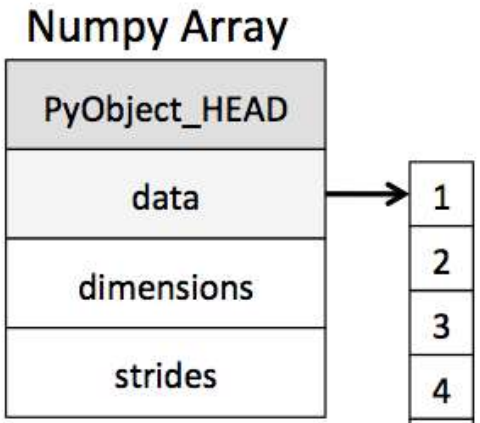
Numpy

김지성 강사

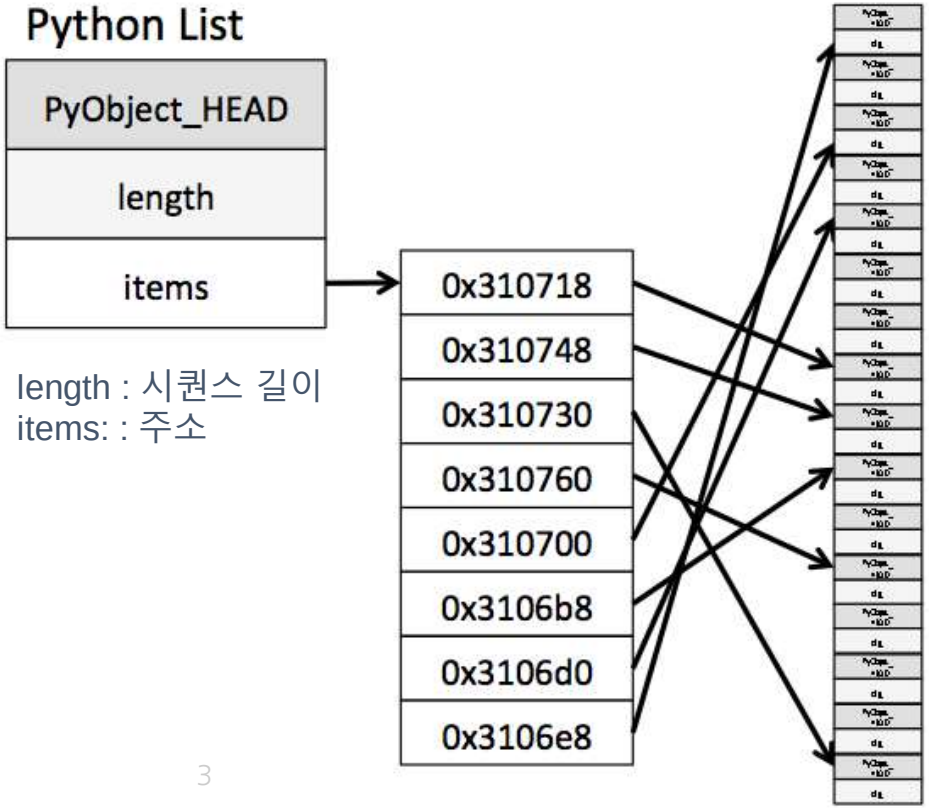
NumPy의 장점

1. NumPy는 Numerical Python의 줄임말로, 산술 계산을 위한 패키지
2. NumPy 배열 객체는 데이터 교환을 위한 공통 언어처럼 사용
3. Python의 연속된 자료형들보다 훨씬 더 적은 메모리 사용
4. 다양한 계산을 수행하기 위해 적합

NumPy 구조



data : 데이터
dimensions : 배열 구조
strides : Type 비트 수



length : 시퀀스 길이
items: : 주소

NumPy 속도 계산

```
1  numpy_ = np.arange(1000000)
2  list_ = list(range(1000000))
3  %time for _ in range(10): my_arr = numpy_ * 2
4  %time for _ in range(10): my_list = [x * 2 for x in list_]
```

✓ 0.6s

CPU times: user 12 ms, sys: 3.97 ms, total: 16 ms

Wall time: 16 ms

CPU times: user 524 ms, sys: 101 ms, total: 626 ms

Wall time: 629 ms

NumPy 계산의 편리함 (벡터화)

행렬합 구하기

matrix_1 = [[1,2],[3,4]]
matrix_2 = [[5,6],[7,8]]

list 계산

```
for i in range(len(matrix_1)):
    tmp = []
    for j in range(len(matrix_2)):

                tmp.append(matrix_1[i][j]+matrix_
2[i][j])
    matrix_result.append(tmp)
```

numpy 계산

```
matrix_result = np.array(matrix_1) + np.array(matrix_2)
```

NumPy의 구조상 이점

NumPy의 장점

- 모든 데이터를 하나의 리스트 시퀀스로 다루게 되면 속도⬇ 메모리⬆

NumPy와 List의 차이점

- NumPy는 모든 원소(items)가 같은 자료형(type)이다.
- NumPy는 내부 배열의 원소 개수가 같아야 한다.

  Ex) np.array([1],[1,2],[1,2,3]) (X)

NumPy 패키지 임포트

1.NumPy 패키지의 별칭은 np라는 이름으로 임포트하는 것이 관례 !

2.아래와 같이 임포트를 해주면 커널이 동작하는 동안 계속 사용 가능 !

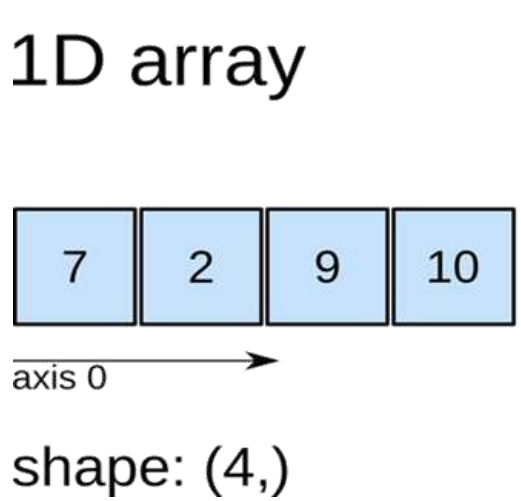
```
import numpy as np
```

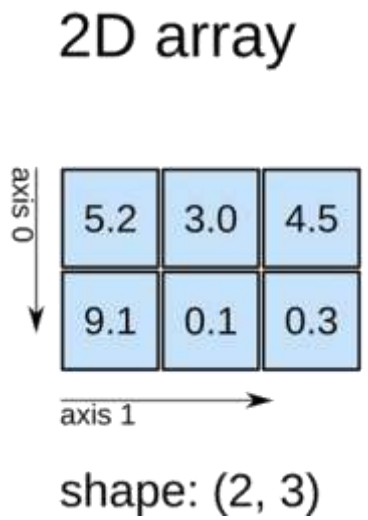
✓ 0.0s Python

배열 만들기 ndarray

- 넘파이 배열 객체인 ndarray 는 N-dimensional Array의 약자입니다.

- 1차원 배열, 2차원 배열, 3차원 배열 등의 다차원 배열 자료 구조를 지원합니다.





다차원 ndarray

0차원
np.array(1)

1차원
np.array([1,2])

2차원 혹은 그 이상
np.array([1,2,3],[4,5,6])

구조 확인
ndarray.shape
ndarray.ndim
ndarray.size

```
x = np.array([[1, 2, 3],[4, 5, 6]])
print('배열의 구조 : ',x.shape)
print('배열의 차원 : ', x.ndim)
print('전체 원소의 개수 : ',x.size)
```

✓ 0.0s

```
배열의 구조 :  (2, 3)
배열의 차원 :  2
전체 원소의 개수 :  6
```

자료형

int(8bit, 16bit, 32bit, 64bit)
부호가 있는 정수형

uint(8bit, 16bit, 32bit, 64bit)
부호가 없는 정수형

float(16bit, 32bit, 64bit, 128bit)
부호가 있는 실수형

복소수형
complex64 : 실수(float32),허수(float32)를 가진 복소수
complex128 : 실수(float64),허수(float64)를 가진 복소수

bool (1bit)
True, False

자료형

dtype 접두 사	설명	사용 예제
‘?’	boolean	?(True, False)
‘i’	int	i8(64비트)
‘u’	unsigned int	u8(64비트)
‘f’	float	f8(64비트)
‘c’	complex	c16(128비트)
‘O’	Object	0(객체에 대한 포인터)
‘U’	Unicode string	U24(24 유니코드 글자)

자료형

dtype=’f’ : 실수 자료형으로 지정

x.dtype : x의 자료형 확인

유니코드 문자열로 지정된 자료형의 덧셈은 concatenate의 결과를 보여준다.
Ex) ‘1’+’2’=’12’

```
1  x = np.array([1, 2, 3], dtype='f')
2  x.dtype
```

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

```
1  x[0] + x[1]
```

```
3.0
```

```
1  x = np.array([1, 2, 3], dtype='U')
2  x.dtype
```

```
dtype('<U1')
```

```
1  x[0] + x[1]
```

```
'12'
```

자료형

NumPy에서는 **무한대**를 표현하기 위해서는 **np.inf**(infinity)로 표현,
정의할 수 없는 숫자는 **np.nan**(not a number)으로 표현한다.

```
1  np.array([0, 1, -1, 0]) / np.array([1, 0, 0, 0])
```

```
<ipython-input-11-6ab12ec6e7a4>:1: RuntimeWarning: divide by zero encountered in true_divide
  np.array([0, 1, -1, 0]) / np.array([1, 0, 0, 0])
<ipython-input-11-6ab12ec6e7a4>:1: RuntimeWarning: invalid value encountered in true_divide
  np.array([0, 1, -1, 0]) / np.array([1, 0, 0, 0])
array([  0.,  inf, -inf,  nan])
```

```
1  np.log(0)
```

```
<ipython-input-12-f6e7c0610b57>:1: RuntimeWarning: divide by zero encountered in log
  np.log(0)
-inf
```

```
1  np.exp(-np.inf)
```

```
0.0
```

자료형 연습 문제

1~18의 값을 가지는 shape이 (3, 2, 3) ndarray를 만들어봅시다. 단 자료형은 실수로 합니다.

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

       [[ 7.,  8.,  9.],
        [10., 11., 12.]],

       [[13., 14., 15.],
        [16., 17., 18.]]])
```

벡터화 연산(Vectorized Operation)

NumPy 배열 객체는 배열의 각 요소에 대한 반복 연산을 하나의 명령어로 처리 하는 벡터화 연산(Vectorized Operation)을 지원

```
data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
answer = [2 * d for d in data]
```

```
answer
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
x = np.array(data)
x
```

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

```
2 * x
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

벡터화 연산(Vectorized Operation)

벡터화 연산은 비교 연산과 논리 연산을 포함한 모든 종류의 수학 연산에 적용

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
```

```
2 * a + b
```

```
array([12, 24, 36])
```

```
a == 2
```

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

```
b > 10
```

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

```
(a == 2) & (b > 10)
```

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

벡터화 연산(Vectorized Operation) 연습 문제

array([[1, 2, 3], [4, 5, 6]]) 과 array([[7, 8, 9], [10, 11, 12]]를 만들고 이 두 넘파이 배열을 활용하여 아래의 계산 결과를 구해보세요.

```
실행 결과:
array([[ 8, 10, 12],
       [14, 16, 18]])
```

벡터화 연산(Vectorized Operation) 연습 문제

1부터 9까지 정수 값을 갖는 1차원 ndarray를 생성하세요.

그리고 그 ndarray를 활용하여 3부터 27까지의 3의 배수를 갖는 ndarray를 출력해보세요.

출력 결과 : array([3,6,9,12,15,18,21,24,27])

Slicing

1차원 배열의 인덱스 슬라이싱은 리스트와 동일

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

```
a[2]
```

```
2
```

```
a[-1]
```

```
4
```

Slicing

- 다차원 배열의 인덱스 슬라이싱 콤마(comma ,)를 사용하여 접근 가능
- 콤마로 구분된 차원을 축(axis)라고 부르며 순서대로 가장 큰 차원부터 접근

x배열(2x3)

1	2	3
4	5	6

x[첫번째 차원,두번째 차원]

1. x[1,2] == 6

2. x[0,1] == 2

Slicing

```
x = np.array([[[1,2,3],[4,5,6],[7,8,9]],
              [[10,20,30],[40,50,60],[70,80,90]]])
display(x)
print(x.shape)
print('x[1,2] : ',x[1,2])
print('x[0,2] : ',x[0,2])
```

✓ 0.0s

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

       [[10, 20, 30],
        [40, 50, 60],
        [70, 80, 90]]])

(2, 3, 3)
x[1,2] :  [70 80 90]
x[0,2] :  [7 8 9]
```

Slicing 다차원

2x3x3 차원이 존재하는 배열 x
x[0:,2] => x[0][2], x[1][2]
x[0:,1:] => x[0][1], x[0][2], x[1][1], x[1][2]

```
display('x[0:,2] : ',x[0:,2])
display('x[0:,1:] : ',x[0:,1:])
```

✓ 0.0s

```
'x[0:,2] : '

array([[ 7,  8,  9],
       [70, 80, 90]])

'x[0:,1:] : '

array([[[ 4,  5,  6],
        [ 7,  8,  9]],

       [[40, 50, 60],
        [70, 80, 90]]])
```

Slicing 다차원

2x3x3 차원이 존재하는 배열 x
x[0:,::2] => x[0][0], x[0][2], x[1][0], x[1][2]

```
display('x[0:,::2] : ',x[0:,::2])
```

✓ 0.0s

'x[0:,::2] : '

```
array([[[ 1,  2,  3],
        [ 7,  8,  9]],

       [[10, 20, 30],
        [70, 80, 90]]])
```

Slicing 연습 문제

```
m = np.array([[ 0,  1,  2,  3,  4],
              [ 5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14]])
```

1.이 행렬에서 값 7 을 인덱싱한다.
2.이 행렬에서 값 14 을 인덱싱한다.
3.이 행렬에서 배열 [6, 7] 을 슬라이싱한다.
4.이 행렬에서 배열 [7, 12] 을 슬라이싱한다.
5.이 행렬에서 배열 [[3, 4], [8, 9]] 을 슬라이싱한다.

Slicing 연습 문제

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

       [[ 10,  20,  30],
        [ 40,  50,  60],
        [ 70,  80,  90]],

       [[100, 200, 300],
        [400, 500, 600],
        [700, 800, 900]]])
```

왼쪽과 같은 배열을 만들고 다음 값을 출력하세요.

```
array([[ 2,  5,  8],
       [ 20,  50,  80],
       [200, 500, 800]])
```

Slicing 연습 문제

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

       [[ 10,  20,  30],
        [ 40,  50,  60],
        [ 70,  80,  90]],

       [[100, 200, 300],
        [400, 500, 600],
        [700, 800, 900]]])
```

왼쪽과 같은 배열을 만들고 다음 값을 출력하세요.

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

       [[100, 300]]])
```

Slicing 연습 문제

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

       [[ 10,  20,  30],
        [ 40,  50,  60],
        [ 70,  80,  90]],

       [[100, 200, 300],
        [400, 500, 600],
        [700, 800, 900]]])
```

왼쪽과 같은 배열을 만들고 다음 값을 출력하세요.

```
array([[[900, 800, 700],
        [600, 500, 400],
        [300, 200, 100]],

       [[ 90,  80,  70],
        [ 60,  50,  40],
        [ 30,  20,  10]],

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

Slicing 연습 문제

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

       [[ 10,  20,  30],
        [ 40,  50,  60],
        [ 70,  80,  90]],

       [[100, 200, 300],
        [400, 500, 600],
        [700, 800, 900]]])
```

왼쪽과 같은 배열을 만들고 다음 값을 출력하세요.

```
array([[[ 3,  2,  1],
        [ 9,  8,  7]],

       [[30, 20, 10],
        [90, 80, 70]]])
```

배열 생성

NumPy는 몇가지 단순한 배열을 생성하는 명령을 제공

- zeros, ones
- zeros_like, ones_like
- empty
- arange
- linspace, logspace

배열 생성 (zeros, ones)

- np.zeros((shape), dtype="type")
- shape크기 만큼의 0의 값을 가진 배열을 생성
  (type은 지정된 type으로 변환)

```
1   c = np.zeros((5, 2), dtype="i")
2   c
```

```
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int32)
```

배열 생성 (zeros, ones)

- np.ones((shape), dtype=”type”)
- shape크기 만큼의 1의 값을 가진 배열을 생성

  (type은 지정된 type으로 변환)

```
1   e = np.ones((2, 3, 4), dtype="i8")
2   e
```

```
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]]])
```

배열 생성 (zeros_like, ones_like)

- np.ones_like(array, dtype=”type”)
- array크기 만큼의 1의 값을 가진 배열을 생성
  (type은 지정된 type으로 변환)

```
1   f = np.ones_like(b, dtype="f")
2   f
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

배열 생성 연습 문제

빈 문자열을 갖는 shape이 (3, 3, 3) ndarray를 만들어봅시다. 단 dtype은 'U10'으로 합니다.

```
array([[['', '', ''],
        ['', '', ''],
        ['', '', '']],

       [['', '', ''],
        ['', '', ''],
        ['', '', '']],

       [['', '', ''],
        ['', '', ''],
        ['', '', '']]], dtype='<U10')
```

배열 생성, Slicing 연습 문제

zeros/ones 함수를 사용해서 배열을 만들고 아래와 같은 배열을 만들어보세요.

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

배열 생성, Slicing 연습 문제

zeros/ones 함수를 사용해서 배열을 만들고 아래와 같은 배열을 만들어보세요.

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

배열 생성 (empty)

- np.empty((shape))
- 배열의 크기가 커지면 초기화 하는데 시간이 증가,

->특정한 값으로 초기화 하지 않은 empty명령어를 사용

```
1   g = np.empty((4, 3))
2   g
```

```
array([[1.62876606e-316, 2.70810582e-316, 4.64421707e-322],
       [2.70810582e-316, 4.64421707e-322, 2.70810582e-316],
       [5.63923366e-091, 0.00000000e+000, 2.70840305e-316],
       [2.70840305e-316, 2.70840305e-316, 0.00000000e+000]])
```

배열 생성 (arange)

- np.arange(start,end,step)
- NumPy의 range명령어, 특정 규칙에 따라 증가하는 수열 생성

```
1   np.arange(10)  # 0 .. n-1
```

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

```
1   np.arange(3, 21, 2)  # 시작, 끝(포함하지 않음), 단계
```

```
array([ 3,  5,  7,  9, 11, 13, 15, 17, 19])
```

배열 생성 (random)

- np.random.random((shape)) : 0~1 균일 분포(모든 값이 동일한 확률)를 따르는 난수를 생성.
- np.random.rand(shape) : 0~1의 균일 분포에서 난수 생성
- np.random.randint(start,end,size) : start~(end-1)의 균일 분포에서 난수 생성
- np.random.randn(shape) : 가우시안 분포를 따르는 난수 생성

```
print('random : \n', np.random.random((3, 3)))
print('rand : \n', np.random.rand(3,3))
print('randomint : \n', np.random.randint(1, 10, (2, 3)))
print('randn : \n', np.random.randn(3, 3))
```

배열 생성 (linspace, logspace)

- np.linspace(start,stop,num)
- np.logspace(start,stop,num)
- 선형(linspace) 혹은 로그(logspcae) 구간을 지정한 구간의 수만큼 분할

```
ex = np.linspace(0, 100, 5)
ex
```

```
array([  0.,  25.,  50.,  75., 100.])
```

```
ex2 = np.logspace(0, 100, 5)
ex2
```

```
array([1.e+000, 1.e+025, 1.e+050, 1.e+075, 1.e+100])
```

배열 생성 (linspace, logspace) 연습 문제

linspace()를 활용하여 다음과 같은 1차원 ndarray를 만들어보세요

[0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]

배열의 크기 변경 (reshape)

- 배열의 내부 데이터를 보존한 채 형태만 변경하기 위해서 reshape을 사용
- 12개의 원소를 가진 1차원 행렬이 존재
    1. 3x4 2차원 행렬 변경 가능
    2. 2x2x3 3차원 행렬 변경 가능

```
1   a = np.arange(12)
2   a
```

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

```
1   b = a.reshape(3, 4)
2   b
```

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

배열의 크기 변경 (reshape)

- 배열의 변경할 수 있는 크기가 정해져있기 때문에 튜플의 값 중 하나는 -1로 대체 가능

```
1   a.reshape(3, -1)
```

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

```
1   a.reshape(2, 2, -1)
```

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

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

```
1   a.reshape(2, -1, 2)
```

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

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

배열의 크기 변경 (reshape) 연습 문제

1부터 40까지의 짝수를 갖는 ndarray를 arange()로 만듭니다. 그리고 shape을 변경하여 (2, 2, 5)로 변경해보세요.

```
[ 2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40]
```

```
[[[ 2  4  6  8 10]
  [12 14 16 18 20]]

 [[22 24 26 28 30]
  [32 34 36 38 40]]]
```

배열의 크기 변경 (flatten, ravel)

- 다차원 배열을 1차원으로 축소하기 위해서는 flatten, ravel 메소드 사용

```
1  a.flatten()
```

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

```
1  a.ravel()
```

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

배열의 크기 변경 (newaxis)

- 배열의 차원을 증가시키는 경우 np.newaxis를 사용

```
x=np.arange(12)
print(f'x.shape : {x.shape}, x.ndim : {x.ndim}')
x=x[:,np.newaxis]
print(f'x.shape : {x.shape}, x.ndim : {x.ndim}')
```

✓ 0.0s

```
x.shape : (12,), x.ndim : 1
x.shape : (12, 1), x.ndim : 2
```

전치 (Transpose)

- 배열의 행과 열의 축을 서로 변경

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

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

```
1   A.T
```

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

배열 결합 (hstack, vstack)

- hstack : 배열을 수평으로 연결
- vstack : 배열을 수직으로 연결

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[10,20,30],[40,50,60]])
print('hstack(수평연결) : \n',np.hstack([a,b]))
print()
print('vstack(수직연결) : \n',np.vstack([a,b]))
```

✓ 0.0s

```
hstack(수평연결) :
 [[ 1  2  3 10 20 30]
 [ 4  5  6 40 50 60]]

vstack(수직연결) :
 [[ 1  2  3]
 [ 4  5  6]
 [10 20 30]
 [40 50 60]]
```

배열 결합 연습 문제

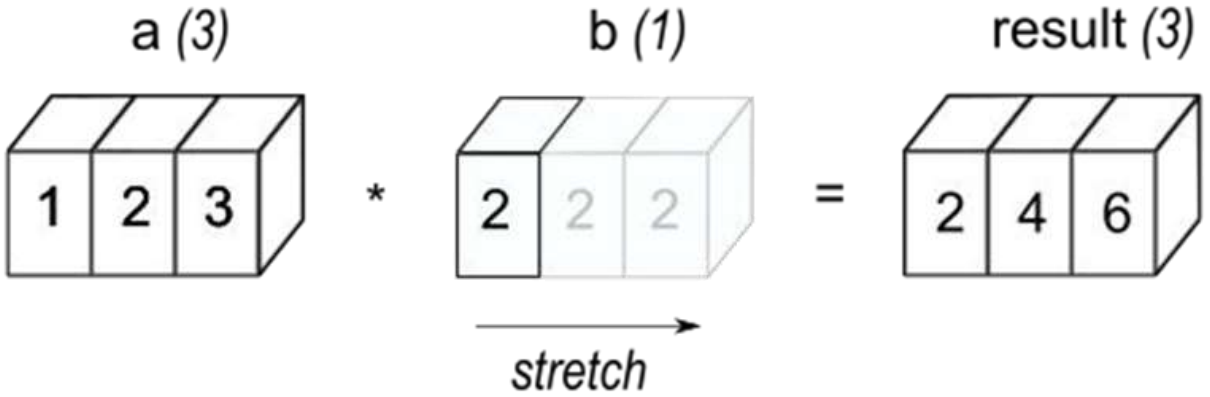
- 다음과 동일한 배열을 만들어 보세요.

```
array([[  0.,   0.,   0.,   1.,   1.],
       [  0.,   0.,   0.,   1.,   1.],
       [  0.,   0.,   0.,   1.,   1.],
       [ 10.,  20.,  30.,  40.,  50.],
       [ 60.,  70.,  80.,  90., 100.],
       [110., 120., 130., 140., 150.],
       [  0.,   0.,   0.,   1.,   1.],
       [  0.,   0.,   0.,   1.,   1.],
       [  0.,   0.,   0.,   1.,   1.],
       [ 10.,  20.,  30.,  40.,  50.],
       [ 60.,  70.,  80.,  90., 100.],
       [110., 120., 130., 140., 150.]])
```

브로드캐스팅

- NumPy 배열은 모양이 다른 배열 간의 연산이 가능하도록 배열의 크기를 변환시켜주는 브로드캐스팅(broadcasting)을 지원

- 크기가 작은 배열이 크기가 큰 배열의 크기로 변환



브로드캐스팅

- 브로드캐스팅이 가능하기 때문에 **배열x배열**보다 **배열x스칼라**를 이용하는게 더 적은 메모리를 사용

```
1   a = np.array([1.0, 2.0, 3.0])
2   b = 2.0
3   a * b
```

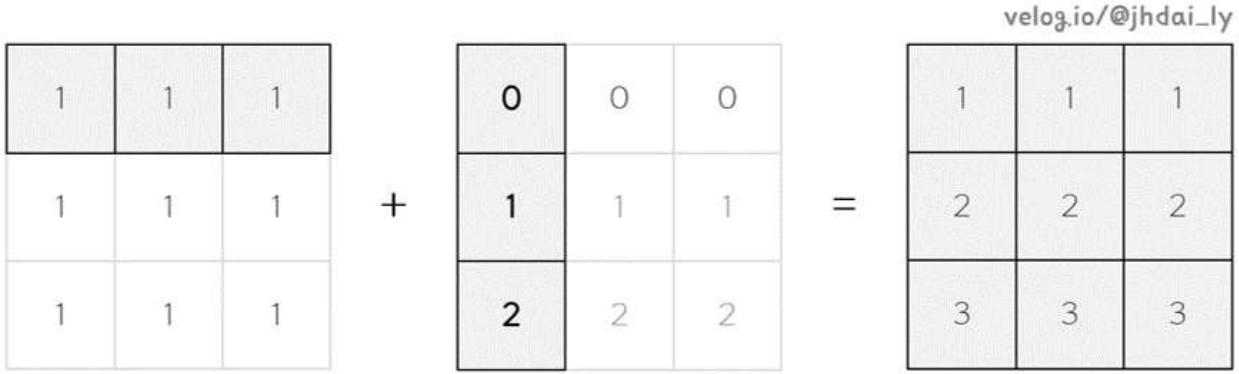
```
array([2., 4., 6.])
```

브로드캐스팅 조건

- 브로드캐스팅이 가능하기 위해서는 아래의 조건을 충족해야 한다.
    1. 차원 중 하나가 1인 경우
    2. 차원의 축의 크기가 동일한 경우

브로드캐스팅 조건

- 브로드캐스팅이 가능하기 위해서는 아래의 조건을 충족해야 한다.
    1. 차원 중 하나가 1인 경우
    2. 차원의 축의 크기가 동일한 경우



NumPy 다양한 메소드

NumPy는 다음과 같은 다양한 메소드를 지원

- 최대/최소 : min, max, argmin, argmax
- 통계 : sum, mean, median, std, var
- 불리언 : all, any

argmin/argmax

다차원 배열일 경우 배열을 1차원 배열로 축소시켜 인덱스를 반환

- argmin : 해당 배열의 제일 작은 값의 인덱스 반환
- argmax : 해당 배열의 제일 큰 값의 인덱스 반환

```
arr = np.reshape(np.random.randint(100,size=(12)),(2,3,-1))
display(arr)
print('Index : ', np.argmax(arr), ' Value : ',arr.flatten()[np.argmax(arr)])
print('Index : ', np.argmin(arr),' Value : ', arr.flatten()[np.argmin(arr)])
```

✓ 0.0s

argmin/argmax

- axis를 설정하면 해당 축의 가장 큰/작은 값의 인덱스를 반환한다.
- 다차원 배열에서의 axis에 따른 반환값은 해당 차원의 배열들을 비교

```
np.random.seed(1)
arr = np.reshape(np.random.randint(100,size=(24)),(2,3,2,-1))
print(arr.shape)
print(arr)
print('axis=0\n',np.argmax(arr,axis=0))
print('axis=1 \n',np.argmax(arr,axis=1))
print('axis=2 \n',np.argmax(arr,axis=2))
```

✓ 0.0s

정렬

- np.sort(array, axis)를 사용하면 축에 따라 배열을 정렬

```
print(np.sort(arr,axis=0))
```

✓ 0.0s

```
[[3 0 3 3]
 [5 2 4 7]
 [5 7 9 11]]
```

```
print(np.sort(arr,axis=1))
```

✓ 0.0s

```
[[0 3 5 11]
 [3 3 7 9]
 [2 4 5 7]]
```

정렬

- np.argsort(array, axis)를 사용하면 축에 따라 배열의 순서를 반환

```
print(np.argsort(arr,axis=1))
```

✓ 0.0s

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

배열값이 아닌 **인덱스** 순서

```
print(np.argsort(arr,axis=0))
```

✓ 0.0s

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