CH. 6. Numpy와 Matplotlib

Section 01 Numpy

❑ 이미지와 Numpy

- OpenCV**에서 이미지나 동영상을 읽어들이는 함수** cv2.imread() **는** NumPy **배열을 반환한다**.
- **따라서** OpenCV**를 파이썬 언어로 프로그래밍한다는 것은** NumPy **배열을 다룬다는 것과 같은 말이다**.
- **그만큼** NumPy**에 대해 잘 알지 못하면 파이썬언어로** OpenCV**를 사용하는 게 불가능하다는 뜻이다**.
- NumPy **배열에서 정보를 얻는 기본 속성은 다음과 같다**.
  - ndim : **차원(축)의 수**
  - shape : **각차원의 크기(튜플)**
  - size : **전체 요소의 개수**, shape**의 각 항목의 곱**
  - dtype : **요소의 데이터 타입**
  - itemsize : **각 요소의 바이트 크기**

Section 01 Numpy

❑ 이미지와 Numpy

- 이미지는 여러 개의 픽셀들의 값으로 구성되므로 수많은 행과 열로 구성된 픽셀 데이터의 모음이라고 볼 수 있다.
- 이 픽셀 데이터들을 프로그래밍 영역에서 다루려면 픽셀 값들을 저장하고 관리할 적절한 자료구조가 필요하기 마련이다.
- OpenCV-Python은 예전에는 독자적인 자료구조를 사용했지만, 버전 2부터 NumPy 라이브러리의 ndarray(N-Dimensional Array)를 가져다 쓰고 있다.
- 다음 코드는 파이썬 대화형 콘솔에서 OpenCV로 읽어 들인 500 x 500 픽셀 이미지 정보를 담은 NumPy 배열의 속성 정보를 출력한다.

```
>>> import cv2
>>> img = cv2.imread('./img/blank_500.jpg')
>>> type(img)
<class 'numpy.ndarray'>
```

Section 01 Numpy

❑ 이미지와 Numpy

- ndarray는 N-Dimensional Array의 약자로 N차원 배열, 즉 다차원 배열을 의미한다.
- OpenCV는 기본적으로 이미지를 3차원 배열, 즉, '행 × 열 × 채널'로 표현한다.
- 행과 열은 이미지의 크기, 즉 높이와 폭만큼의 길이를 갖고 채널은 컬러인 경우 파랑, 초록, 빨강 3개의 길이를 갖는다.
- 따라서 일반적인 이미지를 읽었을 때 3차원 배열은 '높이 × 폭 x 3'의 형태이다.

```
>>> img.ndim
3

>>> img.shape
(500, 500, 3)

>>> img.size
750000
```

Section 01 Numpy

❑ 이미지와 Numpy

- 파이썬 언어는 데이터 타입이나 데이터 크기를 따로 지정하지 않지만, 수많은 데이터를 처리해야 하는 NumPy 배열은 명시적인 데이터 타입을 지정하는 것이 효과적일 수밖에 없다.
- 이미지 픽셀 데이터는 음수이거나 소수점을 갖는 경우가 없고 값의 크기도 최대 255 이므로 부호없는 8비트, 그러니까 uint8을 데이터 타입으로 사용한다.
- 다음과 같이 dtype 속성으로 데이터 타입을 확인할 수 있다.
- img.itemsize 결과 값은 각 요소의 크기가 1 바이트인 것을 나타낸다.

```
>>> img.dtype
dtype('uint8')

>>> img.itemsize
1
```

Section 01 Numpy

❑ Numpy 배열 생성

- NumPy **배열을 만드는 방법은 값을 가지고 생성하는 방법과 크기만 지정해서 생성하는 방법으로 나눌 수 있다.**
- **크기만 지정해서 생성하는 방법은 다시 특정한 초기 값을 모든 요소에 지정하는 경우와 값의 범위를 지정 해서 순차적으로 증가 또는 감소하는 값을 갖게 하는 방법으로 나눌 수 있다.**
- **다음 목록은** NumPy **배열 생성에 사용할 함수들이다.**
  - **값으로 생성** array( )
  - **초기 값으로 생성** : empty(), zeros(), ones(), full()
  - **기존 배열로 생성** : empty_like(), zeros_like(), ones_like(), full_like()
  - **순차적인 값으로 생성** : arange()
  - **난수로 생성** : random.rand(), random.randn()

Section 01 Numpy

❑ 값으로 생성

- **배열 생성에 사용할 값을 가지고 있는 경우에는** numpy.array() **함수로 간단히 생성할 수 있다**.
- numpy.array(list [, dtype]) : **지정한 값들로** NumPy **배열 생성**
- list: **배열 생성에 사용할 값을 갖는 파이썬 리스트 객체**
- dtype: **데이터 타입**(**생략하면 값에 의해 자동 결정**)
  - int8, int16, int32, int64 : **부호 있는 정수**
  - uint8, uint16, uint32, uint64 : **부호 없는 정수**
  - float16, float32, float64, float128 : **부동 소수점을 갖는 실수**
  - complex64, complex128, complex256 : **부동 소수점을 갖는 복소수**
  - bool : **불**(boolean)

Section 01 Numpy

❑ 값으로 생성

- NumPy 모듈을 사용하기 위해서는 numpy 모듈을 임포트해야 한다.
- 이제 numpy.array() 함수로 파이썬 리스트에 값을 지정해서 생성하는 코드를 작성해보자.
- 다음 코드에서 배열을 생성할 때 dtype을 따로 지정하지 않았지만, 리스트 항목이 모두 정수라서 알아서 int32가 된 것을 볼 수 있다.

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

>>> a.dtype
dtype('int32')

>>> a.shape
(4,)
```

Section 01 Numpy

❑ 값으로 생성

- 이번엔 배열을 만들 때 2차원 리스트를 전달했다.
- shape 속성을 보니까 (2 ,4), 즉 2행 4 열인 것을 알 수 있다.

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

>>> b.shape
(2, 4)
```

- 다음 코드에서는 정수와 소수점이 있는 실수를 섞어서 배열을 만들었더니 dtype은 float64로 알아서 정해졌다.

```
>>> c = np.array([1, 2, 3.14, 4])
>>> c
array([1.  , 2.  , 3.14, 4.  ])

>>> c.dtype
dtype('float64')
```

Section 01 Numpy

❑ 값으로 생성

- 앞의 코드는 정수만을 가지고 배열을 만들었지만 명시적으로 dtype=np.float32로 지정해서 데이터 타입이 float32이다.

```
>>> d = np.array([1, 2, 3, 4], dtype=np.float32)
>>> d
array([1., 2., 3., 4.], dtype=float32)
```

- 명시적인 dtype을 지정해서 생성해야 할 때를 위해서 NumPy 생성함수는 dtype과 동일한 이름의 함수를 제공한다.
- 예를 들어 dtype=np.uint8로 생성하려고 한다면 아래와 같이 할 수 있다.

```
>>> a = np.uint8([1, 2, 3, 4])
```

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- NumPy **배열을 생성할 때 더 많이 사용하는 방법은 배열의 차수와 크기 그리고 초기 값을 지정해서 생성하는 방법이다.**
- **이 때 쓸 수 있는 함수는 초기 값을 지정하는 방법에 따라 여러 가지가 있는데, 튜플로 차수와 크기를 지정하는 방법은 모두 같다.**
  - numpy.empty(shape [, dtype]) : **초기화되지 않은 값(쓰레기 값)으로 배열 생성**
    - shape : **튜플, 배열의 각 차수의 크기 지정**
  - numpy.zeros(shape [, dtype]) : 0(**영**, zero)**으로 초기화된 배열 생성**
  - numpy.ones (shape [, dtype]): 1**로 초기화된 배열 생성**
  - numpy.full(shape, fill_value [, dtype]) : fill_value**로 초기화된 배열 생성**

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- 아래 코드는 numpy.empty() 함수의 사용 사례를 보여주고 있다.

```
>>> a = np.empty([2, 3])
>>> a
array([[7.614445e-317, 0.000000e+000, 0.000000e+000],
       [0.000000e+000, 0.000000e+000, 0.000000e+000]])

>>> a.dtype
dtype('float64')
```

- 위의 코드에서 보는 것처럼 2 행 3 열 배열이 만들어졌지만 초기 값은 제각각이고, dtype은 float64가 기본 값으로 사용된 것을 알 수 있다.
- 쓰레기 값을 갖는 배열이므로 어떤 값으로 초기화를 하고 싶으면 fill() 함수를 사용하는 것이 좋다.
  - ndarray.fill(value) : 배열의 모든 요소를 value로 채움

```
>>> a.fill(255)
>>> a
array([[255., 255., 255.],
       [255., 255., 255.]])
```

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- 이렇게 배열을 만들고 어떤 특정한 값으로 모든 요소를 초기화하는 일이 많을텐데, 이런 작업을 한번에 해 주는 함수가 zeros(), ones(), full() 이다.
- 함수의 이름만 봐도 알 수 있듯이 초기화해 주는 값만 서로 다르다.

```
>>> b = np.zeros((2, 3))
>>> b
array([[0., 0., 0.],
       [0., 0., 0.]])

>>> b.dtype
dtype('float64')
```

- zeros() 함수로 생성한 2열 3행 배열은 모두 0으로 채워져 있지만, dtype을 따로 지정하지 않아서 역시 float64로 생성된 것을 알 수 있다.

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- 다음 코드처럼 생성할 때 원하는 dtype을 명시적으로 지정할 수 있다.

```
>>> c = np.zeros((2, 3), dtype=np.int8)
>>> c
array([[0, 0, 0],
       [0, 0, 0]], dtype=int8)
```

- 다음 코드는 1로 채워진 배열을 만든다.

```
>>> d = np.ones((2, 3), dtype=np.int16)
>>> d
array([[1, 1, 1],
       [1, 1, 1]], dtype=int16)
```

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- 0(영) 이나 1이 아닌 다른 원하는 초기 값을 지정하고 싶을 때는 앞의 코드처럼 full() 함수를 쓸 수 있다.
- 다음 코드는 2 x 3 x 4 배열을 255로 초기화하는 코드이다.

```
>>> e = np.full((2, 3, 4), 255, dtype=np.uint8)
>>> e
array([[[255, 255, 255, 255],
        [255, 255, 255, 255],
        [255, 255, 255, 255]],

       [[255, 255, 255, 255],
        [255, 255, 255, 255],
        [255, 255, 255, 255]]], dtype=uint8)
```

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- **새로운 배열을 생성할 때 기존에 있던 배열과 같은 크기의 배열을 만들어야 할 때도 있는데, 그때 사용하는 함수는 다음과 같다.**
  - empty_like(array [, dtype]) : **초기화되지 않은**, array**와 같은** shape**와** dtype**의 배열 생성**
  - zeros_like(array [, dtype]) : 0(**영**, zero)**으로 초기화된**, array**와 같은** shape**와** dtype**의 배열 생성**
  - ones_like(array [, dtype]) : 1**로 초기화된**, array**와 같은** shape**와** dtype**의 배열 생성**
  - full_like(array, fill_value [, dtype]) : fill_value**로 초기화된**, array**와 같은** shape**와** dtype**의 배열 생성**

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- 다음 코드는 사진을 배열로 읽어들인다.

```
>>> img = cv2.imread('./img/lena.jpg')
>>> img
array([[[ 14,  45,  84],
        [ 13,  44,  83],
        [ 13,  43,  84],
        ...,
        [ 89, 134, 185],
        [ 89, 134, 185],
        [ 89, 134, 185]],

       [[ 15,  31,  44],
        [ 13,  29,  42],
        [ 10,  26,  42],
        ...,
        [ 10,  26,  49],
        [  9,  25,  48],
        [  9,  25,  48]]], dtype=uint8)

>>> img.shape
(822, 1200, 3)
```

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- 이 이미지와 동일한 크기의 배열을 생성하는 코드를 작성하면 다음과 같다.

```
>>> a = np.empty_like(img)
>>> b = np.zeros_like(img)
>>> c = np.ones_like(img)
>>> d = np.full_like(img, 255)
>>> a
array([[[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]], dtype=uint8)
>>> a.shape
(822, 1200, 3)
```

```
>>> b
array([[[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]], dtype=uint8)
>>> b.shape
(822, 1200, 3)
```

Section 01 Numpy

❑ 크기와 초기 값으로 생성

- 이 이미지와 동일한 크기의 배열을 생성하는 코드를 작성하면 다음과 같다.

```
>>> c
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],
        [1, 1, 1],
        ...,
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]]], dtype=uint8)
```

```
>>> c.shape
(822, 1200, 3)
```

```
>>> d
array([[[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]]], dtype=uint8)
```

```
>>> d.shape
(822, 1200, 3)
```

Section 01 Numpy

❑ 시퀀스와 난수로 생성

- NumPy **배열을 생성하는 방법 중에는 일정한 범위 내에서 순차적인 값을 갖게 하는 방법과 난수로 채우는 방법이 있다**.
- numpy.arange([start=0, 1 stop [, step=1, dtype=float64]) : **순차적인 값으로 생성**
  - start : **시작값**
  - stop : **종료 값**, **범위에 포함하는 수는** stop - 1**까지**
  - step : **증가값**
  - dtype : **데이터 타입**
- numpy.random.rand([d0 [, d1 [... , dn]]]) : 0**과** 1 **사이의 무작위 수로 생성**
  - d0, d1..dn : shape, **생략하면 난수 한 개 반환**
- numpy.random.randn([d0 [, d1 [... , dn]]]) **표준정규분포**(**평균**: 0, **분산**: 1)**를 따르는 무작위 수로 생성**

Section 01 Numpy

❑ 시퀀스와 난수로 생성

- 숫자 하나를 인자로 전달하면 0부터 시작하는 값을 순차적으로 갖는 차원 배열을 반환한다.
- 이때 dtype은 요소들이 정수만으로 이루어 졌으므로 int32를 갖는다.

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

>>> a.dtype
dtype('int32')

>>> a.shape
(5,)
```

Section 01 Numpy

❑ 시퀀스와 난수로 생성

- 만약 아래의 코드처럼 인자로 소수점이 있는 수를 사용하면 dtype은 float64로 지정되며, 필요에 따라 명시적으로 지정할 수도 있다.
- 이 함수는 시작 값과 종료 값 그리고 증가 값을 모두 지정해서 생성할 수도 있다.
- 반드시 기억해야 할 점은 지정한 범위의 마지막 값은 항목에 포함되지 않는다는 것이다.

```
>>> b = np.arange(5.0)
>>> b
array([0., 1., 2., 3., 4.])

>>> b.dtype
dtype('float64')
```

Section 01 Numpy

❑ 시퀀스와 난수로 생성

- 다음 코드는 3에서 시작해서 9의 바로 앞 그러니까 8까지 2씩 증가하는 수를 갖는 배열을 생성한다.

```
>>> c = np.arange(3, 9, 2)
>>> c
array([3, 5, 7])
```

- arange( ) 함수는 차원 배열만을 생성할 수 있으므로 다차원 배열, 특히 이미지 데이터를 갖는 3차원 배열로 만들기 위해서는 '차원 변경' 함수와 함께 써야하는 경우가 많다.

Section 01 Numpy

❑ 시퀀스와 난수로 생성

- 난수를 발생하는 함수로는 random.rand() 와 random.randn()이 있다.
- rand() 함수는 0과 1사이의 값을 무작위로 만들고, randn() 함수는 평균이 0이고 분산이 1인 정규분포를 따르는 무작위 수를 만들어 낸다.
- 두 함수 모두 인자없이 호출하면 난수 1개를 반환하고, 원하는 차수(shape)에 맞게 인자를 전달하면 해당 차수에 맞는 배열을 난수로 채워서 반환한다.

Section 01 Numpy

❑ 시퀀스와 난수로 생성

- 다음 코드의 a와 b 배열은 난수로 채워진 2행 3열의 배열이다.
- 이 함수들은 결과 값이 소수점을 갖는 데다가 특정 범위 내에서 난수를 추출하므로 이미지 작업에 필요한 원하는 범위 내에서 난수를 발생하기 위해서는 뒤에서 설명하는 브로드캐스팅 연산과 dtype 변경이 필요할 때가 많다.

```
>>> np.random.rand()
0.1111738423194517

>>> np.random.randn()
-0.6296419669076598

>>> a = np.random.rand(2, 3)
>>> a
array([[0.9176118 , 0.16488181, 0.26636115],
       [0.54629538, 0.05092636, 0.83685965]])

>>> b = np.random.randn(2, 3)
>>> b
array([[ 0.26144281, -1.14066652,  0.52960206],
       [-0.5531953 , -0.32104923, -0.36007499]])
```

Section 01 Numpy

❑ dtype 변경

- **배열의 데이터 타입을 변경하는 함수 다음과 같다.**
- ndarray.astype(dtype)
  - dtype : **변경하고 싶은** dtype, **문자열 또는** dtype
- numpy.uintXX(array) : array**를 부호 없는 정수**(uint) **타입으로 변경해서 반환**
  - uintXX : uint8, unü16, uint32, uint64
- numpy.intXX(array) : array**를** int **타입으로 변경해서 반환**
  - intXX: int8, int16, int32, int64
- numpy.floatXX(array) : array**를** float **타입으로 변경해서 반환**
  - floatXX : float16, float32, float64, float128
- numpy.complexXX(array) : array**를 복소수**(complex) **타입으로 변경해서 반환**
  - complexXX : complex64, complex128, complex256

Section 01 Numpy

❑ dtype 변경

- 다음 코드는 처음 int32 타입으로 생성한 배열을 float32로 변경하는 모습을 보여주고 있다.
- 이때 astype('float32') 함수에 전달한 인자는 문자열을 사용하고 있는데, 이것은 앞서 설명한 dtype 이름을 그대로 문자열로 작성하면 된다.
- 문자열에 오타가 있거나 일치하지 않는 경우 오류가 발생하니 주의해야 한다.

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

>>> a.dtype
dtype('int32')

>>> b = a.astype('float32')
>>> b
array([0., 1., 2., 3., 4.], dtype=float32)
```

Section 01 Numpy

❑ dtype 변경

- **아래 코드처럼** astype(np.float64) **함수에** NumPy **모듈에 선언된 변수를 사용하는 방법도 있다**.

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

>>> c = a.astype(np.float64)
>>> c
array([0., 1., 2., 3., 4.])

>>> c.dtype
dtype('float64')
```

Section 01 Numpy

❑ dtype 변경

- 배열 객체의 dtype을 변경하는 방법으로는 배열 객체의 astype() 메서드를 호출하는 방법이 있는 반면 또 다른 방법도 있다.
- NumPy 모듈 정적 함수에는 NumPy에서 지원하는 dtype들과 같은 이름의 함수들이 있는데, 이 함수들 중에 변경을 원하는 dtype 이름의 함수를 호출하면서 배열 객체를 인자로 전달하는 방법도 있다.
- 아래 코드는 원래 데이터 타입이 int32 였던 배열을 np.uint8() 함수에 전달했더니 uint8로 변경돼서 반환되는 것을 보여준다.

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

>>> d = np.uint8(a)
>>> d
array([0, 1, 2, 3, 4], dtype=uint8)
```

Section 01 Numpy

❑ 차원 변경

- **원래는 1차원이던 배열을 2행 3열 배열로 바꾼다든지**, 100 x 200 x 3**인 배열을 1차원으로 바꾸는 식의 작업이 필요할 때가 많은데, 이때 필요한 함수는 다음과 같다.**
- ndarray.reshape(newshape) : ndarray **의** shape**를** newshape**로 차원 변경**
- numpy.reshape(ndarray, newshape) : ndarray **의** shape**를** newshape**로 차원 변경**
  - ndarray : **원본 배열 객체**
  - newshape : **변경하고자 하는 새로운** shape(**튜플**)
- numpy.ravel (ndarray) : **차원 배열로 차원 변경**
  - ndarray : **변경할 원본 배열**
- ndarray.T : **전치 배열** (transpose)

Section 01 Numpy

❑ 차원 변경

- numpy.arange() 함수는 차원 배열만을 생성할 수 있으므로 원하는 차원의 모양으로 변경하는 함수가 거의 대부분 필요하다.
- 모양을 변경하는 함수는 원본 배열의 메서드로 호출하거나 NumPy 모듈에 있는 정적함수에 배열을 인자로 전달해서 호출한다.
- 다음 코드는 1차원 배열 a를 2행 3열로 바꾸는 작업을 두 가지 함수로 각각 보여주고 있다.

```
>>> a = np.arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])

>>> b = a.reshape(2, 3)
>>> b
array([[0, 1, 2],
       [3, 4, 5]])

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

Section 01 Numpy

❑ 차원 변경

- 다음 코드는 100개의 1차원 배열의 차원을 변경하기 위해 지정한 shape에 -1을 쓴 예를 보여주고 있다.
- (2, -1)의 의미는 2행으로 나누고 열은 알아서 맞추라는 뜻이다.
- 결국 100개의 요소를 2행으로 나누어 열이 50개가 되므로 2행 50 열이 된다.

```
>>> d = np.arange(100).reshape(2, -1)
>>> d
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, 24, 25, 26, 27, 28, 29, 30, 31,
        32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
        48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65,
        66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
        82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
        98, 99]])

>>> d.shape
(2, 50)
```

Section 01 Numpy

❑ 차원 변경

- (-1, 5)는 -1행 5열을 생성하겠다는 것인데, 5열에 맞춰서 행을 알아서 계산하면 20행이 나오므로 (20, 5) 가 출력된다.

```
>>> e = np.arange(100).reshape(-1, 5)
>>> e
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, 24],
       [25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34],
       [35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44],
       [45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54],
       [55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64],
       [65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74],
       [75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84],
       [85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94],
       [95, 96, 97, 98, 99]])
```

```
>>> e.shape
(20, 5)
```

Section 01 Numpy

❑ 차원 변경

- 다음 코드는 2행 3열 배열을 차원 배열로 바꾸는 방법을 보여주고 있다.
- 첫 번째 방법은 f. reshape((6,)) 과 같이 요소의 수를 직접 전달하는 방법이며, 일일이 계산해야 하니 불편하다.
- 그래서 조금 전 설명한 -1 을 사용하면 간단히 해결할 수 있다.
- 마지막으로 np.ravel() 함수로도 똑같은 결과를 얻을 수 있다.

```
>>> f = np.zeros((2, 3))
>>> f
array([[0., 0., 0.],
       [0., 0., 0.]])

>>> f.reshape((6,))
array([0., 0., 0., 0., 0., 0.])

>>> f.reshape(-1)
array([0., 0., 0., 0., 0., 0.])

>>> np.ravel(f)
array([0., 0., 0., 0., 0., 0.])
```

Section 01 Numpy

❑ 차원 변경

- NumPy **배열** (ndarray) **객체에는** ndarray.T **는 속성이 있다**.
- **이 속성을 이용하면 행과 열을 서로 바꾸는 전치 배열을 얻을 수 있다**.

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

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

Section 01 Numpy

❑ 브로드캐스팅 연산

- 다음 코드의 예처럼 0부터 9까지 있는 파이썬 리스트의 모든 항목 값을 1씩 증가시키려면 반복문을 작성해야 한다.

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

>>> for i in range(len(mylist)):
        mylist[i] = mylist[i] + 1


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

- 하지만, NumPy 배열에 +1(더하기 1) 연산을 한 번만 해도 같은 결과를 얻게 되는데, 이것을 브로드캐스팅 연산이라고 한다.

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

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

>>> a + 1
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Section 01 Numpy

❑ 브로드캐스팅 연산

- **아래 코드는** NumPy **배열과 스칼라**(Scalar, **스케일러**) **값 간의 여러 가지 연산의 예를 보여주고 있다**.

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

>>> a + 5
array([5, 6, 7, 8, 9])

>>> a - 2
array([-2, -1,  0,  1,  2])

>>> a * 2
array([0, 2, 4, 6, 8])

>>> a / 2
array([0. , 0.5, 1. , 1.5, 2. ])

>>> a ** 2
array([ 0,  1,  4,  9, 16], dtype=int32)

>>> b = np.arange(6).reshape(2, -1)
>>> b
array([[0, 1, 2],
       [3, 4, 5]])

>>> b * 2
array([[ 0,  2,  4],
       [ 6,  8, 10]])
```

Section 01 Numpy

❑ 브로드캐스팅 연산

- 산술 연산뿐만 아니라 비교 연산도 가능하다.
- 비교 연산의 결과는 각 항목에 대해 만족 여부를 불 값(True/False)으로 갖는 동일한 크기의 배열로 반환한다.

```
>>> a
array([0, 1, 2, 3, 4])
>>> a > 2
array([False, False, False,  True,  True])
```

Section 01 Numpy

❑ 브로드캐스팅 연산

- 배열과 숫자 값간의 연산뿐만 아니라 배열끼리의 연산도 가능하다.

```
>>> a = np.arange(10, 60, 10)
>>> b = np.arange(1, 6)
>>> a
array([10, 20, 30, 40, 50])

>>> b
array([1, 2, 3, 4, 5])

>>> a + b
array([11, 22, 33, 44, 55])

>>> a - b
array([ 9, 18, 27, 36, 45])

>>> a * b
array([ 10,  40,  90, 160, 250])

>>> a / b
array([10., 10., 10., 10., 10.])

>>> a ** b
array([       10,       400,     27000,   2560000, 312500000], dtype=int32)
```

Section 01 Numpy

❑ 브로드캐스팅 연산

- 하지만, 배열 간의 연산에는 약간의 제약이 있다.
- 두 배열의 shape가 완전히 동일하거나 둘중 하나가 1차원이면서 열의 축의 길이가 같아야 한다.
- 다음 코드는 두 배열의 shape가 일치하지 않아서 연산에 실패한다.

```
>>> a = np.ones((2, 3))
>>> b = np.ones((3, 2))
>>> a + b
  Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
  ValueError: operands could not be broadcast together with shapes (2,3) (3,2)
```

Section 01 Numpy

❑ 브로드캐스팅 연산

- 다음 코드는 배열 c가 1차원이고 1차원 배열의 열의 개수가 a 배열의 열의 개수와 같아서 연산이 가능하다.

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

>>> c = np.arange(3)
>>> c
array([0, 1, 2])

>>> a + c
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Section 01 Numpy

❑ 브로드캐스팅 연산

- 만약 1차원 배열이라고 해도 열의 개수가 맞지 않으면 아래 코드처럼 연산은 실패한다.

```
>>> d = np.arange(2)
>>> a + d
  Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
  ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

- 이 경우 열단위 연산을 하려면 배열 d의 모양을 바꾸어 연산할 수 있다.

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

>>> d = np.arange(2).reshape(2, 1)
>>> d
array([[0],
       [1]])

>>> a + d
array([[1., 1., 1.],
       [2., 2., 2.]])
```

Section 01 Numpy

❑ 인덱싱과 슬라이싱

- 위 코드에서 배열 a는 1차원, b는 2차원이다. 이때 2차원인 배열 b에 1개의 인덱스만 사용하면 1개의 행 모두가 선택된다.
- 2차원일 때는 인덱스 2개를 사용해서 열과 행을 지정해야 1개의 요소를 선택할 수 있다.

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

>>> a[5]
5

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

>>> b[1]
array([4, 5, 6, 7])

>>> b[1, 2]
6
```

Section 01 Numpy

❑ 인덱싱과 슬라이싱

- 다음 코드에서 보여주는 것처럼 값의 변경도 마찬가지로 인덱스로 정확히 1개의 요소를 지정하면 1개의 요소만 변경되지만, 인덱스를 적게 지정해서 행 단위로 지정하면 브로드캐스팅 연산이 일어나서 해당 단위 모두를 같은 값으로 변경한다

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

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

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

>>> b[0] = 0
>>> b
array([[ 0,  0,  0,  0],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> b[1, 2] = 99
>>> b
array([[ 0,  0,  0,  0],
       [ 4,  5, 99,  7],
       [ 8,  9, 10, 11]])
```

Section 01 Numpy

❑ 인덱싱과 슬라이싱

- 인덱스 자리에 콜론(:)을 이용해서 범위를 지정하면 슬라이싱(slicing)을 할 수 있다.
- 이때 범위의 끝 인덱스는 슬라이싱 결과에 포함되지 않는다.
- 시작과 끝 인덱스를 각각 생략하면 처음부터 끝까지라는 의미이다.

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

>>> a[2:5]
array([2, 3, 4])

>>> a[5:]
array([5, 6, 7, 8, 9])

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

```
>>> b[0:2, 1]
array([1, 5])

>>> b[0:2, 1:3]
array([[1, 2],
       [5, 6]])

>>> b[2, :]
array([ 8,  9, 10, 11])

>>> b[:, 1]
array([1, 5, 9])

>>> b[0:2, 1:3] = 0
>>> b
array([[ 0,  0,  0,  3],
       [ 4,  0,  0,  7],
       [ 8,  9, 10, 11]])
```

Section 01 Numpy

❑ 인덱싱과 슬라이싱

- 값의 할당은 앞서 살펴본 것처럼 브로드캐스팅 연산으로 이루어진다.
- 파이썬 기본형인 리스트와의 큰 차이점은 슬라이싱의 결과가 복제본이 아닌 원본이라는 것이다.
- 특히 다음 코드와 같이 슬라이싱으로 전체 배열 중 일부를 다른 변수에 할당하는 경우 별도의 배열로 착각하는 경우가 많다.
- 만약 파이썬 리스트처럼 복제본을 얻고 싶다면 ndarray.copy() 함수를 명시적으로 호출해야 한다.

```
>>> b
array([[ 0,  0,  0,  3],
       [ 4,  0,  0,  7],
       [ 8,  9, 10, 11]])

>>> bb = b[0:2, 1:3]
>>> bb
array([[0, 0],
       [0, 0]])

>>> bb[0] = 99
>>> b
array([[ 0, 99, 99,  3],
       [ 4,  0,  0,  7],
       [ 8,  9, 10, 11]])
```

Section 01 Numpy

❑ 팬시 인덱싱

- **배열 인덱스에 다른 배열을 전달해서 원하는 요소를 선택하는 방법을 팬시 인덱싱**(fancy indexing) **라고 한다**.
- **전달하는 배열에 숫자를 포함하고 있으면 해당 인덱스에 맞게 선택되고, 배열에 불**(boolean) **값을 포함하면** True **인 값을 갖는 요소만 선택된다**.

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

>>> a[[1, 3]]
array([1, 3])

>>> a[[True, False, True, False, True]]
array([0, 2, 4])
```

Section 01 Numpy

❑ 팬시 인덱싱

- NumPy 배열에 비교 연산을 하면 개별 요소들이 조건을 만족하는지를 알 수 있다.
- 반대로 불 값을 갖는 배열을 배열의 인덱스 대신 사용하면 True 값 위치의 값들만 얻을 수 있는데, 이 둘을 한번에 합해서 실행하면 원하는 조건의 값만을 얻을 수 있다.

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

>>> b = a > 5
>>> b
array([False, False, False, False, False, False,  True,  True,  True,
        True])

>>> a[b]
array([6, 7, 8, 9])

>>> a[a > 5]
array([6, 7, 8, 9])

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

Section 01 Numpy

❑ 팬시 인덱싱

- 다차원인 경우 인덱스 배열도 다차원으로 지정할 수 있고, 이때는 교차하는 인덱스의 것이 선택된다.

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

>>> b[[0, 2]]
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])

>>> b[[0, 2], [2, 3]]
array([ 2, 11])
```

Section 01 Numpy

❑ 병합과 분리

- 2개 이상의 NumPy 배열을 병합하는 방법은 크게 두 가지가 있다.
- 단순히 배열들을 이어 붙여서 크기를 키우는 방법과 새로운 차원을 만들어 서로서로 끼워넣는 방법이다.
- 여기서부터는 축(axis) 이라는 용어를 꼭 알아야 한다.
- NumPy 배열의 shape 속성을 확인하면 튜플 형식으로 몇 개의 숫자가 나오는 것은 이미 알아보았다.
- 각 숫자의 수는 차원을 의미하며, 맨 앞부터 0, 1, 2, ... 순으로 축 번호 사용한다.
- 예를 들어 어느 배열의 shape가 (10, 20, 3) 이면 3개의 축이 있고 10은 0번 축, 20은 1번 축, 3은 2번 축을 나타낸다.
- 축을 기준으로 작업을 한다는 뜻은 바로 shape의 각 순서에 따라 작업을 한다는 뜻이다.

Section 01 Numpy

❑ 병합과 분리

- **알아볼 병합에 사용하는 함수는 다음과 같다.**
- numpy.hstack(arrays) : arrays **열을 수평으로 병합**
- numpy.vstack(arrays) : arrays **열을 수직으로 병합**
- numpy.concatenate(arrays, axis=0) : arrays **열을 지정한 축 기준으로 병합**
- numpy.stack(arrays, axis=0) : arrays **열을 새로운 축으로 병합**
  - arrays: **병합 대상 열(튜플)**
  - axis: **작업할 대상 축 번호**

❑ 병합과 분리

- **다음 코드는 2행 2열인 배열 a와 b를 numpy.vstack() 로 수직 병합해서 4행 2열 배열로 만들고, numpy.hstack() 으로 수평 병합해서 2행 4열 배열로 만든다.**

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

>>> b = np.arange(10, 14).reshape(2, 2)
>>> b
array([[10, 11],
       [12, 13]])

>>> np.vstack((a, b))
array([[ 0,  1],
       [ 2,  3],
       [10, 11],
       [12, 13]])

>>> np.hstack((a, b))
array([[ 0,  1, 10, 11],
       [ 2,  3, 12, 13]])

>>> np.concatenate((a, b), 0)
array([[ 0,  1],
       [ 2,  3],
       [10, 11],
       [12, 13]])
```

```
>>> np.concatenate((a, b), 1)
array([[ 0,  1, 10, 11],
       [ 2,  3, 12, 13]])
```

Section 01 Numpy

❑ 병합과 분리

- **위 코드는** (4, 3) **배열** 2**개를** np.stack ((a, b), 0) **함수로 병합하고 있으므로 원래** 4**행** 3**열인** 2**차원 배열은 병합하고 나면** 3**차원이 되고 축 번호도** 0, 1, 2**로** 3**개가 된다**.
- **이때 축 번호로** 0**을 전달하므로 맨 앞 축 번호가 새로 생성되어** (2, 4, 3)**인 배열이 만들어진다**.
- **이때 축 번호를 생략하면** 0**과 같다**.

```
>>> a = np.arange(12).reshape(4, 3)
>>> b = np.arange(10, 130, 10).reshape(4, 3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

>>> b
array([[ 10,  20,  30],
       [ 40,  50,  60],
       [ 70,  80,  90],
       [100, 110, 120]])

>>> c = np.stack((a, b), 0)
>>> c.shape
(2, 4, 3)
```

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

       [[ 10,  20,  30],
        [ 40,  50,  60],
        [ 70,  80,  90],
        [100, 110, 120]]])
```

Section 01 Numpy

❑ 병합과 분리

- 다음은 같은 배열을 축 번호 1과 2로 각각 생성한 코드이다.
- 다음 코드는 앞서 사용했던 코드에서 (4, 3) 인 배열 a와 b를 병합한다.
- 이 함수들은 이미지 작업을 완료하고 작업 전과 후 이미지를 병합해서 나란히 출력할 때 자주 쓴다.

```
>>> d = np.stack((a, b), 1)
>>> d.shape
(4, 2, 3)

>>> d
array([[[  0,   1,   2],
        [ 10,  20,  30]],

       [[  3,   4,   5],
        [ 40,  50,  60]],

       [[  6,   7,   8],
        [ 70,  80,  90]],

       [[  9,  10,  11],
        [100, 110, 120]]])

>>> e = np.stack((a, b), 2)
>>> e.shape
(4, 3, 2)
```

```
>>> e
array([[[  0,  10],
        [  1,  20],
        [  2,  30]],

       [[  3,  40],
        [  4,  50],
        [  5,  60]],

       [[  6,  70],
        [  7,  80],
        [  8,  90]],

       [[  9, 100],
        [ 10, 110],
        [ 11, 120]]])

>>> ee = np.stack((a, b), -1)
>>> ee.shape
(4, 3, 2)
```

Section 01 Numpy

❑ 병합과 분리

- **배열을 분리할 때 사용하는 함수는 아래와 같다.**
- numpy.vsplit(array, indice) : array **배열을 수평으로 분리**
- numpy.hsplit(array, indice) : array **배열을 수직으로 분리**
- numpy.split(array, indice, axis=0) : array **배열을** axis **축으로 분리**
  - array : **분리할 배열**
  - indice : **분리할 개수 또는 인덱스**
  - axis : **기준 축 번호**

- indice**는 어떻게 나눌지를 정하는 인자인데, 정수 또는** 1**차원 배열을 사용할 수 있다.**
- **정수를 전달하면 배열을 그 수로 나누고,** 1**차원 배열을 전달하면 나누고자 하는 인덱스로 사용한다.**

❑ 병합과 분리

- indice 인자의 사용 예를 들면 아래와 같다.
- 다음 코드에서 np.hsplit(a, 3) 은 12 개의 요소를 갖는 배열을 수평으로 분리하는데, indice 항목에 3이 전달되었으므로 배열을 3개로 쪼개어 각 열은 4개 요소씩 갖는다.

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

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

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

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

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

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

Section 01 Numpy

❑ 병합과 분리

- 다음 코드는 나누려는 배열이 2차원일 때 numpy.vsplit() 를 사용한 것과 numpy.split()에 축 번호 0을 지정했을 때 같은 결과가 나오는 것을 보여주고 있다.
- 배열의 shape가 (4, 3)이므로 vsplit(b, 2) 는 수직으로 나누어 4행을 2로 나누어 2행씩 갖게 하고, split(b, 2, 0) 함수는 0번 축을 기준으로 2로 나누게 된다.

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

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

❑ 병합과 분리

- **다음 코드는 나누려는 배열이 2차원일 때 numpy.vsplit() 를 사용한 것과 numpy.split()에 축 번호 0을 지정했을 때 같은 결과가 나오는 것을 보여주고 있다.**

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

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

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

Section 01 Numpy

❑ 검색

- NumPy 배열을 사용하는 이유는 수많은 데이터를 쉽고 빠르게 다루려는 이유가 가장 크며, 이미지 작업도 마찬가지이다.
- 그래서 NumPy를 쓰다 보면 배열 안에서 관심있는 데이터만을 찾거나 찾아서 바꾸는 일이 자주 필요하다.
- 이와 관련한 함수는 다음과 같다.
- • ret = numpy.where(condition [, t , f]) : 조건에 맞는 요소를 찾기
  - ret : 검색 조건에 맞는 요소의 인덱스 또는 변경된 값으로 채워진 배열(튜플)
  - condition : 검색에 사용할 조건식
  - t, f : 조건에 따라 지정할 값 또는 열, 열의 경우 조건에 사용한 열과 같은 shape
    - t : 조건에 맞는 값에 지정할 값이나 배열
    - f : 조건에 틀린 값에 지정할 값이나 배열
- • numpy.nonzero(array) : array에서 요소 중에 0(영, zero) 이 아닌 요소의 인덱스들을 반환(튜플)
- • numpy.all(array [, axis]): array 의 모든 요소가 True 인지 검색

Section 01 Numpy

❑ 검색

- 이와 관련한 함수는 다음과 같다.
- numpy.nonzero(array) : array에서 요소 중에 0(영, zero) 이 아닌 요소의 인덱스들을 반환(튜플)
- numpy.all(array [, axis]): array 의 모든 요소가 True 인지 검색
  - array : 검색 대상 배열
  - axis : 검색할 기준 축, 생략하면 모든 요소 검색, 지정하면 축 개수별로 결과 반환
- numpy.any(array [, axis]) : array의 어느 요소이든 True가 있는지 검색

Section 01 Numpy

❑ 검색

- 아래 코드는 배열에서 조건에 맞는 인덱스를 찾아오는 사례, 그리고 찾은 값을 새로운 값으로 변경한 배열을 구하는 사례를 보여준다.
- 다음 코드 np.where(a > 15) 는 10부터 19까지 값을 갖는 배열 a에서 15보다 큰 값을 갖는 요소의 인덱스를 구한다

```
>>> a = np.arange(10, 20)
>>> a
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

>>> np.where(a > 15)
(array([6, 7, 8, 9], dtype=int32),)

>>> np.where(a > 15, 1, 0)
array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1])

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

❑ 검색

- 만약 조건에 맞는 요소만 특정한 값으로 변경하고 맞지 않는 요소는 기존 값을 그대로 갖게 하려면 다음과 같은 코드로 할 수 있다.
- 아래 코드는 조건에 맞거나 틀린 경우에 할당할 값으로, 원래의 검색 대상 배열을 그대로 지정해서 조건에 맞는 값만 새로운 값으로 지정하거나 그 반대로도 가능하다.
- 결과는 새로운 배열을 반환하므로 원본 배열은 그대로 유지된다.

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

>>> np.where(a > 15, 99, a)
array([10, 11, 12, 13, 14, 15, 99, 99, 99, 99])

>>> np.where(a > 15, a, 0)
array([ 0,  0,  0,  0,  0,  0, 16, 17, 18, 19])

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

❑ 검색

- 다차원 배열인 경우 원하는 요소를 검색만 한다면 해당하는 요소의 인덱스는 여러 개를 반환한다.
- 위 코드는 3행 4열의 배열에서 6보다 큰 수만 검색하는 코드인데, 검색 결과는 행 번호(axis=0)만 갖는 배열과 열 번호(axis=1) 만 갖는 배열 2개를 반환한다.

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

>>> coords = np.where(b > 6)
>>> coords
(array([1, 2, 2, 2, 2], dtype=int32), array([3, 0, 1, 2, 3], dtype=int32))

>>> np.stack((coords[0], coords[1]), -1)
array([[1, 3],
       [2, 0],
       [2, 1],
       [2, 2],
       [2, 3]], dtype=int32)
```

Section 01 Numpy

❑ 검색

- **배열 요소 중에** 0(zero) **이 아닌 요소를 찾을 때는** numpy.nonzero() **함수를 사용할 수 있다**.
- **이 함수는** 0**이 아닌 요소의 인덱스를 배열로 만들어서 반환한다**.

```
>>> z = np.array([0, 1, 2, 0, 1, 2])
>>> np.nonzero(z)
(array([1, 2, 4, 5], dtype=int32),)

>>> zz = np.array([[0, 1, 2], [1, 2, 0], [2, 0, 1]])
>>> zz
array([[0, 1, 2],
       [1, 2, 0],
       [2, 0, 1]])

>>> coords = np.nonzero(zz)
>>> coords
(array([0, 0, 1, 1, 2, 2], dtype=int32), array([1, 2, 0, 1, 0, 2], dtype=int32))

>>> np.stack((coords[0], coords[1]), -1)
array([[0, 1],
       [0, 2],
       [1, 0],
       [1, 1],
       [2, 0],
       [2, 2]], dtype=int32)
```

Section 01 Numpy

❑ 검색

- numpy.nonzero() **함수는** True**나** False **같은 불 값에 대해서는** False**를** 0(**영**, zero)**으로 간주하고 동작하므로** numpy.where() **함수처럼 조건을 만족하는 요소의 인덱스를 찾을 수도 있다**.

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

>>> np.nonzero(a > 15)
(array([6, 7, 8, 9], dtype=int32),)

>>> np.where(a > 15)
(array([6, 7, 8, 9], dtype=int32),)

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

>>> np.nonzero(b > 6)
(array([1, 2, 2, 2, 2], dtype=int32), array([3, 0, 1, 2, 3], dtype=int32))

>>> np.where(b > 6)
(array([1, 2, 2, 2, 2], dtype=int32), array([3, 0, 1, 2, 3], dtype=int32))
```

Section 01 Numpy

❑ 검색

- NumPy 배열에 모든 요소가 참 또는 거짓인지 확인할 때는 all() 함수를 사용할 수 있다.
- 다음 코드에서 배열의 모든 요소가 True 일 때는 np.all(t) 가 True를 반환하지만 1개의 요소를 False로 바꾸자 그 결과가 False로 나타나는 것을 보여준다.

```
>>> t = np.array([True, True, True])
>>> np.all(t)
True

>>> t[1] = False
>>> t
array([ True, False,  True])

>>> np.all(t)
False
```

❑ 검색

- all() 함수에 축(axis) 인자를 지정하지 않으면 모든 요소에 대해서 True를 만족하는지 검색하지만, 축 인자를 지정하면 해당 축을 기준으로 True를 만족하는지 반환한다.

```
>>> tt = np.array([[True, True], [False, True], [True, True]])
>>> tt
array([[ True,  True],
       [False,  True],
       [ True,  True]])

>>> np.all(tt, 0)
array([False,  True])

>>> np.all(tt, 1)
array([ True, False,  True])
```

Section 01 Numpy

❑ 검색

- **다음 코드에서 배열 a와 b는 처음에는 동일하지만 b[5] = -1 연산으로 다르게 했다.**

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

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

>>> a == b
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True])

>>> np.all(a == b)
True

>>> b[5] = -1
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> b
array([ 0,  1,  2,  3,  4, -1,  6,  7,  8,  9])

>>> np.all( a == b)
False

>>> np.where(a == b)
(array([0, 1, 2, 3, 4, 6, 7, 8, 9], dtype=int32),)

>>> np.where(a != b)
(array([5], dtype=int32),)
```

Section 01 Numpy

❑ 기초 통계 함수

- **배열의 값이 하나하나 확인할 수 없을 만큼 많을 때는 평균, 최대 값, 최소 값같은 통계값들이 의미있는 정보가 될 때가 많다.**
- **대표적인 함수는 다음과 같다.**
- numpy.sum(array [, axis]) : **배열의 합계 계산**
- numpy.mean(array [, axis]) : **배열의 평균 계산**
- numpy.amin(array [, axis]) : **배열의 최소 값 계산**
- numpy.min(array [, axis]) : numpy.amin() **과 동일**
- numpy.amax(array [, axis]) : **배열의 최대 값 계산**
- numpy.max(array [, axis]) : numpy.amax()**와 동일**
  - array : **계산의 대상 배열**
  - axis : **계산 기준, 생략하면 모든 요소를 대상**

Section 01 Numpy

❑ 기초 통계 함수

- **다음 코드는 0부터 11 까지 값으로 갖는 배열을 3행 4열로 만들었다.**
- np.sum(a) **함수는 모든 요소 값의 합계를 계산한다.**
- **축 번호를 지정하면 행과 열을 기준으로 각각 합산하는 것을 알 수 있다.**

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

>>> np.sum(a)
66

>>> np.sum(a, 0)
array([12, 15, 18, 21])

>>> np.sum(a, 1)
array([ 6, 22, 38])
```

Section 01 Numpy

❑ 기초 통계 함수

- 다음 코드는 같은 방법으로 평균, 최소 값, 최대 값을 각각 계산하고 있다.

```
>>> np.mean(a)
5.5

>>> np.mean(a, 0)
array([4., 5., 6., 7.])

>>> np.mean(a, 1)
array([1.5, 5.5, 9.5])

>>> np.amin(a)
0

>>> np.amin(a, 0)
array([0, 1, 2, 3])

>>> np.amin(a, 1)
array([0, 4, 8])

>>> np.amax(a, 0)
array([ 8,  9, 10, 11])

>>> np.amax(a, 1)
array([ 3,  7, 11])
```

Section 01 Numpy

❑ 기초 통계 함수

- **다음 코드는** amin() **과** min(), amax() **와** max() **함수가 내부적으로 동일하다는 것을 보여 주고 있다**.
- **따라서 각각의 함수는 어느 것을 사용하든 차이가 없다**.

```
>>> np.amin is np.min
True

>>> np.amax is np.amax
True
```

Section 01 Numpy

❑ 이미지 생성

- 다음 코드는 지금까지 다룬 지식으로 간단한 이미지를 생성해 보는 사례이다.

```
1   import cv2
2   import numpy as np
3
4   img = np.zeros((120, 120), dtype=np.uint8)
5   img[25:35, :] = 45
6   img[55:65, :] = 115
7   img[85:95, :] = 160
8   img[:, 35:45] = 205
9   img[:, 75:85] = 255
10  cv2.imshow('Gray', img)
11  cv2.waitKey(0)
12  cv2.destroyAllWindows()
```

Section 01 Numpy

❑ 이미지 생성

- **다음 예제는** np.zeros((120, 120, 3) , dtype=np.uint8) **로** 120**행**, 120 **열 크기의** 3**개의 채널을 갖는** 3**차원 배열을 생성했다**.

```
1   import cv2
2   import numpy as np
3
4   img = np.zeros((120, 120, 3), dtype=np.uint8)
5   img[25:35, :] = [255, 0, 0]
6   img[55:65, :] = [0, 255, 0]
7   img[85:95, :] = [0, 0, 255]
8   img[:, 35:45] = [255, 0, 0]
9   img[:, 75:85] = [255, 0, 255]
10  cv2.imshow('BGR', img)
11  cv2.waitKey(0)
12  cv2.destroyAllWindows()
```

Section 02 Matplotlib

❑ Matplotlib의 설치

- 라즈베리파이의 경우 데비안패키지 관리자인 APT 명령어로 설치해야 한다.
- 설치 명령어는 다음과 같다.

```
$ sudo apt-get update
$ sudo apt-get install python3-matplotlib
```

- 설치가 모두 끝나면 파이썬3 대화형 콘솔에서 아래의 명령어로 버전 번호를 확인할 수 있다.

```
>>> import matplotlib.pyplot
>>> matplotlib.__version__
'3.0.2'
```

Section 02 Matplotlib

❑ plot

- 그래프를 그리는 가장 간단한 방법은 plot() 함수를 사용하는 것이다.
- 1차원 배열을 인자로 전달하면 배열의 인덱스를 x 좌표로, 배열의 값을 y 좌표로 써서 그래프를 그린다.
- 아래의 코드는 가장 간단한 방법으로 그래프를 그리는 예제이다.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  a = np.array([2, 6, 7, 3, 12, 8, 4, 5])
5  plt.plot(a)
6  plt.show()
```

Section 02 Matplotlib

❑ plot

- 두 배열의 상관관계를 그래프로 표시하려면 plot() 함수의 인자로 배열을 순차적으로 전달하면 차례대로 좌표로 사용해서 그래프를 그린다.
- 2개의 배열로 그래프로 표시하는 예시는 아래와 같다.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.arange(10)
5  y = x**2
6  plt.plot(x, y)
7  plt.show()
```

Section 02 Matplotlib

❑ color와 style

- **그래프의 선에 색상과 스타일을 지정할 수 있다.**
- plot() **함수의 마지막 인자에 아래의 색상 기호 중 하나를 선택해서 문자로 전달하면 색상이 적용된다.**
- **색상기호**
  - b : **파란색** (Blue)
  - g : **초록색** (Green)
  - r : **빨간색** (Red)
  - c : **청록색** (Cyan)
  - m : **자홍색** (Magenta)
  - y : **노란색** (Yellow)
  - k : **검은색** (black)
  - w : **흰색** (White)

Section 02 Matplotlib

❑ color와 style

- 다음 코드는 선을 빨간색으로 표시한 예제이다.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.arange(10)
5  y = x**2
6  plt.plot(x, y, 'r')
7  plt.show()
```

Section 02 Matplotlib

❑ color와 style

- **색상과 함께 스타일도 지정할 수 있는데, 다음의 스타일 기호 중 하나를 색상 값에 이어 붙여서 사용한다.**

기호	스타일	기호	스타일
-	실선	--	이음선
-.	점 이음선	:	점선
.	점	,	픽셀
o	원	v	역삼각형
^	정삼각형	<	좌삼각형
>	우삼각형	l	작은 역삼각형
2	작은 정삼각형	3	작은 좌삼각형
4	작은 우삼각형	s	사각형
p	오각형	*	별표
h	육각형	+	더하기
D	다이아몬드	x	엑스표

Section 02 Matplotlib

❑ color와 style

- 다음 예제는 다양한 색상과 스타일의 그래프를 표시하는 코드이다.

```
1   import matplotlib.pyplot as plt
2   import numpy as np
3
4   x = np.arange(10)
5   f1 = x * 5
6   f2 = x**2
7   f3 = x**2 + x*2
8
9   plt.plot(x, ‘r--')
10  plt.plot(f1, ‘g.’)
11  plt.plot(f2, ‘bv’)
12  plt.plot(f3, ‘ks’)
13  plt.show()
```

Section 02 Matplotlib

❑ subplot

- 각각의 그래프를 분리해서 따로 그려야 할 때는 plt.subplot() 함수를 이용한다.
- 이 함수는 3개의 인자를 이용해서 몇 행 몇 열로 분할된 그래프에 몇 번째 그래프를 그릴지를 먼저 지정한 후에 plt.plot() 함수를 호출하면 그 자리에 그래프를 그리게 된다.

```
1   import matplotlib.pyplot as plt
2   import numpy as np
3
4   x = np.arange(10)
5
6   plt.subplot(2, 2, 1)
7   plt.plot(x, x**2)
8
9   plt.subplot(2, 2, 2)
10  plt.plot(x, x * 5)
11
```

Section 02 Matplotlib

❑ subplot

- 각각의 그래프를 분리해서 따로 그려야 할 때는 plt.subplot() 함수를 이용한다.
- 이 함수는 3개의 인자를 이용해서 몇 행 몇 열로 분할된 그래프에 몇 번째 그래프를 그릴지를 먼저 지정한 후에 plt.plot() 함수를 호출하면 그 자리에 그래프를 그리게 된다.

```
12  plt.subplot(2, 2, 3)
13  plt.plot(x, np.sin(x))
14
15  plt.subplot(2, 2, 4)
16  plt.plot(x, np.cos(x))
17
18  plt.show()
```

Section 02 Matplotlib

❑ 이미지 표시

- plt.plot() **대신에** plt.imshow() **함수를 호출하면** OpenCV**로 읽어들인 이미지를 그래프 영역에 출력할 수 있다**.

```
1  import cv2
2  import matplotlib.pyplot as plt
3
4  img = cv2.imread('./img/lena.jpg')
5
6  plt.imshow(img)
7  plt.show()
```

Section 02 Matplotlib

❑ 이미지 표시

- plt.imshow() **함수는 컬러 이미지인 경우 컬러 채널을** R, G, B **순으로 해석하지만** OpenCV **이미지는** B, G, R **순으로 만들어져서 색상의 위치가 반대이다**.
- **그래서** OpenCV**로 읽은 이미지를** R, G, B **순으로 순서를 바꾸어서** plt.imshow() **함수에 전달해야 제대로 된 색상으로 출력할 수 있다**.

```
1  import cv2
2  import matplotlib.pyplot as plt
3
4  img = cv2.imread('./img/lena.jpg')
5
6  plt.imshow(img[:, :, ::-1])
7  plt.xticks([])
8  plt.yticks([])
9  plt.show()
```

Section 02 Matplotlib

❑ 이미지 표시

- **프로그램의 결과로 이미지를 여러 개 출력해야 하는 경우**, OpenCV**의** cv2.imshow() **함수는 여러 번 호출하면 매번 새로운 창이 열리기 때문에 귀찮다**.
- plt.imshow()**함수는** plt.subplot() **함수와 함께 사용하면 하나의 창에 여러 개의 이미지를 동시 출력할 수 있으니 이런 경우 좋은 대안이 될 수 있다**.

```
1   import matplotlib.pyplot as plt
2   import numpy as np
3   import cv2
4
5   img1 = cv2.imread('./img/model.jpg')
6   img2 = cv2.imread('./img/model2.jpg')
7   img3 = cv2.imread('./img/model3.jpg')
8
9   plt.subplot(1, 3, 1)
10  plt.imshow(img1[:, :, (2, 1, 0)])
11  plt.xticks([]); plt.ytics([])
12
```

Section 02 Matplotlib

❑ 이미지 표시

- **프로그램의 결과로 이미지를 여러 개 출력해야 하는 경우**, OpenCV**의** cv2.imshow() **함수는 여러 번 호출하면 매번 새로운 창이 열리기 때문에 귀찮다**.
- plt.imshow()**함수는** plt.subplot() **함수와 함께 사용하면 하나의 창에 여러 개의 이미지를 동시 출력할 수 있으니 이런 경우 좋은 대안이 될 수 있다**.

```
13  plt.subplot(1, 3, 2)
14  plt.imshow(img2[:, :,::-1])
15  plt.xticks([]); plt.ytics([])
16
17  plt.subplot(1, 3, 3)
18  plt.imshow(img3[:, :,::-1])
19  plt.xticks([]); plt.ytics([])
20
21  plt.show()
```

Section 02 Matplotlib

❑ 이미지 표시

- **프로그램의 결과로 이미지를 여러 개 출력해야 하는 경우**, OpenCV**의** cv2.imshow() **함수는 여러 번 호출하면 매번 새로운 창이 열리기 때문에 귀찮다**.
- plt.imshow()**함수는** plt.subplot() **함수와 함께 사용하면 하나의 창에 여러 개의 이미지를 동시 출력할 수 있으니 이런 경우 좋은 대안이 될 수 있다**.

```
13  plt.subplot(1, 3, 2)
14  plt.imshow(img2[:, :,::-1])
15  plt.xticks([]); plt.ytics([])
16
17  plt.subplot(1, 3, 3)
18  plt.imshow(img3[:, :,::-1])
19  plt.xticks([]); plt.ytics([])
20
21  plt.show()
```

Q&A