Python 기초 Chap4: 벡터와 친해지기

Table of contents

4.1 NumPy 소개 및 설치 가이드

NumPy는 과학 계산을 위해 널리 사용되는 파이썬 라이브러리입니다. 다차원 배열 객체와 배열 작업을 위한 다양한 도구를 제공하며, 데이터 분석, 기계 학습, 공학 설계 등 많은 과학적 계산 분야에서 필수적인 도구로 자리잡고 있습니다.

4.1.1 NumPy란?

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.

4.1.2 설치 방법

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

4.1.2.a **pip**를 이용한 설치

NumPy를 설치하려면, 터미널이나 명령 프롬프트에서 다음 명령을 입력하세요:

```
pip install numpy
```

4.1.2.b **Mini Conda**를 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
```

또는 이미 설치된 NumPy를 업데이트하려면:

```
conda  update numpy
```

4.1.3 불러오기 및 사용방법

NumPy를 설치한 후에는, 파이썬 스크립트나 인터프리터에서 간단히 임포트하여 사용할 수 있습니다.

4.1.3.a **NumPy** 불러오기

```
import numpy as np
```

4.1.4 **NumPy** 배열 생성

Python의 NumPy 라이브러리를 사용하여 벡터(vector)를 다루는 방법에 대해 살펴봅니다. 벡터는 동일한 데이터 타입의 값들을 순서대로 나열한 것입니다.

4.1.4.a 벡터를 만드는 2가지 방법

Python에서 벡터를 생성하는 가장 일반적인 방법은 NumPy의 `np.array()` 함수를 사용하는 것입니다. 이 함수를 사용하면 여러 개의 값을 하나의 벡터로 묶을 수 있습니다. 벡터를 생성할 때는 동일한 데이터 타입의 값을 사용해야 합니다.

- 예제: 벡터 생성하기

아래 코드는 넘파이를 사용해서 각각 숫자형 벡터, 문자형 벡터, 논리형 벡터를 만드는 코드입니다.

```
import numpy as np

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

```
Numeric Vector: [1 2 3 4 5]
String Vector: ['apple' 'banana' 'orange']
Boolean Vector: [ True False  True  True]
```

벡터를 생성하는 방법은 크게 두 가지로 나뉩니다:

1. 빈 배열 선언 후 채우기
2. 배열을 생성하면서 채우기

- 빈 배열 선언 후 채우기

NumPy에서 빈 배열을 생성하는 방법은 `np.empty()` 또는 `np.zeros()` 함수를 사용할 수 있습니다. 다음의 코드는 길이가 3인 빈 배열을 생성하고, 이를 채워 넣는 예제입니다.

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

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

```
빈 벡터 생성하기: [0.  0.5 1. ]
채워진 벡터: [3. 5. 3.]
```

결과에서 e-10은 10의 −10승을 나타냅니다. 따라서 엄청 0에 가까운 수라서 0으로 봐도 무당한 수를 말합니다.

- 배열을 생성하면서 채우기

NumPy에서 배열을 생성하면서 채우는 방법에는 여러 가지가 있습니다:

- `np.array()` 함수를 직접 사용
- `np.arange()` 함수를 사용하여 일정한 간격의 숫자 배열 생성
- `np.linspace()` 함수를 사용하여 지정된 범위를 균일하게 나눈 숫자 배열 생성

4.1.4.a.a `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.

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]
```

예제 코드 2. 0부터 2 미만까지 0.5 간격으로 배열 생성

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

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

4.1.4.a.b `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.

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.  ]
```

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]
```

4.1.4.a.c `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.

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]
```

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]
```

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]
```

4.1.5 넘파이 벡터 길이 재는 방법

넘파이 배열의 길이를 재는 방법은 여러 가지가 있습니다. 가장 일반적으로 사용하는 방법은 `len()` 함수를 사용하는 것입니다. 또한, 배열의 크기와 모양을 확인하기 위해 `numpy` 모듈의 `shape` 속성을 사용할 수도 있습니다.

4.1.5.a `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
```

4.1.5.b `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,)
```

4.1.5.c 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
```

4.1.5.d (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)
```

이와 같이 넘파이 배열의 길이와 크기를 재는 다양한 방법이 있습니다. 배열의 구조와 용도에 따라 적절한 방법을 선택하여 사용할 수 있습니다.

4.1.6 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.

다음은 Python의 NumPy 라이브러리를 사용하여 벡터 연산을 수행하는 예제입니다.

```
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 간 뺄셈
sub_result = a - b
print("벡터 뺄셈:", sub_result)

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

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

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

```
벡터 덧셈: [5 7 9]
벡터 뺄셈: [-3 -3 -3]
벡터 곱셈: [ 4 10 18]
벡터 나눗셈: [0.25 0.4  0.5 ]
벡터 나머지 연산: [1 2 3]
```

4.1.7 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.

4.1.7.a 벡터 연산 예제

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

$$\begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix} + \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 9 \end{pmatrix}$$

위와 같은 연산을 Python에서는 다음과 같이 표현할 수 있습니다.

```
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]
```

다음과 같이 상수를 곱하는 연산 역시 마찬가지입니다.

$$2\begin{pmatrix}1\\2\\4\\5\end{pmatrix}=\begin{pmatrix}2\\4\\8\\10\end{pmatrix}$$

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

```
상수 곱셈: [ 2  4  8 10]
```

벡터화(Vectorized) 코드를 사용하면 반복문을 사용하지 않고도 벡터를 한 번에 처리할 수 있습니다.

4.1.8 NumPy의 브로드캐스팅(Broadcasting) 개념

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.

4.1.8.a 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.

4.1.8.a.a 브로드캐스팅 안되는 경우

Python에서 NumPy를 사용하여, 길이가 다른 두 벡터의 덧셈을 살펴보겠습니다.

```
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
print("브로드캐스팅 결과:", result)
```

```
ValueError: operands could not be broadcast together with shapes (4,) (2,)
```

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

```
# 배열의 shape 확인
print("a의 shape:", a.shape)
print("b의 shape:", b.shape)
```

```
a의 shape: (4,)
b의 shape: (2,)
```

이 경우, 배열의 `shape`을 맞춰 연산을 수행할 수 있습니다:

```
# b 배열을 반복 확장하여 a의 길이에 맞춤
b_repeated = np.tile(b, 2)
print("반복된 b 배열:", b_repeated)

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

```
반복된 b 배열: [1 2 1 2]
브로드캐스팅 결과: [2 4 4 6]
```

4.1.8.a.b 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.])
```

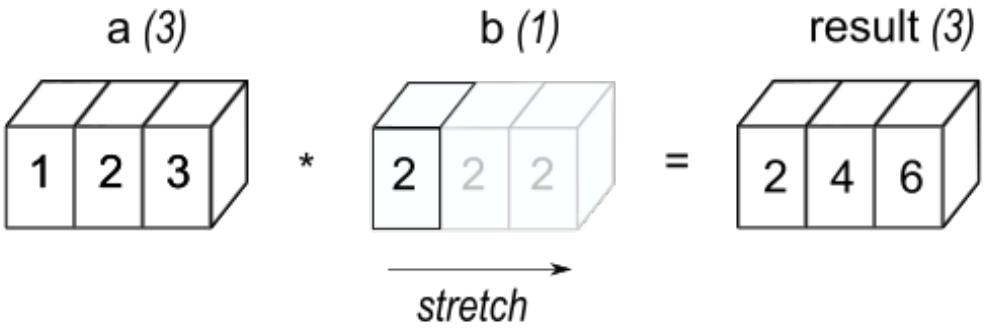

Figure 1: 1차원 브로드캐스팅

다음 예제에서는 2차원 배열과 1차원 배열을 사용하여 브로드캐스팅을 통한 연산을 수행합니다.

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]])
matrix.shape
# 1차원 배열 생성
vector = np.array([1.0, 2.0, 3.0])
vector.shape
# 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.]]
```

위 예제에서, `matrix`는 4x3 크기의 2차원 배열이고, `vector`는 3 요소를 가진 1차원 배열입니다. 브로드캐스팅을 사용하면 `vector`가 `matrix`의 각 행에 반복적으로 더해집니다.

- `matrix`의 shape: (4, 3)

- `vector`의 shape: (3,)

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

결과는 다음과 같습니다:

$$\begin{pmatrix} 0+1 & 0+2 & 0+3 \\ 10+1 & 10+2 & 10+3 \\ 20+1 & 20+2 & 20+3 \\ 30+1 & 30+2 & 30+3 \end{pmatrix}$$

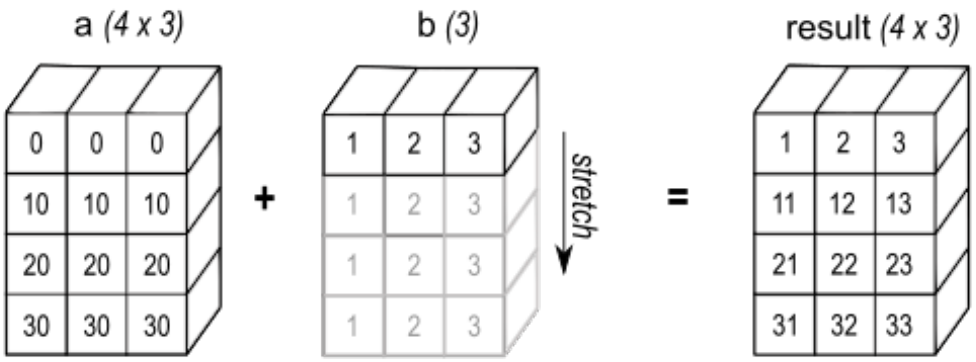

Figure 2: 2차원 브로드캐스팅

4.1.8.a.c 배열에 세로 벡터 더하기

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

```
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]

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

```
vector.shape
matrix.shape
```

```
SyntaxError: invalid syntax (1440457060.py, line 13)
```

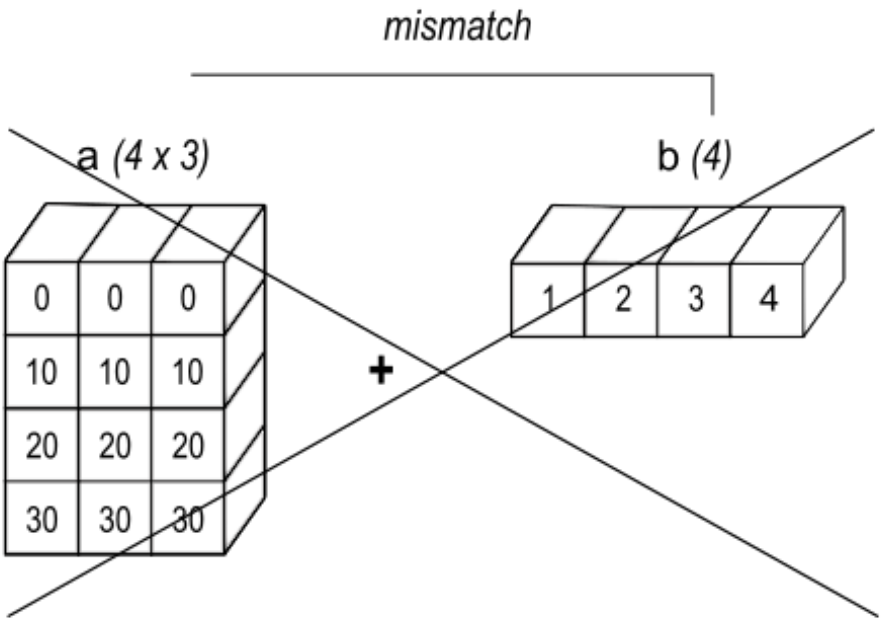

Figure 3: 배열에 세로 벡터가 안 더해지는 이유

이런 경우, 벡터를 세로벡터로 바꿔준 후, shape을 맞춰 더해주면 브로드캐스트가 작동하게 됩니다.

```
# 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.]]
```

4.1.9 참고자료

- 넘파이 브로드캐스팅