파이썬이란

- 네덜란드의 수학 및 컴퓨터 과학 국립연구소에서 Guido van rossum (귀도 반 로섬) 이 만든 인터프리터 언어
- 인간다운 언어: 사람이 생각하는 방식을 그대로 표현할 수 있다. 컴퓨터 사고 체계에 맞추어서 프로그래밍을 하려고 힘쓸 필요가 없다.

- 강력한 언어: 오픈소스이며 만들고자 하는 프로그램 대부분을 만들 수 있다. 시스템 프로그래밍이나 하드웨어 제어와 같은 매우 복잡하여 연산이 많은 프로그램은 파이썬과 어울리지 않지만 프로그램의 전반적인 뼈대를 파이썬으로 만들고 빠른 실행 속도를 필요로 하는 부분은 c 등 다른 언어로 만들어서 파이썬 프로그램에 포함시킬 수 있다.

- 간결한 언어: 줄을 맞추지 않으면 실행이 안되서 가독성에 큰 도움을 준다. 다른 사람들의 소스 코드가 한 눈에 들어오기 때문에 공동 작업에 좋은 영향을 준다.

- (개발) 속도가 빠른 언어: 제공되는 라이브러리가 많아 구현할 필요없이 바로 가져다 쓸 수 있다.

실습환경 구성

아나콘다 설치

Anaconda 는 패키지 관리 및 배포를 단순화하는 것을 목표로 하는 Python 과 R 언어의 배포판이다.
데이터 사이언스와 대용량 데이터 처리, 예측 분석 등의 머신 러닝에 적합한 오픈소스이다.

Anaconda 를 설치하면 데이터 분석 및 처리하는 패키지가 기본 내장되어 있어
사용자가 개별적으로 설치할 필요가 없어진다.

질문) 가상환경은 꼭 설정해줘야 하나요?
그렇지는 않습니다. 가상환경은 내가 설치한 패키지 등을 다 포함하기 때문에 다음번에 그 가상환경만 불러오면 패키지를 새로 설치할 필요가 없어서 어디서든 편하게 작업할 수 있습니다.

질문) 윈도우 10 은 왜 우분투를 설치하고 아나콘다를 설치하나요?
대부분의 분석 작업은 리눅스를 통해 이루어지기 때문에 협업을 하기 위해서는 같은 환경을 구성하는 것이 좋습니다. 윈도우 10 이 아니라 우분투를 설치할 수 없는 경우에는 버추얼 박스라는 가상머신을 활용하여 우분투를 설치하면 됩니다.

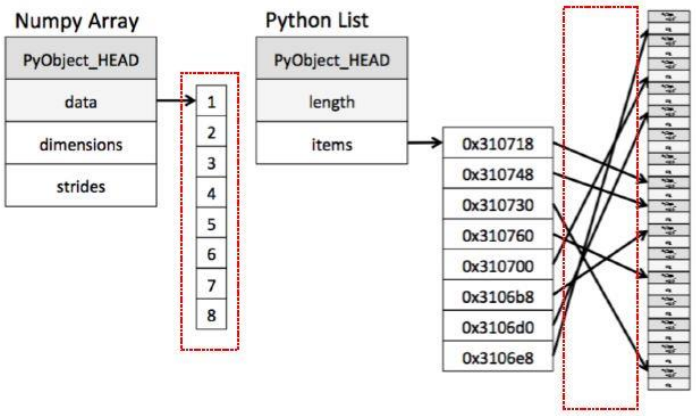
주피터 노트북: 기존의 Ipython Notebook 에서 시작해서 분석자가 노트북처럼 이미지나 코드를 저장해둘 수 있게 해서 다른 사람들이 분석의 흐름을 읽기 쉽게 만드는 툴이다.

Numpy 라이브러리

- **Numpy 란** Numerical Python 의 약자로
  대규모 다차원 배열과 행렬 연산에 필요한 다양한 함수를 제공한다.
  데이터 분석할 때 사용되는 다른 라이브러리 pandas 와 matplotlib 의 기반이 된다.
  기본적으로 array 라는 단위로 데이터를 관리하는데, 행렬 개념으로 생각하면 된다.

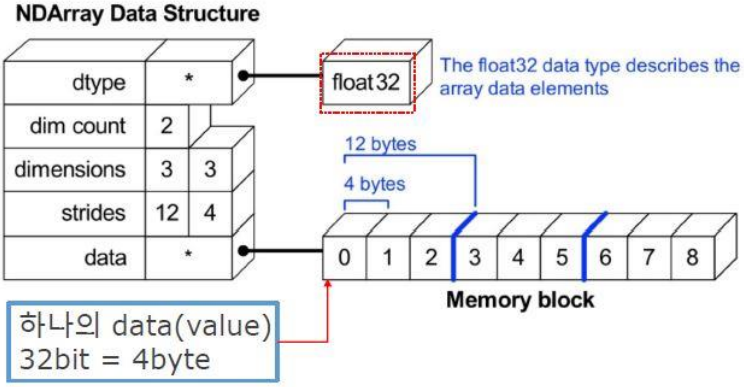
- **Numpy 특징:** 일반 list 에 비해 빠르고 메모리에 효율적이다.
  선형대수와 관련된 다양한 기능을 제공하고,
  for 문, while 문 같은 반복문 없이 데이터 배열에 대한 처리를 지원한다.

- **Numpy 가 빠른 이유:** numpy 는 메모리에 차례대로 생성/할당을 해준다.
  반면 기존의 List 는 이 값(value)가 어디에 있는지 주소만 저장을 해놓고 그 주소를 알려준다.
  그래서 List 를 for 문을 돌리면 그 주소마다 하나하나씩 다 찾아가면서 연산을 해줘야 하는데,
  numpy 는 같은 곳에 몰려있기 때문에 연산이 더 빠르게 이루어진다.



- **Numpy 호출:** "import numpy as np"로 numpy 를 호출하는데
  모두 np 라는 별칭 (alias)로 호출하지만 특별한 이유는 없다.

- **Numpy 로 array 생성하는 방법**: ex) test_array = np.array([1,3,5,7],float)
type(test_array[3])을 하면 4 바이트씩 numpy.float64 라는 값이 반환된다.
float32 같은 데이터 타입은 하나씩 모여서 메모리 블럭을 구성한다.
32bit(비트) = 4byte(바이트)이다. (8bit 가 1byte)



- **shape** 는 array 의 크기를 나타내 준다.
  다음의 예시에서 데이터는 string 까지고 float 타입으로 만들었기 때문에
  전체가 다 float 타입으로 생성되었다. (array([1., 3., 5., 7.,])을 보면 다 .이 붙어있다)

```
test_array = np.array([1, 3, 5, "7"], float)
test_array
```

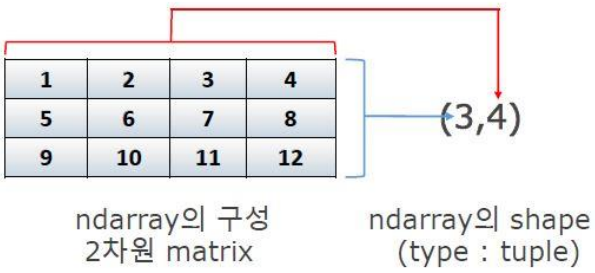
```
array([ 1., 3., 5., 7. ])
```

이건 1 차원의 벡터 형식이라고 부른다.

vector 는 일차원의 행렬을 말하고 하나의 행에 열만 있는 것이다.
(위의 그림 예시에서는 1 차원에 4 개의 element 만 있음)
각 숫자는 value(요소)라고도 부른다.
shape 를 보는 코드 예시는 그림 상에 없지만 결과적으로 (4,) 의 결과를 보여줄 것이다.

```
matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
np.array(matrix, int).shape
```

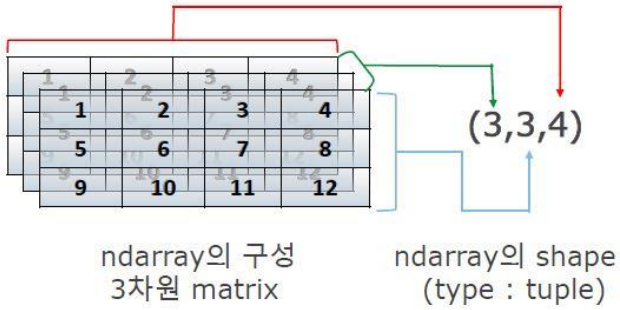
```
(3,4)
```



매트릭스는 행과 열이 같이 있는 것을 의미한다.
matrix 의 shape 를 찍어보면 3 행 4 열의 shape 가 나온다. (3,4)

```
tensor = [ [[1,2,3,4],[5,6,7,8],[9,10,11,12]],
           [[1,2,3,4],[5,6,7,8],[9,10,11,12]],
           [[1,2,3,4],[5,6,7,8],[9,10,11,12]] ]
np.array(tensor, int).shape
```

(3, 3, 4)



tensor 는 매트릭스가 여러 개 있는 것으로 3 차원, 4 차원, 5 차원...이 다 표현된다.

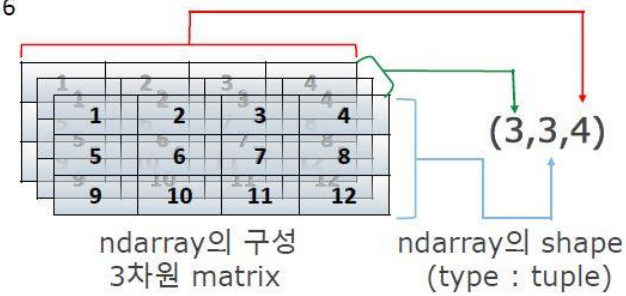
벡터는 (4,), 매트릭스는(3,4) 이런 식으로 2 차원부터는 앞에 하나씩 생긴다.
텐서는 (3,3,4)가 되었다.
새로 만들어낸 차원인 3 차원이 제일 앞에 들어간 걸 볼 수 있다.
(3 차원, 행, 열)로 밀려나게 되었다.

```
tensor = [ [[1,2,3,4],[5,6,7,8],[9,10,11,12]],
           [[1,2,3,4],[5,6,7,8],[9,10,11,12]],
           [[1,2,3,4],[5,6,7,8],[9,10,11,12]] ]
np.array(tensor, int).ndim  # number of dimension
```

3

```
np.array(tensor, int).size  # data의 개수
```

36



np 형식의 array 가 '몇 차원인지 나타내라'와 size 는
요소/ 데이터가 몇 개인지 나타내라는 함수이다.

* tensor 는 딥러닝 가면 이미지 분석할 때 많이 쓴다.

데이터 타입

- Ndarray의 single element가 가지는 data type
- 각 element가 차지하는 memory의 크기가 결정됨

```
np.array( [[1, 2.6, 3.2], [4, 5.1, 6]], dtype = int )
```

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

```
np.array( [[1, 2.6, 3.2], [4, "5", 6]], dtype = np.float32 )
```

```
array([[1. , 2.6 , 3.2],
       [4. , 5. , 6. ]]. dtype=float32 )
```

각 요소마다 데이터 타입을 지정해주면 그 데이터 타입으로 변환이 되는 걸 볼 수 있다.

아래의 예시를 보면 여기는 실수형 이고 여기는 string 타입인데 소수점 타입으로 바꾸면 결과물에 .이 찍혀있는 걸 볼 수 있다.

- nbyte : ndarray object의 메모리 크기 리턴

```
np.array( [[1, 2.6, 3.2], [4, "5", 6]], dtype = np.float32 ).nbytes
```

24 ➔ 32bits = 4bytes → 6 * 4bytes

```
np.array( [[1, 2.6, 3.2], [4, "5", 6]], dtype = np. float64 ).nbytes
```

48 ➔ 64bits = 8bytes → 6 * 8bytes

```
np.array( [[1, 2.6, 3.2], [4, "5", 6]], dtype = np.int8 ).nbytes
```

6 ➔ 8bits = 1bytes → 6 * 1bytes

하나의 value 가 4 바이트를 가지는데
요소가 6 개 있으니까, 이게 메모리에서 차지하는 건 총 24 바이트가 된다.
그 다음 타입은 하나가 8 바이트이니까 48 바이트를 차지한다.

array 생성할 때 데이터가 수 만개 수십 만개 있으면 메모리 차지 비율이 매우 늘어나서,
만약 숫자로 12 이런 실수형 데이터가 있으면
float type 이나 bit 타입을 줄여주는 게 성능에 도움이 된다.

- Array의 shape의 크기를 변경함 (element의 개수는 동일)

```
t_matrix = [[1, 2, 3, 4], [5, 6, 7, 8]]
np.array(t_matrix).shape
```

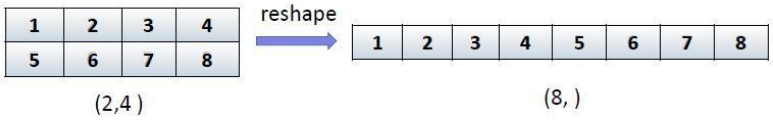
```
(2, 4)
```

```
np.array(t_matrix).reshape(8, )
```

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

```
np.array(t_matrix).reshape(8, ).shape
```

```
(8, )
```



(2,4) (8,)

reshape 은 크기 변경이 가능하다.
[[1,2,3,4],[5,6,7,8]] 은 2x4 (투 바이 포) 매트릭스인데,
이걸 reshape 를 통해 변경할 수 있다.
reshape(8,) 이렇게 콤마만 써준 경우는 벡터로 변경한다는 것인데
요소의 개수가 똑같아야 한다. (2x4 = 8)

- Array의 size만 같다면 다차원으로 자유로이 변형 가능

```
np.array(t_matrix).reshape(2, 4).shape
```

(2, 4)

```
np.array(t_matrix).reshape(-1, 2).shape
```

(4, 2) ➔ -1은 size를 기반으로 row개수 산정한다

```
np.array(t_matrix).reshape(2, 2, 2)
```

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

       [[5, 6],
        [7, 8]]])
```

```
np.array(t_matrix).reshape(2, 2, 2).shape
```

(2, 2, 2)

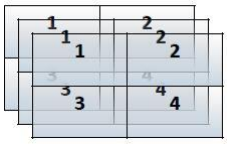
-1 을 이용하면, 뒤에 있는 값에 따라서 알아서 변환시켜 준다는 뜻이다.
reshape(-1,2)를 쓰면 뒤에 2 열이기 때문에 8 에서 2 를 나눈 4 행이 앞에 들어가게 된다.

3 차원 텐서까지 reshape 를 확장시켜보면, 2 행 2 열이 2 개가 되도록 만들어 본다.

flatten
- 다차원 array를 1차원 array로 변환

```
t_matrix = [ [[1, 2], [3, 4]], [[1, 2], [3, 4]], [[1, 2], [3, 4]] ]
np.array(t_matrix).flatten()
```

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



(3, 2, 2)

(16,)

flatten 은 쭉 펴준다. 다차원이 와도 다 1 차원의 벡터 형태로 바꿔준다.

indexing

```
a = np.array([[1, 2.2, 3], [4, 5, 6.3], int)
print(a)
```

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

```
print(a[0,0])
```

```
1
```

```
print(a[0][0])
```

```
1
```

```
a[0, 0] = 7
print(a)
```

```
[[7 2 3]
 [4 5 6]]
```

```
a[0][0] = 8
print(a)
```

```
[[8 2 3]
 [4 5 6]]
```

인덱싱은 내가 찾아가는 주소르 의미한다.
2 x 3 매트릭스를 맨 위에 만들었는데
여기서 print(a[0,0])은 row 에서 0 번째 자리이고
column 에서 0 번째 자리에 있는 값을 출력하라는 뜻이다.

아래에 있는 print(a[0][0])도 똑같은 말이고
같은 결과를 나타내는 형식이 2 개 있다는 걸 보여준 것이다.

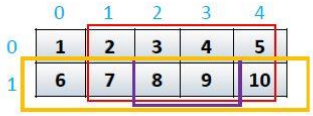
np.array 가 아닌 list 형식은 밑에 방식만 지원해줄 것이다.
(나중에 list 와 matrix 의 인덱스 비교해보기)

slicing

- list와 달리 행과 열 부분을 나눠서 slicing이 가능함
- matrix 부분 집합 추출할 때 유용

```
a = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], int)

a[:, 1:]  # 전체 row의 1 열 이상
a[1, 2:4] # 1 row의 2 열~3 열
a[1:3]  # 1 row ~ 로우 전체
```




```
a = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]], int)
print(a)
```

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

```
a[:, 1:]
```

```
array([[ 2,  3,  4,  5],
       [ 7,  8,  9, 10]])
```

```
a[1, 2:4]
```

```
array([8, 9])
```

```
a[1:3]
```

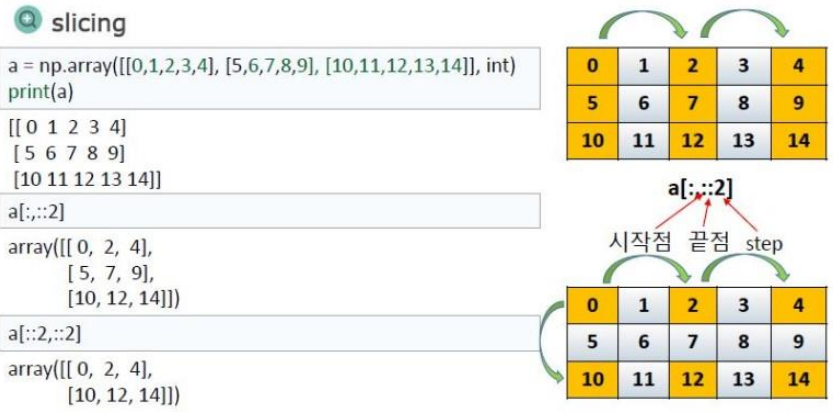
```
array([[ 6,  7,  8,  9, 10]])
```

slicing 은 말 그대로 자르는 것을 의미한다.
내가 원하는 만큼 잘라서 출력하고 싶을 때 사용한다.

2x5 의 매트릭스 형태의 배열을 만들고서 a[:, 1:] 를 써줬는데 ":"는 전체라는 뜻이다.
왼쪽의 :는 전체 row 를 나타낸다.
그리고 1:은 위에 그림에서 빨간 색으로 표시해두었는데,
column 의 1 부터 전체를 다 의미한다.

주의할 점은 1:이 아닌 a[1, 2:4] 처럼 숫자:숫자의 형식인 경우에 마지막 숫자는 포함되지 않는다.
그림에서 보라색 부분인데 2 에서 3 까지만 잘라진다.

또한 a[1,3:-1]과 같은 식으로 마지막에 -1 을 써주면 맨 끝 열을 의미한다.
가장 끝에 있는 것을 써주었으니까 슬라이싱할 때는 뒤에서 두 번째 값까지 나오게 된다.



::를 두번 사용하면 원하는 만큼 step 을 넘어가며 슬라이싱도 가능하다.

- array의 범위를 지정하여, 값의 list를 생성하는 명령어

```
np.arange(20) # list의 range와 같은 역할, integer로 0부터 19까지 배열추출
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19])
np.arange(0, 1, 0.2) # float 가능
array([0. , 0.2, 0.4, 0.6, 0.8])
np.arange(20).reshape(4,5)
array([[ 0, 1, 2, 3, 4],
[ 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14],
[15, 16, 17, 18, 19]])
```

arange(20)은 총 20 개의 value 를 vector 형식으로 쭉 뽑게 된다.
이것도 스텝을 쓸 수 있는데
0 에서 1 까지 0.2 씩 커지라고 np.arange(0, 1, 0.2) 이런 식으로 array 를 생성할 수 있다.

아까 배운 reshape 통해서 매트릭스 형태로 바꿔서 나타낼 수도 있다.

zeros, ones, empty 가 있는데 0 으로 채우고 1 로 채우고..... 하는 애들이다.

```
np.zeros(shape = (5,2), dtype = np.int8) # 5 by 2 zero matrix 생성, int8
```

```
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

```
np.ones(shape = (5,2), dtype = np.int8) # 5 by 2 one matrix 생성, int8
```

```
array([[1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1]], dtype=int8)
```

```
np.empty(shape = (3,2), dtype = np.int8)
```

```
array([[64,  0],
       [ 0,  0],
       [ 0,  0]], dtype=int8)
```

empty 는 주어진 shape 대로 비어있는 것을 생성한다.
이런 식으로 array 를 만드는데 메모리를 어느 정도 할당 시켜준다.
그런데 메모리에 기종에 있었던 값을 보여준다!!

zeros 나 ones 는 0 과 1 로 메모리 할당 값을 초기화 시켜주는데
empty 는 초기화시키지 않고 기존에 메모리에 있는 찌꺼기 그대로 보여준다.

np.empty()함수를 칠 때마다 결과값이 변하는 걸 볼 수 있고,
때때로 아까 연습할 때 만들어 둔 배열이 보이기도 한다.

메모리에 원래 값이 있는 걸 1 로 새로 채우면 속도가 느려지지만,
메모리 부분을 그대로 써서 나타내면 만드는 속도가 있어서 빨라진다.

* 실제로 jupyter 에서 코드로 실습을 하실 때 보면
  함수 위에 커서를 놓고 shift+tab 을 누르면
  함수에 들어갈 argument 나 이 함수가 어떤 걸 뜻하는지, 예제 등이 아래에 보인다.
  (이건 numpy 를 구글에 쳤을 때, numpy.org 홈페이지에서 튜토리얼에 나와있는 것이기도 하다.)

- 기존 ndarray의 shape 크기만큼 1 or 0 or empty array 반환

```
t_matrix = np.arange(15).reshape(3,5)
np.ones_like(t_matrix)
```

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

```
t_matrix1 = np.arange(15).reshape(3,5)
np.zeros_like(t_matrix1)
```

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

```
t_matrix2 = np.arange(15).reshape(3,5)
np.empty_like(t_matrix2)
```

```
array([[    0,     0,     0,     0,     0],
       [    0,     0,     0,     0,     0],
       [ 1224,     0,     0, 6357104,     0]])
```

something_like 는 ones 나 zeros 같은 것인데,
기존의 matrix 를 새로 채우고 싶을 때 사용한다.
원래 값이 있었던 1 로 다 채우라, 0 으로 채우라는 예제가 위에 있다.

- 단위 행렬(i 행렬)을 생성
  n → number of rows

```
np.identity(n = 3, dtype = np.int8)
```

```
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]], dtype=int8)
```

```
np.identity(n = 5)
```

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

identity 는 단위행렬인 대각선이 1 로 채워진 행렬을 의미한다.

- 대각선이 1인 행렬, k값이 시작 index 변경 가능

```
np.eye(N = 3, M = 4, dtype = np.int)
```

```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0]], dtype=int8)
```

```
np.eye(4)  # identity 행렬과 같게 출력
```

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

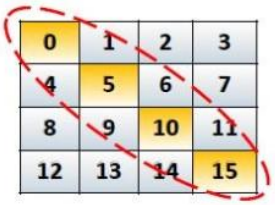
```
np.eye(3, 6, k = 3)  # k → start index
```

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

대각선은 1 로 채워지지만 시작점을 다르게 하려면 np.eye()를 사용하면 된다.
N 은 row 이고 M 은 column 이다.
숫자를 하나만 쓰면 즉, np.eye(4)처럼 쓰면 identity 결과랑 똑같이 나온다.
k 는 시작점을 의미해서, k 를 3 으로 바꿔주면 3 번째 열부터 대각선으로 생성되는 걸 볼 수 있다.

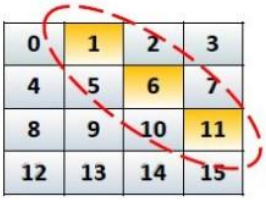
```
t_matrix = np.arange(16).reshape(4, 4)
np.diag(t_matrix)
```

```
array([ 0,  5, 10, 15])
```




```
np.diag(t_matrix, k = 1)
```

```
array([1, 6, 11])
```



np.diag 하면 대각선 요소만을 추출해주고
마찬가지로 k 값을 정의해주면 거기부터 대각선에 있는 걸 뽑아준다.

* matrix 랑 tensor 를 구분하기 힘들면 앞의 대괄호를 본다.
  대괄호가 4 개면 4 차원이니까 tensor 이다.
  대괄호라는 건 [[[[1,2,3,4],[2,3,4,5].... 이런 식... 맨 앞에 대괄호 몇 개인지 보기.

Random Sampling

```
np.random.uniform(0, 1, 12).reshape(4, 3) # 균등분포
 # np.random.uniform(최소값, 최대값, data 개수)
```

```
array([[0.6471031 , 0.70506654, 0.59970038],
       [0.97119081, 0.87124496, 0.68000872],
       [0.70800846, 0.53908685, 0.57826248],
       [0.54314476, 0.89906667, 0.24140212]])
```

```
np.random.normal(0, 1, 12).reshape(4, 3) # 정규분포
```

```
array([[-0.29703021, -0.16169222,  0.52078926],
       [ 0.85924274, -0.88387902, -1.43606194],
       [ 1.57495957, -0.18813136, -1.94351926],
       [-0.52597883, -1.30939238, -0.83384486]])
```

uniform 은 일자로 균등하게 생긴 함수를 의미한다.
'0 에서 1 까지 12 개를 뽑아라 근데 값을 균등분포로 추출해라'가 첫 번째 예시이다.
그리고 평균이 0 이고 표준편차가 1 인 정규분포를 표준정규분포라고 부른다.
np.random.normal(0, 1, ?)를 쓰면 여기서 뽑으라는 명령어가 된다.

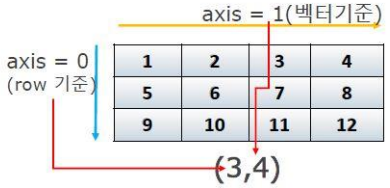
axis
- 모든 operation function을 실행할 때, 기준이 되는 dimension 축

```
t_array = np.arange(1, 13).reshape(3, 4)
t_array
```

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

```
t_array.sum(axis = 0), t_array.sum(axis = 1)
```

```
(array([15, 18, 21, 24]), array([10, 26, 42]))
```



사칙연산할 때 axis 가 중요하다!

아래에 있는 예제에서

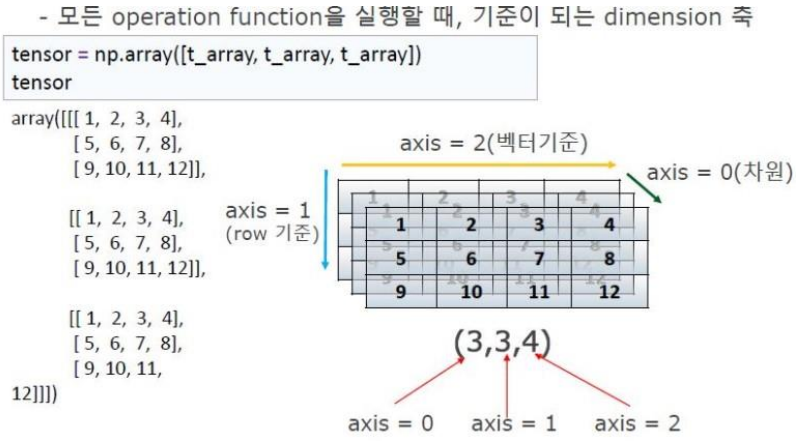
t_array.sum(axis=0)을 하면 row 기준으로 더하기 때문에

1, 5, 9 를 아래로 더하면 15 가 나오고

그 옆에 2, 6, 10 을 더하면 18 이 나온다.

그리고 t_array.sum(axis=1)로 vector 기준으로 하면

1+2+3+4 는 10 이 나온다. 5+6+7+8 은 26 이 나온다.



매트릭스가 3 개라면 axis 도 3 개가 나온다는 걸 볼 수 있다.

```
t_array = np.arange(1, 13).reshape(3, 4)
t_array
```

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

```
t_array.mean(), t_array.mean(axis = 0)
```

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

```
t_array.std(), t_array.std(axis = 0)
```

```
(3.452052529534663, array([3.26598632, 3.26598632,
                           3.26598632, 3.26598632]))
```

mean 은 평균 std 는 표준편차를 구해준다.

1 에서 12 까지를 평균을 구하면 6.5 가 나온다.

이걸 axis 기준을 0 으로 해주면 row 기준이기 때문에

1, 5, 9 의 평균을 내고 2, 6, 10 의 평균을 내고... 해서 구해준다.

표준편차는

전체에 대한 표준편차를 기본으로 구해주다가

이것도 axis=0 기준으로 하면 row 기준으로 나오게 된다.

Mathematical functions

지수함수(exponential)
- exp, expm1, exp2, log, log2, log10, log1p, power, sqrt

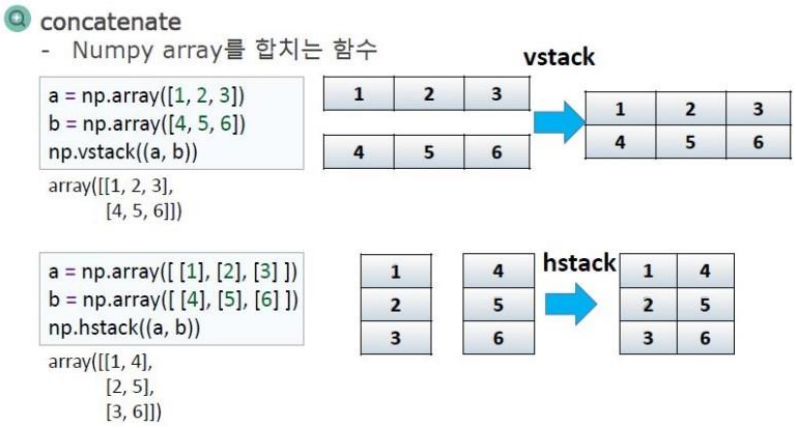
삼각함수(trigonometric)
- sin, cos, tan, acsin, arccos, atctan

쌍곡선함수(hyperbolic)
- sinh, cosh, tanh, acsinh, arccosh, atctanh

지수함수 로그함수 제곱이나, 루트 상삼각함수, 쌍곡선함수 등이
numpy 에서 제공되는데 필요할 때마다 쓰시면 될 것이다.

```
np.exp(t_array) # 지수함수
```
```
array([[2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 5.45981500e+01],
       [1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.98095799e+03],
       [8.10308393e+03, 2.20264658e+04, 5.98741417e+04, 1.62754791e+05]])
```
```
np.sqrt(t_array) # 루트(√ )
```
```
array([[1.        , 1.41421356, 1.73205081, 2.        ],
       [2.23606798, 2.44948974, 2.64575131, 2.82842712],
       [3.        , 3.16227766, 3.31662479, 3.46410162]])
```
```
np.sin(t_array) # sin 함수
```
```
array([[ 0.84147098,  0.90929743,  0.14112001, -0.7568025 ],
       [-0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825],
       [ 0.41211849, -0.54402111, -0.99999021, -0.53657292]])
```

그냥 보면, 이렇게 하나하나마다 다 연산이 된다.
이걸 브로드캐스팅이라고 부르는데 하나하나마다 연산이 되는 걸 뜻한다.

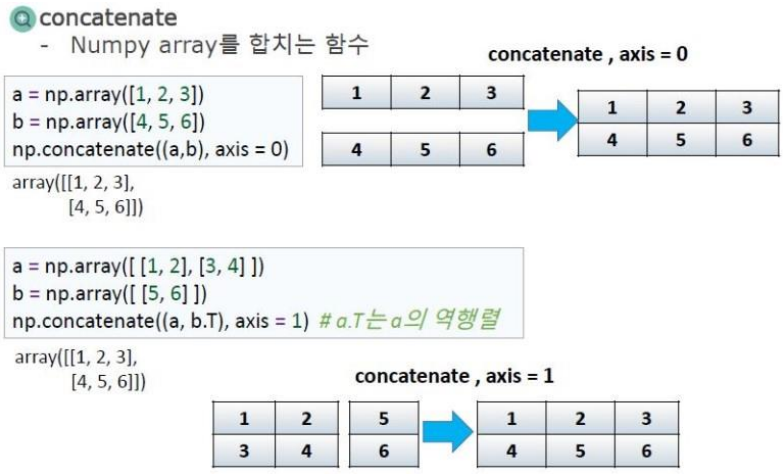


concatenate 는 합치는 것을 의미한다.
vstack 이라는 함수는 row 를 기준으로 합친다.
1, 2, 3 과 4, 5, 6 벡터를 아래로 붙였다.

함수에 쓸 때는 튜플 형식으로 (a,b)라고 쓴다.

hstack 은 컬럼 기준으로 합친다.
그런데 넣을 때 [1],[2],[3] 이런 식으로 넣지 않고
[1,2,3] 이렇게 리스트 형식으로 넣게 되면
위의 예제랑 똑같아져서

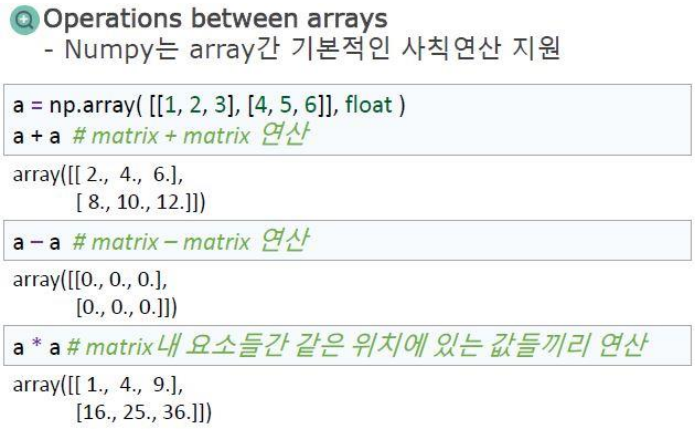
우리가 원하는 대로 세로로 붙지 않고 옆으로 길게 붙어버리니까 주의해야 한다.



vstack 이랑 hstack 똑같은 함수인데 axis 로 결정된다.
axis=0 같은 경우는 vstack 이 된다.

아래 예시를 보면 b.T 라고 표현되어있는데,
이것은 전치행렬로 행과 열을 바꿔주는 것이다.
b 는 원래 [5,6] 이런 벡터 형태였는데,
b.T 가 되면서 세로로 긴 컬럼 벡터 형식으로 변하였다.

array 연산



a 는 [[1,2,3], [4,5,6]]이라는 매트릭스인데
같은 shape 을 가지면 같은 index 가진 값끼리 더하고 빼고 곱해줄 수 있다.
이걸 **operation** 이라고 한다.

먼저 a+a 를 해보면,
같은 위치에 있는 1 과 1 이 더해져서 2 가 되고
그 옆에 2 와 2 가 더해져서 4 가 된다.

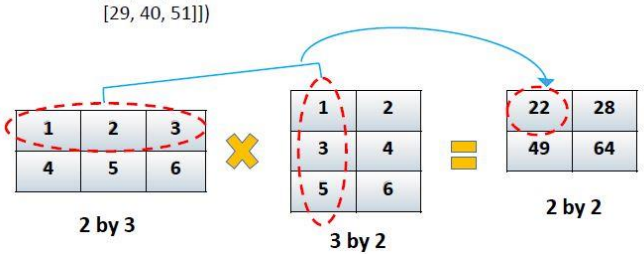
이렇게 같은 index 에 있는 것 끼리 더하고 빼고 곱해줘서 그 자리에 결과값을 써준다.
= **Element-wise Operation** 이라고 한다.

우리가 기본적으로 아는 행렬의 곱을 구할 때는
앞 행렬의 열과 뒤 행렬의 행 개수가 같아야 하는데 이런 곱을 위해서는 **dot 이란 함수를** 쓴다.

Dot product
- matrix의 기본 연산
- dot 함수 사용

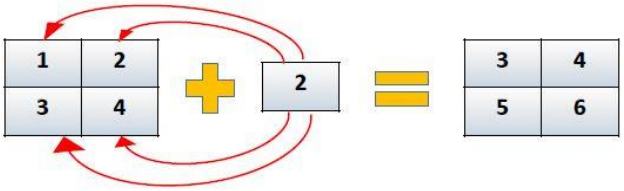
```
dot_a = np.arrange(1, 7).reshape(2, 3)
dot_b = np.arrange(1, 7).reshape(3, 2)
dot_a.dot(dot_b)
```

```
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51]])
```



dot 함수를 써서 2 by 3 matrix 와 3 by 2 matrix 를 dot 함수로 곱하면
결과는 2 by 2 가 나온다.

broadcasting
- shape이 다른 배열 간 연산을 지원하는 기능




```
t_matrix = np.array( [[1, 2], [3, 4]], float )
scalar = 2
t_matrix + scalar  # matrix , scalar 덧셈
```

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

```
t_matrix - scalar  # matrix , scalar 뺄셈
```

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

```
t_matrix * 3  # matrix , scalar 곱셈
```

```
array([[ 3., 6.],
       [ 9., 12.]])
```

```
t_matrix / 3  # matrix , scalar 나눗셈
```

```
array([[0.33333333, 0.66666667],
       [1.        , 1.33333333]])
```

```
t_matrix // 3  # matrix , scalar 몫
```

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

```
t_matrix ** 3  # matrix , scalar 제곱
```

```
array([[ 1., 8.],
       [27., 64.]])
```

아까 본 연산은 각 요소마다 하나씩 개개인 적으로 되는데
broadcasting 은 shape 이 달라도 연산이 지원된다.

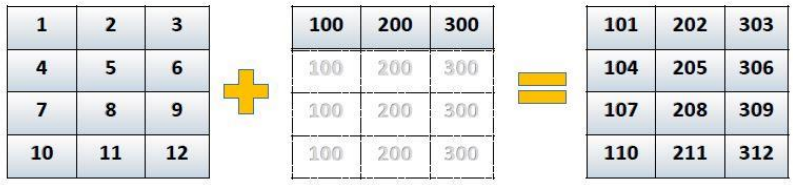
그런데 무조건 되는 건 아니고
행과 열 중 하나가 같거나
하나의 스칼라 값만 주어졌을 때만 이 연산이 가능하다.

t_matrix - 2 는 1, 2, 3, 4 에서 2 를 뺀 값이 되니까 -1, 0, 1, 2 가 나온다.
매트릭스와 스칼라 뿐 아니라 벡터와 매트릭스도 가능하다.

- scalar-matrix 외에도, vector-matrix간의 연산도 지원

```
t_matrix = np.arange(1, 13).reshape(3, 4)
t_vector = np.arange(100, 400, 100)
t_matrix + t_vector
```

```
array([[101, 202, 303],
       [104, 205, 306],
       [107, 208, 309],
       [110, 211, 312]])
```



크기가 다르지만 알아서 똑같은 벡터를 아래에 넣어서 연산을 해준다.
그 값이 매트릭스 형태로 나오게 된다.

Numpy performance
- jupyter notebook 환경에서 코드의 퍼포먼스를 체크하는 timeit 이용

```
def sclar_vector_product(scalar, vector):  # scalar와 vector 끼리 곱셈
    result = []
    for value in vector:
        result.append(scalar * value)
    return result

i_max = 100000000

vector = list(range(i_max))
scalr = 2

%timeit sclar_vector_product(scalr, vector)  # for loop을 이용한 성능
%timeit[scalar * value for value in range(i_max)] # list comprehension을
                                                  이용한 성능
%timeit np.arange(i_max) * scalar # numpy를 이용한 성능
```

numpy 가 왜 list 를 사용하는 것보다 빠른가 & for 를 사용한 연산보다 빠른가

주피터에서 timeit 이라는 함수는 연산이 얼마나 걸리는지 타임을 알려주는 매직 커맨드이다.
(1 억이 너무 많아서 오래 걸릴까봐 i_max 는 100 만번으로 줄여서 돌렸다)
백만번까지 list 랑 스칼라값 곱해줄 것이다.

두 번째에 있는 list 에 comprehension 이라는 함수 있는데
이걸 통해서 연산을 할 수 있다. (이건 리스트 부분을 공부해야 한다.)

마지막은 매트릭스와 numpy 를 이용한 연산이다.
for 루프 = 246 ms, list = 150ms, numpy = 57ms 로 성능 차이를 보였다.

Q: 원래 float 으로 소수점이 있던 숫자를 int 로 변환하면 뒤 숫자는 버리나요? 반올림인가요?
Q: reshape 로 벡터나 매트릭스를 바꿀 때 어떤 순서로 모양에 들어가지는 건지 천천히 설명해주실 수 있나요? 그 순서를 제가 옵션을 사용해서 바꿀 수도 있나요?
Q: 마지막에 본 함수 앞에 붙은 %는 무엇을 의미하나요? -> 이것은 파이썬이 아닌 주피터에서 사용되는 커맨드를 의미합니다. 매직 커맨드라고 부릅니다.
Q: 코드를 쓸 때 탭을 쓰는 게 좋을까요 스페이스를 쓰는 게 좋을까요? -> 탭 보다 스페이스를 권유합니다.