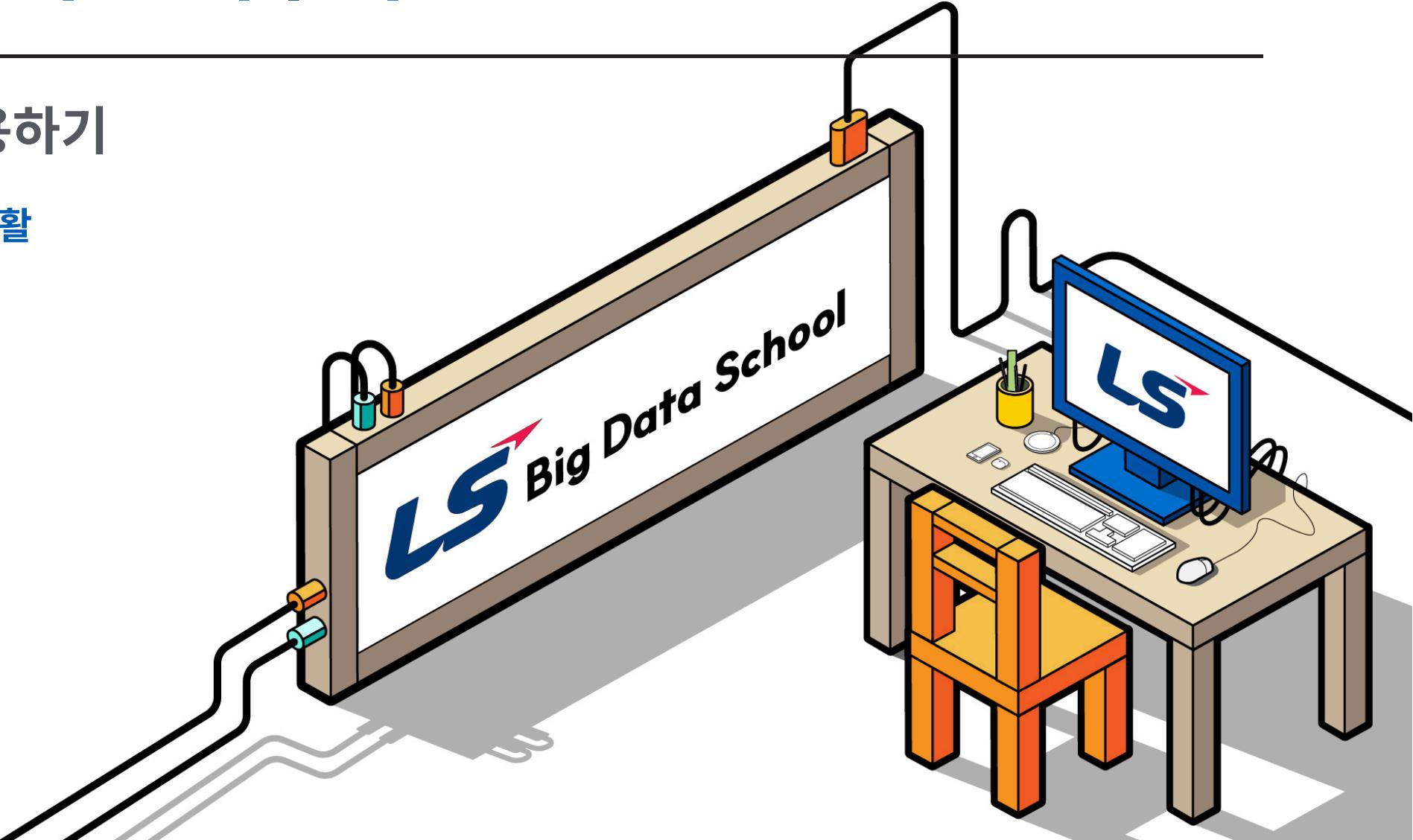


파이썬 기초 배우기

넘파이 활용하기

슬기로운통계생활



코스 훑어보기.

파이썬의 넘파이 라이브러리에 대해 학습합니다.



NumPy 소개

NumPy(Numerical Python의 약자)는 파이썬에서 강력한 수치 계산을 수행하기 위해 개발된 라이브러리입니다. 주로 다음과 같은 기능을 제공합니다:

- **다차원 배열 객체**: 고성능의 다차원 배열(ndarray)을 지원하며, 이를 통해 벡터화된 연산을 수행할 수 있어 계산 속도가 매우 빠릅니다.
- **방대한 수학 함수 라이브러리**: 선형 대수, 통계, 푸리에 변환 등과 같은 수학적 연산을 위한 함수를 대규모로 제공합니다.
- **배열 기반의 데이터 처리 도구**: 데이터 조작, 정제, 부분집합 생성, 필터링, 변형, 그리고 다른 종류의 계산을 위한 편리한 방법을 제공합니다.



설치 방법

NumPy를 설치하는 가장 간단한 방법은 pip, 파이썬의 패키지 관리자를 사용하는 것입니다. 대부분의 파이썬 설치에는 pip가 포함되어 있습니다.

pip를 이용한 설치

NumPy를 설치하려면, 터미널이나 명령 프롬프트에서 다음 명령을 입력하세요:

```
pip install numpy
```

Mini Conda를 이용한 설치

Mini Conda를 사용하는 경우, NumPy는 대부분 기본적으로 설치되어 있습니다만, NumPy를 설치하거나 업데이트하려면, 다음 명령을 사용할 수 있습니다:

```
conda activate '원하는 가상환경'  
conda install numpy
```



NumPy 불러오기

NumPy를 설치한 후에는, 파이썬 스크립트나 인터프리터에서 간단히 임포트하여 사용할 수 있습니다. 여기서 np는 numpy의 약자입니다.

```
import numpy as np
```



NumPy 배열 생성

Python의 NumPy 라이브러리를 사용하여 벡터(vector)를 다루는 방법에 대해 살펴봅니다. 벡터는 동일한 데이터 타입의 값들을 순서대로 나열한 것입니다. 벡터를 만들기 위해 NumPy의 `np.array()` 함수를 활용합니다. 벡터를 생성할 때는 동일한 데이터 타입의 값을 사용해야 합니다.

```
import numpy as np

# 벡터 생성하기 예제
a = np.array([1, 2, 3, 4, 5]) # 숫자형 벡터
b = np.array(["apple", "banana", "orange"])
c = np.array([True, False, True, True]) #
print("Numeric Vector:", a)
```

```
## Numeric Vector: [1 2 3 4 5]
```

```
print("String Vector:", b)
```

```
## String Vector: ['apple' 'banana' 'orange']
```

```
print("Boolean Vector:", c)
```

```
## Boolean Vector: [ True False  True  True]
```



빈 배열 선언 후 채우기

NumPy에서 빈 배열을 생성하는 방법은 `np.empty()` 또는 `np.zeros()` 함수를 사용할 수 있습니다.

```
# 빈 배열 생성  
x = np.empty(3)  
print("빈 벡터 생성하기:", x)
```

```
## 빈 벡터 생성하기: [1.04e-322 1.48e-322 2.03e-322]
```

```
# 배열 채우기  
x[0] = 3  
x[1] = 5  
x[2] = 3  
print("채워진 벡터:", x)
```

```
## 채워진 벡터: [3. 5. 3.]
```



배열을 생성하면서 채우기

NumPy에서 배열을 생성하면서 채우는 방법에는 여러 가지가 있습니다.

np.arange() 함수

np.arange() 함수는 일정한 간격으로 숫자를 생성하여 배열을 반환합니다. 이 함수는 Python의 내장 함수 range()와 유사하지만, 배열을 반환하며 더 넓은 범위의 숫자 타입을 지원합니다.

문법:

```
np.arange([start, ]stop, [step, ]dtype=None)
```

1. start: 배열의 시작값, 생략 시 0부터 시작합니다.
2. stop: 배열 생성을 멈출 종료값, 이 값은 배열에 포함되지 않습니다.
3. step: 각 배열 요소 간의 간격, 기본값은 1입니다.
4. dtype: 배열의 데이터 타입을 명시적으로 지정, 생략 시 입력 데이터를 기반으로 유추합니다.



배열을 생성하면서 채우기

예제 코드 1. 0부터 10 미만까지의 정수 배열 생성

```
import numpy as np  
  
arr1 = np.arange(10)  
print("Array from 0 to 9:", arr1)
```

```
## Array from 0 to 9: [0 1 2 3 4 5 6 7 8 9]
```



배열을 생성하면서 채우기

예제 코드 2. 0부터 2 미만까지 0.5 간격으로 배열 생성

```
arr2 = np.arange(0, 2, 0.5)  
print("0부터 1.5까지 0.5 간격으로 발생:", arr2)
```

```
## 0부터 1.5까지 0.5 간격으로 발생: [0.  0.5 1.  1.5]
```



np.linspace() 함수

np.linspace() 함수는 지정된 시작점과 종료점 사이에서 균일한 간격의 숫자 배열을 생성합니다. 이 함수는 주로 데이터 플롯이나 수학적 계산에서 필요한 특정 개수의 포인트를 생성할 때 사용됩니다.

문법:

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

1. start: 시퀀스의 시작값입니다.
2. stop: 시퀀스의 종료값입니다. endpoint=True이면 이 값이 배열에 포함됩니다.
3. num: 생성할 샘플의 수, 기본값은 50입니다.
4. endpoint: True인 경우 stop이 마지막 샘플로 포함됩니다.
5. retstep: True인 경우 결과와 함께 샘플 간의 간격도 반환됩니다.
6. dtype: 배열의 데이터 타입을 지정합니다.



배열을 생성하면서 채우기

예제 코드 1. 0부터 1까지 총 5개의 요소로 구성된 배열 생성

```
linear_space1 = np.linspace(0, 1, 5)
print("0부터 1까지 5개 원소:", linear_space1)
```

```
## 0부터 1까지 5개 원소: [0.    0.25  0.5   0.75  1.  ]
```



배열을 생성하면서 채우기

예제 코드 2. endpoint 옵션 변경

0부터 1까지 총 5개의 요소로 구성되지만, 1은 포함하지 않는 배열을 생성합니다.

```
linear_space2 = np.linspace(0, 1, 5, endpoint=False)
print("0부터 1까지 5개 원소, endpoint 제외:", linear_space2)
```

```
## 0부터 1까지 5개 원소, endpoint 제외: [0.  0.2 0.4 0.6 0.8]
```



배열을 생성하면서 채우기

np.repeat() 함수, 값을 반복해서 벡터 만들기

NumPy 라이브러리를 사용하여 같은 숫자들로 벡터를 채워서 만드는 방법을 살펴봅니다.

np.repeat() 함수를 사용하여 반복된 요소로 배열을 생성할 수 있습니다.

문법:

```
np.repeat(a, repeats, axis=None)
```

- **a**: 반복할 입력 배열입니다.
- **repeats**: 각 요소를 반복할 횟수입니다.
- **axis**: 반복을 적용할 축을 지정합니다. 기본값은 None으로, 배열을 평평하게 만든 후 반복합니다.



배열을 생성하면서 채우기

예제 코드 1. 단일 값 반복

```
import numpy as np

# 숫자 8을 4번 반복
repeated_vals = np.repeat(8, 4)
print("Repeated 8 four times:", repeated_vals)
```

```
## Repeated 8 four times: [8 8 8 8]
```

배열을 생성하면서 채우기



예제 코드 2. 배열 반복

```
# 배열 [1, 2, 4]를 2번 반복
repeated_array = np.repeat([1, 2, 4], 2)
print("Repeated array [1, 2, 4] two times:", repeated_array)
```

```
## Repeated array [1, 2, 4] two times: [1 1 2 2 4 4]
```



예제 코드 3. 각 요소를 반복

NumPy에서 각 요소를 개별적으로 반복하려면 `np.repeat()` 함수의 `repeats` 인수를 배열로 사용하면 됩니다.

```
# 배열 [1, 2, 4]의 각 요소를 각각 1, 2, 3번 반복
repeated_each = np.repeat([1, 2, 4], repeats=[1, 2, 3])
print("Repeated each element in [1, 2, 4] two times:", repeated_each)
```

```
## Repeated each element in [1, 2, 4] two times: [1 2 2 4 4 4]
```



배열을 생성하면서 채우기

예제 코드 4. 벡터 전체를 반복해서 붙이기

NumPy에서 벡터 전체를 반복하려면 `np.tile()` 함수를 사용합니다.

```
# 배열 [1, 2, 4]를 2번 반복
repeated_whole = np.tile([1, 2, 4], 2)
print("벡터 전체를 두번 반복:", repeated_whole)
```

```
## 벡터 전체를 두번 반복: [1 2 4 1 2 4]
```



넘파이 벡터 길이 재는 방법

넘파이 배열의 길이를 재는 방법은 여러 가지가 있습니다. 가장 일반적으로 사용하는 방법은 `len()` 함수를 사용하는 것입니다.

`len()` 함수 사용하기

`len()` 함수는 배열의 첫 번째 차원의 길이를 반환합니다. 이는 1차원 배열의 경우 배열의 요소 수를 의미합니다.

```
import numpy as np  
  
# 1차원 배열  
a = np.array([1, 2, 3, 4, 5])  
len(a)
```

```
## 5
```

shape 속성 사용하기



shape 속성은 배열의 각 차원의 크기를 튜플 형태로 반환합니다. 이를 통해 배열의 전체 크기를 알 수 있습니다.

```
import numpy as np  
  
# 1차원 배열  
a = np.array([1, 2, 3, 4, 5])  
a.shape
```

```
## (5, )
```



배열의 전체 요소 수 구하기

배열의 전체 요소 수를 구하려면 `size` 속성을 사용할 수 있습니다. 이는 배열의 모든 요소의 개수를 반환합니다.

```
import numpy as np  
  
# 1차원 배열  
a = np.array([1, 2, 3, 4, 5])  
a.size
```

```
## 5
```



(미리 학습) 다차원 배열의 길이 재기

다차원 배열에서도 `len()` 함수, `shape` 속성, `size` 속성을 사용할 수 있습니다. `len()` 함수는 첫 번째 차원의 길이를 반환하며, `shape` 속성은 각 차원의 크기를 튜플로 반환하고, `size` 속성은 전체 요소의 개수를 반환합니다.

```
import numpy as np

# 2차원 배열
b = np.array([[1, 2, 3], [4, 5, 6]])
length = len(b)          # 첫 번째 차원의 길이
shape = b.shape           # 각 차원의 크기
size = b.size              # 전체 요소의 개수

length, shape, size
```

```
## (2, (2, 3), 6)
```



NumPy를 사용하여 벡터 연산하기

벡터 간 덧셈, 뺄셈, 곱셈, 나눗셈 등의 연산은 벡터의 각 요소에 대해 동시에 수행됩니다. 연산을 수행할 때는 벡터 간 길이가 같아야 합니다.

NumPy를 사용하여 벡터 연산 예제

```
import numpy as np

# 벡터 생성
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# 벡터 간 덧셈
add_result = a + b
print("벡터 덧셈:", add_result)
```

```
## 벡터 덧셈: [5 7 9]
```



NumPy를 사용하여 벡터 연산 예제

```
import numpy as np  
  
# 벡터 간 뺄셈  
sub_result = a - b  
print("벡터 뺄셈:", sub_result)
```

벡터 뺄셈: [-3 -3 -3]

```
# 벡터 간 곱셈  
mul_result = a * b  
print("벡터 곱셈:", mul_result)
```

벡터 곱셈: [4 10 18]



NumPy를 사용하여 벡터 연산 예제

```
import numpy as np

# 벡터 간 나눗셈
div_result = a / b
print("벡터 나눗셈:", div_result)
```

벡터 나눗셈: [0.25 0.4 0.5]

```
# 벡터 간 나머지 연산
mod_result = a % b
print("벡터 나머지 연산:", mod_result)
```

벡터 나머지 연산: [1 2 3]



벡터화(Vectorized) 코드

벡터화(Vectorized) 코드는 반복문을 사용하지 않고 벡터를 한 번에 처리할 수 있게 해줍니다. 이를 이용하여 여러 값을 동시에 처리할 수 있으며, 코드의 가독성과 성능을 높이는 역할을 합니다.

벡터 연산 예제

수학에서 배우는 벡터 연산을 기본적으로 지원하는 NumPy를 사용하여 벡터 연산을 수행해보겠습니다. 예를 들어, 다음과 같은 벡터 덧셈 연산을 고려해봅시다.

$$(1 \ 2 \ 4) + (2 \ 3 \ 5) = (3 \ 5 \ 9)$$

```
import numpy as np  
  
a = np.array([1, 2, 4])  
b = np.array([2, 3, 5])  
c = a + b  
print("벡터 덧셈:", c)
```

```
## 벡터 덧셈: [3 5 9]
```



벡터 연산 예제

다음과 같이 상수를 곱하는 연산 역시 마찬가지입니다.

$$2(1\ 2\ 4\ 5) = (2\ 4\ 8\ 10)$$

```
x = np.array([1, 2, 4, 5])
y = x * 2
print("상수 곱셈:", y)
```

```
## 상수 곱셈: [ 2  4  8 10]
```



NumPy의 브로드캐스팅(Broadcasting)

NumPy에서 브로드캐스팅은 길이가 다른 배열 간의 연산을 가능하게 해주는 강력한 메커니즘입니다. 작은 배열이 큰 배열의 길이에 맞추어 자동으로 확장되어 연산이 수행됩니다.

파이썬의 루프를 사용하지 않고 C 수준에서 효율적인 배열 연산을 가능하게 하여, 데이터 복사를 최소화하고 연산을 최적화합니다. 이를 통해 벡터 연산을 간단하게 처리할 수 있습니다.

브로드캐스팅의 기본 원리

브로드캐스팅은 두 배열의 차원을 비교할 때, 끝 차원부터 시작하여 앞으로 진행합니다. 연산이 가능하려면, 각 차원에서:

- 차원의 크기가 같거나
- 차원 중 하나의 크기가 1인 경우

위 조건을 만족해야 합니다. 이 조건을 충족하지 않을 경우, `ValueError`가 발생하여 두 배열이 브로드캐스팅할 수 없다는 예외가 발생합니다.



브로드캐스팅 안되는 경우

NumPy에서 배열 간 연산을 수행할 때, 배열의 `shape`이 중요합니다. 배열의 차원과 크기가 맞지 않으면 연산이 불가능하며, `ValueError`가 발생합니다.

```
import numpy as np  
  
# 길이가 다른 두 벡터  
a = np.array([1, 2, 3, 4])  
b = np.array([1, 2])  
  
# NumPy 브로드캐스팅을 사용한 벡터 덧셈  
result = a + b
```

```
## ValueError: operands could not be broadcast
```

```
print("브로드캐스팅 결과:", result)
```

```
## 브로드캐스팅 결과: 25
```



브로드캐스팅 안되는 경우

이 경우, 배열의 `shape`을 맞춰 연산을 수행할 수 있습니다:

```
print("a의 shape:", a.shape)
```

```
## a의 shape: (4, )
```

```
print("b의 shape:", b.shape)
```

```
## b의 shape: (2, )
```

```
b_repeated = np.tile(b, 2) # b 배열을 반복 확장  
print("반복된 b 배열:", b_repeated)
```

```
## 반복된 b 배열: [1 2 1 2]
```

```
result = a + b_repeated # 브로드캐스팅을 사용한!  
print("브로드캐스팅 결과:", result)
```

```
## 브로드캐스팅 결과: [2 4 4 6]
```

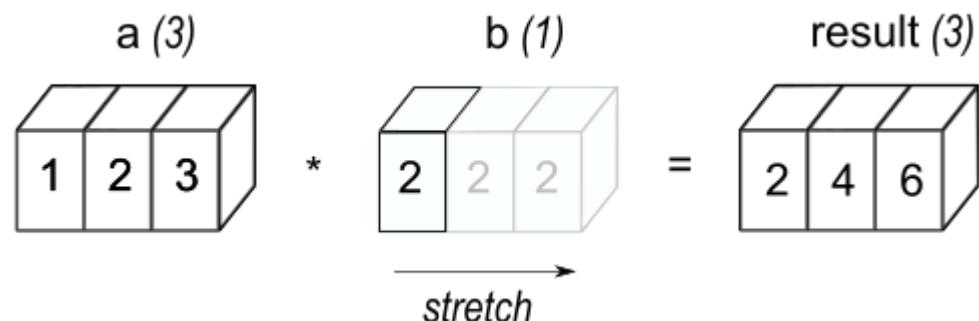


브로드캐스팅이 되는 경우

가장 간단한 브로드캐스팅 예는 스칼라 값과 배열을 연산할 때 발생합니다.

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

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



브로드캐스팅이 되는 경우



예제: 2차원 배열과 1차원 배열의 덧셈

```
import numpy as np  
  
# 2차원 배열 생성  
  
matrix = np.array([[ 0.0,  0.0,  0.0],  
                  [10.0, 10.0, 10.0],  
                  [20.0, 20.0, 20.0],  
                  [30.0, 30.0, 30.0]])  
  
# 1차원 배열 생성  
  
vector = np.array([1.0, 2.0, 3.0])  
print(matrix.shape, vector.shape)
```

```
## (4, 3) (3, )
```

```
# 브로드캐스팅을 이용한 배열 덧셈  
  
result = matrix + vector  
print("브로드캐스팅 결과:\n", result)
```

```
## 브로드캐스팅 결과:  
## [[ 1.  2.  3. ]  
## [11. 12. 13. ]  
## [21. 22. 23. ]  
## [31. 32. 33. ]]
```

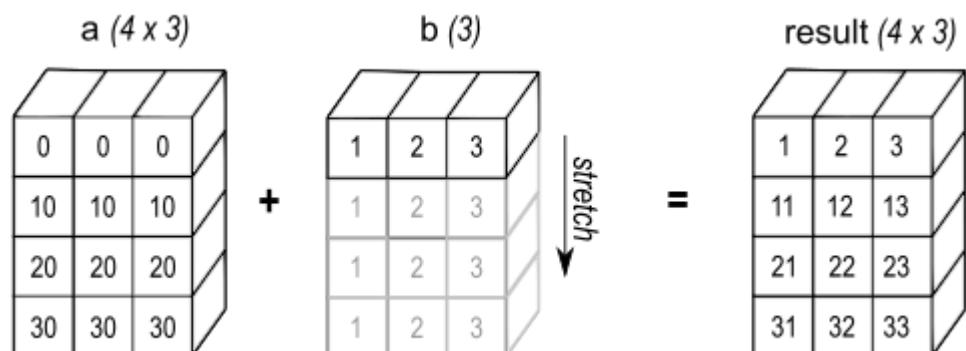


브로드캐스팅이 되는 경우

예제: 2차원 배열과 1차원 배열의 덧셈

브로드캐스팅 규칙에 따라, `vector`는 $(4, 3)$ 형태로 확장되어 `matrix`와 같은 `shape`을 가집니다. 이는 `vector`의 각 요소가 `matrix`의 각 행에 더해지는 효과를 가져옵니다.

결과는 다음과 같습니다:





배열에 세로 벡터 더하기

배열에 세로 벡터를 더하고 싶은 경우가 있습니다. 이 경우에도 그냥 더하면 에러가 발생합니다.

```
import numpy as np

# 2차원 배열 생성
matrix = np.array([[ 0.0,  0.0,  0.0],
                   [10.0, 10.0, 10.0],
                   [20.0, 20.0, 20.0],
                   [30.0, 30.0, 30.0]])
# 벡터 생성
vector = np.array([1.0, 2.0, 3.0, 4.0])
print(vector.shape, matrix.shape)
```

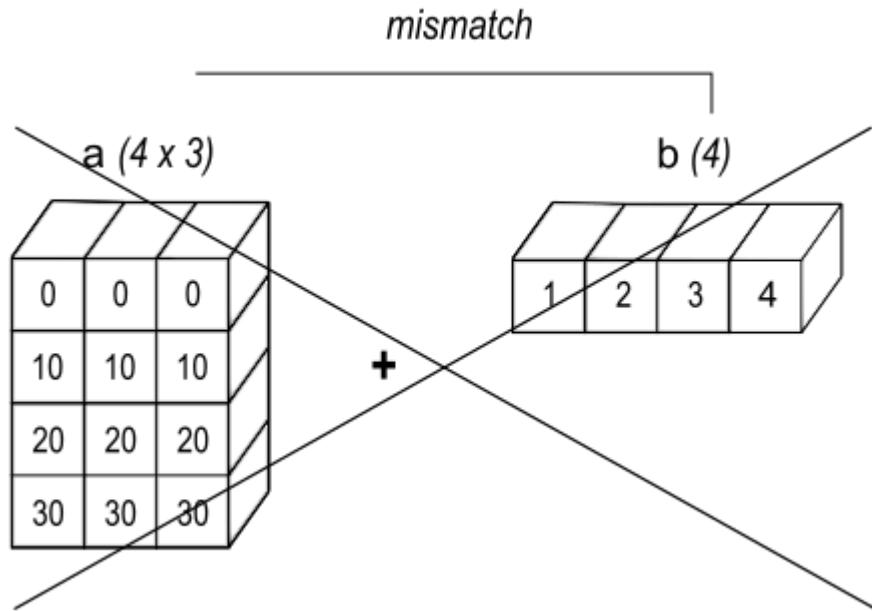
```
## (4, ) (4, 3)
```

```
# 브로드캐스팅을 이용한 배열 덧셈
result = matrix + vector
```

```
## ValueError: operands could not be broadcast
```



배열에 세로 벡터 더하기



이런 경우, 벡터를 세로벡터로 바꿔준 후, `shape`을 맞춰 더해주면 브로드캐스트가 작동하게 됩니다.



배열에 세로 벡터 더하기

```
# 세로 벡터 생성  
vector = np.array([1.0, 2.0, 3.0, 4.0]).reshape(4, 1)  
  
# 브로드캐스팅을 이용한 배열 덧셈  
result = matrix + vector  
print("브로드캐스팅 결과:\n", result)
```

```
## 브로드캐스팅 결과:  
## [[ 1.  1.  1. ]  
## [12. 12. 12.]  
## [23. 23. 23.]  
## [34. 34. 34.]]
```



벡터 내적 활용하기

벡터 내적(dot product)은 두 벡터의 요소를 곱한 후 합산하는 연산입니다.

$$\mathbf{a} \cdot \mathbf{b} = (1 \times 4) + (2 \times 5) + (3 \times 6) = 32$$

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

dot_product = np.dot(a, b)
print("벡터 내적:", dot_product)
```

```
## 벡터 내적: 32
```

코스 훑어보기.

파이썬의 넘파이 라이브러리에 대해 심화 학습합니다.



넘파이(NumPy) 벡터 슬라이싱

벡터의 일부를 추출할 때는 대괄호([])를 사용합니다. 대괄호 안에는 추출하려는 요소의 위치나 인덱스를 지정합니다.

```
import numpy as np

# 벡터 슬라이싱 예제, a를 랜덤하게 채움
np.random.seed(42)
a = np.random.randint(1, 21, 10)
print(a)
```

```
## [ 7 20 15 11  8  7 19 11 11  4]
```

```
print(a[1]) # 두 번째 값 추출
```

```
## 20
```



파이썬 인덱싱 특징

파이썬 인덱싱의 특징은 다음과 같습니다:

- 인덱스는 0부터 시작
- 양의 인덱스는 앞에서부터 세고, 음의 인덱스는 뒤에서부터 셹니다.
- 슬라이싱 구문 `[start:stop:step]`에서 `stop`은 포함되지 않습니다.

예를 들어 `a = [0, 1, 2, 3, 4, 5]`라는 리스트가 있다면:

- `a[0] = 0` (첫 번째 값)
- `a[5] = 5` (마지막 값)
- `a[-1] = 5` (마지막 값, 음수 인덱스 사용)
- `a[1:4] = [1, 2, 3]` (인덱스 1부터 3까지 추출, 4는 미포함)
- `a[::-2] = [0, 2, 4]` (처음부터 끝까지, 스텝은 2)

이런 식으로 인덱싱과 슬라이싱을 활용하여 원하는 값들을 추출할 수 있습니다.



파이썬 인덱싱 예제

```
# 세 번째부터 다섯 번째 값 추출  
print(a[2:5])
```

```
## [15 11 8]
```

- 주의 할 점: 시작값에 해당하는 포지션은 포함, 마지막 값에 해당하는 포지션은 미포함.



파이썬 슬라이싱 연산자

`a[2:5]`에서 `2:5`는 슬라이싱 연산자입니다. 이 연산자는 `start:stop:step` 형태를 가지며, 다음과 같이 동작합니다.

- `start`: 슬라이싱을 시작할 인덱스 위치입니다. 이 인덱스는 포함됩니다.
- `stop`: 슬라이싱을 종료할 인덱스 위치입니다. 이 인덱스는 제외됩니다.
- `step`: 인덱스를 움직일 간격입니다. 기본값은 1입니다.

따라서 `a[2:5]`는 벡터 `a`에서 인덱스 2부터 인덱스 4까지의 값을 추출합니다. 파이썬의 인덱싱이 0부터 시작하므로, 인덱스 2는 실제로 세 번째 값이고, 인덱스 4는 다섯 번째 값입니다. 그리고 인덱스 5는 포함되지 않습니다.

```
# 첫 번째, 세 번째, 다섯 번째 값 추출  
print(a[[0, 2, 4]])
```

```
## [ 7 15 8 ]
```



파이썬 슬라이싱 연산자

```
# 두 번째 값 제외하고 추출  
print(np.delete(a, 1))
```

```
## [ 7 15 11  8  7 19 11 11  4]
```

인덱싱 안에 리스트가 들어가도 됩니다. 또한, 인덱싱을 중복해서 선택이 가능합니다.

```
# 인덱싱 중복 선택  
print(a)
```

```
## [ 7 20 15 11  8  7 19 11 11  4]
```

```
print(a[[1, 1, 3, 2]])
```

```
## [20 20 11 15]
```



파이썬 슬라이싱 연산자 예제

대괄호 연산자와 비교 연산자를 사용한 벡터 슬라이싱은 다음과 같이 수행할 수 있습니다.

```
b = a[a > 3]
print(b)
```

```
## [ 7 20 15 11  8  7 19 11 11  4]
```

위 예제에서는 벡터 `a`의 값이 `3`보다 큰 요소만 추출하여 `b`에 할당합니다.



파이썬 슬라이싱 연산자 예제

대괄호 안에 논리 연산자와 비교 연산자를 조합하여 원하는 값을 추출할 수도 있습니다.

```
b = a[(a > 2) & (a < 9)]  
print(b)
```

```
## [7 8 7 4]
```



파이썬 슬라이싱 연산자 예제

대괄호 안에 비교 연산자를 조합하여 원하는 값을 추출할 수도 있습니다.

- `==`, `!=`

```
print(a[a == 8])
```

```
## [8]
```

```
print(a[a != 8])
```

```
## [ 7 20 15 11  7 19 11 11  4]
```



파이썬 슬라이싱 연산자 예제

- % (나머지), // (몫)

나머지 연산자(%)를 사용하여 벡터 슬라이싱을 수행할 수도 있습니다. 나머지 연산자를 사용하여 특정 패턴의 값만 추출할 수 있습니다.

```
b = a[np.arange(1, 11) % 2 == 1]
print(b)
```

```
## [ 7 15  8 19 11]
```

위 예제에서는 `np.arange(1, 11)`을 사용하여 인덱스 벡터를 생성합니다. 이후, % 연산자를 사용하여 인덱스 벡터를 2로 나눈 나머지가 1인 요소만 추출합니다. 이를 통해 벡터 `a`에서 홀수 번째 요소만 추출할 수 있습니다.



파이썬 슬라이싱 연산자 예제

- & (AND)

```
x = np.array([True, True, False])
y = np.array([True, False, False])
print(x & y)
```

```
## [ True False False]
```

- | (OR)

```
print(x | y)
```

```
## [ True  True False]
```



필터링을 이용한 벡터 변경

앞에서 배운 슬라이싱을 이용하면, 벡터에 대한 조건문을 사용하여 벡터의 일부 값을 변경할 수 있습니다.

```
import numpy as np  
a = np.array([5, 10, 15, 20, 25, 30]) # 예시 배열  
  
a[a >= 10] = 10  
a
```

```
## array([ 5, 10, 10, 10, 10, 10])
```

`a[a >= 10]`는 `a` 벡터에서 10 이상인 원소들을 선택한 것을 의미합니다. 이 선택된 원소들에 대해서 10이 할당되면서, `a` 벡터에서 10 이상인 값들은 모두 10으로 변경됩니다. 이후에 `a`를 출력하면, `a` 벡터에서 10 이하인 값들은 그대로 유지되고, 10 이상인 값들은 모두 10으로 변경되어 있음을 확인할 수 있습니다.



조건을 만족하는 위치 탐색 np.where()

벡터에 대한 조건문과 `np.where()` 함수를 사용하여, 조건을 만족하는 원소의 위치를 선택할 수 있습니다. `a < 7`은 `a` 벡터에서 7보다 작은 원소들에 대해 논리값 `True`를, 7 이상인 원소들에 대해 논리값 `False`를 반환합니다. `a < 7`의 결과는 다음과 같습니다.

```
import numpy as np  
a = np.array([1, 5, 7, 8, 10]) # 예시 배열  
  
result = a < 7  
result
```

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



조건을 만족하는 위치 탐색 np.where()

np.where() 함수를 이용하여 논리값이 True인 원소의 위치를 선택합니다. 따라서 np.where(a < 7)은 a 벡터에서 7보다 작은 원소들의 위치를 반환하게 됩니다.

```
import numpy as np  
a = np.array([1, 5, 7, 8, 10]) # 예시 배열  
  
result = np.where(a < 7)  
result
```

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

이렇게 np.where() 함수를 이용하면 선택된 원소의 위치를 반환할 수 있으며, 이를 활용하여 다양한 계산을 수행할 수 있습니다.



벡터 함수 사용하기

파이썬에서도 다양한 벡터 함수를 제공합니다. 이러한 함수를 사용하면 벡터의 합계, 평균, 중앙값, 표준편차 등을 계산할 수 있습니다.

```
import numpy as np

# 벡터 함수 사용하기 예제
a = np.array([1, 2, 3, 4, 5])
sum_a = np.sum(a)          # 합계 계산
mean_a = np.mean(a)         # 평균 계산
median_a = np.median(a)     # 중앙값 계산
std_a = np.std(a, ddof=1)   # 표준편차 계산

sum_a, mean_a, median_a, std_a
```

```
## (15, 3.0, 3.0, 1.5811388300841898)
```



빈 칸을 나타내는 방법

데이터가 정의 되지 않은 np.nan

np.nan은 정의 되지 않은 값(not a number)을 나타냅니다. np.nan를 벡터에 추가하면 해당 위치에는 nan라는 이상치가 들어갑니다. nan은 실제로 값을 가지고 있지 않지만, 벡터의 길이나 타입을 유지하기 위해 존재하는 것입니다.

- nan: not a number

```
import numpy as np  
  
a = np.array([20, np.nan, 13, 24, 309])  
a
```

```
## array([ 20.,  nan,  13.,  24., 309.])
```



데이터가 정의 되지 않은 np.nan

벡터 안에 nan가 들어있는 경우, 계산 값이 nan로 나오게 됩니다. 이유는 숫자에 nan를 더하면 nan가 되기 때문입니다. 이러한 것을 방지하기 위해서, 많은 함수들에는 nan 무시 옵션이 들어있습니다.

```
np.mean(a)
```

```
## nan
```

- nan 무시 옵션

```
np.nanmean(a) # nan 무시 함수
```

```
## 91.5
```



데이터가 정의 되지 않은 np.nan

- 타입: float 타입입니다. numpy 라이브러리에서 제공하는 상수입니다.
- 사용처: 주로 데이터 분석에서 결측값(missing value)을 나타내기 위해 사용됩니다.
- 비교: np.nan과의 비교는 직접적으로 == 연산자를 사용할 수 없고, numpy의 np.isnan() 함수를 사용해야 합니다.

```
import numpy as np
value = np.nan
if np.isnan(value):
    print("값이 NaN입니다.")
```

```
## 값이 NaN입니다.
```

- 수치연산 가능

```
np.nan + 1
```

```
## nan
```



값이 없음을 나타내는 None

None은 아무런 값도 없는 상태를 나타냅니다.

- 타입: NoneType 타입입니다.
- 사용처: 주로 변수 초기화, 함수 반환 값, 조건문, 기본 인자값 등에 사용됩니다.
- 비교: None과의 비교는 is 연산자를 사용하여 이루어집니다.

```
my_variable = None  
if my_variable is None:  
    print("변수에 값이 없습니다.")
```

변수에 값이 없습니다.

- 수치연산 불가

```
None + 1
```

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'



빈 칸을 제거하는 방법

빈 칸을 제거하는 방법은 다음과 같습니다. `np.isnan()` 함수는 벡터 `a`의 원소가 `nan`인지를 아닌지를 알려주는 함수입니다. `nan`인 경우 `True`를 반환하고, 그렇지 않은 경우 `False`를 반환합니다.

따라서, 이러한 논리 벡터를 사용하여 벡터 필터링을 하게 되면, 다음과 같이 `nan`이 생략된 벡터를 얻을 수 있습니다.

```
a_filtered = a[~np.isnan(a)]  
a_filtered
```

```
## array([ 20.,  13.,  24., 309.])
```



벡터 합치기

벡터는 같은 타입의 정보 (숫자, 문자)를 묶어놓은 것입니다. 즉, 숫자면 숫자, 문자면 문자이기만 하면, 묶을 수 있습니다. 다음은 문자열로 이루어진 벡터입니다.

```
import numpy as np  
str_vec = np.array([ "사과", "배", "수박", "참외" ])  
str_vec
```

```
## array(['사과', '배', '수박', '참외'], dtype='<U2')
```

```
str_vec[[0, 2]]
```

```
## array(['사과', '수박'], dtype='<U2')
```



벡터 합치기

그렇다면, 문자와 숫자를 섞어서 벡터를 만든다면 어떨까요?

```
import numpy as np  
mix_vec = np.array(["사과", 12, "수박", "참외"], dtype=str)  
mix_vec
```

```
## array(['사과', '12', '수박', '참외'], dtype='<U2')
```

결과를 살펴보면, 파이썬은 자동으로 통일할 수 있는 타입(문자) 정보로 바꿔서, 벡터로 저장하는 것을 관찰할 수 있습니다.



여러 벡터들을 묶기

여러 개의 벡터들을 하나로 묶을 수 있는 방법이 세 가지 존재합니다. 첫 번째 방법은 `np.concatenate()` 함수를 사용하는 것입니다. 앞에서 정의한 `str_vec`와 `mix_vec`을 묶어 하나의 벡터로 만들려면 다음과 같이 할 수 있습니다.

```
combined_vec = np.concatenate((str_vec, mix_vec))  
combined_vec
```

```
## array(['사과', '배', '수박', '참외', '사과', '12', '수박', '참외'], dtype='<U2')
```



np.column_stack() 와 np.row_stack()

np.column_stack() 함수는 벡터들을 세로로, np.row_stack() 함수는 벡터들을 가로로 쌓아 줍니다.

```
col_stacked = np.column_stack((np.arange(1, 5), np.arange(12, 16)))
col_stacked
```

```
## array([[ 1, 12],
##          [ 2, 13],
##          [ 3, 14],
##          [ 4, 15]])
```

```
row_stacked = np.row_stack((np.arange(1, 5), np.arange(12, 16)))
row_stacked
```

```
## array([[ 1,  2,  3,  4],
##          [12, 13, 14, 15]])
```



길이가 다른 벡터 합치기

만약 앞의 방법으로 합칠 때 길이가 다른 벡터를 합치게 되면 어떤 일이 벌어질까요? 다음의 코드를 보겠습니다.

```
uneven_stacked = np.column_stack((np.arange(1, 5), np.arange(12, 18)))
```

```
## ValueError: all the input array dimensions except for the concatenation axis must match exactly
```

```
uneven_stacked
```

```
## NameError: name 'uneven_stacked' is not defined
```



길이가 다른 벡터 합치기

```
import numpy as np

# 길이가 다른 벡터
vec1 = np.arange(1, 5)
vec2 = np.arange(12, 18)
vec1 = np.resize(vec1, len(vec2))
vec1
```

```
## array([1, 2, 3, 4, 1, 2])
```

`np.resize()` 함수를 사용하면 길이를 강제로 맞춰주고, 값을 앞에서부터 채워줍니다.



길이가 다른 벡터 합치기

```
# 두 벡터를 세로로 쌓기
uneven_stacked = np.column_stack((vec1, vec2))
uneven_stacked
```

```
## array([[ 1, 12],
##          [ 2, 13],
##          [ 3, 14],
##          [ 4, 15],
##          [ 1, 16],
##          [ 2, 17]])
```



여러 조건을 처리하기

여러 개의 조건을 처리할 때는 `numpy.select()`를 활용합니다. 각 조건에 대한 결과를 리스트로 작성하여 조건에 따라 결과를 반환할 수 있습니다. `if-else` 문법을 활용하는 것과 결과는 동일합니다.

```
import numpy as np

x = np.array([1, -2, 3, -4, 0])
conditions = [x > 0, x == 0, x < 0]
choices = ["양수", "0", "음수"]
result = np.select(conditions, choices, default="기타") # 기본값을 문자열로 설정
print(result)
```

```
## ['양수' '음수' '양수' '음수' '0']
```



NumPy 성능 최적화

NumPy는 기본적으로 C 기반으로 구현되어 있어 일반적인 Python 반복문보다 훨씬 빠른 속도로 연산을 수행할 수 있습니다. 하지만 대규모 데이터셋을 처리할 때는 성능을 더욱 최적화하는 것이 중요합니다. NumPy에서 성능을 최적화하는 몇 가지 방법을 살펴보겠습니다.

Python for-loop vs NumPy 벡터 연산

NumPy의 벡터 연산이 Python for 반복문보다 훨씬 빠른 이유는 내부적으로 C에서 최적화된 행렬 연산이 수행되기 때문입니다.

일반적인 Python for 반복문 사용 (느림)



```
import numpy as np
import time

size = 10**6
a = np.random.rand(size)
b = np.random.rand(size)
start = time.time()
c = np.zeros(size)
for i in range(size):
    c[i] = a[i] + b[i]
end = time.time()
print("For-loop 실행 시간:", end - start, "초")
```

```
## For-loop 실행 시간: 0.2225499153137207 초
```



NumPy 벡터 연산 (빠름)

NumPy의 벡터 연산은 반복문보다 훨씬 빠릅니다.

```
start = time.time()
c = a + b # 벡터 연산
end = time.time()

print("NumPy 벡터 연산 실행 시간:", end - start, "초")
```

```
## NumPy 벡터 연산 실행 시간: 0.0037801265716552734 초
```



NumPy 벡터화 (Vectorization)

np.vectorize()를 사용하면 Python의 for 루프를 사용하지 않고 빠르게 벡터 연산을 수행할 수 있습니다.

python for 반복문 사용 (느림)

```
def my_function(x):
    return x ** 2 + 10

data = np.arange(1, 1000000)

start = time.time()
result = np.array([my_function(x) for x in data]) # 일반적인 for문 사용
end = time.time()

print("For-loop 실행 시간:", end - start, "초")
```

```
## For-loop 실행 시간: 0.14769697189331055 초
```



np.vectorize()를 사용한 벡터화 (빠름)

np.vectorize()를 사용하면 Python for 루프보다 훨씬 빠르게 연산할 수 있습니다. 단순히 함수를 np.vectorize()로 감싸주면 됩니다.

```
start = time.time()
vectorized_function = np.vectorize(my_function)
result = vectorized_function(data)
end = time.time()

print("NumPy 벡터화 실행 시간:", end - start, "초")
```

```
## NumPy 벡터화 실행 시간: 0.1085507869720459 초
```



메모리 절약을 위한 데이터 타입 설정 (dtype)

NumPy 배열을 생성할 때, 기본적으로 float64 타입이 사용됩니다. 하지만 float32 또는 int32 같은 작은 데이터 타입을 사용하면 메모리를 절약할 수 있습니다.

```
import numpy as np  
a = np.array([1.0, 2.0, 3.0], dtype=np.float64) # 기본 dtype: float64  
b = np.array([1.0, 2.0, 3.0], dtype=np.float32) # float32로 저장  
  
print("float64 배열 크기:", a.nbytes, "bytes") # 8 bytes * 3 = 24 bytes
```

```
## float64 배열 크기: 24 bytes
```

```
print("float32 배열 크기:", b.nbytes, "bytes") # 4 bytes * 3 = 12 bytes
```

```
## float32 배열 크기: 12 bytes
```

코스 훑어보기.

파이썬의 넘파이 라이브러리를 활용한 행렬 연산에 대해 심화 학습합니다.



행렬이란 무엇일까?

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

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.

행렬 생성 예제



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)
```

```
## 행렬의 크기: (4, 2)
```

```
## 행렬:
## [[ 1 12]
## [ 2 13]
## [ 3 14]
## [ 4 15]]
```



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.

NumPy 함수 요약

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



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.]]
```

채우면서 만들기



아래의 코드는 1부터 4까지의 수를 원소로 갖는 2행 2열의 행렬 `y`를 생성합니다. 행렬의 크기는 무조건 사각형이므로, `reshape()` 함수를 사용하여 행의 수(`nrow`)와 열의 수(`ncol`)를 지정하면 자동으로 행렬의 크기를 계산하여 만들어 줍니다.

```
# 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]
```



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.

행렬을 채우는 방법 - `order` 옵션 예제



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("가로 방향으로 채운 행렬 y:\n", y)
```

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



행렬을 채우는 방법 - order 옵션 예제

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]]
```



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 `x`를 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]]
```



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 x의 1행 2열에 위치한 원소: x[0, 1]

그리고 x[0, 1]는 행렬 x에서 1행 2열의 원소를 반환합니다.

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

```
## 1행 2열의 원소: 4
```



행렬 인덱싱 (Indexing)

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  
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]]
```

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

```
## 두 번째 열의 모든 원소: [ 4  8 12 16 20]
```



행렬 인덱싱 (Indexing)

만약 행렬 `x`의 세 번째 행을 추출하기 위해서는 어떻게 해야 할까요? 이번에는 반대로 `x`의 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]
```



선택적으로 골라오기

특정 열에서 선택적으로 원소를 골라오려면, 행의 위치를 나타내는 곳에 원하는 행의 번호를 넣습니다. 예를 들어 `x[[1, 2, 4], 1]`는 행렬 `x`의 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]
```



행렬 필터링

True, False를 원소로 가진 배열을 사용하여 행렬에서 원하는 원소를 필터링할 수 있습니다. 예를 들어, `x[[True, True, False, False, True], 0]`은 행렬 `x`의 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
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]]
```

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

```
## 첫 번째 열의 첫 번째, 두 번째, 다섯 번째 행의 원소
## [ 2  6 18]
```



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]
```



사진은 행렬이다.

행렬을 이용해 이미지 생성하기

이미지는 흑백인 경우 0과 1 사이의 숫자로 표현할 수 있습니다. 이 때, 0은 검은색을, 1은 흰색을 의미합니다. 예를 들어, 아래의 코드에서는 `np.random.rand(3, 3)`으로 0과 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)

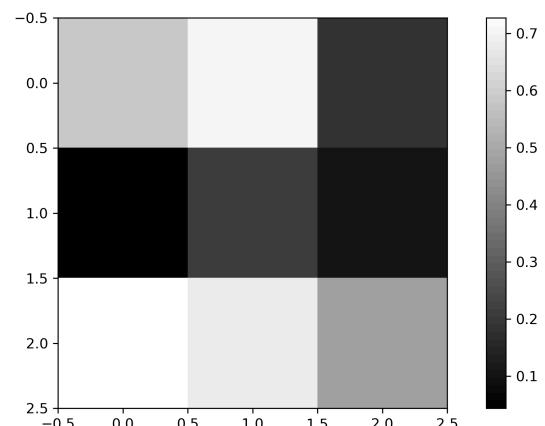
## 이미지 행렬 img1:
## [[ 0.58801452  0.69910875  0.18815196]
##  [ 0.04380856  0.20501895  0.10606287]
##  [ 0.72724014  0.67940052  0.4738457 ]]
```



행렬을 이용해 이미지 생성하기

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

```
# 행렬을 이미지로 표시
plt.figure(figsize=(10, 5)) # (가로, 세로) 크기 설정
plt.imshow(img1, cmap='gray', interpolation='nearest');
plt.colorbar();
plt.show();
```





행렬 다운로드하기

링크를 눌러, 미리 정의된 행렬을 다운로드 받아보겠습니다. 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)
```



행렬 다운로드하기

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 크기: (88, 50)
```

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

```
## 행렬의 일부:  
## [[ 132. 131. 134. 132.]  
## [ 135. 137. 137. 138.]  
## [ 143. 142. 145. 146.]]
```



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())
```

```
## 최대값: 255.0
```

```
print("최소값:", img_mat.min())
```

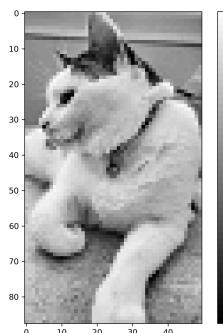
```
## 최소값: 0.0
```



행렬에서 사진으로

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





행렬의 연산

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)
```

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



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 y:  
## [[1 2 3]  
## [4 5 6]]
```

```
print("행렬 x, y의 크기:", x.shape, y.shape)
```

```
## 행렬 x, y의 크기: (5, 2) (2, 3)
```

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

```
## 행렬곱 x * y:  
## [[ 18  24  30]  
## [ 38  52  66]  
## [ 58  80 102]  
## [ 78 108 138]  
## [ 98 136 174]]
```



행렬의 곱셈 (dot product)

NumPy에서는 `matmul()` 메서드를 사용하여 두 행렬의 곱을 계산할 수도 있습니다. `matmul()`은 행렬 곱셈 전용으로 `dot()`과 달리 벡터 간 내적은 수행하지 않습니다.

```
# 행렬 곱셈 (matmul 사용)
matrix_product = np.matmul(x, y) # 행렬 곱셈
print("행렬 곱셈 (matmul 사용):\n", matrix_product)
```

```
## 행렬 곱셈 (matmul 사용):
## [[ 18  24  30]
## [ 38  52  66]
## [ 58  80 102]
## [ 78 108 138]
## [ 98 136 174]]
```

다차원 배열(3D 이상)에서 dot() vs matmul() 비교



```
C = np.random.rand(2, 3, 4)
D = np.random.rand(2, 4, 5)
dot_result = np.dot(C, D)      # 다차원 배열에서
matmul_result = np.matmul(C, D) # 다차원 배열
```

```
print("np.dot() 결과 shape:", dot_result.sha
```

```
## np.dot() 결과 shape: (2, 3, 2, 5)
```

```
print("np.matmul() 결과 shape:", matmul_resu
```

```
## np.matmul() 결과 shape: (2, 3, 5)
```

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.

```
z = np.arange(10, 14).reshape((2, 2))
print(z)
```

```
## [[10 11]
##  [12 13]]
```

```
y = np.array([[1, 2], [3, 4]])
print(y)
```

```
## [[1 2]
##  [3 4]]
```

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

```
## 원소별 곱셈 y * z:
## [[10 22]
##  [36 52]]
```



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 y:  
## [[1 2]  
## [3 4]]
```

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

```
## 행렬 y의 역행렬:  
## [[-2. 1.]  
## [1.5 -0.5]]
```



행렬의 역행렬

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)
```

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

```
np.linalg.inv(no_inverse)
```

```
## numpy.linalg.LinAlgError: Singular matrix
```



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

```
## 행렬 y:  
## [[1 2]  
##  [3 4]]
```

```
result = x * y  
print('x, y shape: ', x.shape, y.shape)
```

```
## x, y shape: (2,) (2, 2)
```

```
print(result)
```

```
## [[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)
```

```
## 행렬 y:  
## [[1 2]  
## [3 4]]
```

```
print("행렬 z:\n", z)
```

```
## 행렬 z:  
## [[1]  
## [2]]
```

```
y.shape
```

```
## (2, 2)
```

```
z.shape
```

```
## (2, 1)
```

```
result = y * z  
print(result)
```

```
## [[1 2]  
## [6 8]]
```

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)
```

```
## 행렬 y:  
## [[1 2]  
## [3 4]]
```

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

```
## 첫 번째 행:  
## [1 2]
```

```
print("첫 번째 행의 차원:", first_row.shape)
```

```
## 첫 번째 행의 차원: (2, )
```



차원 축소 효과

이런 경우에는 `np.newaxis`를 사용하여 차원을 유지할 수 있습니다. `np.newaxis`는 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)
```

```
## 첫 번째 행을 뽑아 가로 차원으로 확장:  
## [[1 2]]
```

```
print("모양 확인:", first_row_2d.shape)
```

```
## 모양 확인: (1, 2)
```



차원 축소 효과

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)
```

```
## 첫 번째 행을 뽑아 세로 차원으로 확장:  
## [ [1]  
## [2] ]
```

```
print("모양 확인:", first_row_2d.shape)
```

```
## 모양 확인: (2, 1)
```



고차원 행렬, 배열

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

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)
# 3차원 배열로 합치기
my_array = np.array([mat1, mat2])
```

array() 함수로 3차원 배열 만들기



```
print("3차원 배열 my_array:\n", my_array)
```

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

```
print("3차원 배열의 크기 (shape):", my_array.shape)
```

```
## 3차원 배열의 크기 (shape): (2, 2, 3)
```



배열 다루기

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

```
import numpy as np  
mat1 = np.arange(1, 7).reshape(2, 3) # 두 개  
mat2 = np.arange(7, 13).reshape(2, 3)  
my_array = np.array([mat1, mat2])  
print("3차원 배열 my_array:\n", my_array)
```

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

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

```
## 첫 번째 2차원 배열:  
## [[1 2 3]  
##  [4 5 6]]
```



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.



배열의 차원 바꾸기

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)
```

```
## 원래 배열 my_array:  
## [[ 1  2  3]  
##  [ 4  5  6]]  
##  
## [[ 7  8  9]  
##  [10 11 12]]]
```

```
my_array.shape
```

```
## (2, 2, 3)
```

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

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



사진은 배열이다.

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 object at 0x31402d650>)
```

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")
```

```
## <string>:3: DeprecationWarning: Starting
```

```
print("이미지 클래스:", type(jelly))
```

```
## 이미지 클래스: <class 'numpy.ndarray'>
```

```
print("이미지 차원:", jelly.shape)
```

```
print("이미지 첫 4x4 픽셀, 첫 번째 채널:\n", jel
```

```
## 이미지 첫 4x4 픽셀, 첫 번째 채널:
## [[156 151 146 142]
## [118 121 123 121]
## [105 107 110 110]
## [104 108 110 112]]
```



RGB, Opacity

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

```
import matplotlib.pyplot as plt  
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)  
plt.imshow(bw_jelly, cmap='gray');  
plt.axis('off');  
plt.show();
```



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 것입니다.



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.



넘파이 배열 기본 제공 함수들 정리

메서드	문법	설명
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.



넘파이 배열 기본 제공 함수들 정리

메서드	문법	설명
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()	sort(axis=-1)	배열을 정렬합니다.
argsort()	argsort(axis=-1)	배열 원소들의 정렬된 인덱스를 반환합니다.



각 함수들 간략 설명

sum()

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

```
import numpy as np  
a = np.array([[1, 2, 3], [4, 5, 6]])  
print("전체 합계:", a.sum())
```

전체 합계: 21

```
print("열별 합계:", a.sum(axis=0))
```

열별 합계: [5 7 9]

```
print("행별 합계:", a.sum(axis=1))
```

행별 합계: [6 15]

mean()



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

```
print("전체 평균:", a.mean())
```

```
## 전체 평균: 3.5
```

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

```
## 열별 평균: [2.5 3.5 4.5]
```

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

```
## 행별 평균: [2. 5.]
```



max()

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

```
print("전체 최댓값:", a.max())
```

```
## 전체 최댓값: 6
```

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

```
## 열별 최댓값: [4 5 6]
```

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

```
## 행별 최댓값: [3 6]
```

min()



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

```
print("전체 최솟값:", a.min())
```

```
## 전체 최솟값: 1
```

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

```
## 열별 최솟값: [1 2 3]
```

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

```
## 행별 최솟값: [1 4]
```



std()

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

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

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

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
```

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())
```

```
## 전체 누적 합계: [ 1  3  6 10 15 21]
```

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

```
## 열별 누적 합계: [[1 2 3]
##   [5 7 9]]
```

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

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



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())
```

```
## 전체 누적 곱: [ 1 2 6 24 120 720]
```

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

```
## 열별 누적 곱: [[ 1 2 3]
##   [ 4 10 18]]
```

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

```
## 행별 누적 곱: [[ 1 2 6]
##   [ 4 20 120]]
```



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())
```

```
## 최댓값의 인덱스 (전체) : 5
```

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

```
## 최댓값의 인덱스 (열별) : [1 1 1]
```

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

```
## 최댓값의 인덱스 (행별) : [2 2]
```



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())
```

```
## 최솟값의 인덱스 (전체) : 0
```

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

```
## 최솟값의 인덱스 (열별) : [0 0 0]
```

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

```
## 최솟값의 인덱스 (행별) : [0 0]
```



reshape()

배열의 형상을 변경합니다.

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

```
## 원본 배열:
## [1 2 3 4 5 6]
```

```
print("형상 변경:\n", b.reshape((2, 3)))
```

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



transpose()

배열을 전치합니다.

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

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

```
print("전치 배열:\n", c.transpose())
```

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

`flatten()`



1차원 배열로 변환합니다.

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

```
## 1차원 배열:  
## [1 2 3 4 5 6]
```



clip()

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

예제를 보면 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.



tolist()

배열을 리스트로 변환합니다.

```
print("리스트:", d.tolist())
```

```
## 리스트: [1, 2, 3, 4, 5]
```

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]
```

copy()



배열의 복사본을 반환합니다.

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

```
## 복사본 배열: [1 2 3 4 5]
```



얕은 복사와 깊은 복사 개념 이해하기

이전 챕터에서 배운 것처럼 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]
```

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

```
print("f:", f) # 출력: f: [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.



얕은 복사와 깊은 복사 개념 이해하기

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]
```

```
## d: [1 2 3 4 5]
```

```
print("f:", f) # 출력: f: [10  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?



sort()

배열을 정렬합니다.

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

```
## 정렬된 배열: [1 2 3]
```

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]
```

argsort()



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 인덱스: [3 1 4 2 0]

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

정렬된 배열: [1 5 7 8 10]



넘파이(Numpy) 배열에 apply 함수 적용

Python에서는 `numpy` 패키지를 사용하여 2차원이나 3차원 배열에 함수를 적용할 수 있습니다. 예제를 만들어 공부해봅시다. 2차원 배열을 하나 만들어 보겠습니다.

```
import numpy as np  
array_2d = np.arange(1, 13).reshape((3, 4), order='F')  
print(array_2d)
```

```
## [[ 1  4  7 10]  
##  [ 2  5  8 11]  
##  [ 3  6  9 12]]
```



apply_along_axis()

넘파이 배열에 함수를 적용할 때에는 `apply_along_axis()` 라는 메서드를 사용합니다.

```
np.apply_along_axis(max, axis=0, arr=array_2d)
```

```
## array([ 3,  6,  9, 12])
```

위 코드에서 `np.apply_along_axis(max, axis=0, arr=array_2d)`는 2차원 배열 `array_2d`의 각 열에 대해 `max` 함수를 적용하여 합계를 계산합니다.

- `axis=0`은 열 방향으로 함수를 적용하라는 의미입니다.
- 결과는 각 열의 합계로 구성된 1차원 배열입니다.



apply_along_axis()

```
> apply(a, 1, max)
```

	[,1]	[,2]	[,3]	[,4]	결과값
[1,]	1	4	7	10	10
[2,]	2	5	8	11	11
[3,]	3	6	9	12	12

```
> apply(a, 2, max)
```

	[,1]	[,2]	[,3]	[,4]	결과값
[1,]	1	4	7	10	3
[2,]	2	5	8	11	6
[3,]	3	6	9	12	9



apply_along_axis()

적용할 수 있는 함수를 직접 만들어서 사용할 수도 있습니다. 또한, 차원이 하나더 늘어서 3차원 배열에도 적용 가능하죠.

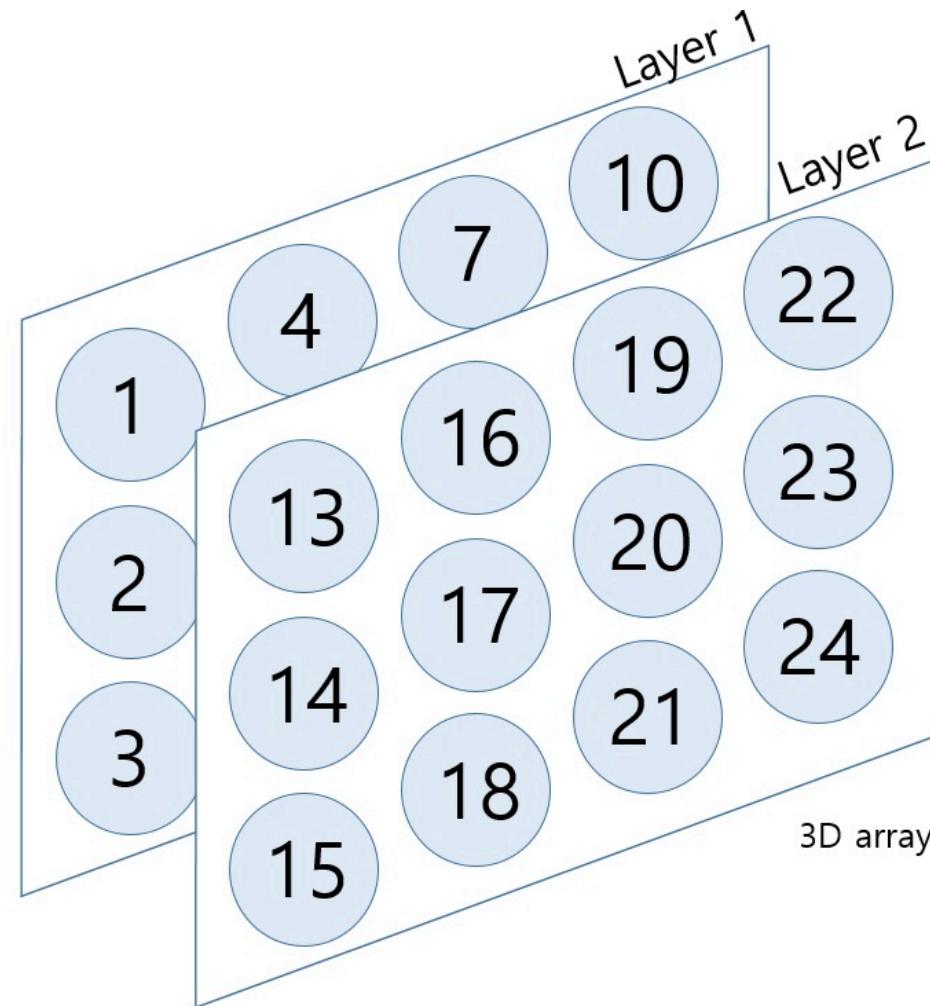
```
array_3d = np.arange(1, 25).reshape(2, 4, 3).transpose(0, 2, 1)
print(array_3d)
```

```
## [[[ 1  4  7 10]
##     [ 2  5  8 11]
##     [ 3  6  9 12]]
##
##     [[13 16 19 22]
##      [14 17 20 23]
##      [15 18 21 24]]]
```



apply_along_axis()

3차원 배열 예제를 그림으로 보면 다음과 같습니다.





apply_along_axis()

배열의 최대값을 반환하는 사용자 정의 함수 `my_func()`을 만들어서 3차원 배열에 적용해봅시다.

- `axis=0`: 깊이 방향 → 같은 위치에 있는 값들끼리 비교

```
def my_func(x):
    return np.min(x)

np.apply_along_axis(my_func, axis=0, arr=array_3d)
```

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



apply_along_axis()

- axis=1: 행 방향 → 같은 깊이의 행별 최소값

```
np.apply_along_axis(my_func, axis=1, arr=array_3d)
```

```
## array([[ 1,  4,  7, 10],  
##          [13, 16, 19, 22]])
```

- axis=2: 열 방향 → 같은 깊이의 열별 최소값

```
np.apply_along_axis(my_func, axis=2, arr=array_3d)
```

```
## array([[ 1,  2,  3],  
##          [13, 14, 15]])
```



각 레이어 별 함수 적용

때로는 각 층별로 함수를 적용하고 싶은 경우가 있습니다.

```
import numpy as np

# 1에서 24까지의 숫자를 생성하고 reshape하여 원하는 형태로 변환
array_3d = np.arange(1, 25).reshape(2, 4, 3).transpose(0, 2, 1)
print(array_3d)
```

```
## [[[ 1  4  7 10]
##     [ 2  5  8 11]
##     [ 3  6  9 12]]
##
##     [[13 16 19 22]
##      [14 17 20 23]
##      [15 18 21 24]]]
```



각 레이어 별 함수 적용

위 3차원 배열에서 각 행렬별 숫자들의 합을 구하고 싶은 경우, 앞에서 배운 apply를 적용할 수도 있겠지만, sum() 함수를 사용해서도 쉽게 구할 수 있습니다.

```
# 각 층의 합을 계산  
layer_sums = np.sum(array_3d, axis=(1, 2))  
print("Layer sums:", layer_sums)
```

```
## Layer sums: [ 78 222]
```



각 레이어 별 함수 적용

사용자 정의 함수를 사용하고자 하는 경우도 있겠죠? 이런 경우 for 루프를 사용해서 레이어 별로 함수를 적용하면 됩니다.

```
# 각 원소의 제곱에 3을 더한 후 합을 구하는 복잡한 사용자 정의 함수
def my_f(layer):
    modified_layer = (layer ** 2) + 3
    layer_sum = np.sum(modified_layer)
    return layer_sum

# 각 층에 대해 사용자 정의 함수를 적용
layer_sums = np.array([my_f(array_3d[i, :, :]) for i in range(2)])
print("Layer sums:", layer_sums)
```

```
## Layer sums: [ 686 4286 ]
```



연습 문제 1

주어진 벡터의 각 요소에 5를 더한 새로운 벡터를 생성하세요.

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

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



연습 문제 1 해답

주어진 벡터의 각 요소에 5를 더한 새로운 벡터를 생성하세요.

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

```
## array([ 6,  7,  8,  9, 10])
```



연습 문제 2

주어진 벡터의 홀수 번째 요소만 추출하여 새로운 벡터를 생성하세요.

```
a = np.array([12, 21, 35, 48, 5])  
a
```

```
## array([12, 21, 35, 48, 5])
```



연습 문제 2 해답

주어진 벡터의 홀수 번째 요소만 추출하여 새로운 벡터를 생성하세요.

```
a = np.array([12, 21, 35, 48, 5])  
a[::2]
```

```
## array([12, 35, 5])
```



연습 문제 3

주어진 벡터에서 최대값을 찾으세요.

```
a = np.array([1, 22, 93, 64, 54])  
a
```

```
## array([ 1, 22, 93, 64, 54])
```



연습 문제 3 해답

주어진 벡터에서 최대값을 찾으세요.

```
a = np.array([1, 22, 93, 64, 54])  
np.max(a)
```

```
## 93
```



연습 문제 4

주어진 벡터에서 중복된 값을 제거한 새로운 벡터를 생성하세요.

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

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



연습 문제 4 해답

주어진 벡터에서 중복된 값을 제거한 새로운 벡터를 생성하세요.

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

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



연습 문제 5

주어진 두 벡터의 요소를 번갈아 가면서 합쳐서 새로운 벡터를 생성하세요.

```
a = np.array([21, 31, 58])  
b = np.array([24, 44, 67])  
a
```

```
## array([21, 31, 58])
```

```
b
```

```
## array([24, 44, 67])
```



연습 문제 5 해답

주어진 두 벡터의 요소를 번갈아 가면서 합쳐서 새로운 벡터를 생성하세요.

```
a = np.array([21, 31, 58])
b = np.array([24, 44, 67])
c = np.empty(a.size + b.size, dtype=a.dtype)
c[0::2] = a
c[1::2] = b
c
```

```
## array([21, 24, 31, 44, 58, 67])
```



연습 문제 6

다음 a 벡터의 마지막 값은 제외한 두 벡터 a와 b를 더한 결과를 구하세요.

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



연습 문제 6 해답

다음 a 벡터의 마지막 값은 제외한 두 벡터 a와 b를 더한 결과를 구하세요.

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([6, 7, 8, 9])
c = a[:-1] + b
c
```

```
## array([ 7,  9, 11, 13])
```



연습 문제 7

주어진 벡터에서 가장 자주 등장하는 숫자를 찾아 출력하세요.

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

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



연습 문제 7 해답

```
unique, counts = np.unique(a, return_counts=True)
most_frequent = unique[np.argmax(counts)]
most_frequent
```

```
## 2
```



연습 문제 8

주어진 벡터에서 3의 배수만 추출한 새로운 벡터를 만드세요.

```
a = np.array([12, 5, 18, 21, 7, 9, 30, 25, 3, 6])  
a
```

```
## array([12, 5, 18, 21, 7, 9, 30, 25, 3, 6])
```



연습 문제 8 해답

```
multiples_of_three = a[a % 3 == 0]  
multiples_of_three
```

```
## array([12, 18, 21, 9, 30, 3, 6])
```



연습 문제 9

주어진 벡터를 중앙값을 기준으로 두 개의 벡터로 분리하세요.

```
a = np.array([10, 20, 5, 7, 15, 30, 25, 8])  
a
```

```
## array([10, 20, 5, 7, 15, 30, 25, 8])
```



연습 문제 9 해답

```
median_value = np.median(a)
lower_half = a[a < median_value]
upper_half = a[a >= median_value]

lower_half, upper_half
```

```
## (array([10,  5,  7,  8]), array([20, 15, 30, 25]))
```



연습 문제 10

주어진 벡터에서 중앙값과 가장 가까운 값을 찾으시오.

```
a = np.array([12, 45, 8, 20, 33, 50, 19])  
a
```

```
## array([12, 45, 8, 20, 33, 50, 19])
```



연습 문제 10 해답

```
median_value = np.median(a)
differences = np.abs(a - median_value) # 중앙값과 각 원소 간 차이 계산
closest_value = a[np.where(differences == np.min(differences))][0] # 최소 차이를 갖는 원소 찾기
closest_value
```

```
## 20
```



연습 문제 11

다음과 같은 행렬 A를 만들어 보세요!

```
## 행렬 A:  
## [[3 5 7]  
## [2 3 6]]
```



연습 문제 11 해답

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

```
## 행렬 A:  
## [[3 5 7]  
##  [2 3 6]]
```



연습 문제 12

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 ] ]
```



연습 문제 12 해답

```
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]]
```



연습 문제 13

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



연습 문제 13 해답

```
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]]
```



연습 문제 14

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.



연습 문제 14 해답

```
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]]
```



연습 문제 15

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.



연습 문제 15 해답

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

```
## 각 열별 평균:  
## [5.6 5.8 5. 5.6]
```

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

```
## 평균이 5보다 큰 열의 인덱스:  
## [0 1 3]
```



연습 문제 16

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
```

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

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



연습 문제 16 해답

```
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]]
```