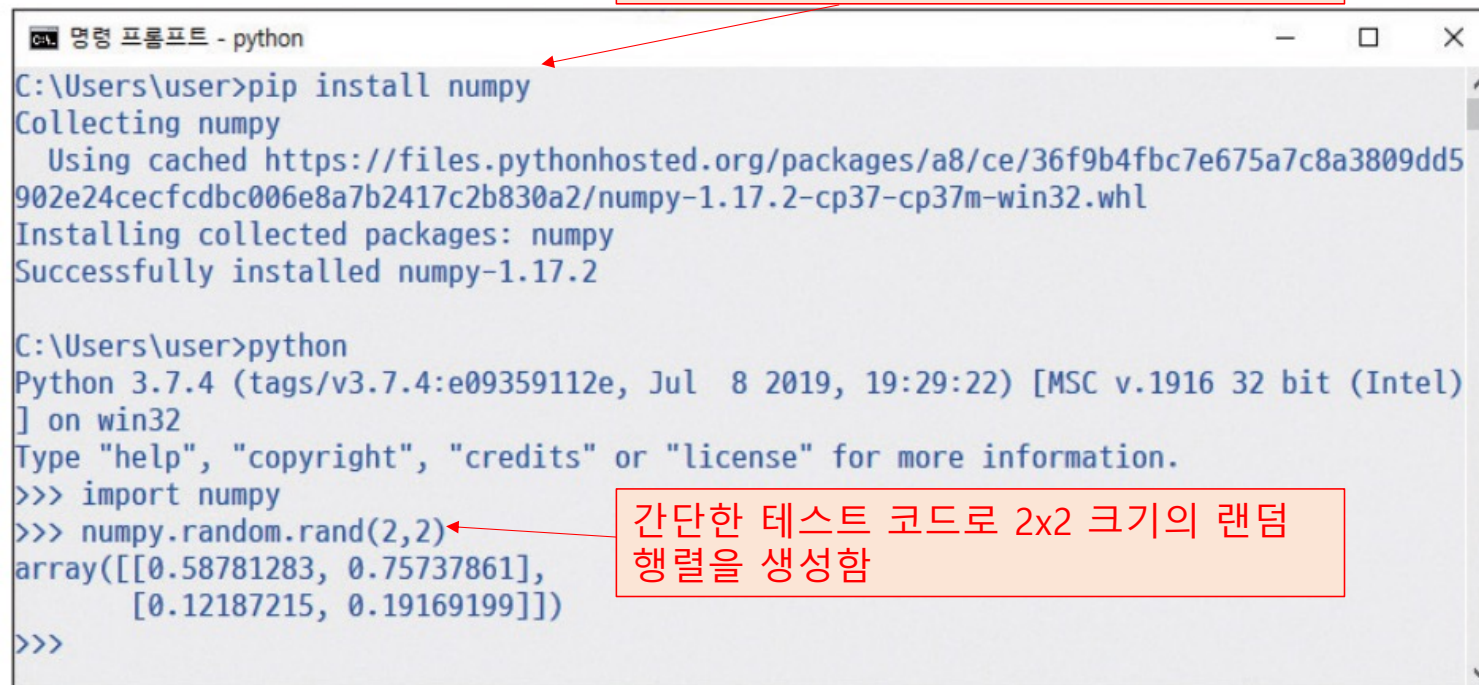
- 파이썬의 과학계산을 위한 가장 기본적인 라이브러리
- 행렬, 벡터 연산을 위한 사실상의 표준 라이브러리로 빠른 처리속도가 장점
- 다차원 배열과 행렬 객체가 포함
- pip를 이용하여 명령행에서 다음과 같이 입력한다

```
$ pip install numpy
```

- 아나콘다, 미니콘다라는 패키지를 설치하면 자동 설치가 됨
- 구글 colab 환경에서는 설치가 필요없음

설치와 테스트

명령 프롬프트에서 설치 명령어
pip는 파이썬 패키지 관리 프로그램임



```
C:\Users\user>pip install numpy
Collecting numpy
  Using cached https://files.pythonhosted.org/packages/a8/ce/36f9b4fbc7e675a7c8a3809dd5902e24cecfcdabc006e8a7b2417c2b830a2/numpy-1.17.2-cp37-cp37m-win32.whl
Installing collected packages: numpy
Successfully installed numpy-1.17.2

C:\Users\user>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.random.rand(2,2)
array([[0.58781283, 0.75737861],
       [0.12187215, 0.19169199]])
>>>
```

간단한 테스트 코드로 2x2 크기의 랜덤 행렬을 생성함

[그림 11-1] pip를 이용한 numpy 설치화면과 간단한 테스트 코드

11.1 넘파이 라이브러리

대화창 실습 : numpy의 ndarray 사용

numpy의 별칭은 np

```
>>> import numpy as np
>>> a = np.array([1, 2, 3]) # 넘파이 ndarray 객체의 생성
>>> a
array([1, 2, 3])
>>> a.shape # a 객체의 형태(shape)
(3,)
>>> a.ndim # a 객체의 차원
1
>>> a.dtype # a 객체 내부 자료형
dtype('int32')
>>> a.itemsize # a 객체 내부 자료형이 차지하는 메모리 크기(byte)
4
>>> a.size # a 객체의 전체 크기(항목의 수)
3
```

ndarray

- 넘파이의 가장 핵심적인 객체, 다차원 배열을 처리하며 다음과 같은 속성들이 있다

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

1	2	3
---	---	---

shape은 생성된 배열객체의 형태를 튜플 타입으로 반환함

```
a.shape : (3,)
```

```
a.ndim : 1
```

```
a.dtype : dtype('int32')
```

```
a.size : 3
```

```
a.itemsize : 4
```

[그림 11-2] 넘파이의 ndarray a와 그 속성들

대화창 실습 : 넘파이와 데이터 형

정수는 8, 16, 32비트 64비트 자료를 사용할 수 있다.

```
>>> a = np.array([1, 2, 3], dtype = 'int32')
```

```
>>> b = np.array([4, 5, 6], dtype = 'int64')
```

```
>>> a.dtype
```

```
dtype('int32')
```

```
>>> b.dtype
```

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

```
>>> c = a + b
```

```
>>> c.dtype
```

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

32비트와 64비트 정수의 덧셈결과는 자동 스케일 업으로 64비트가 됨

주의



주의 : ndarray 배열을 생성할 때 주의할 점

1. 넘파이의 배열 ndarray을 생성할 때, 반드시 대괄호를 사용하여 리스트 형식의 데이터를 만들어서 array() 함수의 인자로 넣어야 한다.

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

만일 리스트 형식으로 하지 않고 쉼표로 구분해서 입력할 경우 다음과 같은 오류가 발생된다.

```
>>> a = np.array(1, 2, 3, 4) # 잘못된 입력
```

```
...
```

```
ValueError: only 2 non-keyword arguments accepted
```


주의

2. 넘파이의 ndarray는 리스트와는 달리, 서로 다른 자료형의 값을 원소로 가질 수 없다.

```
>>> a = np.array([1, 'two', 3, 4], dtype = np.int32)
...
ValueError: invalid literal for int() with base 10: 'two'
```

만일 다음과 같이 자료형을 명시하지 않을 경우 'two'라는 원소의 자료형인 str 형으로 자동 형 변환이 일어난다. 이 경우 모든 원소들은 문자열 형이되어 정수의 덧셈, 뺄셈 등의 연산을 사용할 수 없다.

```
>>> a = np.array([1, 'two', 3, 4])
>>> a
array(['1', 'two', '3', '4'], dtype = '<U21')
```

노트



NOTE : 넘파이 배열의 데이터 타입을 지정하는 두 가지 방법

넘파이 배열의 데이터 타입을 지정하는 방법에는 두 가지가 있다.

1. `dtype = np.int32` 와 같이 `np`의 `int32` 속성 값으로 지정하기

```
>>> a = np.array([1, 2, 3, 4], dtype = np.int32)
```

2. `dtype = 'int32'` 와 같이 문자열 형식으로 속성 값 지정하기

```
>>> a = np.array([1, 2, 3, 4], dtype = 'int32')
```

- numpy의 ndarray의 속성

속성	설명
ndim	배열 축 혹은 차원의 갯수.
shape	배열의 차원으로 (m, n) 형식의 튜플 형이다. 이 때, m과 n은 각 차원의 원소의 크기를 알려주는 정수 값이다.
size	배열의 원소의 갯수이다. 이 갯수는 shape내의 원소의 크기의 곱과 같다. 즉 (m, n) shape 배열의 size는 $m*n$ 이다.
dtype	배열내의 원소의 형을 기술하는 객체이다. numpy는 파이썬 표준 형을 사용할 수 있으나 numpy 자체의 자료형인 bool_, character, int_, int8, int16, int32, int64, float, float8, float16_, float32, float64, complex_, complex64, object_ 형을 사용할 수 있다.
itemsize	배열내의 원소의 크기를 바이트 단위 로 기술한다. 예를 들어 int32 자료형의 크기는 $32/8=4$ 바이트가 된다.
data	배열의 실제 원소를 포함하고 있는 버퍼.



LAB 11-1 : ndarray 객체 생성하기 그리고 속성 알아보기

1. 0에서 9까지의 정수 값을 가지는 ndarray 객체 a를 넘파이를 이용하여 작성하여 다음과 같이 출력하여라.

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

2. range() 함수를 사용하여 0에서 9까지의 정수 값을 가지는 ndarray 객체 b를 만들고 문제 1의 결과와 같이 나타나도록 하여라.

3. 문제 2의 코드를 수정하여 0에서 9까지의 정수 값 중에서 다음과 같이 짝수를 가지는 ndarray 객체 c를 출력하여라.

```
array([0, 2, 4, 6, 8])
```

4. 문제 3번 ndarray 객체 c의 shape, ndim, dtype, size, itemsize를 다음과 같이 출력하여라.

```
c.shape = (5,)
c.ndim = 1
c.dtype = int64
c.size = 5
c.itemsize = 8
```

11.2 ndarray의 메소드와 주요 함수

ndarray

대화창 실습 : ndarray의 메소드

```
>>> a = np.array([1, 2, 3]) # 1차원 ndarray 배열 생성
>>> a.max()      # 가장 큰 값을 반환
3
>>> a.min()      # 가장 작은 값을 반환
1
>>> a.mean()     # 평균 값을 반환
2.0
```

flatten()

대화창 실습 : ndarray의 flatten() 메소드

```
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a.flatten() # ndarray 배열의 평탄화 메소드
array([1, 1, 2, 2, 3, 3])
```

append() 함수

대화창 실습 : ndarray의 append() 함수

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([[4, 5, 6], [7, 8, 9]])
>>> np.append(a, b)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.append([a], b, axis = 0) # [a]를 통해 2차원 배열로 만들어야 함
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

rand() 함수

대화창 실습 : ndarray의 rand() 함수

```
>>> np.random.rand(3, 3) # (3, 3) shape의 난수 생성
array([[0.63208882, 0.72259476, 0.15125742],
       [0.60731818, 0.20682056, 0.51958311],
       [0.644331 , 0.91940484, 0.83990604]])
```

randint() 함수

Ndarray의 randint() 함수

```
>>> np.random.randint(0, 10, size = 10) # 0에서 10까지의 10개의 난수 생성  
array([2, 0, 8, 2, 7, 0, 1, 3, 1, 3])
```



LAB 11-2 : ndarray 객체의 메소드와 함수

1. [23, 45, 67, 7, 2, 30, 34, 82]의 정수 값을 가지는 ndarray 객체 a를 생성하여 다음과 같이 출력하여라.

```
a = array([23 45 67 7 2 30 34 82])
```

이 ndarray의 max(), min(), mean() 메소드를 이용하여 최댓값, 최솟값, 평균을 다음과 같이 출력하여라.

최댓값 : 82

최솟값 : 2

평균 : 36.25

2. numpy.random.randint() 함수를 사용하여 0에서 99까지의 정수 값 10개를 랜덤하게 생성하시오. 이 10개의 값을 b라는 ndarray에 넣고 ndarray에서 최댓값, 최솟값, 평균을 다음과 같이 출력하여라(b 값은 랜덤하게 생성되므로 다음 화면의 값과 일치하지 않음).

```
b = [25 2 9 86 93 73 15 53 67 20]
```

최댓값 : 93

최솟값 : 2

평균 : 44.3

3. 위의 문제 1의 a 배열과 문제 2의 b 배열을 append() 하여 다음과 같은 배열 c를 생성하여 출력하시오.

```
c = [23 45 67 7 2 30 34 82, 25 2 9 86 93 73 15 53 67 20]
```


11.3 ndarray의 연산

대화창 실습 : ndarray의 덧셈(shape이 같을 경우)

```
>>> a = np.array([1, 2, 3]) # 1, 2, 3 원소를 가지는 1차원 ndarray
>>> b = np.array([4, 5, 6]) # 4, 5, 6 원소를 가지는 1차원 ndarray
>>> c = a + b                # 1차원 ndarray의 덧셈
>>> c
array([5, 7, 9])
```

1	2	3
+		
4	5	6
=		
5	7	9

대화창 실습 : ndarray의 덧셈(shape이 다를 경우)

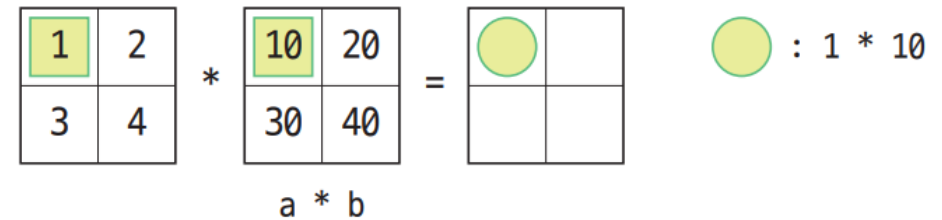
```
>>> a = np.array([1, 2])    # 2개의 원소를 가지는 1차원 배열 : (2,) shape
>>> b = np.array([4, 5, 6]) # 3개의 원소를 가지는 1차원 배열 : (3,) shape
>>> c = a + b                # shape이 다른 1차원 배열의 합
...
ValueError: operands could not be broadcast together with shapes (2,) (3,)
```

ndarray의 사칙연산

대화창 실습 : 2차원 ndarray의 사칙연산

```
>>> a = np.array([[1, 2], [3, 4]])      # 2차원 배열 a
>>> b = np.array([[10, 20], [30, 40]]) # 2차원 배열 b
>>> a + b
array([[11, 22],
       [33, 44]])
>>> a - b
array([[ -9, -18],
       [-27, -36]])
>>> a * b
array([[ 10, 40],
       [ 90, 160]])
>>> a / b
array([[0.1, 0.1],
       [0.1, 0.1]])
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 1*10 & 2*20 \\ 3*30 & 4*40 \end{bmatrix}$$



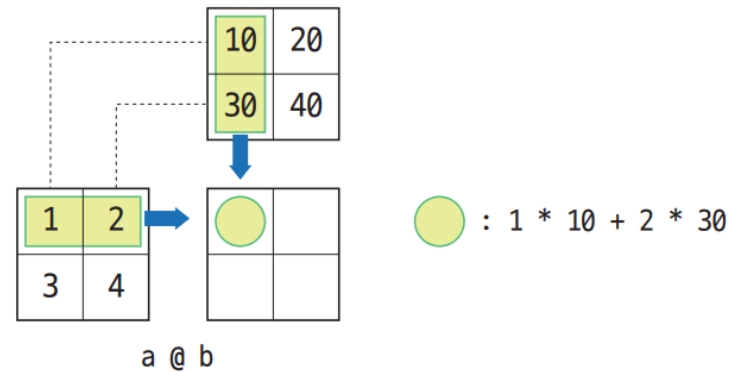
[그림 11-4] 넘파이의 $a * b$ 연산

행렬 곱 matmul()

대화창 실습 : 행렬 곱 함수 matmul() 실습

```
>>> np.matmul(a, b)
array([[ 70, 100],
       [150, 220]])
>>> a @ b
array([[ 70, 100],
       [150, 220]])
>>> a[0,0] * b[0,0] + a[0,1] * b[1,0]
70
>>> a[0,0] * b[0,1] + a[0,1] * b[1,1]
100
>>> a[1,0] * b[0,0] + a[1,1] * b[1,0]
150
>>> a[1,0] * b[0,1] + a[1,1] * b[1,1]
220
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} @ \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 1*10+2*30 & 1*20+2*40 \\ 3*10+4*30 & 3*20+4*40 \end{bmatrix}$$



[그림 11-5] 넘파이의 $a @ b$ 연산

대화창 실습 : 단위행렬에 대한 행렬 곱

```
>>> a = [[1, 2], [3, 4]]
>>> b = [[1, 0], [0, 1]] # b는 단위행렬
>>> np.matmul(a, b)      # a 행렬과 단위행렬 b의 곱의 결과
array([[ 1,  2],
       [ 3,  4]])
```

$$\begin{aligned} AE &= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} a \times 1 + b \times 0 & a \times 0 + b \times 1 \\ c \times 1 + d \times 0 & c \times 0 + d \times 1 \end{pmatrix} \\ &= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{aligned}$$

대화창 실습 : 2차원 ndarray의 덧셈, 뺄셈, 곱셈, 나눗셈, 제곱 연산

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a + 1          # 행렬의 각 성분에 대한 덧셈
array([[2, 3],
       [4, 5]])
>>> a - 1          # 행렬의 각 성분에 대한 뺄셈
array([[0, 1],
       [2, 3]])
>>> a * 100        # 행렬의 각 성분에 대한 곱셈
array([[100, 200],
       [300, 400]])
>>> a / 100        # 행렬의 각 성분에 대한 나눗셈
array([[0.01, 0.02],
       [0.03, 0.04]])
>>> a ** 2         # 행렬의 각 성분에 대한 제곱연산
array([[ 1,  4],
       [ 9, 16]])
```

1	2
3	4

+ 1 =

1	2
3	4

* 100 =

1	2
3	4

** 2 =



LAB 11-3 : 다차원 행렬의 생성과 활용

1. 1에서 9까지의 모든 정수 값을 크기 순서대로 가지는 3x3 크기의 행렬 a를 생성하고, 모든 성분의 값이 2인 3x3 크기의 행렬 b를 생성하여라.

```
a = [[1 2 3]
      [4 5 6]
      [7 8 9]]
```

```
b = [[2 2 2]
      [2 2 2]
      [2 2 2]]
```

2. 다음 행렬 연산의 결과를 예상한 후 실행하고 그 결과를 적으시오.

- 1) $a + b$
- 2) $a - b$
- 3) $a * b$
- 4) a / b
- 5) $a @ b$
- 6) $a ** 2$

11.4 ndarray의 생성

- 넘파이는 배열을 쉽게 생성하는 함수도 지원하고 있다.
- 첫 번째가 `zeros((n,m))` 함수이다. 이는 $n \times m$ 배열(혹은 행렬)을 생성해서 초기값은 0으로 해준다.
- `eye(n)` 함수를 이용하면 $n \times n$ 크기의 단위행렬(identity matrix)을 생성할 수 있다.
 - 정방행렬 중에서 대각선 성분이 1이고 나머지 성분이 0인 행렬

대화창 실습 : 초기값을 가지는 행렬의 생성

```
>>> np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.ones((2, 3))
array([[1., 1., 1.],
       [1., 1., 1.]])
>>> np.full((2, 3), 100)
array([[100, 100, 100],
       [100, 100, 100]])
>>>... 이어서...
```

모든 값이 0인 2x3 크기 행렬

모든 값이 1인 2x3 크기 행렬

모든 값이 100인 2x3 행렬

대화창 실습 : 초기값을 가지는 행렬의 생성

```
>>> np.zeros((2, 3))
```

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

```
>>> np.ones((2, 3))
```

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

```
>>> np.full((2, 3), 100)
```

```
array([[100, 100, 100],  
       [100, 100, 100]])
```

```
>>> np.eye(3)
```

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

3x3 크기의 단위행렬

2x3 크기의 랜덤행렬

```
>>> np.random.random((2, 3))
```

```
array([[0.87143684, 0.44538612, 0.11908973],  
       [0.87548557, 0.57502347, 0.87641631]])
```


arange(0, 10)

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

[그림 11-6] arange(0,10) 실행 결과

arange(0, 10, 2)

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

step=2

[그림 11-7] arange(0, 10, 2) 실행과 스텝 값

➡

0	2	4	6	8
---	---	---	---	---

[그림 11-8] arange(0, 10, 2) 실행 결과

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 1.0, 0.3))
```

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

list(range(0, 1.0, 0.3))

TypeError: 'float' object cannot be interpreted as an integer

파이썬 리스트는 실수 step 값을 사용할 수 없음

대화창 실습 : arange() 함수를 이용한 배열 생성

```
>>> np.arange(0, 10)
```

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

```
>>> np.arange(0, 10, 2)
```

```
array([0, 2, 4, 6, 8])
```

```
>>> np.arange(0, 10, 3)
```

```
array([0, 3, 6, 9])
```

```
>>> np.arange(0.0, 1.0, 0.2)
```

```
array([0. , 0.2, 0.4, 0.6, 0.8])
```

실수 step 값을 사용할 수 있음

동일한 간격을 가진 연속된 값을 생성

대화창 실습 : linspace() 함수를 이용한 배열 생성

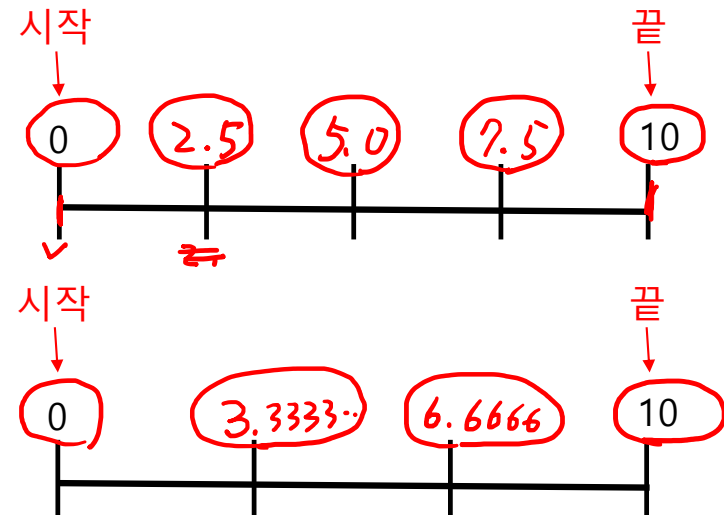
```
>>> np.linspace(0, 10, 5)
```

```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

```
>>> np.linspace(0, 10, 4)
```

```
array([ 0.          ,  3.33333333,  6.66666667, 10.          ])
```

(시작, 끝, 간격의 수)를 인자로 가짐
디폴트 간격의 수는 50개





LAB 11-4 : 행렬의 생성

1. full() 함수를 이용하여 모든 원소의 값이 2인 3x3 크기의 행렬 a1을 생성하여라.

```
a1 = [[2 2 2]
       [2 2 2]
       [2 2 2]]
```

2. arange() 함수를 사용하여 1에서 12까지 12개의 원소를 가지는 1차원 행렬 a2를 생성하여라.

```
a2 = [ 1 2 3 4 5 6 7 8 9 10 11 12]
```

3. arange() 함수의 step 값을 이용하여 1에서 50까지 정수 중에서 3의 배수만을 가지는 행렬 a3을 생성하여라.

```
a3 = [ 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48]
```

4. 0에서 20 사이에서 동일한 간격의 값을 가지는 원소 5개를 가진 a4 행렬을 생성하여라.

```
a4 = [ 0. 5. 10. 15. 20.]
```

11.5 ndarray의 재구성 **reshape**

`np.arange(0, 10).reshape(2, 5)`

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

(10,) : 1차원 배열(10개의 원소)

➡

0	1	2	3	4
5	6	7	8	9

[그림 11-9] `reshape()` 메소드를 이용하여 1차원 배열을 2행 5열의 다차원 행렬로 변환시킨 결과

대화창 실습 : `reshape()`을 이용한 배열의 재구성

```
>>> np.arange(0, 10).reshape(2, 5)
```

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

```
>>> np.arange(0, 10).reshape(5, 2)
```

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

```
>>> np.arange(0, 10).reshape(3, 3)
```

```
...
```

```
ValueError: cannot reshape array of size 10 into shape  
(3,3)
```

대화창 실습 : 다른 차원으로의 reshape() 실습

```
>>> np.arange(0, 24).reshape(4, 3, 2)
```

```
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]])
```

대화창 실습 : 다른 차원으로의 reshape() 실습

```
>>> a = np.arange(6).reshape(3, 2)
```

```
>>> a
```

```
array([[0, 1],  
       [2, 3],  
       [4, 5]])
```

```
>>> np.transpose(a)
```

```
array([[0, 2, 4],  
       [1, 3, 5]])
```



LAB 11-5 : 배열의 재구성

1. `arange()` 함수를 사용하여 1에서 12까지의 원소를 가지는 1차원 배열 `a1`을 생성하여라. 그리고 이 `a1` 배열을 `reshape()` 메소드를 사용하여 2행 6열의 행렬로 재구성하여라.

```
a1 = [[ 1 2 63 4 5 6]
      [ 7 8 9 10 11 12]]
```

2. `arange()` 함수를 사용하여 1에서 30까지의 원소를 가지는 1차원 배열 `a2`을 생성하여라. 그리고 이 `a2` 배열을 `reshape()` 메소드를 사용하여 3행 10열의 행렬로 재구성하여라.

```
a2 = [[ 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]]
```

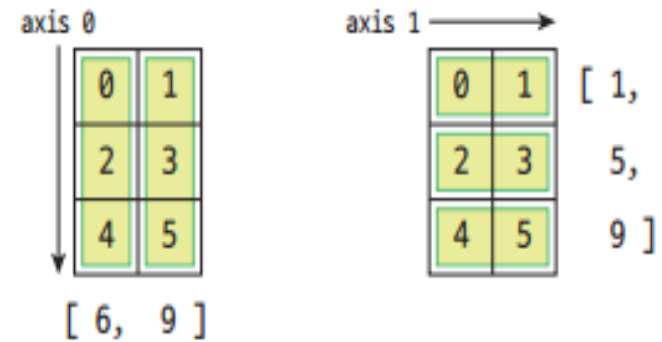
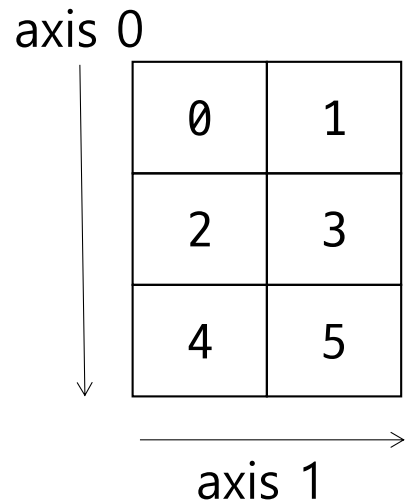
3. 문제 2에서 생성한 `a2` 배열을 `reshape()` 메소드를 사용하여 6행 5열의 행렬로 재구성한 행렬 `a3`을 생성하여라.

```
a3 = [[ 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]]
```

4. 문제 3에서 생성한 `a3` 배열을 `transpose()` 함수를 사용하여 다음과 같은 전치 행렬을 생성하여라.

```
a4 = [[ 1 6 11 16 21 26]
      [ 2 7 12 17 22 27]
      [ 3 8 13 18 23 28]
      [ 4 9 14 19 24 29]
      [ 5 10 15 20 25 30]]
```

11.6 다차원 배열의 축



[그림 11-11] axis 0과 axis 1에 대하여 각각 sum() 함수를 수행한 결과

[그림 14-6] 2차원 배열과 축 1과 축 2의 방향

대화창 실습 : sum() 함수와 axis에 따른 원소의 합

```
>>> a = np.arange(0, 6).reshape(3, 2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> a.sum()           # 행렬의 모든 원소의 합
15
>>> a.sum(axis = 0)   # 0축 방향(행 방향) 원소의 합
array([6, 9])
```

대화창 실습 : 1축 방향(열 방향) 원소의 합

```
>>> a.sum(axis = 1) # 1축 방향(열 방향) 원소의 합
array([1, 5, 9])
```


대화창 실습 : 0축과 1축 방향 원소의 최솟값, 최댓값

```
>>> a.min(axis = 0)  # 0축 방향 원소의 최솟값  
array([0, 1])
```

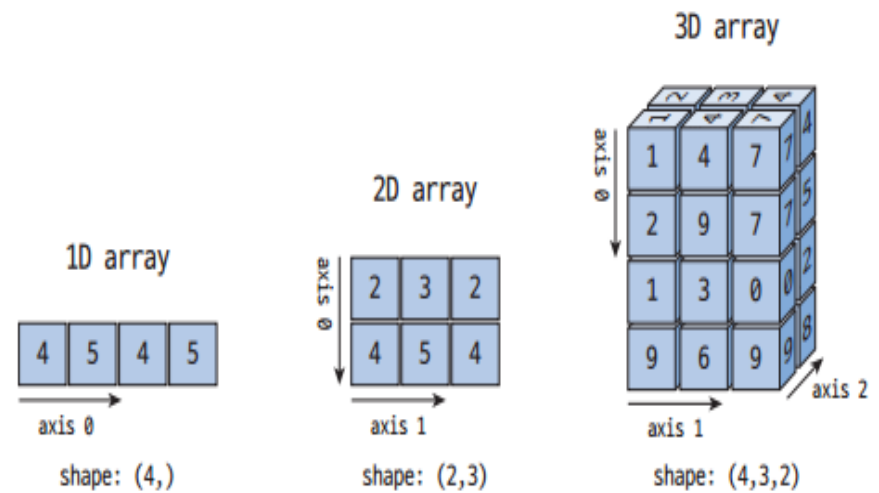
```
>>> a.min(axis = 1)  # 1축 방향 원소의 최솟값  
array([0, 2, 4])
```

```
>>> a.max(axis = 0)  # 0축 방향 원소의 최댓값  
array([4, 5])
```

```
>>> a.max(axis = 1)  # 1축 방향 원소의 최댓값  
array([1, 3, 5])
```

대화창 실습 : ndarray의 insert() 함수

```
>>> a = np.array([1, 3, 4])
>>> np.insert(a, 1, 2)
array([1, 2, 3, 4])
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> np.insert(a, 1, 4, axis = 0)
array([[1, 1],
       [4, 4],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 4, axis = 1)
array([[1, 4, 1],
       [2, 4, 2],
       [3, 4, 3]])
```



[그림 11-12] 1차원, 2차원, 3차원 배열의 형태와 각 배열의 축 방향



LAB 11-6 : 다차원 배열의 축과 삽입 함수

1. 1에서 100까지의 랜덤 정수 15개를 np.random.randint()를 통해서 생성하여라. 이렇게 생성한 정수 값들을 (3, 5)의 shape을 가지는 행렬 a에 다음과 같이 저장하여 출력하여라.

```
a = [[82 30 87 64 21]
      [60 10 26 29 87]
      [29 99 37 21 96]]
```

2. 문제 1에서 생성한 a 행렬에 대해 열방향의 최댓값, 최솟값, 평균을 각각 다음과 같이 출력하시오.

```
a의 열방향 최댓값 : [82 99 87 64 96]
a의 열방향 최솟값 : [29 10 26 21 21]
a의 열방향 평균 : [57. 46.33333333 50. 38. 68. ]
```

3. 문제 1에서 생성한 a 행렬에 대해 행방향의 최댓값, 최솟값, 평균을 각각 다음과 같이 출력하시오.

```
a의 행방향 최댓값 : [87 87 99]
a의 행방향 최솟값 : [21 10 21]
a의 행방향 평균 : [56.8 42.4 56.4]
```

11.7 배열의 인덱싱과 슬라이싱

대화창 실습 : 1차원 배열의 인덱싱하기

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

```
>>> print(a[0], a[1], a[2])
```

```
1 2 3
```

```
>>> print(a[-1], a[-2], a[-3]) # 음수 인덱싱
```

```
3 2 1
```

대화창 실습 : 1차원 배열에서 여러 개의 원소를 인덱싱하기

```
>>> a = np.array([1, 2, 3, 4, 5])
>>> print(a[np.array([0, 1])]) # 인덱싱을 위하여 ndarray를 사용
[1 2]
>>> print(a[np.array([0, 1, 2])])
[1 2 3]
>>> print(a[np.array([0, 1, 3])])
[1 2 4]
>>> print(a[np.array([1, 1, 1, 1])])
[2 2 2 2]
>>> a[3, 4] # 3은 axis 0, 4는 axis 1로 해석-오류, Try this!
```

대화창 실습 : 리스트와 슬라이싱(비교)

```
>>> a_list = [10, 20, 30, 40, 50, 60, 70, 80]
>>> a_list[1:5] # 리스트 슬라이싱 방식
[20, 30, 40, 50]
```

대화창 실습 : 넘파이의 슬라이싱(리스트 슬라이싱과 비슷)

```
>>> a = np.array([10, 20, 30, 40, 50, 60, 70, 80])
>>> a[1:5]          # 슬라이싱 구간 [시작:끝] 인덱스
array([20, 30, 40, 50])
>>> a[1:]
array([20, 30, 40, 50, 60, 70, 80])
>>> a[:]            # 전체를 슬라이싱
array([10, 20, 30, 40, 50, 60, 70, 80])
>>> a[::2]          # 양수 2의 스텝값
array([10, 30, 50, 70])
>>> a[::-1]         # 음수 스텝값
array([80, 70, 60, 50, 40, 30, 20, 10])
```



LAB 11-7 : 배열의 인덱싱과 슬라이싱

1. 1에서 10까지의 원소를 가지는 1차원 배열 `a`를 생성하여라. 이 `a`를 인덱싱하여 `[2, 4, 6, 8]`의 원소를 가진 배열 `b`를 생성하여 다음과 같이 출력하여라.

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

```
b = [2 4 6 8]
```

2. 문제 1에서 생성한 배열 `a`를 슬라이싱하여 다음과 같은 배열 `b, c, d, e, f`를 생성하여라.

```
b = [6 7 8 9 10]
```

```
c = [7 8 9 10]
```

```
d = [1 2 3]
```

```
e = [1 3 5 7 9]
```

```
f = [10 8 6 4 2]
```

11.8 2차원 배열의 인덱싱

행 방향 인덱스 i axis = 0

0	1
2	3
4	5

열 방향 인덱스 j axis = 1

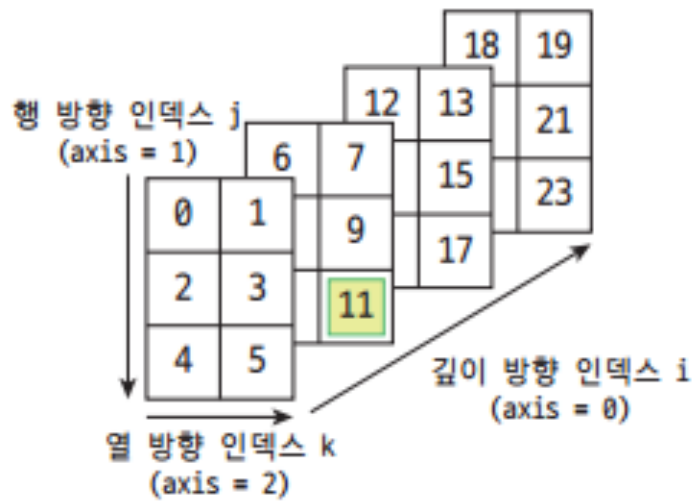
a[0, 0]	a[0, 1]
a[1, 0]	a[1, 1]
a[2, 0]	a[2, 1]

a[i, j] 인덱스의 위치

[그림 11-13] 2차원 배열의 인덱싱

대화창 실습 : 2차원 배열의 인덱싱

```
>>> a = np.arange(0, 6).reshape(3, 2) # 30 | axis 0 방향
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> print(a[0, 0]) # [행(row), 열(column)] 형식
0
>>> print(a[0, 1])
1
>>> print(a[0, 2]) # 범위를 벗어남
...
IndexError: index 2 is out of bounds for axis 1 with size 2
```

[그림 11-14] 3차원 배열의 인덱스

대화창 실습 : 3차원 배열의 인덱스

```
>>> a = np.arange(0, 24).reshape(4, 3, 2)
```

```
>>> a
```

```
array([[[ 0, 1],
        [ 2, 3],
        [ 4, 5]],

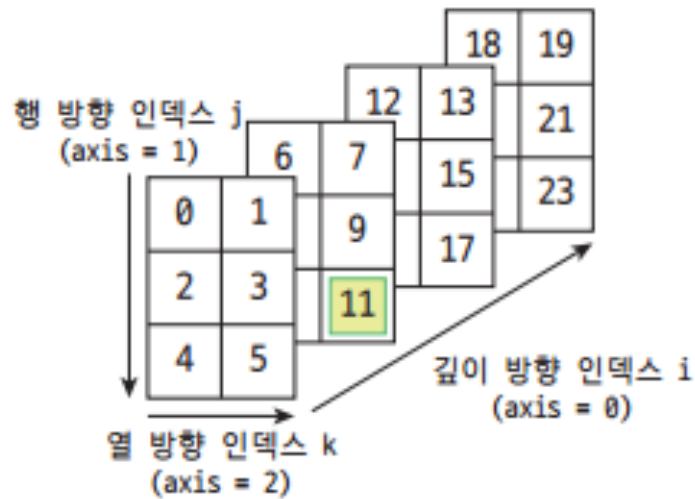
       [[ 6, 7],
        [ 8, 9],
        [10, 11]],

       [[12, 13],
        [14, 15],
        [16, 17]],

       [[18, 19],
        [20, 21],
        [22, 23]])
```

```
>>> print(a[1, 2, 1])
```

```
11
```



[그림 11-14] 3차원 배열의 인덱스

대화창 실습 : 3차원 배열의 인덱스와 concatenate() 함수

```
>>> a = np.arange(0, 24).reshape(4, 3, 2)
>>> a[0]
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5]])
>>> a[0, 0]
array([ 0,  1])
>>> a[0, 1]
array([ 2,  3])
>>> a[0, 2]
array([ 4,  5])
>>> np.concatenate((a[0, 0], a[0, 2]), axis = 0)
array([0, 1, 4, 5])
```



LAB 11-8 : 3차원 배열의 인덱싱

1. [그림 11-14]의 (4, 3, 2) 형태의 3차원 배열을 다음과 같이 인덱스와 원소 값이 나타나도록 출력하시오(힌트 : for 문과 `nindex(a. shape)` 함수를 사용하도록 한다).

index	element
(0, 0, 0)	0
(0, 0, 1)	1
(0, 1, 0)	2
(0, 1, 1)	3
...(생략)	
(3, 2, 0)	22
(3, 2, 1)	23

2. [그림 11-14]의 (4, 3, 2) 형태의 3차원 배열을 `concatenate()` 함수를 사용하여 다음과 같은 배열로 만드시오.

```
>>> 1) _____
array([0, 1, 6, 7])
>>> 2) _____
array([ 0, 1, 14, 15])
>>> 3) _____
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])
>>> 4) _____
array([[ 0, 1, 6, 7],
       [ 2, 3, 8, 9],
       [ 4, 5, 10, 11]])
```

11.9 2차원 배열의 슬라이싱

```
>>> a = np.arange(0, 9).reshape(3, 3)
>>> print(a[0])      # print를 이용한 출력시 array()는 나타나지 않음
[0 1 2]
>>> print(a[0, :])
[0 1 2]
>>> print(a[:, 0])
[0 3 6]
```

첫 번째 행의 두 성분 읽어오기

```
>>> print(a[0, 0:2])
```

```
[0 1]
```

```
>>> print(a[0, :2])
```

```
[0 1]
```

2 x 2 배열 얻기

```
>>> print(a[0:2, 0:2])
```

```
[[0 1]
```

```
 [3 4]]
```

```
>>> print(a[:2, :2])
```

```
[[0 1]
```

```
 [3 4]]
```

```
>>> print(a[1:, 1:])
```

```
[[4 5]
```

```
[7 8]]
```

```
>>> print(a[1, 1:])
```

```
[4 5]
```

```
>>> a[1, 1:].shape
```

```
(2,)
```

```
>>> a[1:2, 1:].shape
```

```
(1, 2)
```

0	1	2
3	4	5
6	7	8
a[0] a[0, :]		

0	1	2
3	4	5
6	7	8
a[0, 0:2]		

0	1	2
3	4	5
6	7	8
a[:2, :2]		

0	1	2
3	4	5
6	7	8
a[1:, 1:]		

0	1	2
3	4	5
6	7	8
a[1, 1:]		

[그림 11-15] 2차원 배열의 슬라이싱과 구간 값



LAB 11-9 : 2차원 배열의 슬라이싱

1. 아래 그림과 같이 0에서 15까지의 연속적인 값을 원소로 가지는 4x4 크기의 2차원 배열 a를 생성하여라. 이 a에 대하여 인덱싱과 슬라이싱을 적용하여 다음과 같은 b, c, d, e 배열을 구하여라.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

b

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

c

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

d

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

e

b = [1 5 9 13]

c = [9 10 11]

d = [[0 1] [4 5]]

e = [[5 6] [9 10]]

2. 문제 1에서 생성한 배열 a를 슬라이싱하고 평탄화 연산을 하여 다음과 같은 배열 f, g, h를 생성하여라.

f = [0 1 2 4 5 6]

g = [8 9 10 11 12 13 14 15]

h = [5 6 7 13 14 15]

11.10 선형 방정식 풀이, 행렬식

$$2x + 3y = 1$$

$$x - 2y = 4$$

[수식 11-1] 선형 연립방정식

코드 11-1 : 선형 연립 방정식 풀이

numpy_linear_ex.py

```
import numpy as np

a = np.array([[2, 3], [1, -2]])
b = np.array([1, 4])
x = np.linalg.solve(a, b)
print(x)
```

실행결과

[2. -1.]

• 2×2행렬의 행렬식 $\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$

• 3×3행렬의 행렬식 $\det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = aei + bgf + cdh - ceg - bdi - afh$

[수식 11-2] 2×2 행렬의 행렬식과 3×3 행렬의 행렬식

대화창 실습 : 행렬식과 det() 함수

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.linalg.det(a)
-2.0000000000000004
>>> b = np.array([[1, 2], [3, -6]])
>>> np.linalg.det(b)
-12.0
>>> b = np.array([[1, 2], [1, 2]])
>>> np.linalg.det(b)
0.0
```



LAB 11-10 : 연립방정식 풀이

1. 다음과 같은 연립방정식의 해를 구하시오.

$$\begin{aligned}x + y - z &= 0 \\ 2x - y + 3z &= 9 \\ x + 2y + z &= 8\end{aligned}$$

이 연립방정식의 해 x , y , z 를 다음과 같이 출력하시오.

$$x = 1.0, y = 2.0, z = 3.0$$

2. 1번 문제의 연립방정식은 다음과 같은 행렬 A 를 가진다.

$$A = \text{np.array}([[1, 1, -1], [2, -1, 3], [1, 2, 1]], \text{dtype} = 'int32')$$

이 행렬의 행렬식을 `linalg.det()` 함수를 사용하여 다음과 같이 구하여라.

$$\det(A) = -11.0$$

3. 3×3 행렬 A 의 행렬식은 다음과 같이 정의할 수 있다. 문제 2번에 있는 A 행렬의 행렬식 -11.0 이 아래 수식의 가장 오른쪽 항의 행렬식의 연산과 그 결과가 같음을 보이시오. 이를 위하여 3개의 2×2 행렬의 행렬식과 각각의 행렬식에 a , $-b$, c 를 곱한 값을 더하시오.

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{vmatrix} - b \begin{vmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{vmatrix} + c \begin{vmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

- 2 x 2 행렬의 행렬식

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

- 3 x 3 행렬의 행렬식

$$\det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = aei + bgf + cdh - ceg - bdi - afh$$

대화창 실습 : 행렬식

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.linalg.det(a)
-2.0000000000000004

>>> b = np.array([[1, 2], [3, -6]])
>>> np.linalg.det(b)
-12.0

>>> b = np.array([[1, 2], [1, 2]])
>>> np.linalg.det(b)
0.0
```



Questions?