2. Numpy

Understanding of Python

1. NumPy 배열 개요

최희윤 강사

NumPy란?

Numerical Python

- ✓ 산술 계산을 위한 패키지
- ✓ 데이터 과학 및 데이터 분석에 많이 사용되는 파이썬 패키지
- ✓ NumPy 배열 객체는 데이터 교환을 위한 공통 언어처럼 사용
- ✓ Python의 연속된 자료형들보다 훨씬 더 적은 메모리 사용
- ✓ 다양한 계산을 수행하기 위해 적합

NumPy의 장점

- ✓ 파이썬 내장 모듈이 아니기 때문에 별도의 설치가 필요
- ✓ 아나콘다가 설치 돼있을 경우 NumPy 패키지 설치 없이 사용 가능

```
!conda list
```

nltk	3.6.1	pyhd3eb1b0_0
nose	1.3.7	pyhd3eb1b0_1006
notebook	6.3.0	py38haa95532_0
numba	0.53.1	py38hf11a4ad_0
numexpr	2.7.3	py38hb80d3ca_1
numpy	1.20.1	py38h34a8a5c_0
numpy-base	1.20.1	py38haf7ebc8_0
numpydoc	1.1.0	pyhd3eb1b0_1
olefile	0.46	py_0
openjpeg	2.3.0	h5ec785f_1
openpyxl	3.0.7	pyhd3eb1b0_0

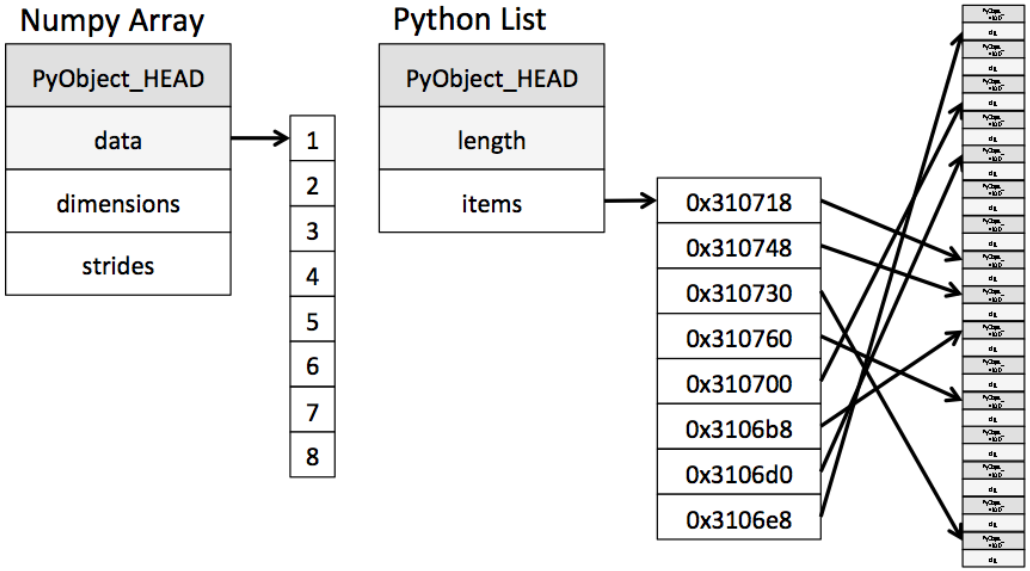
NumPy 배열

- ✓ NumPy는 **다차원의 배열 자료구조** 클래스인 **ndarray**클래스를 지원
- ✓ 벡터와 행렬을 사용하는 선형대수 계산에 주로 사용 됨

```
import numpy as np
np.eye(3)
>> array([[1., 0., 0.],
          [0., 1., 0.],
          [0., 0., 1.]])

np.eye(3, 5)
>> array([[1., 0., 0., 0., 0.],
          [0., 1., 0., 0., 0.],
          [0., 0., 1., 0., 0.]])
```

NumPy 배열

- ✓ **배열 (Array)**이란, **순서가 있는 같은 종류(type)의 데이터가 저장**되는 자료형
- ✓ 리스트에 비해서 속도가 빠르고 메모리를 더 적게 사용



NumPy 배열

✓ 리스트에 비해서 **속도가 빠르고** 메모리를 더 적게 사용

```
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 배열

- ✓ **벡터화 연산**: NumPy는 내부적으로 C로 구현 돼있어 루프 없이 배열의 모든 요소에 대해 연산을 병렬적으로 처리
- ✓ **메모리 레이아웃**: NumPy 배열은 데이터가 연속된 메모리 블록에 저장되므로 캐시 효율이 높은 반면, Python 리스트는 각 요소가 개별 메모리 위치를 참조하므로 메모리 접근 비용이 높음
- ✓ 데이터 타입 강제: NumPy 배열은 요소가 동일한 데이터 타입을 가지므로, 추가적인 타입 체크 없이 효율적인 연산이 가능

```
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 배열

✓ 행렬 합 연산 비교

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

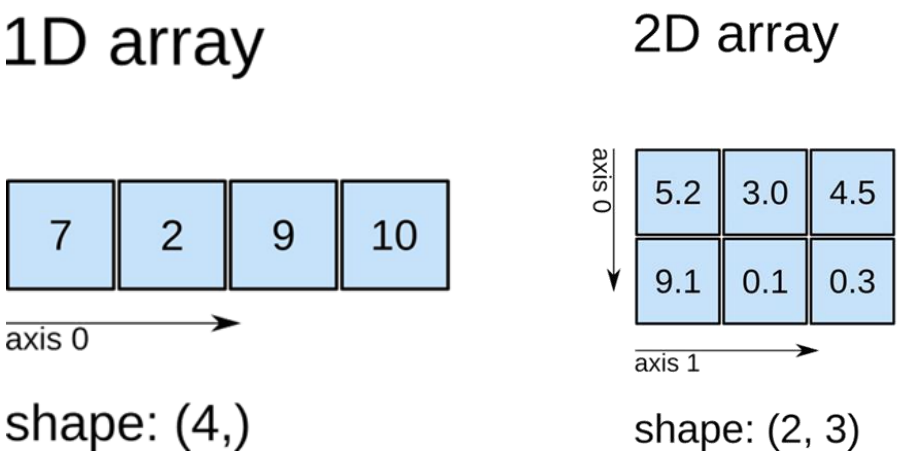
```
# list 계산
matrix_result = []

for i in range(len(matrix_1))
        tmp = []
        for j in range(len(matrix_2)):
              tmp.append(matrix_1[i][j] + matrix_22[i][j])
        matrix_result.append(tmp)

# numpy 계산
matrix_result = np.array(matrix_1) + np.array(matrix_2)
```

2. 배열생성 1

최희윤 강사

배열 만들기 ndarray

- ✓ 넘파이 배열 객체인 ndarray는 N-dimension Array의 약자
- ✓ 1차원 배열, 2차원 배열, 3차원 배열 등의 다차원 배열 자료 구조를 지원



1차원 배열 만들기 ndarray

- ✓ 넘파이의 array 메서드 **인수로 시퀀스 자료형**을 넣으면 ndarray 클래스 객체,
  즉 **넘파이 배열로 변환**해줌
- ✓ 시퀀스 자료형은 주로 튜플 보단 **리스트**를 많이 활용
- ✓ 만들어진 ndarray 객체의 표현식(representation)을 보면 바깥쪽에 array()란 것이 붙어있을 뿐 리스트와 동일한 구조처럼 보임
- ✓ 그러나 배열 객체와 리스트 객체는 많은 차이가 있음

```
import numpy as np

ar = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
ar
>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

시퀀스 자료형

ndarray 객체 반환

배열 모양 확인

- ✓ 넘파이 배열 객체(ndarray) 속성 중 shape를 사용하면 넘파이 배열의 모양을 확인할 수 있음
- ✓ 즉, 배열의 모양과 차원 확인 가능

```
import numpy as np

ar = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
ar
>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

ar.shape
>> (10, ) # 개수가 10개인 1차원 배열이 생성된 것을 확인할 수 있음
```

type 확인

✓ 파이썬의 내장함수 type()을 통해 넘파이 배열의 자료형을 확인하면 numpy.ndarray임을 확인할 수 있음

```
import numpy as np

ar = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

print(type(ar))
>> <class 'numpy.ndarray'> # 자료형(type)이 numpy.ndarray임을 알 수 있음
```

모든 요소가 같은 자료형

- ✓ 넘파이 배열 객체는 **모든 요소가 같은 자료형**이어야하고, 각 차원의 요소 개수가 동일해야 함
- ✓ 때문에, 실수와 정수가 혼합된 리스트를 활용하여 ndarray객체 생성해보면

  표현 범위가 더 큰 자료형으로 변환됨 (정수가 실수로 변환됨)

```
import numpy as np

ar2 = np.array([0.1, 5, 4, 12, 0.5])
ar2
>> [0.1, 5., 4., 12., 0.5]
```

넘파이의 자료형

✓ 넘파이 배열의 자료형 확인하는 방법: dtype 속성 활용

```
import numpy as np

ar3 = np.array([1, 2, 3, 4, 5])
print(ar3.dtype)
>> int64
```

모든 요소가 같은 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

모든 요소가 같은 자료형

✓ 무한대 표현과 정의할 수 없는 숫자 표현

- **무한대** 표현: **np.inf** (infinity)
- **정의할 수 없는 숫자**: **np.nan** (not a number)

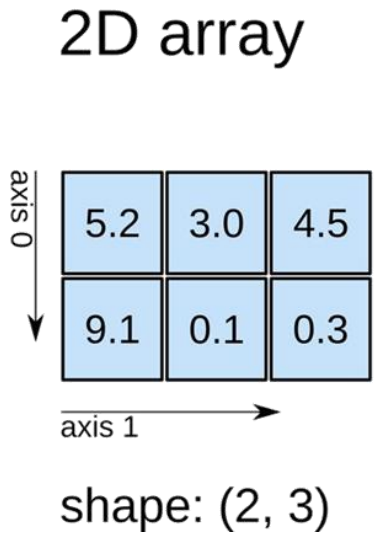
✓ 1을 0으로 나누려고 하거나 0에 대한 로그 값을 계산하면 무한대인 np.inf가 나옴

✓ 0을 0으로 나누려고 시도하면 np.nan이 나옴

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

np.log(0)
>> -inf
```

2차원 배열 만들기

- ✓ 2차원 배열은 수학에서 행렬(matrix)임
- ✓ 행렬에서 가로줄을 행(row), 세로줄을 열(column)이라고 부름



2차원 배열 만들기

- ✓ 리스트를 중첩하여 리스트의 리스트(list of list)를 이용해서 2차원 배열처럼 생성 가능
- ✓ 안쪽 리스트의 길이는 행렬의 열의 수 (즉, 가로 크기)
  바깥쪽 리스트의 길이는 행렬의 행의 수 (즉, 세로 크기)가 됨
- ✓ 예를 들어 2개의 행과 3개의 열을 갖는 2x3 배열을 만들어보자

```
c = np.array([[0, 1, 2], [3, 4, 5]]) # 2 x 3 array

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

2차원 배열 만들기

✓ 2차원 배열의 행과 열의 개수 구하는 방법: len()

```
c = np.array([[0, 1, 2], [3, 4, 5]]) # 2 x 3 array

# 행의 개수
len(c)
>> 2

# 열의 개수
len(c[0])
>> 3
```

3차원 배열 만들기

- ✓ 리스트의 리스트의 리스트를 이용하면 3차원 배열 생성 가능
- ✓ 크기를 나타낼 때는 가장 바깥쪽 리스트의 길이부터 가장 안쪽 리스트 길이의 순서로 표시
- ✓ 2 x 3 x 4 배열은 아래와 같이 만듦

```
# 2 x 3 x 4
d = np.array([[[1, 2, 3, 4],
               [5, 6, 7, 8],
               [9, 10, 11, 12]],
              [[11, 12, 13, 14],
               [15, 16, 17, 18],
               [19, 20, 21, 22]]])
```

3차원 배열 만들기

✓ 3차원 배열의 **깊이**, **행**, **열**은 아래와 같이 구할 수 있음

```
# 2 x 3 x 4
d = np.array([[[1, 2, 3, 4],
               [5, 6, 7, 8],
               [9, 10, 11, 12]],
              [[11, 12, 13, 14],
               [15, 16, 17, 18],
               [19, 20, 21, 22]]])

len(d), len(d[0]), len(d[0][0])
>> (2, 3, 4)
```

배열의 차원과 크기 알아내기

✓ 배열의 차원 및 크기를 구하는 방법: **ndim**, **shape** 속성 활용

```
# a = np.array([1, 2, 3])
print(a.ndim) # 차원
print(a.shape) # 크기
>> 1
>> (3, )

# c = np.array([[0, 1, 2], [3, 4, 5]])
print(c.ndim)
print(c.shape)
>> 2
>> (2, 3)
```

배열 생성 실습 01

✓ [문제] 넘파이를 사용하여 다음과 같은 행렬을 만들어 보세요

```
array([[10, 20, 30, 40],
       [50, 60, 70, 80]])
```

배열 생성 실습 02

✓ [문제]

- **1~18의 값을 가지는 shape이 (3, 2, 3) ndarray를 만들어 봅시다.**
- **단, 자료형은 실수로 합니다.**

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

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

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

3. 벡터화 연산

최희윤 강사

벡터화 연산 (Vectorized Operation)

✓ 넘파이 배열 객체는 배열의 각 요소에 대한 반복 연산을 하나의 명령어로 처리하는 벡터화 연산을 지원

- 아래는 리스트 내 모든 요소에 2를 곱해야 하는 경우이다.

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

answer = [2*d for d in data]

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

벡터화 연산 (Vectorized Operation)

✓ 넘파이 배열 객체는 배열의 각 요소에 대한 반복 연산을 하나의 명령어로 처리하는 벡터화 연산을 지원

- 넘파이 배열 객체의 벡터화 연산을 사용하면 반복문 없이 간단하게 단 한 번의 연산식으로 표현 가능
- 계산속도도 리스트에 비해 훨씬 빠름

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

x = np.array(data)

x*2
>> 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 대해 적용 됨

```
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])
```

벡터화 연산 실습 01

✓ [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 실습 02

✓ [문제]

- 1부터 9까지 정수 값을 갖는 1차원 ndarray를 생성하세요.
- 그리고 그 ndarray를 활용항 3부터 27까지의 3배수를 갖는 ndarray를 출력해보세요.

➢ 실행 결과:

```
array([3, 6, 9, 12, 15, 18, 21, 24, 27])
```

4. 배열의 인덱싱과 슬라이싱

최희윤 강사

배열의 인덱싱

✓ 1차원 배열 인덱싱

- 1차원 배열의 인덱싱은 리스트의 인덱싱과 동일

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

a[2]
>> 2

a[-1]
>> 4
```

배열의 인덱싱

✓ 다차원 배열의 인덱싱

- 다차원 배열일 때는 콤마(comma, 쉼표)를 사용하여 접근할 수 있음
- 콤마로 구분된 차원을 축(axis)이라고도 표현
- 그래프의 x축과 y축을 떠올리면 됨

배열 a (2x3)

1	2	3
4	5	6

a[첫 번째 차원, 두 번째 차원]

a[x축, y축]

1. a[1, 2] == 6
2. a[0, 1] == 2

배열의 인덱싱

✓ 다차원 배열(2차원)의 인덱싱

```
a = np.array([[0, 1, 2], [3, 4, 5]]) # 2차원 배열

a[0, 1] # 첫 번째 행의 두 번째 열
>> 1

a[-1, -1] # 마지막 행의 마지막 열
>> 5
```

배열의 인덱싱

✓ 다차원 배열(3차원) 인덱싱

```
a = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
              [[10, 20, 30], [40, 50, 60], [70, 80, 90]]])

a.shape
>> (2, 3, 3)

a[0, 2] # 첫 번째 행의 두 번째 열
>> array([7, 8, 9])

a[1, 2] # 마지막 행의 마지막 열
>> array([70, 80, 90])
```

배열의 슬라이싱

✓ 배열 객체로 구현한 다차원 배열의 원소 중 복수 개를 접근하려면 일반적인 파이썬 슬라이싱과 콤마를 함께 사용

배열 a (2x3)

1	2	3
4	5	6

a[행 슬라이스(또는 인덱스), 열 슬라이스(또는 인덱스)]

a[시작:끝, 시작:끝]

```
a[:1, :2] == a[0][0], a[0][1]
```

배열의 슬라이싱

✓ 2차원 배열 슬라이싱

```
a = np.array([[0, 1, 2], [3, 4, 5]])

a[0, :] # 첫 번째 행 전체 (a[0][0], a[0][1], a[0][2])
>> array([0, 1, 2])

a[:, -1] # 두 번째 열 전체 (a[0][-1], a[1][-1])
>> array([2, 5])

a[1, 1:] # 두 번째 행의 두 번째 열부터 끝까지 (a[1][1], a[1][2])
>> array([4, 5])
```

배열의 슬라이싱

✓ 3차원 배열 슬라이싱 (2,3, 3)

```
a = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
              [[10, 20, 30], [40, 50, 60], [70, 80, 90]]])

a[0:, 2] # a[0][2], a[1][2]
>> array([[7, 8, 9],
          [70, 80, 90]])

a[0:, 1:] # a[0][1], a[0][2], a[1][1], a[1][2]
>> array([[[ 4, 5, 6], [ 7, 8, 9]],
          [[40, 50, 60], [70, 80, 90]]])
```

배열의 슬라이싱

✓ 증감을 표현한 슬라이싱

```
a = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
              [[10, 20, 30], [40, 50, 60], [70, 80, 90]]])

a[0:, ::2] # a[0][0], a[0][2], a[1][0], a[1][2]
>> array([[[ 1, 2, 3], [ 7, 8, 9]],
          [[10, 20, 30], [70, 80, 90]]])
```

슬라이싱 실습 01

✓ [문제] 주어진 행렬에 대하여 각 문제를 해결해보세요.

- 값 7 인덱싱
- 값 14 인덱싱
- 배열 [6, 7] 슬라이싱
- 배열 [7, 12] 슬라이싱
- 배열 [[3, 4], [8, 9]]

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

슬라이싱 실습 02

✓ [문제] 주어진 행렬에 대하여 각 문제를 해결해보세요.

1. 오른쪽과 같은 배열을 만든 후 아래의 값을 출력하세요.

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

```
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]]])
```

슬라이싱 실습 02

✓ [문제] 주어진 행렬에 대하여 각 문제를 해결해보세요.

2. 오른쪽과 같은 배열을 만든 후 아래의 값을 출력하세요.

```
array([[1, 3],
       [100, 300]])
```

```
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]]])
```

슬라이싱 실습 02

✓ [문제] 주어진 행렬에 대하여 각 문제를 해결해보세요.

3. 오른쪽과 같은 배열을 만든 후 아래의 값을 출력하세요.

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

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

```
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]]])
```

5. 배열생성 2

최희윤 강사

배열 생성

✓ NumPy는 몇가지 단순한 배열을 생성하는 명령을 제공

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

zeros

- ✓ np.zeros((shape), dtype='type')
- ✓ shape 크기 만큼의 0의 값을 가진 배열을 생성
- ✓ dtype은 지정된 type으로 변환 (default = float)

```
c = np.zeros((5, 2), dtype='i')
c
>> array([[0, 0],
          [0, 0],
          [0, 0],
          [0, 0],
          [0, 0]], dtype=int32)
```

ones

- ✓ np.ones((shape), dtype='type')
- ✓ shape 크기 만큼의 1의 값을 가진 배열을 생성
- ✓ dtype은 지정된 type으로 변환

```
e = np.ones((2, 3, 4), dtype='i8')
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]]])
```

zeros_like, ones_like

- ✓ zeros_like(array, dtype='type'), np.ones_like(array, dtype='type')
- ✓ 주어진 array크기 만큼의 1의 값을 가진 배열 생성
- ✓ dtype은 주어진 dtype를 따름 (필요 시 변경 가능)
- ✓ 즉, 주어진 배열과 같은 shape의 같은 dtype을 가진 배열 생성

```
f = np.ones_like(a, dtype='f')
f
>> array([[1., 1., 1.],
          [1., 1., 1.]], dtype=float32)
```

배열 생성2 실습 01

✓ [문제] 빈 문자열을 갖는 shape이 (3, 3, 3) ndarray를 만들어 봅시다.

- 단, dtype은 'U10'으로 합니다.

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

배열 생성2 실습 02

✓ [문제] zeros/ones 함수를 사용해서 배열을 만들고 아래와 같은 배열을 만들어 보세요

```
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 생성2 실습 03

✓ [문제] zeros/ones 함수를 사용해서 배열을 만들고 아래와 같은 배열을 만들어 보세요

```
array([[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.]])
```

empyt

- ✓ np.empty((shape))
- ✓ 배열의 크기가 커지면 초기화 하는데 시간이 증가
- ✓ 특정한 값으로 초기화 하지 않은 empty 명령어를 사용

```
g = np.empty((4, 3))
g
>> array([[0., 0., 0.],
          [0., 0., 0.],
          [0., 0., 0.],
          [0., 0., 0.]])
```

arange

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

```
np.arange(10) # 0, …, n-1
>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.arange(3, 21, 2) # 시작, 끝(포함하지 않음), 단계
>> array([3, 5, 7, 9, 11, 13, 15, 17, 19])
```

random

- ✓ 난수: 특정 규칙이나 패턴 없이 임의로 생성된 숫자로 예측할 수 없는 값을 가짐
- ✓ np.random.rand(size): 0~1의 균일 분포에서 난수 생성
- ✓ np.random.randint(start, end, size): start ~ (end-1)의 균일 분포에서 난수 생성
- ✓ np.random.randn(size): 가우시안 분포를 따르는 난수 생성

```
print(np.random.random((3, 3)))
print(np.random.randint(1, 10, (2, 3)))
print(np.random.randn(3, 3))

# 아래는 random((3, 3))의 예시 입니다.
>> [[0.33124121 0.10197451 0.68562082]
    [0.94938095 0.96975406 0.77657014]
    [0.26792989 0.01847137 0.27188331]]
```

❖ 가우시안 분포(정규분포):
- 데이터가 평균을 중심으로 대칭적으로 분포
- np.random.randn()에서는 평균이 0, 표준편차가 1인 표준 정규분포에서 난수 생성

random

✓ 난수: 특정 규칙이나 패턴 없이 임의로 생성된 숫자로 예측할 수 없는 값을 가짐

✓ np.random.rand(size): 0~1의 균일 분포에서 난수 생성

✓ np.random.randint(start, end, size): start ~ (end-1)의 균일 분포에서 난수 생성

✓ np.random.randn(size): 가우시안 분포(정규 분포)를 따르는 난수 생성

```
print(np.random.rand(3, 3))
print(np.random.randint(1, 10, (2, 3)))
print(np.random.randn(3, 3))

# 아래는 random((3, 3))의 예시 입니다.
>> [[0.33124121 0.10197451 0.68562082]
    [0.94938095 0.96975406 0.77657014]
    [0.26792989 0.01847137 0.27188331]]
```

linspace, logspace

- ✓ np.linspace(start, stop, num)
- ✓ np.logspace(start, stop, num)
- ✓ 선형(linspace), 혹은 로그(logspace) 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])
```

linspace, logspace

- ✓ np.linspace(start, stop, num)
- ✓ np.logspace(start, stop, num)
- ✓ 선형(linspace), 혹은 로그(logspace) 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 생성2 실습 04

✓ [문제] linspace()를 활용하여 다음과 같은 1차원 ndarray를 만들어 보세요.

```
[0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

6. 배열 크기 변경

최희윤 강사

reshape

- ✓ 배열 내부 데이터를 보존한 채 형태만 변경하기 위해서 reshape를 사용
- ✓ 예제)
  - 12개의 원소를 가진 1차원 행렬이 존재한다면
  - 3x4 2차원 행렬 변경 가능
  - 2x2x3차원 행렬 변경 가능

```
a = np.arange(12)
a
>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

b = a.reshape(3, 4)
b
>> array([[0, 1, 2, 3],
          [4, 5, 6, 7],
          [8, 9, 10, 11]])
```

reshape

✓ 변경할 수 있는 크기가 정해져 있기 때문에 튜플의 값 중 하나는 -1로 대체 가능

```
a.reshape(3, -1)
>> array([[0, 1, 2, 3],
          [4, 5, 6, 7],
          [8, 9, 10, 11]])

a.reshape(2, 2, -1)
>> array([[[0, 1, 2],
           [3, 4, 5]],
          [[6, 7, 8],
           [9, 10, 11]]])
```

```
a.reshape(2, -1, 2)
>> array([[[0, 1],
           [2, 3],
           [4, 5]],
          [[6, 7],
           [8, 9],
           [10, 11]]])
```

배열크기변경 실습 01

✓ [문제]

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

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

➢ 결과

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

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

flatten, ravel

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

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

a.flatten()
>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

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

newaxis

✓ 배열의 차원을 증가시키는 경우 np.newaxis를 사용

```
a = np.arange(12)
print(a.shape, a.ndim)
>> (12, ) 1

a = a[:, np.newaxis]
print(a.shape, a.ndim)
>> (12, 1) 2
```

전치

✓ 배열의 행과 열의 축을 서로 변경

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

a.T # (3, 2)
>> array([[1, 4],
          [2, 5],
          [3, 6]])
```

배열 결합 (hstack, vstack)

✓ hstack

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[10, 20, 30], [40, 50, 60]])

c_hstack = np.hstack([a, b])
c_hstack
>> array([[1, 2, 3, 10, 20, 30],
          [4, 5, 6, 40, 50, 60]])
```

배열 결합 (hstack, vstack)

✓ vstack

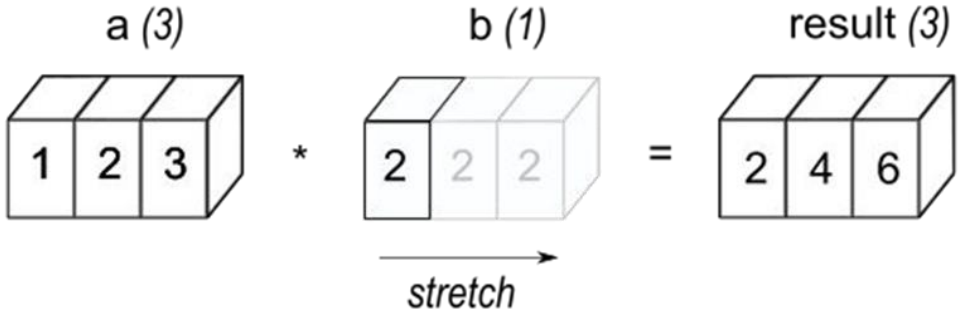
```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[10, 20, 30], [40, 50, 60]])

c_vstack = np.vstack([a, b])
c_vstack
>> array([[1, 2, 3],
          [4, 5, 6],
          [10, 20, 30],
`         [40, 50, 60]])
```

7. 브로드캐스팅

최희윤 강사

브로드캐스팅

- ✓ NumPy 배열은 모양이 다른 배열 간의 연산이 가능하도록 배열의 크기를 변환시켜주는 브로드 캐스팅(broadcasting)을 지원
- ✓ 크기가 작은 배열이 크기가 큰 배열의 크기로 전환



브로드캐스팅

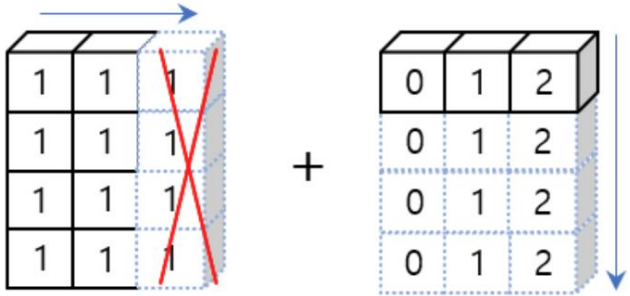
✓ 브로드캐스팅이 가능하기 때문에 배열x배열 보다 **배열x스칼라**를 이용하는 게 더 적은 메모리를 사용

```
a = np.array([1.0, 2.0, 3.0])
b = 2.0

a * b
>> array([2., 4., 6.])
```

브로드캐스팅

✓ 브로드캐스팅이 가능하기 위해서는 아래의 조건을 충족해야 함

- **차원 수**가 다를 때: 차원이 적은 배열은 차원이 큰 배열의 뒤쪽에 1로 확장하여 비교
- **크기**다 다른 차원: 크기가 1인 차원은 그 차원을 다른 배열의 크기에 맞게 확장
- 배열의 각 차원에서 크기가 같거나 하나가 1이어야 연산 가능



차원 수도 안 맞고 차원의 크기도 1이 아니어서 에러 발생

NumPy 다양한 메서드

✓ NumPy는 다음과 같은 다양한 메서드를 지원

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

NumPy 다양한 메서드 실습 01

- ✓ [문제] 1부터 100까지의 3의 배수를 갖는 1차원 ndarray를 생성하고 그 전체의 합을 구하세요
  - 답: 1683

NumPy 다양한 메서드 실습 02

✓ [문제]

- 서버 응답시간의 데이터가 다음과 같이 존재합니다.
- 해당 값의 단위는 ms 입니다.
- 이 때 200ms를 넘기는 값의 개수가 총 몇개인지 구해보세요

```
resp_time = np. array([158, 85, 205, 24, 42, 175, 188, 149, 153,  171, 137, 55, 108,
             199, 177, 209, 218, 76, 103, 189, 110, 192, 136, 119, 4, 102,
             177, 212, 110, 37, 163, 32, 77, 22, 125, 184, 46, 139, 72,
             168, 133, 209, 148, 47, 102, 65, 160, 56, 11, 96, 169, 118,
             55, 135, 217, 49, 100, 85, 179, 15, 139, 199, 124, 142, 15,
             49, 168, 171, 168, 177, 147, 23, 199, 26, 51, 42, 90, 128,
             186, 169, 129, 70, 198, 111, 81, 150, 17, 177, 201, 10, 75,
             199, 55, 151, 183, 152, 119, 193, 110, 204])
```

정렬

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

```
arr = np.reshape(np.random.randint(20, size=12), (3, -1))
arr
>> array([[ 8, 15, 13, 8],
          [11, 18, 11, 8],
          [ 7, 2, 17, 11]])

print(np.sort(arr, axis=0)) # 행 기준
>> array([[ 8, 2, 11, 8],
          [7, 15, 13, 8],
          [ 11, 18, 17, 11]])
```

정렬

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

```
arr = np.reshape(np.random.randint(20, size=12), (3, -1))
arr
>> array([[ 8, 15, 13, 8],
          [11, 18, 11, 8],
          [ 7, 2, 17, 11]])

print(np.sort(arr, axis=1)) # 열 기준
>> array([[ 8, 8, 13, 15],
          [8, 11, 11, 18],
          [ 2, 7, 11, 17]])
```