

A red square with a white border containing the text "Numpy Basic".

Numpy Basic

Numpy Basic

- Numpy 시작하기
- Shape Manipulation
- Indexing & Slicing

Numpy 시작하기

Numpy 시작하기

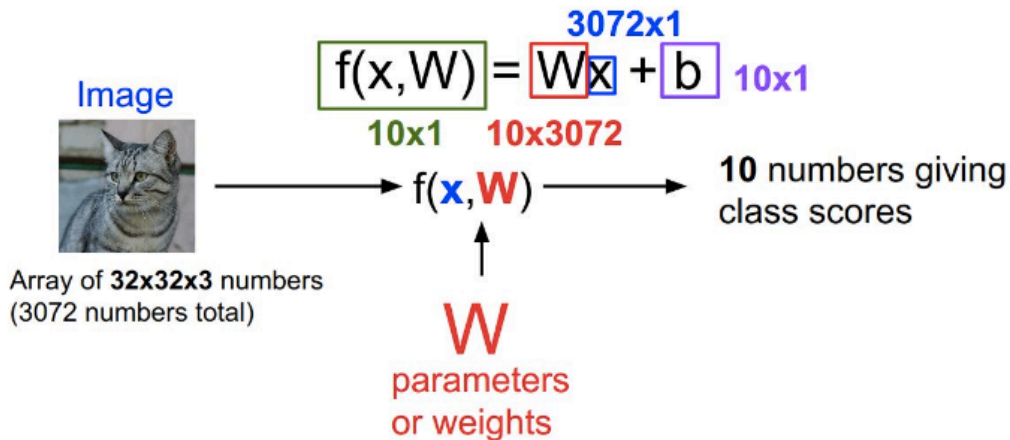
- Numpy란
- ndarray
- ndarray attributes
- ndarray 생성하기

Numpy(넘파이)

- Numpy란

- Numerical Python의 약자로 산술계산용 라이브러리

Parametric Approach: Linear Classifier



Numpy(넘파이)

- Numpy란

- Numerical Python의 약자로 산술계산용 라이브러리

- Numpy 특징

- ndarray(다차원 배열객체) → 빠르고 효율적인 메모리 사용, 유연한 브로드캐스팅
- 디스크로부터 배열 기반의 데이터를 읽거나 쓸 수 있는 도구
- C, C++, 포트란 등으로 쓰여진 코드를 통합하는 도구
- 선형대수 계산, 푸리에 변환, 난수 생성기 등

ndarray

- ndarray란
 - numpy에서 제공하는 자료구조, N차원의 배열 객체
 - 대규모의 데이터 집합을 담을 수 있는 자료구조

ndarray

- ndarray vs. list

```
[2] n = 1000000  
    numpy_arr = np.arange(n)  
    python_list = list(range(n))
```

```
[3] %%time  
    python_list = [x**3+10 for x in python_list]
```

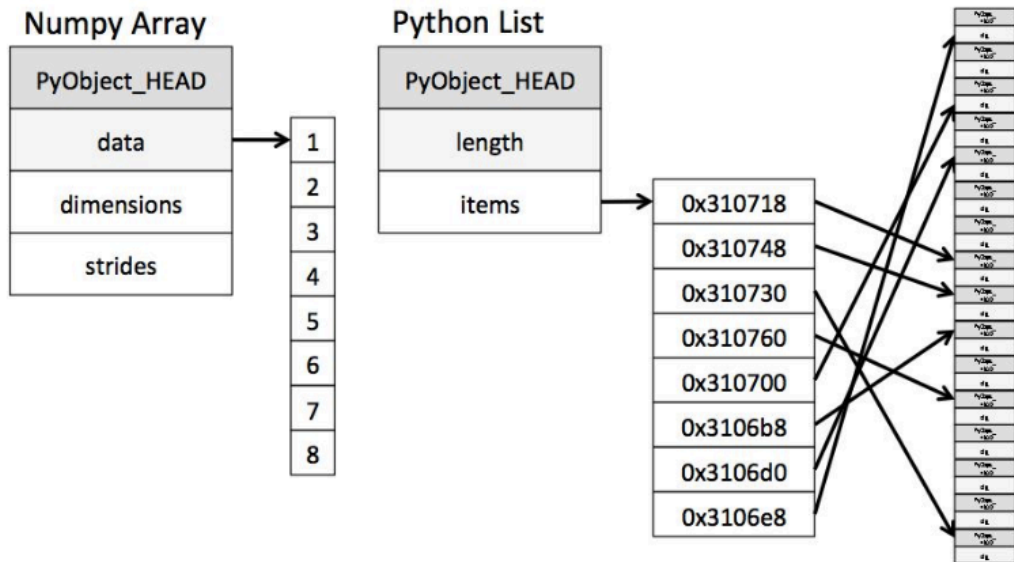
↳ CPU times: user 328 ms, sys: 52.9 ms, total: 381 ms
Wall time: 382 ms

```
▶ %%time  
    numpy_arr = numpy_arr**3+10
```

↳ CPU times: user 4.37 ms, sys: 850 µs, total: 5.22 ms
Wall time: 9.3 ms

ndarray

- ndarray vs. list
 - 연속된 메모리 블록에 데이터를 저장
 - 같은 종류의 데이터를 담음



ndarray attributes

1. dtype

- 배열에 담긴 원소의 자료형 (ndarray는 같은 자료형을 담음)

```
[7] arr = np.array([10, 20, 30, 40])  
arr
```

```
↳ array([10, 20, 30, 40])
```

```
[8] arr.dtype
```

```
↳ dtype('int64')
```

```
[10] arr2 = np.array([[0.1, 0.6], [-2, 6]])  
arr2
```

```
↳ array([[ 0.1,  0.6],  
        [-2. ,  6. ]])
```

```
▶ arr2.dtype
```

```
↳ dtype('float64')
```

- int8, float16, complex32, bool, object, string_, unicode_ 등 다양한 데이터 타입이 있음([참고](#))

ndarray attributes

1. dtype

- dtype으로 데이터 타입을 명시하지 않은 경우 자료형을 추론하여 저장
- dtype이 있어 C나 포트란 같은 저수준 언어로 작성된 코드와 쉽게 연동이 가능

```
[2] arr = np.array([10, 20, 30, 40])  
arr
```

```
↳ array([10, 20, 30, 40])
```

```
[3] arr.dtype
```

```
↳ dtype('int64')
```

```
[8] float_arr = arr.astype(np.float64)  
float_arr
```

```
↳ array([10., 20., 30., 40.])
```

```
9 float_arr.dtype
```

```
↳ dtype('float64')
```

→ astype이라는 메소드를 이용하여
dtype을 명시적으로 변환할 수 있음

ndarray attributes

2. size

- 배열에 있는 원소의 전체 갯수

arr

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

```
[12] arr.size
```

```
↳ 8
```

arr2

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

```
[16] arr2.size
```

```
↳ 12
```

ndarray attributes

3. ndim

- 배열의 차원의 갯수

arr

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

```
[14] arr.ndim
```

```
↳ 1
```

arr2

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

```
[▶] arr2.ndim
```

```
↳ 2
```

ndarray attributes

4. shape

- 배열의 각 차원의 크기
- 튜플의 형태로 리턴

arr

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

```
[14] arr.ndim
```

```
↳ 1
```

```
[13] arr.shape
```

```
↳ (8,)
```

arr2

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

```
[▶] arr2.ndim
```

```
↳ 2
```

```
[17] arr2.shape
```

```
↳ (2, 6)
```

연습문제 1

ndarray 생성하기

1. array함수

- 기존에 있던 데이터(자료형)를 이용하여 새로운 배열을 생성

np.array(데이터, dtype=)

```
[2] arr = np.array([10, 20, 30])  
arr
```

```
↳ array([10, 20, 30])
```

```
[3] arr2 = np.array([10, 20, 30], dtype=np.float16)  
arr2
```

```
↳ array([10., 20., 30.], dtype=float16)
```

→ 리스트를 이용하여 배열을 생성

ndarray 생성하기

1. array함수

- 기존에 있던 데이터(자료형)를 이용하여 새로운 배열을 생성

```
[4] arr3 = np.array(((1, 0), (0, 1)), dtype=np.float32)
arr3
```

```
↳ array([[1., 0.],
          [0., 1.]], dtype=float32)
```

→ 튜플을 이용하여 배열을 생성

```
[5] arr4 = np.array(range(20))
arr4
```

→ range 함수를 이용하여 배열을 생성

```
↳ array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
          17, 18, 19])
```

ndarray 생성하기

1. array함수

- 기존에 있던 데이터(자료형)를 이용하여 새로운 배열을 생성

```
[6] arr5 = np.array([(10, 20), (40, 50)])  
arr5
```

```
↳ array([[10, 20],  
         [40, 50]])
```

```
[7] arr6 = np.array(((1, 2), (3)))  
arr6
```

```
↳ array([(1, 2), 3], dtype=object)
```

```
▶ arr6.size
```

```
↳ 2
```

ndarray 생성하기

2. 배열 생성 함수

- numpy의 표준 배열 함수(참고)
- 자료형을 명시하지 않으면 float64

함수이름	설명
zeros	모두 0으로 초기화
ones	모두 1로 초기화
full	어떠한 값으로 모두 채워 초기화
empty	초기화되지 않은 배열을 생성
identity, eye	NxN 크기의 단위행렬
_likes	주어진 어떤 배열과 같은 shape의 배열을 생성
arange	range함수와 유사함, 범위와 stepsize
linspace	range함수와 유사함, sample 갯수

ndarray 생성하기

2. 배열 생성 함수

1) zeros

```
[16] np.zeros((5))
```

```
↳ array([0., 0., 0., 0., 0.])
```

```
[17] np.zeros((2, 4), dtype=np.int8)
```

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

→ shape을 명시

2) ones

```
[18] np.ones((3,3))
```

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

ndarray 생성하기

2. 배열 생성 함수

3) full

```
[19] np.full((4), 5)
```

```
↳ array([5, 5, 5, 5])
```

```
[▶] np.full((2, 5), -1.0)
```

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

4) empty

```
[25] np.empty((2,3), dtype=np.float64)
```

```
↳ array([[1.8155276e-316, 0.0000000e+000, 0.0000000e+000],  
        [0.0000000e+000, 0.0000000e+000, 0.0000000e+000]])
```

ndarray 생성하기

2. 배열 생성 함수

5) identity, eye

```
[22] np.identity(5, dtype=int)
```

→ N x N 정방 행렬만 생성가능

```
↳ 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]])
```

```
▶ np.eye(5, dtype=int)
```

```
↳ 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]])
```

```
▶ np.eye(5, 10, dtype=int, k=5)
```

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

→ N x M 행렬도 생성가능

ndarray 생성하기

2. 배열 생성 함수

6) _like

- zeros_like, ones_like, full_like, empty_like

```
[29] arr1 = np.array([[1, 2, 3, 1], [2, 4, 5, 6]])  
      arr1
```

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

```
▶ arr2 = np.ones_like(arr1)  
      arr2
```

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

→ shape을 명시하지 않고 기존에 존재하는 배열을 인자로 넘겨줌

ndarray 생성하기

2. 배열 생성 함수

7) arange

- 파이썬 내장함수 range와 유사한 역할, ndarray를 반환

```
[43] np.arange(5)
```

```
↳ array([0, 1, 2, 3, 4])
```

```
[44] np.arange(-3, 3)
```

```
↳ array([-3, -2, -1,  0,  1,  2])
```

```
[45] np.arange(3,50,5)
```

```
↳ array([ 3,  8, 13, 18, 23, 28, 33, 38, 43, 48])
```


→ 세번째 인자는 step size
(range와 동일)


ndarray 생성하기

2. 배열 생성 함수

8) linspace

- 범위 내에서 주어진 sample의 갯수만큼 생성

```
 np.linspace(0, 1, 6)
```

```
 array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

ndarray 생성하기

3. 난수 생성

- numpy.random 모듈을 이용하여 다양한 종류의 확률분포로부터 표본값을 생성
(참고)

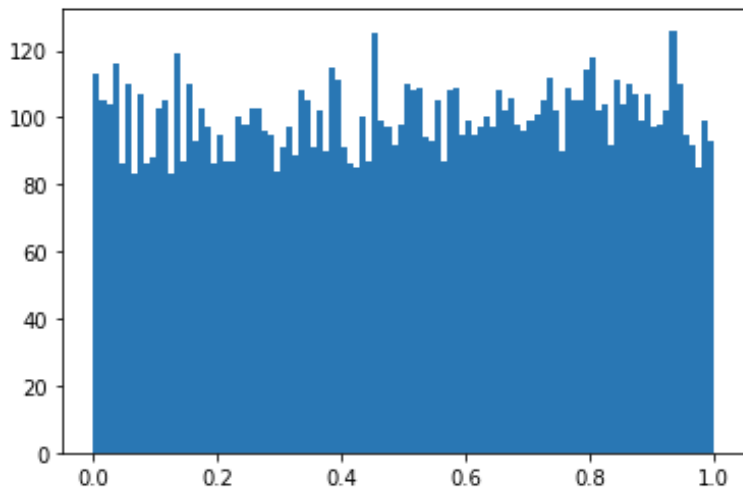
함수이름	설명
np.random.seed()	난수 생성기의 시드를 지정
np.random.rand()	[0, 1) 범위의 균등분포에서 표본을 추출
np.random.randn()	표준편차 1, 평균값 0인 정규분포에서 표본을 추출
np.random.randint()	주어진 범위 내에서 임의의 난수를 추출
np.random.permutation()	순서를 임의로 바꾸거나 임의의 순열을 반환
np.random.shuffle()	리스트나 배열의 순서를 뒤섞음


ndarray 생성하기

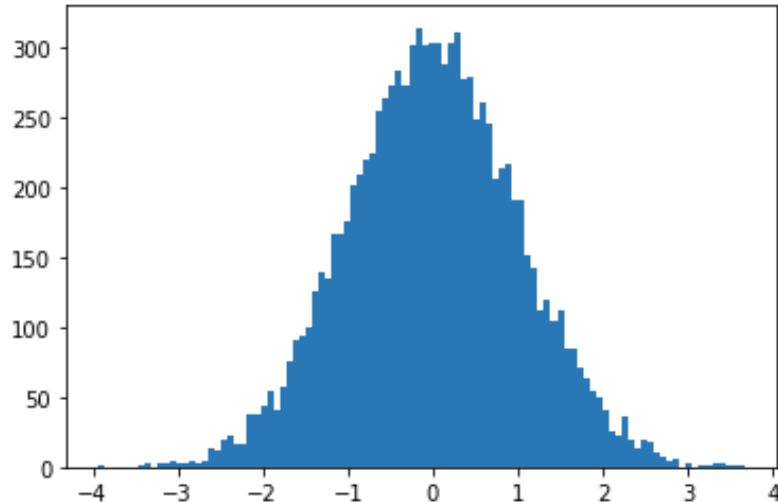
3. 난수 생성

- numpy.random 모듈을 이용하여 다양한 종류의 확률분포로부터 표본값을 생성

 `data = np.random.rand(10000)`



 `data = np.random.randn(10000)`



Numpy 시작하기

- Numpy란
- ndarray
- ndarray attributes
- ndarray 생성하기

Shape Manipulation

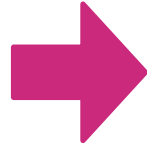
Shape Manipulation

- Flatten
- Reshape
- Transpose, T

Flatten

N dim \rightarrow 1 dim

1	2	3
4	5	6
7	8	9



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

```
[60] arr = np.zeros((3,2))  
arr
```

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



```
[61] arr.flatten()
```

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

Reshape

이미 존재하는 배열을 내가 원하는대로 shape을 조정하는 함수

`np.reshape(arr, shape)`
`arr.reshape(shape)`

```
[76] arr = np.arange(12)
      arr
```

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

```
[77] arr.reshape(3, 4)
```

주의

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

원래 주어진 shape의 약수로 이뤄진 shape만 가능

Reshape

이미 존재하는 배열을 내가 원하는대로 shape을 조정하는 함수

```
[80] arr = np.arange(20)  
      arr
```

```
↳ array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,  
         17, 18, 19])
```

```
[81] arr.reshape(-1, 10)
```

→ -1을 사용하면 shape을 명시하지 않아도 자동으로 채워줌

```
↳ array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],  
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
▶ arr.reshape(5, -1)
```

```
↳ array([[ 0,  1,  2,  3],  
        [ 4,  5,  6,  7],  
        [ 8,  9, 10, 11],  
        [12, 13, 14, 15],  
        [16, 17, 18, 19]])
```

Transpose

1	2	3
4	5	6
7	8	9

A



1	4	7
2	5	8
3	6	9

A^T

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Transpose

`np.transpose(arr, axes)`
`arr.transpose(axes)`

```
[83] arr = np.arange(20).reshape(4, 5)  
arr
```

```
↳ array([[ 0,  1,  2,  3,  4],  
         [ 5,  6,  7,  8,  9],  
         [10, 11, 12, 13, 14],  
         [15, 16, 17, 18, 19]])
```



```
[84] print(arr.transpose().shape)  
arr.transpose()
```

```
↳ (5, 4)  
array([[ 0,  5, 10, 15],  
       [ 1,  6, 11, 16],  
       [ 2,  7, 12, 17],  
       [ 3,  8, 13, 18],  
       [ 4,  9, 14, 19]])
```

Transpose

- axes를 지정하지 않으면?

$a.shape=(i[0], i[1], \dots, i[n-1]) \rightarrow a^t.shape=(i[n-1], i[n-2], \dots, i[0])$

```
[2] arr = np.arange(30).reshape(3,2,5)  
    arr.shape
```

↳ (3, 2, 5)

```
[3] print(arr.transpose().shape)
```

↳ (5, 2, 3)

Transpose

- axes는 tuple이나 list로 지정해줄 수 있음

```
[4] print(arr.shape)
     arr.transpose((1, 0, 2)).shape
```

```
↳ (3, 2, 5)
   (2, 3, 5)
```

```
▶ arr = np.arange(120).reshape(2,3,4,5)
  print(arr.shape)
  arr.transpose((1, 0, 2, 3)).shape
```

```
↳ (2, 3, 4, 5)
   (3, 2, 4, 5)
```

T

- transpose와 같은 역할을 하는 ndarray의 attribute
- 단, T의 경우 axes를 지정할 수 없음

```
[7] x = np.arange(6).reshape((-1,3))  
x
```

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

▶ x.T

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

Shape Manipulation

- Flatten
- Reshape
- Transpose, T

연습문제 2

Indexing & Scling

Indexing & Slicing

- indexing
- slicing
- boolean indexing
- fancy indexing

indexing & slicing

- 1차원

파이썬의 리스트와 유사함

```
[9] arr1d = np.arange(8)
    arr1d
```

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

```
[10] arr1d[1]
```

```
↳ 1
```

```
[11] arr1d[-1]
```

```
↳ 7
```

```
[12] arr1d[:4]
```

```
↳ array([0, 1, 2, 3])
```

indexing & slicing

- 1차원

ndarray vs. list

1) 브로드캐스팅 지원

- 브로드캐스팅이란? 다른 모양의 배열 간의 산술 연산을 수행할 수 있도록 해 주는 numpy의 기능

indexing & slicing

- 1차원

ndarray vs. list

```
[15] lst = list(range(6))  
      lst
```

```
↳ [0, 1, 2, 3, 4, 5]
```

```
[16] lst[2:5] = -1  
      lst
```

```
↳ -----  
TypeError  
<ipython-input-16-b4d2405268a4> in <modu  
----> 1 lst[2:5] = -1  
      2 lst
```

TypeError: can only assign an iterable

```
[17] lst[3] = -1  
      lst
```

```
↳ [0, 1, 2, -1, 4, 5]
```

→ 브로드캐스팅을 지원하지 않음

indexing & slicing

- 1차원

ndarray vs. list

```
[9] arr1d = np.arange(8)  
    arr1d
```

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

```
[13] arr1d[3:6] = 100
```

→ 브로드캐스팅을 지원

```
[▶] arr1d
```

```
↳ array([ 0,  1,  2, 100, 100, 100,  6,  7])
```

indexing & slicing

- 1차원

ndarray vs. list

2) 뷰

- 넘파이의 슬라이싱은 원본데이터의 **뷰**를 제공 : 데이터를 새롭게 복사해 오는 것이 아니라 기존의 데이터와 연결되어있음
- 리스트의 슬라이싱은 데이터를 복사하게 됨

indexing & slicing

- 1차원

ndarray vs. list

```
[17] arr_part = arr1d[:3]  
arr_part
```

```
↳ array([0, 1, 2])
```

→ 슬라이싱

```
[18] arr_part[1:] = -1  
arr_part
```

```
↳ array([ 0, -1, -1])
```

→ 값을 변경

```
[19] arr1d
```

```
↳ array([ 0, -1, -1, 100, 100, 100, 6, 7])
```

→ 원본데이터가 변경되어있음

indexing & slicing

- 1차원

ndarray vs. list

```
[20] lst_part = lst[2:]  
     lst_part
```

```
↳ [2, -1, 4, 5]
```

→ 슬라이싱

```
[21] lst_part[3] = 100  
     lst_part
```

```
↳ [2, -1, 4, 100]
```

→ 값을 변경

```
[▶] lst
```

```
↳ [0, 1, 2, -1, 4, 5]
```

→ 원본데이터는 그대로 유지

indexing & slicing

- 1차원

ndarray vs. list

2) 뷰

- 원본데이터를 훼손하지 않으려면?

arr.copy()

```
[27] new_arr = arr1d.copy()  
      new_arr
```

```
↳ array([ 0, -1, -1, 100, 100, 100,  6,  7])
```

indexing & slicing

- 다차원

```
[3] arr2d = np.arange(20).reshape(4, -1)  
arr2d
```

```
↳ array([[ 0,  1,  2,  3,  4],  
         [ 5,  6,  7,  8,  9],  
         [10, 11, 12, 13, 14],  
         [15, 16, 17, 18, 19]])
```

```
[4] arr2d[0]
```

```
↳ array([0, 1, 2, 3, 4])
```

```
[5] arr2d[1][2]
```

```
↳ 7
```

```
[6] arr2d[1, 2]
```

```
↳ 7
```

→ 다중리스트의 인덱싱과 유사함

→ 재귀적으로 접근

→ 콤마(,)를 이용하여 쉽게 인덱싱을 할 수도 있음

indexing & slicing

- 다차원

```
[3] arr2d = np.arange(20).reshape(4, -1)  
arr2d
```

```
↳ array([[ 0,  1,  2,  3,  4],  
         [ 5,  6,  7,  8,  9],  
         [10, 11, 12, 13, 14],  
         [15, 16, 17, 18, 19]])
```

```
[7] arr2d[:3][:2]
```

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

```
[8] arr2d[:3, :2]
```

```
↳ array([[ 0,  1],  
        [ 5,  6],  
        [10, 11]])
```

→ 재귀적으로 슬라이싱을 함

boolean indexing

- 불리언 배열 : 불리언 값으로 이루어진 배열

```
[2] arr = np.array([True, False])  
arr.dtype
```

```
↳ dtype('bool')
```

- 불리언 인덱싱 : 불리언 배열을 이용한 인덱싱
- True에 해당되는 위치에 있는 값만을 반환

```
[3] arr = np.array([0, 1, 2, 3, 4], int)  
arr[[True, False, True, False, True]]
```

```
↳ array([0, 2, 4])
```

boolean indexing

- 불리언 배열을 이용한 인덱싱

```
[4] arr = np.array([10, 20, 30, 40, 50, 60], int)
arr
```

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

```
[5] arr % 3 == 0
```

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

```
▶ arr[arr%3==0]
```

```
↳ array([30, 60])
```

→ 조건문을 통해 불리언 배열을 만들 수 있음

→ 이를 배열에 인덱싱으로 주면 해당 위치에 값을 반환

- 불리언 인덱싱 후 해당 값을 다시 인덱싱/슬라이싱도 할 수 있음

fancy indexing

정수 배열을 사용한 인덱싱

```
[5] arr = np.arange(10, 20)  
arr
```

```
↳ array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

```
[6] arr[[0, 2, 4, 6]]
```

```
↳ array([10, 12, 14, 16])
```

```
▶ arr[[3, 0, 1]]
```

```
↳ array([13, 10, 11])
```

→ 정수가 담긴 ndarray나 리스트로 특정 위치에 있는 값을 가져올 수 있음

→ 주어지는 순서대로 값을 가져오게됨

fancy indexing

정수 배열을 사용한 인덱싱

```
[7] arr2d = np.arange(20).reshape(4, 5)  
arr2d
```

```
↳ array([[ 0,  1,  2,  3,  4],  
         [ 5,  6,  7,  8,  9],  
         [10, 11, 12, 13, 14],  
         [15, 16, 17, 18, 19]])
```

```
[8] arr2d[[0, 2]]
```

```
↳ array([[ 0,  1,  2,  3,  4],  
         [10, 11, 12, 13, 14]])
```

```
[9] arr2d[[0, 1], [4]]
```

```
↳ array([4, 9])
```


fancy indexing

정수 배열을 사용한 인덱싱

```
[7] arr2d = np.arange(20).reshape(4, 5)  
arr2d
```

```
↳ array([[ 0,  1,  2,  3,  4],  
         [ 5,  6,  7,  8,  9],  
         [10, 11, 12, 13, 14],  
         [15, 16, 17, 18, 19]])
```

```
[10] arr2d[[0, 1], [4, 3]]
```

```
↳ array([4, 8])
```

```
▶ arr2d[[0, 1, 2], [4, 3, 1]]
```

```
↳ array([ 4,  8, 11])
```

→ (0, 4), (1, 3)에 대응되는 원소들을 가져옴

fancy indexing

정수 배열을 사용한 인덱싱

```
[7] arr2d = np.arange(20).reshape(4, 5)  
arr2d
```

```
↳ array([[ 0,  1,  2,  3,  4],  
         [ 5,  6,  7,  8,  9],  
         [10, 11, 12, 13, 14],  
         [15, 16, 17, 18, 19]])
```

```
▶ arr2d[[0, 1, 2]][:, [4, 3, 1]]
```

```
↳ array([[ 4,  3,  1],  
         [ 9,  8,  6],  
         [14, 13, 11]])
```

연습문제3

Indexing & Slicing

- indexing
- slicing
- boolean indexing
- fancy indexing