Python 기초 Chap6: 행렬 (Matrix) 가지고 놀기

Table of contents

6.0.1 행렬이란 무엇일까?

행렬이란, 앞에서 배운 벡터들을 사용하여 만들 수 있는 객체입니다. 좀 더 구체적으로 이야기하면 길이가 같은 벡터들을 사각형 모양으로 묶어 놓았다고 생각하면 좋겠습니다.

6.0.1.a 벡터들을 모아놓은 것

행렬은 일련의 벡터들을 모아놓은 사각형 모양의 구조를 띕니다. 이는 반드시 사각형의 형태를 가져야 하며, 이 사각형의 크기는 `shape` 속성을 통해 측정할 수 있습니다. 아래의 예제에서 `np.column_stack((1, 2, 3, 4), (12, 13, 14, 15))`는 두 개의 벡터를 합쳐 하나의 행렬을 생성합니다.

```
import numpy as np

# 두 개의 벡터를 합쳐 행렬 생성
matrix = np.column_stack((np.arange(1, 5),
                          np.arange(12, 16)))
print("행렬:\n", matrix)

# 행렬의 크기를 재어주는 shape 속성
print("행렬의 크기:", matrix.shape)
```

```
행렬:
 [[ 1 12]
 [ 2 13]
 [ 3 14]
 [ 4 15]]
행렬의 크기: (4, 2)
```

결과를 보니, “가로로 4행, 세로로 2열에 해당하는 크기의 숫자들이 들어있다."라고 해석할 수 있습니다.

6.0.2 행렬 만들기

행렬을 생성하는 데에는 numpy의 `np.zeros()`나 `np.arange()`, `np.reshape()` 같은 함수를 사용합니다. 이 함수들은 행렬을 만들 때 필요한 정보들, 즉 행렬을 채울 숫자들, 행의 개수, 열의 개수 등을 입력값으로 받습니다.

6.0.3 NumPy 함수 요약

함수	문법	설명
`np.zeros()`	`np.zeros((행, 열))`	지정된 형태의 모든 요소가 0인 행렬 생성
`np.reshape()`	`np.reshape((행, 열))`	배열의 형태를 지정된 행과 열로 변환

6.0.3.a 빈 행렬 만들기

빈 벡터를 만들 때 `np.zeros()` 함수를 사용하여 값을 넣지 않고 길이를 지정해주면 됩니다. 다음은 2행 2열에 해당하는 빈 행렬을 만들어서 y 변수에 저장하는 코드입니다.

```
import numpy as np

# 2행 2열 빈 행렬 생성
y = np.zeros((2, 2))
print("빈 행렬 y:\n", y)
```

```
빈 행렬 y:
 [[0. 0.]
 [0. 0.]]
```

6.0.3.b 채우면서 만들기

아래의 코드는 1부터 4까지의 수를 원소로 갖는 2행 2열의 행렬 y를 생성합니다.

```
# 1부터 4까지의 수로 채운 2행 2열 행렬 생성
y = np.arange(1, 5).reshape(2, 2)
print("1부터 4까지의 수로 채운 행렬 y:\n", y)
```

```
1부터 4까지의 수로 채운 행렬 y:
 [[1 2]
 [3 4]]
```

행렬의 크기는 무조건 사각형이므로, `reshape()` 함수를 사용하여 행의 수(nrow)와 열의 수(ncol)를 지정하면 자동으로 행렬의 크기를 계산하여 만들어 줍니다.

6.0.3.c 행렬을 채우는 방법 - `order` 옵션

앞의 결과를 살펴보면 행렬을 생성하며 원소들을 채울 때, 기본 설정이 가로로 채워지도록 설정이 되어있는 것을 확인 할 수 있습니다. `reshape()`의 `order` 옵션을 사용하여 원소들이 행렬에 채워지는 방향을 정할 수 있습니다.

- `order='C'`: 행 우선 순서 (row-major order). 기본값으로, C 언어 스타일로 행을 먼저 채웁니다.
- `order='F'`: 열 우선 순서 (column-major order). Fortran 언어 스타일로 열을 먼저 채웁니다.

아래 코드는 1부터 4까지의 수를 원소로 하는 행렬을 가로 방향으로 채워 나갑니다.

```
import numpy as np

# 가로 방향으로 채우기 (기본값)
y = np.arange(1, 5).reshape((2, 2), order='C')
print("세로 방향으로 채운 행렬 y:\n", y)
```

```
세로 방향으로 채운 행렬 y:
 [[1 2]
 [3 4]]
```

반면에 세로로 원소들을 채우려면, `order='F'`를 설정합니다.

```
import numpy as np

# 가로 방향으로 채우기
y = np.arange(1, 5).reshape((2, 2), order='F')
print("가로 방향으로 채운 행렬 y:\n", y)
```

```
가로 방향으로 채운 행렬 y:
 [[1 3]
 [2 4]]
```

6.0.4 행렬의 원소에 접근하기

행렬 인덱싱을 좀 더 자세히 알아보기 위해서 좀 더 큰 예제 행렬을 만들겠습니다. 아래 코드는 1부터 10까지의 수에 2를 곱한 값들을 원소로 갖는 5행 2열의 행렬 x를 생성합니다.

```
import numpy as np
```

```
# 1부터 10까지의 수에 2를 곱한 값으로 5행 2열의 행렬 생성
x = np.arange(1, 11).reshape((5, 2)) * 2
print("행렬 x:\n", x)
```

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

6.0.4.a 행렬 인덱싱 (Indexing)

행렬의 특정 원소에 접근하는 방법을 인덱싱(Indexing)이라고 합니다. 행렬 내에서 특정 위치의 원소는 접근하고자 하는 행렬 뒤에 대괄호 안 순서쌍 형태로 표시하여 나타낼 수 있습니다. 순서쌍 문법은 `[row, col]`로, 예를 들어 1행 2열에 위치한 원소는 `[0, 1]`로 표현합니다.

- 문법: `행렬[행위치, 열위치]`
- 행렬 `x`의 1행 2열에 위치한 원소: `x[0, 1]`

그리고 `x[0, 1]`는 행렬 `x`에서 1행 2열의 원소를 반환합니다.

```
# 1행 2열의 원소 접근
element = x[0, 1]
print("1행 2열의 원소:", element)
```

```
1행 2열의 원소: 4
```

6.0.4.a.a 전체를 나타내는 빈 칸

특정 열의 모든 원소에 접근하려면, 행의 위치를 나타내는 곳에 빈 칸 대신 콜론(`:`)을 사용합니다. 예를 들어 `x[:, 1]`는 행렬 `x`의 두 번째 열의 모든 원소를 반환합니다.

```
import numpy as np

# 1부터 10까지의 수에 2를 곱한 값으로 5행 2열의 행렬 생성
x = np.arange(1, 11).reshape((5, 2)) * 2
print("행렬 x:\n", x)

# 두 번째 열의 모든 원소 반환
second_column = x[:, 1]
print("두 번째 열의 모든 원소:", second_column)
```

```
행렬 x:
 [[ 2  4]
 [ 6  8]
 [10 12]
 [14 16]
 [18 20]]
두 번째 열의 모든 원소: [ 4  8 12 16 20]
```

만약 행렬 x의 세 번째 행을 추출하기 위해서는 어떻게 해야 할까요? 이번에는 반대로 x의 모든 열을 선택해야 하므로, 다음과 같이 코드를 작성하면 됩니다.

```
# 세 번째 행의 모든 원소 반환
third_row = x[2, :]
print("세 번째 행의 모든 원소:", third_row)
```

```
세 번째 행의 모든 원소: [10 12]
```

6.0.4.a.b 선택적으로 골라오기

특정 열에서 선택적으로 원소를 골라오려면, 행의 위치를 나타내는 곳에 원하는 행의 번호를 넣습니다. 예를 들어 `x[[1, 2, 4], 1]`는 행렬 x의 두 번째 열에서 두 번째, 세 번째, 다섯 번째 행의 원소를 반환합니다.

```
# 두 번째 열에서 두 번째, 세 번째, 다섯 번째 행의 원소 반환
selected_elements = x[[1, 2, 4], 1]
print("두번째 열의 2, 3, 5번째 행의 원소: \n", selected_elements)
```

```
두번째 열의 2, 3, 5번째 행의 원소:
 [ 8 12 20]
```

6.0.5 행렬 필터링

`True`, `False`를 원소로 가진 배열을 사용하여 행렬에서 원하는 원소를 필터링할 수 있습니다. 예를 들어, `x[[True, True, False, False, True], 0]`은 행렬 x의 첫 번째 열에서 첫 번째, 두 번째, 다섯 번째 행의 원소를 선택하여 반환합니다.

```
import numpy as np

# 1부터 10까지의 수에 2를 곱한 값으로 5행 2열의 행렬 생성
x = np.arange(1, 11).reshape((5, 2)) * 2
print("행렬 x:\n", x)
```

```
# 첫 번째 열에서 첫 번째, 두 번째, 다섯 번째 행의 원소 반환
filtered_elements = x[[True, True, False, False, True], 0]
print("첫 번째 열의 첫 번째, 두 번째, 다섯 번째 행의 원소:\n", filtered_elements)
```

```
행렬 x:
 [[ 2  4]
 [ 6  8]
 [10 12]
 [14 16]
 [18 20]]
첫 번째 열의 첫 번째, 두 번째, 다섯 번째 행의 원소:
 [ 2  6 18]
```

6.0.5.a.a 조건문 사용한 필터링

또한, 조건문을 사용하여 원하는 조건을 만족하는 원소를 필터링할 수 있습니다. 예를 들어, `x[x[:, 1] > 15, 0]`은 행렬 `x`의 두 번째 열의 원소가 15보다 큰 행의 첫 번째 열의 원소를 선택하여 반환합니다.

```
# 두 번째 열의 원소가 15보다 큰 행의 첫 번째 열의 원소 반환
filtered_elements = x[x[:, 1] > 15, 0]
print("두 번째 열의 원소가 15보다 큰 행의 첫 번째 열의 원소:\n", filtered_elements)
```

```
두 번째 열의 원소가 15보다 큰 행의 첫 번째 열의 원소:
 [14 18]
```

6.0.6 사진은 행렬이다.

6.0.6.a 행렬을 이용해 이미지 생성하기

이미지는 흑백인 경우 `0`과 `1` 사이의 숫자로 표현할 수 있습니다. 이 때, `0`은 검은색을, `1`은 흰색을 의미합니다. 예를 들어, 아래의 코드에서는 `np.random.rand(3, 3)`으로 `0`과 `1` 사이의 난수 9개를 생성하고, 이를 `3x3` 크기의 행렬 `img1`로 만듭니다.

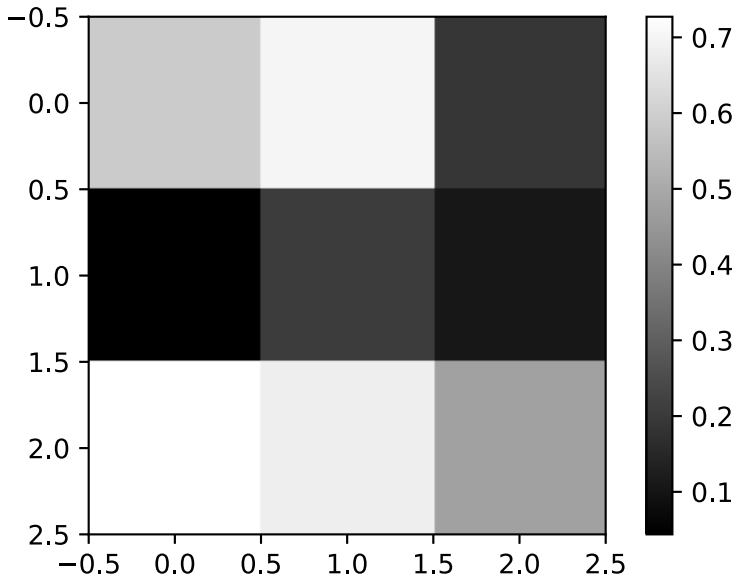
```
import numpy as np
import matplotlib.pyplot as plt

# 난수 생성하여 3x3 크기의 행렬 생성
np.random.seed(2024)
img1 = np.random.rand(3, 3)
print("이미지 행렬 img1:\n", img1)

# 행렬을 이미지로 표시
```

```
plt.imshow(img1, cmap='gray', interpolation='nearest')
plt.colorbar()
plt.show()
```

```
이미지 행렬 img1:
 [[0.58801452 0.69910875 0.18815196]
 [0.04380856 0.20501895 0.10606287]
 [0.72724014 0.67940052 0.4738457 ]]
```



6.0.6.a.a 행렬 다운로드하기

링크를 눌러, 미리 정의된 행렬을 다운로드받아보겠습니다. 다운로드한 압축파일을 풀어보면 `img_mat.csv` 파일이 있는데, 이를 `np.loadtxt()` 함수를 사용하여 읽어 들이고, 이를 `img_mat` 행렬로 저장합니다.

```
import numpy as np

img_mat = np.loadtxt('./data/img_mat.csv', delimiter=',', skiprows=1)
```

이미지가 행렬로 변환되면, 각 행렬의 원소는 이미지의 픽셀 값을 나타냅니다. 이 값을 `shape` 속성과 `head()` 함수를 사용하여 확인할 수 있습니다.

```
# 행렬의 크기 확인
print("행렬의 크기:", img_mat.shape)
```

```
# 행렬의 일부 확인
print("행렬의 일부:\n", img_mat[:3, :4])
```

```
행렬의 크기: (88, 50)
행렬의 일부:
 [[132. 131. 134. 132.]
 [135. 137. 137. 138.]
 [143. 142. 145. 146.]]
```

6.0.6.b 행렬에서 사진으로

앞에서 배운 숫자들이 채워진 행렬을 이미지로 변환하기 위해서는 우선 행렬 안의 값이 적절한 범위 (0과 1사이)에 있어야 합니다. `max()`와 `min()` 함수를 사용해 `img_mat` 행렬의 최대값과 최소값을 확인합니다.

```
# 행렬의 최대값과 최소값 확인
print("최대값:", img_mat.max())
print("최소값:", img_mat.min())
```

```
최대값: 255.0
최소값: 0.0
```

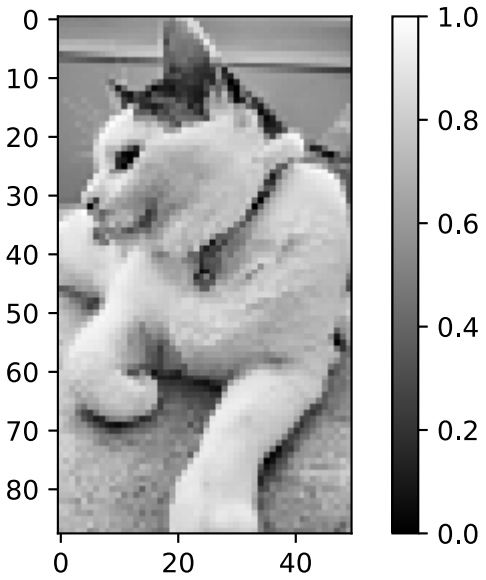
그런데 행렬의 원소값이 0과 255사이에 있는 것을 알 수 있습니다. 이를 0과 1 사이로 변환하기 위해 255로 나눠줍니다.

```
# 행렬 값을 0과 1 사이로 변환
img_mat = img_mat / 255.0
```

그 다음, Matplotlib의 `imshow()` 함수를 이용해 행렬을 이미지로 변환하고, `plt.show()` 함수를 이용해 변환된 이미지를 출력할 수 있습니다.

```
import matplotlib.pyplot as plt

# 행렬을 이미지로 변환하여 출력
plt.imshow(img_mat, cmap='gray', interpolation='nearest')
plt.colorbar()
plt.show()
```



6.0.7 행렬의 연산

6.0.7.a 행렬 뒤집기 (Transpose)

행렬을 뒤집으려면 `transpose()` 메서드를 사용합니다. 이 메서드는 주어진 행렬의 행과 열을 뒤집어 반환합니다.

```
import numpy as np

# 5행 2열의 행렬 생성
x = np.arange(1, 11).reshape((5, 2)) * 2
print("원래 행렬 x:\n", x)

# 행렬을 전치
transposed_x = x.transpose()
print("전치된 행렬 x:\n", transposed_x)
```

```
원래 행렬 x:
 [[ 2  4]
 [ 6  8]
 [10 12]
 [14 16]
 [18 20]]
전치된 행렬 x:
 [[ 2  6 10 14 18]
 [ 4  8 12 16 20]]
```

6.0.7.b 행렬의 곱셈 (dot product)

행렬의 곱셈은 행렬의 크기가 맞아야 가능합니다. NumPy에서는 `dot()` 메서드를 사용하여 두 행렬의 곱을 계산합니다.

```
# 2행 3열의 행렬 y 생성
y = np.arange(1, 7).reshape((2, 3))
print("행렬 y:\n", y)

# 행렬 x와 y의 크기 확인
print("행렬 x의 크기:", x.shape)
print("행렬 y의 크기:", y.shape)

# 행렬곱 계산
dot_product = x.dot(y)
print("행렬곱 x * y:\n", dot_product)
```

```
행렬 y:
 [[1 2 3]
 [4 5 6]]
행렬 x의 크기: (5, 2)
행렬 y의 크기: (2, 3)
행렬곱 x * y:
 [[ 18  24  30]
 [ 38  52  66]
 [ 58  80 102]
 [ 78 108 138]
 [ 98 136 174]]
```

6.0.7.c 원소별 곱셈 (Hadamard product, element-wise product)

두 행렬의 크기가 같을 경우, 각 원소별로 곱셈을 수행할 수 있습니다. 이를 Hadamard 곱셈 또는 원소별 곱셈이라고 합니다.

```
# 2행 2열의 행렬 z 생성
z = np.arange(10, 14).reshape((2, 2))
print("행렬 z:\n", z)

# 2행 2열의 행렬 y 생성
y = np.array([[1, 2], [3, 4]])
print("행렬 y:\n", y)

# 원소별 곱셈 계산
elementwise_product = y * z
print("원소별 곱셈 y * z:\n", elementwise_product)
```

```
행렬 z:
 [[10 11]
 [12 13]]
행렬 y:
 [[1 2]
 [3 4]]
원소별 곱셈 y * z:
 [[10 22]
 [36 52]]
```

6.0.7.d 행렬의 역행렬

행렬의 역행렬은 `np.linalg.inv()` 함수를 사용하여 계산합니다. 이 함수에 행렬을 입력하면, 해당 행렬의 역행렬을 반환합니다.

```
# 2행 2열의 정사각행렬 y 생성
y = np.array([[1, 2], [3, 4]])
print("행렬 y:\n", y)

# 행렬 y의 역행렬 계산
inverse_y = np.linalg.inv(y)
print("행렬 y의 역행렬:\n", inverse_y)
```

```
행렬 y:
 [[1 2]
 [3 4]]
행렬 y의 역행렬:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

그러나 모든 행렬이 역행렬을 가지는 것은 아닙니다. 예를 들어, 역행렬이 존재하지 않는 선형 독립인 행렬의 경우 `np.linalg.inv()` 함수는 오류를 반환합니다.

```
# 역행렬이 존재하지 않는 행렬 no_inverse 생성
no_inverse = np.array([[1, 2], [1, 2]])
print("역행렬이 존재하지 않는 행렬:\n", no_inverse)

np.linalg.inv(no_inverse)
```

```
역행렬이 존재하지 않는 행렬:
 [[1 2]
 [1 2]]
```

```
LinAlgError: Singular matrix
```

6.0.8 행렬의 연산과 브로드캐스팅

벡터에서 사용되는 브로드캐스팅 개념은 행렬에서도 적용됩니다. 행렬에 벡터를 곱할 때 벡터의 길이를 자동으로 맞추어 계산해 줍니다.

- 가로 벡터 브로드캐스팅

```
import numpy as np

# 2행 2열의 행렬 y 생성
x = np.array([1, 2])
y = np.array([[1, 2], [3, 4]])
print("행렬 y:\n", y)

# 벡터와 행렬의 원소별 곱셈 (브로드캐스팅)
result = x * y
x.shape
y.shape

print("벡터와 행렬의 원소별 곱셈 (브로드캐스팅):\n", result)
```

```
행렬 y:
 [[1 2]
 [3 4]]
벡터와 행렬의 원소별 곱셈 (브로드캐스팅):
 [[1 4]
 [3 8]]
```

- 세로 벡터 브로드캐스팅

```
# 행렬 y와 행렬 z 생성
y = np.array([[1, 2], [3, 4]])
z = np.array([[1], [2]])
print("행렬 y:\n", y)
print("행렬 z:\n", z)

y.shape
z.shape
result = y * z
```

```
행렬 y:
 [[1 2]
 [3 4]]
```

```
행렬 z:
 [[1]
 [2]]
```

6.0.9 차원 축소 효과

Python에서는 행렬에서 하나의 행이나 열을 선택할 경우, 해당 부분을 1차원 배열로 반환하는 기능을 제공합니다. 이는 편리할 수 있지만, 때로는 불편할 수도 있습니다.

```
import numpy as np

# 2행 2열의 행렬 y 생성
y = np.array([[1, 2], [3, 4]])
print("행렬 y:\n", y)

# 첫 번째 행 선택
first_row = y[0, :]
print("첫 번째 행:\n", first_row)
print("첫 번째 행의 차원:", first_row.shape)
```

```
행렬 y:
 [[1 2]
 [3 4]]
첫 번째 행:
 [1 2]
첫 번째 행의 차원: (2,)
```

이런 경우에는 np.newaxis를 사용하여 차원을 유지할 수 있습니다. np.newaxis는 NumPy에서 배열의 차원을 확장할 때 사용하는 키워드입니다. 이를 통해 배열에 새로운 축을 추가하여, 배열의 차원이 증가시킬 수 있습니다.

따라서, 위의 예제의 경우, 1차원으로 줄어든 벡터의 차원을 다시 증가 시키는 것이죠.

```
# 첫 번째 행을 2차원 배열로 유지
first_row_2d = y[0, :][np.newaxis, :]
print("첫 번째 행을 뽑아 가로 차원으로 확장:\n", first_row_2d)
print("모양 확인:", first_row_2d.shape)
```

```
첫 번째 행을 뽑아 가로 차원으로 확장:
 [[1 2]]
모양 확인: (1, 2)
```

np.newaxis의 위치에 따라서 확장되는 차원이 달라집니다.

```
# 첫 번째 행을 2차원 배열로 유지
first_row_2d = y[0, :][:, np.newaxis]
print("첫 번째 행을 뽑아 세로 차원으로 확장:\n", first_row_2d)
print("모양 확인:", first_row_2d.shape)
```

```
첫 번째 행을 뽑아 세로 차원으로 확장:
 [[1]
 [2]]
모양 확인: (2, 1)
```

6.0.10 고차원 행렬, 배열

행렬은 2차원의 데이터 구조입니다. 그러나 때때로 데이터를 3차원 이상으로 표현해야 할 필요가 있습니다. NumPy는 이런 고차원 행렬도 무리없이 확장가능합니다.

6.0.10.a `array()` 함수로 3차원 배열 만들기

예를 들어, 두 개의 2x3 행렬 `mat1`과 `mat2`를 가지고 있을 때, 이를 합쳐서 3차원 배열로 만들어봅시다.

```
import numpy as np

# 두 개의 2x3 행렬 생성
mat1 = np.arange(1, 7).reshape(2, 3)
mat2 = np.arange(7, 13).reshape(2, 3)
print("행렬 mat1:\n", mat1)
print("행렬 mat2:\n", mat2)

# 3차원 배열로 합치기
my_array = np.array([mat1, mat2])
print("3차원 배열 my_array:\n", my_array)
print("3차원 배열의 크기 (shape):", my_array.shape)
```

```
행렬 mat1:
 [[1 2 3]
 [4 5 6]]
행렬 mat2:
 [[ 7  8  9]
 [10 11 12]]
3차원 배열 my_array:
 [[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
```

```
  [10 11 12]]]
3차원 배열의 크기 (shape): (2, 2, 3)
```

마지막에 나온 배열의 크기인 $(2, 2, 3)$의 의미는 무엇일까요? 바로 2 by 3 행렬(마지막 2개 숫자)이 2장이 겹쳐있다!(첫번째 숫자 2)라는 의미가 됩니다.

6.0.11 배열 다루기

배열은 행렬과 비슷한 방식으로 다룰 수 있으며, 행렬에서 사용되는 인덱싱 및 필터링 방식이 그대로 적용됩니다.

```python
import numpy as np

# 두 개의 2x3 행렬 생성
mat1 = np.arange(1, 7).reshape(2, 3)
mat2 = np.arange(7, 13).reshape(2, 3)

# 3차원 배열로 합치기
my_array = np.array([mat1, mat2])
print("3차원 배열 my_array:\n", my_array)

# 첫 번째 2차원 배열 선택
first_slice = my_array[0, :, :]
print("첫 번째 2차원 배열:\n", first_slice)

# 두 번째 차원의 세 번째 요소를 제외한 배열 선택
filtered_array = my_array[:, :, :-1]
print("세 번째 요소를 제외한 배열:\n", filtered_array)
```

```
3차원 배열 my_array:
 [[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
첫 번째 2차원 배열:
 [[1 2 3]
 [4 5 6]]
세 번째 요소를 제외한 배열:
 [[[ 1  2]
  [ 4  5]]

 [[ 7  8]
  [10 11]]]
```

6.0.11.a 슬라이싱에서 `:-1`의 의미 알아두기

`:-1`은 슬라이싱에서 처음부터 (start가 생략된 경우 첫 번째 요소) 마지막 요소 직전 (end가 -1)까지 선택하는 것을 의미합니다. 이는 파이썬의 슬라이싱 규칙 중 하나입니다.

- `::` 해당 차원의 모든 요소를 선택합니다.
- `:-1`: 해당 차원의 첫 번째 요소부터 마지막 요소 직전까지 선택합니다.

배열에는 또한 `transpose()` 메서드를 사용하여 차원을 바꿀 수 있습니다. 이는 행렬의 전치 (transpose)를 확장한 개념입니다.

```
# 원래 배열
print("원래 배열 my_array:\n", my_array)
my_array.shape

# 차원 변경
transposed_array = my_array.transpose(0, 2, 1)
print("차원이 변경된 배열:\n", transposed_array)
```

```
원래 배열 my_array:
 [[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
차원이 변경된 배열:
 [[[ 1  4]
  [ 2  5]
  [ 3  6]]

 [[ 7 10]
  [ 8 11]
  [ 9 12]]]
```

여기서 `(0, 2, 1)`은 원래 차원의 순서 `(0, 1, 2)`를 `(0, 2, 1)`로 바꾸라는 것을 의미합니다. 즉, 두번째 차원과 세번째 차원의 위치를 바꿔줍니다.

따라서, 원래는 2 by 3 행렬이 2장 겹쳐있던 것이 3 by 2 행렬 두개가 겹쳐있는 것으로 바뀌게 되는 것이죠!

6.0.12 사진은 배열이다.

6.0.12.a 이미지 다운로드

다음 코드를 통하여 이미지를 다운로드 받도록 하겠습니다.

```
import urllib.request

img_url = "https://bit.ly/3ErnM2Q"
urllib.request.urlretrieve(img_url, "jelly.png")
```

```
('jelly.png', <http.client.HTTPMessage at 0x12c5e3e8688>)
```

6.0.12.b 이미지 읽기

다운로드 받은 파일은 `imageio` 모듈을 통해 Python으로 불러올 수 있습니다.

```
import imageio
import numpy as np

# 이미지 읽기
jelly = imageio.imread("jelly.png")
print("이미지 클래스:", type(jelly))
print("이미지 차원:", jelly.shape)
print("이미지 첫 4x4 픽셀, 첫 번째 채널:\n", jelly[:4, :4, 0])
```

```
이미지 클래스: <class 'numpy.ndarray'>
이미지 차원: (88, 50, 4)
이미지 첫 4x4 픽셀, 첫 번째 채널:
 [[156 151 146 142]
 [118 121 123 121]
 [105 107 110 110]
 [104 108 110 112]]
```

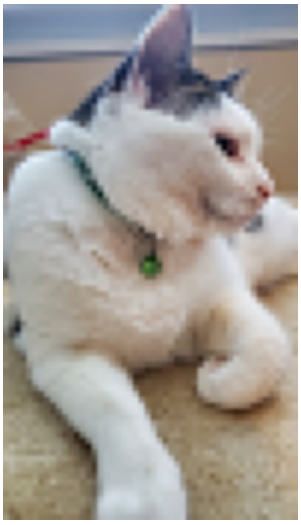
```
C:\Users\issac\.conda\envs\adp-python\lib\site-packages\ipykernel_launcher.py:
5: DeprecationWarning: Starting with ImageIO v3 the behavior of this function
will switch to that of iio.v3.imread. To keep the current behavior (and make this
warning disappear) use `import imageio.v2 as imageio` or call `imageio.v2.imread`
directly.
  """
```

위의 결과로부터 `jelly`가 배열임을 알 수 있습니다. 앞에서 배운 코드들을 통하여 이미지를 이리저리 바꿔 볼 수 있습니다.

- **RGB**, Opacity
  - 첫 3개의 채널은 해당 위치의 빨강, 녹색, 파랑의 색깔 강도를 숫자로 표현합니다.
  - 마지막 채널은 투명도를 결정합니다.

```
import matplotlib.pyplot as plt
```

```
# 원본 이미지 표시
plt.imshow(jelly)
plt.axis('off')
plt.show()
```

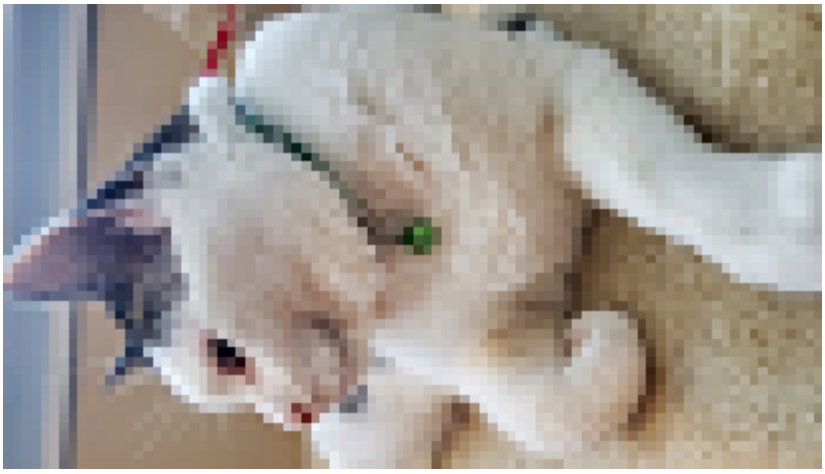


- 사진 뒤집기

  앞에서 배운 배열 뒤집기를 이용한 사진 돌리기

```
# 배열 뒤집기 (전치)
t_jelly = np.transpose(jelly, (1, 0, 2))

# 뒤집힌 이미지 표시
plt.imshow(t_jelly)
plt.axis('off')
plt.show()
```



- 사진 흑백으로 만들기

```
# 흑백으로 변환
bw_jelly = np.mean(jelly[:, :, :3], axis=2)

# 흑백 이미지 표시
plt.imshow(bw_jelly, cmap='gray')
plt.axis('off')
plt.show()
```

6.0.12.c 코드 해석

1. `jelly[:, :, :3]`:
   - `jelly`는 원본 이미지 배열입니다.

- 이 배열은 3차원 배열로, 첫 번째와 두 번째 차원은 이미지의 높이와 너비를 나타내고, 세 번째 차원은 색상 채널 (R, G, B, Alpha)을 나타냅니다.
- `:3`은 세 번째 차원에서 첫 번째, 두 번째, 세 번째 채널 (R, G, B)을 선택합니다. 즉, Alpha 채널 (투명도)은 제외하고 RGB 채널만 선택합니다.

2. `np.mean(jelly[:, :, :3], axis=2)`:
   - `np.mean` 함수는 배열의 평균을 계산합니다.
   - `axis=2`는 평균을 계산할 축을 지정합니다. 여기서 `axis=2`는 세 번째 축 (색상 채널)을 의미합니다.
   - 즉, 각 픽셀의 RGB 값의 평균을 계산하여 흑백 값을 만듭니다.

따라서 색상 축을 중심으로 평균을 내었기 때문에 88행, 50열 크기의 행렬이 하나 반환이 되는 것이고, 이것을 시각화 하면 흑백 사진이 탄생하는 것 입니다.

```
bw_jelly.shape
```

```
(88, 50)
```

6.0.13 넘파이 배열 기본 제공 함수들 정리

넘파이 배열에 제공되는 대표적인 함수들(메서드)은 다음과 같습니다.

메서드	문법	설명
sum()	sum(axis=0)	배열 원소들의 합계를 반환합니다. axis=0은 열별 합계를, axis=1은 행별 합계를 의미합니다.
mean()	mean(axis=0)	배열 원소들의 평균을 반환합니다. axis=0은 열별 평균을, axis=1은 행별 평균을 의미합니다.
max()	max(axis=0)	배열 원소들의 최댓값을 반환합니다. axis=0은 열별 최댓값을, axis=1은 행별 최댓값을 의미합니다.
min()	min(axis=0)	배열 원소들의 최솟값을 반환합니다. axis=0은 열별 최솟값을, axis=1은 행별 최솟값을 의미합니다.
std()	std(ddof=0)	배열 원소들의 표준편차를 반환합니다. ddof 옵션을 사용하여 자유도를 조정할 수 있습니다.
var()	var(ddof=0)	배열 원소들의 분산을 반환합니다. ddof 옵션을 사용하여 자유도를 조정할 수 있습니다.
cumsum()	cumsum(axis=0)	배열 원소들의 누적 합계를 반환합니다. axis=0은 열별 누적 합계를, axis=1은 행별 누적 합계를 의미합니다.
cumprod()	cumprod(axis=0)	배열 원소들의 누적 곱을 반환합니다. axis=0은 열별 누적 곱을, axis=1은 행별 누적 곱을 의미합니다.
argmax()	argmax(axis=0)	배열 원소들 중 최댓값의 인덱스를 반환합니다. axis=0은 열별 최댓값의 인덱스를, axis=1은 행별 최댓값의 인덱스를 의미합니다.
argmin()	argmin(axis=0)	배열 원소들 중 최솟값의 인덱스를 반환합니다. axis=0은 열별 최솟값의 인덱스를, axis=1은 행별 최솟값의 인덱스를 의미합니다.
reshape()	reshape(newshape)	배열의 형상을 변경합니다.
transpose()	transpose(*axes)	배열을 전치합니다.
flatten()	flatten()	1차원 배열로 변환합니다.
clip()	clip(min, max)	배열 원소들을 주어진 범위로 자릅니다.
tolist()	tolist()	배열을 리스트로 변환합니다.
astype()	astype(dtype)	배열 원소들의 타입을 변환합니다.
copy()	copy()	배열의 복사본을 반환합니다.
~~sort()~~ argsort()	~~sort(axis=-1)~~ argsort(axis=-1)	~~배열을 정렬합니다.~~ 배열 원소들의 정렬된 인덱스를 반환합니다.

6.0.13.a 각 함수들 간략 설명

6.0.13.a.a `sum()`

배열 원소들의 합계를 반환합니다. `axis=0`은 열별 합계를, `axis=1`은 행별 합계를 의미합니다.

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
print("전체 합계:", a.sum())
print("열별 합계:", a.sum(axis=0))
print("행별 합계:", a.sum(axis=1))
```

```
전체 합계: 21
열별 합계: [5 7 9]
행별 합계: [ 6 15]
```

6.0.13.a.b `mean()`

배열 원소들의 평균을 반환합니다. `axis=0`은 열별 평균을, `axis=1`은 행별 평균을 의미합니다.

```
print("전체 평균:", a.mean())
print("열별 평균:", a.mean(axis=0))
print("행별 평균:", a.mean(axis=1))
```

```
전체 평균: 3.5
열별 평균: [2.5 3.5 4.5]
행별 평균: [2. 5.]
```

6.0.13.a.c `max()`

배열 원소들의 최댓값을 반환합니다. `axis=0`은 열별 최댓값을, `axis=1`은 행별 최댓값을 의미합니다.

```
print("전체 최댓값:", a.max())
print("열별 최댓값:", a.max(axis=0))
print("행별 최댓값:", a.max(axis=1))
```

```
전체 최댓값: 6
열별 최댓값: [4 5 6]
행별 최댓값: [3 6]
```

6.0.13.a.d `min()`

배열 원소들의 최솟값을 반환합니다. `axis=0`은 열별 최솟값을, `axis=1`은 행별 최솟값을 의미합니다.

```
print("전체 최솟값:", a.min())
print("열별 최솟값:", a.min(axis=0))
print("행별 최솟값:", a.min(axis=1))
```

```
전체 최솟값: 1
열별 최솟값: [1 2 3]
행별 최솟값: [1 4]
```

6.0.13.a.e `std()`

배열 원소들의 표준편차를 반환합니다. `ddof` 옵션을 사용하여 자유도를 조정할 수 있습니다. `ddof=1`의 경우 n-1로 나눠준 것을 표현하므로 표본 표준편차를 의미합니다.

```
print("표본 표준편차 (ddof=1):", a.std(ddof=1))
```

```
표본 표준편차 (ddof=1): 1.8708286933869707
```

6.0.13.a.f `var()`

배열 원소들의 분산을 반환합니다. `ddof` 옵션을 사용하여 자유도를 조정할 수 있습니다.

```
print("표본 분산 (ddof=1):", a.var(ddof=1))
```

```
표본 분산 (ddof=1): 3.5
```

6.0.13.a.g `cumsum()`

배열 원소들의 누적 합계를 반환합니다. `axis=0`은 열별 누적 합계를, `axis=1`은 행별 누적 합계를 의미합니다.

```
print("전체 누적 합계:", a.cumsum())
print("열별 누적 합계:", a.cumsum(axis=0))
print("행별 누적 합계:", a.cumsum(axis=1))
```

```
전체 누적 합계: [ 1  3  6 10 15 21]
열별 누적 합계: [[1 2 3]
 [5 7 9]]
```

```
행별 누적 합계: [[ 1  3  6]
 [ 4  9 15]]
```

6.0.13.a.h `cumprod()`

배열 원소들의 누적 곱을 반환합니다. `axis=0`은 열별 누적 곱을, `axis=1`은 행별 누적 곱을 의미합니다.

```
print("전체 누적 곱:", a.cumprod())
print("열별 누적 곱:", a.cumprod(axis=0))
print("행별 누적 곱:", a.cumprod(axis=1))
```

```
전체 누적 곱: [  1   2   6  24 120 720]
열별 누적 곱: [[ 1  2  3]
 [ 4 10 18]]
행별 누적 곱: [[  1   2   6]
 [  4  20 120]]
```

6.0.13.a.i `argmax()`

배열 원소들 중 최댓값의 인덱스를 반환합니다. `axis=0`은 열별 최댓값의 인덱스를, `axis=1`은 행별 최댓값의 인덱스를 의미합니다.

```
print("최댓값의 인덱스 (전체):", a.argmax())
print("최댓값의 인덱스 (열별):", a.argmax(axis=0))
print("최댓값의 인덱스 (행별):", a.argmax(axis=1))
```

```
최댓값의 인덱스 (전체): 5
최댓값의 인덱스 (열별): [1 1 1]
최댓값의 인덱스 (행별): [2 2]
```

6.0.13.a.j `argmin()`

배열 원소들 중 최솟값의 인덱스를 반환합니다. `axis=0`은 열별 최솟값의 인덱스를, `axis=1`은 행별 최솟값의 인덱스를 의미합니다.

```
print("최솟값의 인덱스 (전체):", a.argmin())
print("최솟값의 인덱스 (열별):", a.argmin(axis=0))
print("최솟값의 인덱스 (행별):", a.argmin(axis=1))
```

```

```

```
최솟값의 인덱스 (전체): 0
최솟값의 인덱스 (열별): [0 0 0]
최솟값의 인덱스 (행별): [0 0]
```

6.0.13.a.k `reshape()`

배열의 형상을 변경합니다.

```
b = np.array([1, 2, 3, 4, 5, 6])
print("원본 배열:\n", b)
print("형상 변경:\n", b.reshape((2, 3)))
```

```
원본 배열:
 [1 2 3 4 5 6]
형상 변경:
 [[1 2 3]
 [4 5 6]]
```

6.0.13.a.l `transpose()`

배열을 전치합니다.

```
c = np.array([[1, 2, 3], [4, 5, 6]])
print("원본 배열:\n", c)
print("전치 배열:\n", c.transpose())
```

```
원본 배열:
 [[1 2 3]
 [4 5 6]]
전치 배열:
 [[1 4]
 [2 5]
 [3 6]]
```

6.0.13.a.m `flatten()`

1차원 배열로 변환합니다.

```
print("1차원 배열:\n", c.flatten())
```

```
1차원 배열:
 [1 2 3 4 5 6]
```

6.0.13.a.n clip()

clip 함수는 배열의 각 원소들을 주어진 최소값과 최대값의 범위로 자르는 역할을 합니다. 주어진 최소값보다 작은 원소는 최소값으로, 최대값보다 큰 원소는 최대값으로 변환합니다.

예제를 통해 자세히 설명드리겠습니다. 이 코드에서 d.clip(2, 4)는 배열 d의 각 원소를 최소값 2와 최대값 4로 제한합니다.

```
d = np.array([1, 2, 3, 4, 5])
print("클립된 배열:", d.clip(2, 4))
```

```
클립된 배열: [2 2 3 4 4]
```

배열 d의 원소 1, 2, 3, 4, 5를 보겠습니다.

- 1은 최소값 2보다 작으므로 2로 변환됩니다.
- 2는 범위 내에 있으므로 그대로 유지됩니다.
- 3은 범위 내에 있으므로 그대로 유지됩니다.
- 4는 범위 내에 있으므로 그대로 유지됩니다.
- 5는 최대값 4보다 크므로 4로 변환됩니다.

결과적으로 clip 함수는 배열의 원소들을 array([2, 2, 3, 4, 4])로 변환합니다.

6.0.13.a.o tolist()

배열을 리스트로 변환합니다.

```
print("리스트:", d.tolist())
```

```
리스트: [1, 2, 3, 4, 5]
```

6.0.13.a.p astype()

배열 원소들의 타입을 변환합니다.

```
e = np.array([1.1, 2.2, 3.3])
print("정수형 배열:", e.astype(int))
```

```
정수형 배열: [1 2 3]
```

6.0.13.a.q copy()

배열의 복사본을 반환합니다.

```

```

```
f = d.copy()
print("복사본 배열:", f)
```

```
복사본 배열: [1 2 3 4 5]
```

- 얕은 복사와 깊은 복사 개념 이해하기

배열을 복사할 때 얕은 복사와 깊은 복사의 차이를 이해하는 것이 중요합니다. 얕은 복사 관련 코드를 보겠습니다.

```
import numpy as np

d = np.array([1, 2, 3, 4, 5])
f = d
f[0] = 10
print("d:", d)  # 출력: d: [10  2  3  4  5]
print("f:", f)  # 출력: f: [10  2  3  4  5]
```

```
d: [10  2  3  4  5]
f: [10  2  3  4  5]
```

변수 d와 f가 연결되어 있어서 f의 값을 변경하면, 연결되어 있는 d의 값 역시 변하는 것은 알 수 있습니다. 반면, 깊은 복사는 어떻게 다를까요?

```
import numpy as np

d = np.array([1, 2, 3, 4, 5])
f = d.copy()
f[0] = 10
print("d:", d)  # 출력: d: [1 2 3 4 5]
print("f:", f)  # 출력: f: [10  2  3  4  5]
```

```
d: [1 2 3 4 5]
f: [10  2  3  4  5]
```

f의 값이 변해도 d값이 변하지 않는, 독립적인 변수가 된 것을 확인 할 수 있습니다. 따라서, 변수를 복사해올때, 두 차이를 정확히 이해하고 사용하는 것이 좋겠죠?

6.0.13.a.r sort()

배열을 정렬합니다.

```

```

```
g = np.array([3, 1, 2])
g.sort()
print("정렬된 배열:", g)
```

```
정렬된 배열: [1 2 3]
```

6.0.13.a.s `argsort()`

`argsort()` 함수는 배열의 원소들을 정렬했을 때의 인덱스를 반환합니다.

```
h = np.array([3, 1, 2])
print("정렬된 인덱스:", h.argsort())
```

```
정렬된 인덱스: [1 2 0]
```

위 코드를 실행하면, 배열 `h`의 원소들이 정렬되었을 때의 인덱스 순서가 반환됩니다. 이 결과는 배열 `[3, 1, 2]`가 `[1, 2, 3]`으로 정렬될 때, 원래 배열 `h`의 인덱스가 `[1, 2, 0]` 순서로 변경된다는 것을 의미합니다.

다음 예제는 `argsort()`를 사용하여 배열의 정렬된 인덱스 응용 코드입니다.

```
import numpy as np

# 배열 생성
h = np.array([10, 5, 8, 1, 7])

# 배열을 정렬했을 때의 인덱스
sorted_indices = h.argsort()
print("정렬된 인덱스:", sorted_indices)

# 정렬된 배열을 인덱스를 사용해 출력
sorted_h = h[sorted_indices]
print("정렬된 배열:", sorted_h)
```

```
정렬된 인덱스: [3 1 4 2 0]
정렬된 배열: [ 1  5  7  8 10]
```

위 코드를 실행하면, 배열 `h`의 원소들이 정렬되었을 때의 인덱스 순서와, 정렬된 배열을 얻을 수 있습니다.

6.0.14 마치며

행렬은 데이터를 사각형으로 모아놓은 것이라 생각할 수 있습니다. 이러한 사각형 데이터는 분석을 할 때 가장 많이 보는 형태입니다. 따라서 행렬의 인덱싱과 필터링 스킬들이 나중에 데이터 분석에서 데이터를 불러온 뒤, 그대로 사용되는 것을 보실 수 있을 것 입니다.

6.0.15 연습 문제

6.0.15.a 연습 문제 1

다음과 같은 행렬 A를 만들어 보세요!

```
행렬 A:
 [[3 5 7]
 [2 3 6]]
```

6.0.15.b 연습 문제 2

다음과 같이 행렬 B가 주어졌을 때, 2번째, 4번째, 5번째 행 만을 선택하여 3 by 4 행렬을 만들어보세요.

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

6.0.15.c 연습 문제 3

연습 문제 2에서 주어진 행렬 B에서 3번째 열의 값이 3보다 큰 행들만 골라내 보세요.

6.0.15.d 연습 문제 4

연습 문제 2에서 주어진 행렬 B의 행별로 합계를 내고 싶을 때 `rowSums()` 함수를 사용할 수 있습니다.

```
# 각 행별 합계 계산
row_sums = np.sum(B, axis=1)
print("각 행별 합계:\n", row_sums)
```

```
각 행별 합계:
 [33 16 21 22 18]
```

각 행 별 합이 20보다 크거나 같은 행 만을 걸러내어 새로운 행렬을 작성해보세요.

6.0.15.e 연습 문제 5

이전 문제에서는 각 행별 합이 20보다 크거나 같은 행을 걸러내어 행렬을 만들었습니다. 이번에는 원래 주어진 행렬 B에서 각 열별 평균이 5보다 크거나 같은 열이 몇 번째 열에 위치하는지 `np.mean()` 함수를 사용하여 알아내는 코드를 작성해보세요.

6.0.15.f 연습 문제 6

연습 문제 2에서 주어진 행렬 B를 5보다 크거나 같은지 물어보는 조건문을 작성하여 돌려보면 다음과 같은 TRUE, FALSE 행렬을 갖게 됩니다.

```
# 5보다 크거나 같은지 확인하는 조건문
B >= 5
print("5보다 크거나 같은지 여부:\n", B >= 5)
```

```
5보다 크거나 같은지 여부:
 [[ True  True  True  True]
 [False False  True  True]
 [ True  True False  True]
 [False  True  True  True]
 [ True False False False]]
```

TRUE는 1, FALSE는 0으로 처리되는 성질을 사용하면, 다음과 같이 행렬 B의 숫자들 중 5보다 크거나 같은 숫자들이 11개 들어있다는 것을 확인 할 수 있습니다.

```
# 5보다 크거나 같은 숫자의 합
np.sum(B >= 5)
print("5보다 크거나 같은 숫자의 개수:\n", np.sum(B >= 5))
```

```
5보다 크거나 같은 숫자의 개수:
 13
```

행렬 B의 각 행에 7보다 큰 숫자가 하나라도 들어있는 행을 걸러내는 코드를 작성해 주세요.

6.0.16 해답

6.0.16.a 연습 문제 1

```
A = np.array([[3, 5, 7],
              [2, 3, 6]])
print("행렬 A:\n", A)
```

```

```

```
행렬 A:
 [[3 5 7]
 [2 3 6]]
```

6.0.16.b 연습 문제 2

```
B_selected = B[[1, 3, 4], :]
print("선택된 행렬:\n", B_selected)
```

```
선택된 행렬:
 [[2 4 5 5]
 [2 6 8 6]
 [9 3 4 2]]
```

6.0.16.c 연습 문제 3

```
B_filtered = B[B[:, 2] > 3, :]
print("3번째 열의 값이 3보다 큰 행들:\n", B_filtered)
```

```
3번째 열의 값이 3보다 큰 행들:
 [[ 8 10  7  8]
 [ 2  4  5  5]
 [ 2  6  8  6]
 [ 9  3  4  2]]
```

6.0.16.d 연습 문제 4

```
B_row_sums_filtered = B[row_sums >= 20, :]
print("합계가 20보다 크거나 같은 행들:\n", B_row_sums_filtered)
```

```
합계가 20보다 크거나 같은 행들:
 [[ 8 10  7  8]
 [ 7  6  1  7]
 [ 2  6  8  6]]
```

6.0.16.e 연습 문제 5

```
# 각 열별 평균 계산
col_means = np.mean(B, axis=0)
print("각 열별 평균:\n", col_means)

# 평균이 5보다 큰 열 선택
```

```
col_indices = np.where(col_means > 5)[0]
print("평균이 5보다 큰 열의 인덱스:\n", col_indices)
```

```
각 열별 평균:
 [5.6 5.8 5.  5.6]
평균이 5보다 큰 열의 인덱스:
 [0 1 3]
```

6.0.16.f 연습 문제 6

```
print("7보다 큰 숫자가 있는 행들:\n", B[np.sum(B > 7, axis=1) > 0, :])
```

```
7보다 큰 숫자가 있는 행들:
 [[ 8 10  7  8]
 [ 2  6  8  6]
 [ 9  3  4  2]]
```