NUMPY CƠ BẢN

Dinh-Tiem Nguyen và Quang-Vinh Dinh

AI VIET NAM - AIO 2024

Mục lục

1 Mở đầu

NumPy là một thư viện mã nguồn mở trong Python được sử dụng để làm việc với mảng (array)-cấu trúc dữ liệu chính của numpy và cho phép thực hiện các thao tác toán học nhanh chóng và hiệu quả. Để bắt đầu sử dụng NumPy, chúng ta cần phải cài đặt thư viện này trong môi trường Python qua cú pháp:

```
pip install numpy
```

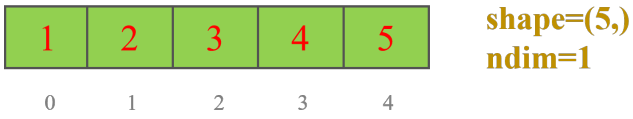
Sau khi cài đặt xong, bạn có thể kiểm tra xem NumPy đã được cài đặt thành công hay chưa bằng cách mở Python interpreter (hoặc Jupyter Notebook) và nhập:

```
import numpy as np
print(np.__version__)
```

Nếu NumPy được cài đặt thành công, bạn sẽ thấy phiên bản của NumPy được in ra.

2 Các phương pháp tạo mảng

2.1 Tạo mảng từ dữ liệu có sẵn



Hình 1: Minh họa mảng

np.array là một hàm trong NumPy dùng để tạo mảng (array) từ dữ liệu có sẵn như list hoặc tuple. Cú pháp:

```
np.array(object)
```

- object: Dữ liệu đầu vào (list, tuple, hoặc các kiểu dữ liệu khác).

Ví dụ sau sẽ tạo một mảng NumPy từ một list gồm các số nguyên [1, 2, 3, 4, 5].

```
#aivietnam
import numpy as np
# Tạo mảng từ list
list_data = [1, 2, 3, 4, 5]
array_from_list = np.array(list_data)
print(array_from_list)
```

```
================ Output ================
[1 2 3 4 5]



========================================
```

Chúng ta có thể kiểm tra các thuộc tính cơ bản của array qua các phương thức sau:

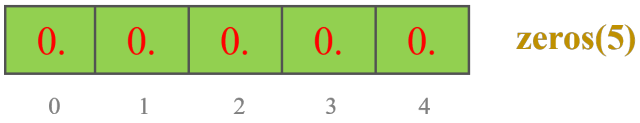
- Shape: kích thước của mảng, tức là số phần tử trong mỗi chiều của chúng.
- dtype: kiểu dữ liệu của các phần tử trong mảng.
- size: số lượng phần tử trong mảng
- ndim: số chiều của mảng

Ví dụ chương trình dưới đây sẽ kiểm tra các thuộc tính của một mảng:

```python
#aivietnam
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print("arr:", arr)
print("Shape of arr:", arr.shape)
print("Size of arr:", arr.size)
print("Data type of arr:", arr.dtype)
print("Number of dimensions of arr:",
        arr.ndim)
```

```
================ Output ================
arr: [1 2 3 4 5]
Shape of arr: (5,)
Size of arr: 5
Data type of arr: int64
Number of dimensions of arr: 1



========================================
```

2.2 Tạo mảng 0-zero()



Hình 2: Minh họa mảng 0

np.zeros là hàm dùng để tạo một mảng với tất cả các phần tử đều là 0. Cú pháp:

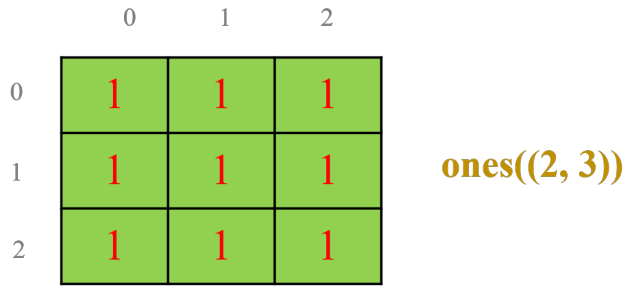
```python
np.zeros(shape)
```

Ví dụ dưới đây sẽ tạo một mảng 1 chiều với 5 phần tử, tất cả đều là 0.

```python
#aivietnam
import numpy as np

# Tạo mảng 1 chiều với 5 phần tử đều là 0
zeros_array = np.zeros(5)
print(zeros_array)
```

```
================ Output ================
[0. 0. 0. 0. 0.]



========================================
```

2.3 Tạo mảng 1-ones()



Hình 3: Minh họa mảng 1

Tương tự với zeros, hàm ones tạo mảng chứa toàn số 1 với đầu vào là kích thước do người dùng chỉ định. Cú pháp:

```python
np.ones(shape)
```

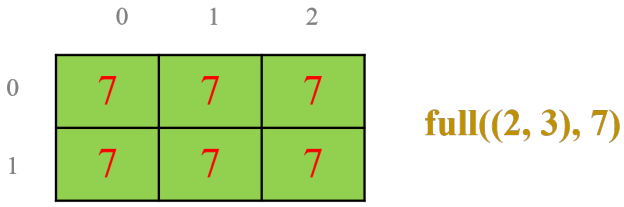
- shape: Kích thước của mảng (có thể là một số nguyên hoặc một tuple).

Ví dụ sau sẽ tạo một mảng 2 chiều có kích thước 3x3 với tất cả các phần tử đều là 1

```python
#aivietnam
import numpy as np

# Tạo mảng 2 chiều 3x3 với tất cả các phần tử đều là 1
ones_array = np.ones((3, 3))
print(ones_array)
```

```
================ Output ================
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]


========================================
```

2.4 Tạo mảng với giá trị chỉ định



Hình 4: Minh họa hàm full

np.full là hàm dùng để tạo một mảng với tất cả các phần tử đều có cùng một giá trị xác định. Cú pháp:

```python
np.full(shape, fill_value)
```

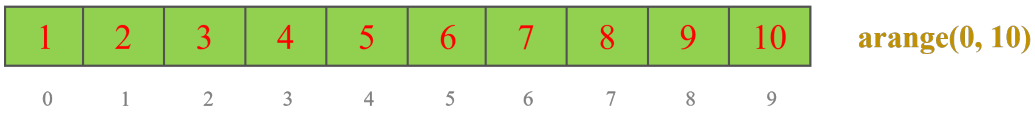
- shape: Kích thước của mảng (có thể là một số nguyên hoặc một tuple).
- fill_value: Giá trị sẽ được gán cho tất cả các phần tử của mảng.

Ví dụ sau sẽ tạo một mảng NumPy 2 chiều, kích thước 2x3 với tất cả các phần tử đều có giá trị là 7.

```python
#aivietnam
import numpy as np

# Tạo mảng 2 chiều 2x3 với tất cả các phần tử đều có giá trị là 7
full_array = np.full((2, 3), 7)
print(full_array)
```

```
================ Output ================
[[7 7 7]
 [7 7 7]]


========================================
```

2.5 Tạo mảng với arange()



Hình 5: Minh họa hàm arange

np.arange là một hàm dùng để tạo một mảng chứa các số nguyên liên tiếp trong một khoảng xác định. Cú pháp:

```python
np.arange(start, stop, step)
```

- start: Giá trị bắt đầu của dãy số(giá trị này bao gồm trong mảng được tạo).

- stop: Giá trị kết thúc của dãy số (giá trị này không bao gôm trong mảng được tạo).
- step: Khoảng cách giữa các giá trị trong dãy số (mặc định là 1).

Ví dụ dưới đây sẽ tạo một mảng chứa các số nguyên từ 0 đến 9.

```
#aivietnam
import numpy as np

# Tạo mảng chứa các số nguyên từ 0 đến 9
arange_array = np.arange(0, 10)
print(arange_array)
```

```
================ Output ================
[0 1 2 3 4 5 6 7 8 9]



========================================
```

2.6 Tạo ma trận đơn vị với eye()

np.eye là một hàm dùng để tạo một ma trận đơn vị, trong đó các phần tử trên đường chéo chính là 1 và các phần tử còn lại là 0. Cú pháp:

```
np.eye(N)
```

- N: Số hàng và số cột của ma trận (ma trận vuông).

Ví dụ sau sẽ tạo một ma trận đơn vị 3x3, với các phần tử trên đường chéo chính là 1 và các phần tử còn lại là 0.

```
#aivietnam
import numpy as np

# Tạo ma trận đơn vị 3x3
eye_matrix = np.eye(3)
print(eye_matrix)
```

```
================ Output ================
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

========================================
```

2.7 Tạo mảng ngẫu nhiên - random

np.random là một module trong NumPy cung cấp các hàm để tạo mảng với các giá trị ngẫu nhiên. Trong phần này, các ví dụ về random sẽ sử dụng ramdom seed=2024 để đảm bảo kết quả đầu ra có thể tái tạo lại.

a) random.rand()

np.random.rand tạo một mảng với các giá trị ngẫu nhiên nằm trong khoảng từ 0 đến 1. Cú pháp:

```
np.random.rand(d0, d1, ..., dn)
```

- d0, d1, ..., dn: Kích thước của mảng.

Ví dụ sau sẽ tạo một mảng NumPy 2x3 với các giá trị ngẫu nhiên từ 0 đến 1.

```
#aivietnam
import numpy as np
np.random.seed(2024)

# Tạo mảng 2x3 với các giá trị ngẫu nhiên
   từ 0 đến 1
rand_array = np.random.rand(2, 3)
print(rand_array)
```

```
================= Output =================
[[0.58801452 0.69910875 0.18815196]
 [0.04380856 0.20501895 0.10606287]]
```

```
==========================================
```

b) random.randint()

np.random.randint tạo một mảng với các giá trị ngẫu nhiên là số nguyên trong một khoảng xác định. Cú pháp:

```
np.random.randint(low, high=None, size=None)
```

- low: Giá trị nhỏ nhất (bao gồm trong mảng đầu ra).
- high: Giá trị lớn nhất (không bao gồm trong mảng đầu ra).
- size: Kích thước của mảng.

Ví dụ sau sẽ tạo một mảng 3x3 với các số nguyên ngẫu nhiên từ 1 đến 10.

```
#aivietnam
import numpy as np
np.random.seed(2024)

# Tạo mảng 3x3 với các số nguyên ngẫu nhiên từ 1 đến 10
randint_array = np.random.randint(1, 10, (3, 3))
print(randint_array)
```

```
================= Output =================
[[9 1 1]
 [5 8 2]
 [4 3 1]]
==========================================
```

3 Các phương pháp truy cập mảng

3.1 Indexing

Indexing trong NumPy cho phép ta truy cập vào từng phần tử của mảng bằng cách sử dụng chỉ số của nó. Để hiểu rõ hơn về cách sử dụng indexing, chúng ta sẽ thực hiện các ví dụ sau đây.
Ví dụ 1: Chúng ta sẽ tạo một mảng 1 chiều và truy cập vào phần tử đầu tiên và phần tử cuối cùng của mảng này.

```
#aivietnam
import numpy as np

# Tạo mảng 1 chiều
arr1d = np.array([1, 2, 3, 4, 5])

# Truy cập vào phần tử đầu tiên của arr1d
first_element = arr1d[0]
print("Phần tử đầu tiên của arr1d:", first_element)

# Truy cập vào phần tử cuối cùng của arr1d
last_element = arr1d[-1]
print("Phần tử cuối cùng của arr1d:", last_element)
```

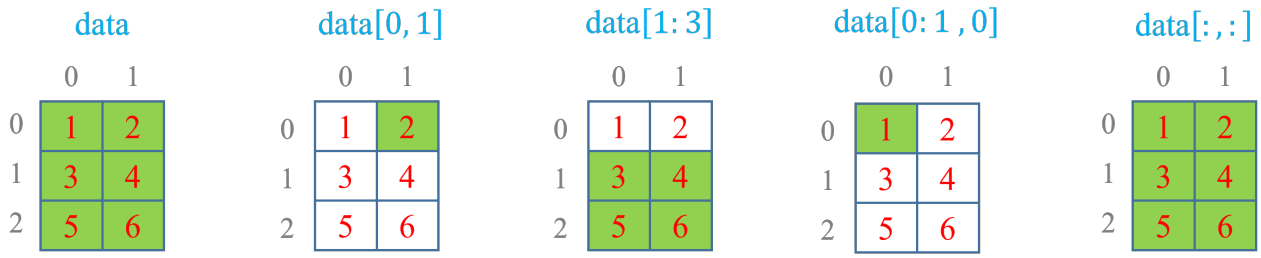
```
================= Output =================
Phần tử đầu tiên của arr1d: 1
Phần tử cuối cùng của arr1d: 5
==========================================
```

Ví dụ 2: Chúng ta sẽ tạo một mảng 2 chiều và truy cập vào một phần tử cụ thể ở hàng thứ nhất và cột thứ hai của mảng.

```
#aivietnam
import numpy as np

# Tạo mảng 2 chiều
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d)

# Truy cập vào phần tử ở hàng thứ nhất và cột thứ hai của arr2d
element_1_2 = arr2d[1, 1]
print("Phần tử ở hàng thứ nhất và cột thứ hai của arr2d:", element_1_2)
```

```
================ Output ================
[[1 2 3]
 [4 5 6]]
Phần tử ở hàng thứ nhất và cột thứ hai của arr2d: 5
========================================
```

3.2 Slicing



Hình 6: Minh họa slicing

Slicing trong NumPy cho phép ta trích xuất một phần của mảng bằng cách chỉ định khoảng chỉ mục. Để hiểu rõ hơn về cách sử dụng slicing, chúng ta sẽ thực hiện các ví dụ sau đây.
Ví dụ 1: Chúng ta sẽ tạo một mảng 1 chiều và lấy ra một phần của mảng này từ chỉ số thứ nhất đến chỉ số thứ tư.

```
#aivietnam
import numpy as np

# Tạo mảng 1 chiều
arr1d = np.array([1, 2, 3, 4, 5])

# Lấy ra một phần của arr1d từ chỉ số thứ nhất đến chỉ số thứ tư
slice_arr1d = arr1d[1:4]
print("arr1d:", arr1d)
print("Sclicing arr1d từ chỉ số 1 đến 3:", slice_arr1d)
```

```
================ Output ================
arr1d: [1 2 3 4 5]
Sclicing arr1d từ chỉ số 1 đến 3: [2 3 4]
========================================
```

Ví dụ 2: Chúng ta sẽ tạo một mảng 2 chiều và lấy ra một phần của mảng này từ hàng thứ nhất đến hàng thứ ba và từ cột thứ hai đến cột cuối cùng.

```python
#aivietnam
import numpy as np

# Tạo mảng 2 chiều
arr2d = np.array([[1, 2, 3], [4, 5, 6],
    [7, 8, 9]])

# Lấy ra một phần của arr2d từ hàng thứ
#nhất đến hàng thứ ba và từ cột thứ hai
#đến cột cuối cùng
slice_arr2d = arr2d[1:3, 1:]
print("arr2d:\n", arr2d)
print("Slicing arr2d từ hàng 1 đến 2 và cộ
    t 1 đến cuối cùng:\n", slice_arr2d)
```

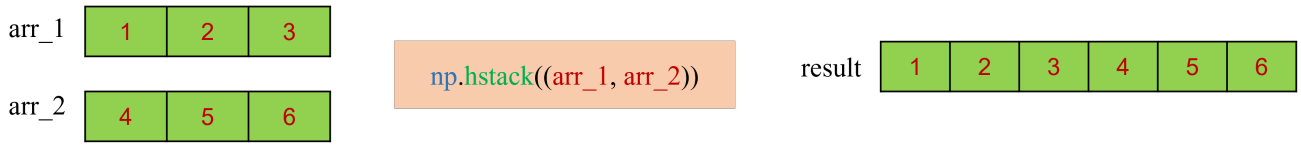
```
================ Output ================
arr2d:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Slicing arr2d từ hàng 1 đến 2 và cột 1 đến
    cuối cùng:
[[5 6]
 [8 9]]
========================================
```

4 Các phương pháp nối mảng

1. Mô tả

Trong các thư viện Numpy, Pytorch, Tensorflow các hàm **hstack**, **vstack**, và **concatenate** được sử dụng để nối các array hoặc các tensor lại với nhau theo các hướng khác nhau. Trong bài tập này, chúng ta sẽ tìm hiểu cách sử dụng ba hàm trên.

4.1 hstack



Hình 7: Minh họa hstack

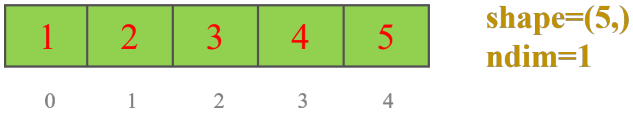
Hàm **hstack** được sử dụng để nối các array theo chiều ngang, tức là nối chúng thành một cấu trúc dữ liệu lớn hơn theo chiều thứ hai. Quá trình này giúp kết hợp dữ liệu từ các nguồn khác nhau hoặc mở rộng kích thước của cấu trúc dữ liệu hiện có.
Ví dụ sau sử dụng hstack để nối các array theo chiều ngang:

```python
#Numpy code
import numpy as np

# Tạo hai mảng
arr_1 = np.array([1, 2, 3])
arr_2 = np.array([4, 5, 6])

# Nối hai mảng theo trục ngang (axis=0)
arr_3 = np.hstack((arr_1, arr_2))

# In kết quả
print("Mảng 1:\n", arr_1)
print("Mảng 2:\n", arr_2)
print("Mảng sau khi nối theo trục
ngang:\n", arr_3)
```

```
================ Output ================
Mảng 1:
 [1 2 3]
Mảng 2:
 [4 5 6]
Mảng sau khi nối theo trục ngang:
 [1 2 3 4 5 6]
========================================
```

Lưu ý khi sử dụng **hstack** là các array hoặc tensor cần phải có cùng độ dài trong chiều dọc (cùng số hàng). Nếu các array/tensor không có cùng độ dài trong chiều dọc sẽ gây ra lỗi khi thực thi chương trình.

4.2 vstack



Hình 8: Minh họa vstack

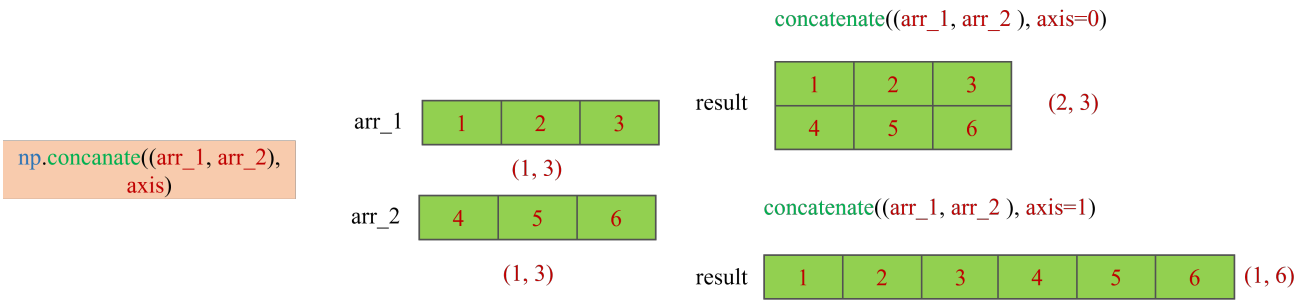
Hàm **vstack** trong NumPy, PyTorch được sử dụng để nối các array hoặc tensor theo chiều dọc, tức là nối chúng thành một cấu trúc dữ liệu lớn hơn theo chiều thứ nhất. Quá trình này giúp kết hợp dữ liệu từ các nguồn khác nhau hoặc mở rộng kích thước của cấu trúc dữ liệu hiện có.
Ví dụ dưới đây sử dụng **vstack** để nối các array theo chiều dọc.

```python
# Numpy code
import numpy as np
# Tạo hai array
arr_1 = np.array([1, 2, 3])
arr_2 = np.array([4, 5, 6])
# Nối hai array theo chiều dọc
arr_3 = np.vstack((arr_1, arr_2))
# In kết quả
print("Array 1:\n", arr_1)
print("Array 2:\n", arr_2)
print("Array sau khi nối theo chiều dọc:\n
    ", arr_3)
```

```
================= Output ================
Array 1:
 [1 2 3]

Array 2:
 [4 5 6]

Array sau khi nối theo chiều dọc:
 [[1 2 3]
 [4 5 6]]

=========================================
```

Lưu ý khi sử dụng **vstack** là các array hoặc tensor cần phải có cùng số cột. Nếu các array/tensor không có cùng số cột sẽ gây ra lỗi khi thực thi chương trình.

4.3 Concatenate



Hình 9: Minh họa concatenate

Hàm **concatenate** được sử dụng để nối các mảng theo chiều cụ thể.
Ví dụ sau sẽ tạo hai mảng từ list 2D là [[1 , 2 , 3] , [4 , 5 , 6]] và [[7 , 8 , 9] , [10 , 11 , 12]]. Sau đó sử dụng hàm **concatenate** để nối theo trục ngang và trục dọc

```python
# Numpy code
import numpy as np
# Tạo hai array 2D
arr_1 = np.array([[1, 2, 3],
                  [4, 5, 6]])
arr_2 = np.array([[7, 8, 9],
                  [10, 11, 12]])
# Nối hai array theo chiều dọc axis = 0
arr_3 = np.concatenate((arr_1, arr_2),
    axis=0)
# Nối hai array theo chiều ngang axis = 1
arr_4 = np.concatenate((arr_1, arr_2),
    axis=1)
# In kết quả
print("Array 1:\n", arr_1)
print("Array 2:\n", arr_2)
print("Nối theo chiều dọc (axis=0):\n",
    arr_3)
print("Nối theo chiều ngang (axis=1):\n",
    arr_4)
```

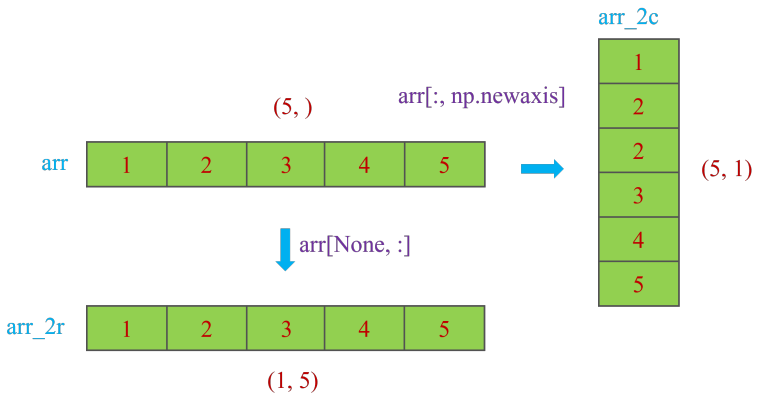
```
================ Output ===============
Array 1:
 [[1 2 3]
 [4 5 6]]

Array 2:
 [[ 7  8  9]
 [10 11 12]]

Nối theo chiều dọc (axis=0):
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Nối theo chiều ngang (axis=1):
 [[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]

=========================================
```

Trong chương trình trên ta sử dụng **np.concatenate** để nối hai array theo chiều dọc (axis=0) và chiều ngang (axis=1). Trong đó **arr_1 và arr_2** là hai array 2D được tạo bằng NumPy. Tiếp theo, **arr_3** được tạo bằng cách nối **arr_1** và **arr_2** theo chiều dọc (axis=0), nghĩa là nối theo hàng. Cuối cùng **arr_4** được tạo bằng cách nối **arr_1** và **arr_2** theo chiều ngang (axis=1), nghĩa là nối theo cột.
Khi sử dụng concatenate các bạn cần lưu ý, đối với việc nối hai array, tensor 1D theo chiều 1 sẽ gặp lỗi, lí do là vì các array, tensor 1D chỉ có 1 chiều là chiều 0. Chính vì vậy mà ta cần chuyển đổi chúng sang dạng 2D trước khi nối chúng lại theo chiều 1. Một lựa chọn dễ dàng hơn là sử dụng vstack.

5 Các phương pháp thay đổi chiều của mảng

5.1 Slicing, expand_dims

Khi thực hiện tính toán với array, tensor, chúng ta sẽ thường xuyên phải điều chỉnh số chiều của chúng. Để thêm một chiều mới, chúng ta sử dụng **newaxis** hoặc **expand_dims**.



Hình 10: Minh họa cách thêm chiều dữ liệu

Ví dụ sau đây thực hiện thêm một chiều mới vào array sử dụng **newaxis**:

```python
#Numpy code
import numpy as np
# Tạo một array 1D
arr_1 = np.array([1, 2, 3])
# 1D -> 2D
arr_2 = arr_1[np.newaxis, :]
# 2D -> 5D
arr_3 = arr_2[np.newaxis, :, np.newaxis,
    :, np.newaxis]
print("array 1: ", arr_1, arr_1.shape)
print("array 2: ", arr_2, arr_2.shape)
print("array 3:\n ", arr_3, arr_3.shape)
```

```
================ Output ===============
array 1:  [1 2 3] (3,)

array 2:  [[1 2 3]] (1, 3)

array 3:
  [[[[[1]
     [2]
     [3]]]]] (1, 1, 1, 3, 1)


=======================================
```

Trong ví dụ trên, ta tạo một array 1D có giá trị [1, 2, 3]. Tiếp theo ta sử dụng **np.newaxis** để thêm một chiều mới ở vị trí 0, chuyển từ array 1D thành array 2D. Cuối cùng ta dụng **np.newaxis** để thêm chiều mới, chuyển từ array 2D thành array 5D. Các chiều mới được thêm vào ở vị trí 0, 2, và 4.
Ở đây, dấu ":"Thể hiện chiều của array cũ, dấu hai chấm đầu tiên là chiều thứ nhất, dấu hai chấm thứ hai là chiều thứ 2 của array cũ. Ta có thể đặt dấu hai chấm này ở các vị trí khác nhau, tùy thuộc vào mục đích tạo array mới. "**np.newaxis**"là chiều mới ta muốn tạo, có thể đặt ở các vị trí khác nhau, ta cũng có thể thay thế nó bằng "**None**"hoặc "**...**".
Ngoài ra ta cũng có thể sử dụng cách thứ hai, **np.expand_dims** để thêm chiều mới cho array.

```python
# Numpy code
import numpy as np
# Tạo một array 1D
arr_1 = np.array([1, 2, 3])
# Thêm chiều mới, chuyển từ 1D -> 2D
arr_2 = np.expand_dims(arr_1, axis=0)
# Thêm nhiều chiều mới, chuyển từ 2D -> 5D
arr_3 = np.expand_dims(arr_2,
        axis=(0, 2, 4))
# In kết quả
print("array 1:", arr_1, arr_1.shape)
print("array 2:", arr_2, arr_2.shape)
print("array 3:\n", arr_3, arr_3.shape)
```

```
================ Output ===============
array 1: [1 2 3] (3,)

array 2: [[1 2 3]] (1, 3)

array 3:
 [[[[[1]
    [2]
    [3]]]]] (1, 1, 1, 3, 1)



=======================================
```

5.2 reshape

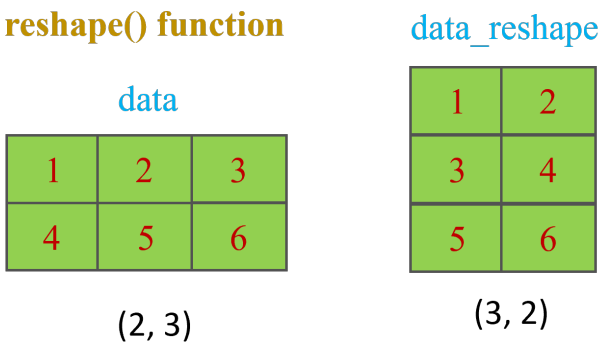
reshape là một hàm được sử dụng để thay đổi hình dạng của một mảng mà không làm thay đổi dữ liệu bên trong nó. Kích thước mảng mới được chỉ định thông qua tham số của hàm **reshape**.
Cú pháp sử dụng reshape:

```python
np.reshape(input, newshape)
```

Trong đó:

- input: là mảng đầu vào.
- newshape là hình dạng mới ta muốn chuyển đổi.

Trong ví dụ sau, ta sẽ sử dụng **reshape** để giảm chiều array, tensor:



Hình 11: Minh họa cách sử dụng hàm reshape

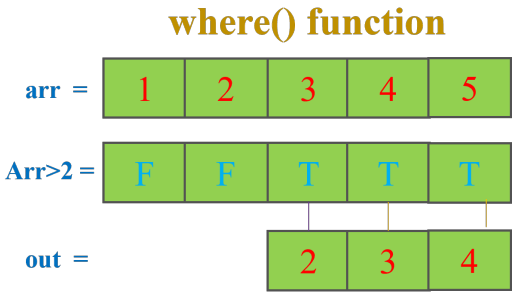
```python
# Numpy code
import numpy as np
# Tạo một array 2D
arr_2D = np.array([[1, 2, 3], [4, 5, 6]])
# (2, 3)->(3, 2)
new_arr_2D = np.reshape(arr_2D, (3, 2))

# Chuyển từ 2D -> 1D
arr_1D = np.reshape(arr_2D,
                    newshape=(6, ))
print("array 2D:\n", arr_2D, arr_2D.shape)
print("new array 2D :\n", new_arr_2D,
    new_arr_2D.shape)
print("array 1D:\n", arr_1D, arr_1D.shape)
```

```
================ Output ===============
array 2D:
 [[1 2 3]
 [4 5 6]] (2, 3)
new array 2D :
 [[1 2]
 [3 4]
 [5 6]] (3, 2)
array 1D:
 [1 2 3 4 5 6] (6,)

=======================================
```

Trong ví dụ trên, ta tạo một array 2D có 2 hàng và 3 cột. Sử dụng **np.reshape** để thay đổi hình dạng của array từ 2D thành 1D với hình dạng mới là (6,), nghĩa là một array có 6 phần tử. Ở đây 6= 2x3, bằng với số phần tử của cũ, ta cần lưu ý khi thay đổi shape thì cần đảm bảo số lượng phần tử không được thay đổi khi **reshape**.

6 Các phương pháp tìm kiếm và sắp xếp mảng

6.1 Tìm kiếm phần tử với điều kiện - where()



Hình 12: Minh họa where

Hàm np.where được sử dụng để tìm vị trí của các phần tử trong mảng thỏa mãn một điều kiện nhất định. Cú pháp:

```
np.where(condition)
```

- condition: Là một mảng boolean hoặc biểu thức trả về mảng boolean.

Để hiểu rõ hơn về cách sử dụng np.where, chúng ta sẽ tạo một mảng và tìm các vị trí của phần tử lớn hơn 2 và in ra giá trị các phần tử đó.

```
#aivietnam
import numpy as np
# Tạo mảng
arr = np.array([1, 2, 3, 4, 5])
# Tìm vị trí của các phần tử lớn hơn 2
result = np.where(arr > 2)
print("Vị trí của các phần tử lớn hơn 2:",
     result)
print("Giá trị các phần tử tại vị trí tìm
    được:", arr[result])
```

```
================ Output ================
Vị trí của các phần tử lớn hơn 2:
(array([2, 3, 4]),)
Giá trị các phần tử tại vị trí tìm được:
    [3 4 5]
========================================
```

6.2 Tìm giá trị lớn nhất và nhỏ nhất

Hàm np.max và np.min được sử dụng để tìm giá trị lớn nhất và nhỏ nhất trong mảng. Cú pháp:

```
np.max(array, axis)
np.min(array, axis)
```

Để hiểu rõ hơn về cách sử dụng np.max và np.min, chúng ta sẽ tạo một mảng và tìm giá trị lớn nhất và nhỏ nhất trong mảng đó.

```
#aivietnam
import numpy as np
# Tạo mảng
arr = np.array([1, 2, 3, 4, 5])
# Tìm giá trị lớn nhất
max_value = np.max(arr)
print("Giá trị lớn nhất trong mảng:",
    max_value)
# Tìm giá trị nhỏ nhất
min_value = np.min(arr)
print("Giá trị nhỏ nhất trong mảng:",
    min_value)
```

```
================ Output ================
Giá trị lớn nhất trong mảng: 5
Giá trị nhỏ nhất trong mảng: 1
========================================
```

Ngoài ra, chúng ta có thể tìm kiếm phần tử nhỏ nhất, lớn nhất theo từng chiều cụ thể đối với mảng nhiều chiều.

```
#aivietnam
import numpy as np

# Tạo mảng 2 chiều
arr2d = np.array([[1, 2, 3], [4, 5, 6]])

# Tìm giá trị lớn nhất theo từng chiều
max_value_row = np.max(arr2d, axis=1)
max_value_col = np.max(arr2d, axis=0)
print("Giá trị lớn nhất theo hàng trong
mảng 2 chiều:", max_value_row)
print("Giá trị lớn nhất theo cột trong
mảng 2 chiều:", max_value_col)

# Tìm giá trị nhỏ nhất theo từng chiều
min_value_row = np.min(arr2d, axis=1)
min_value_col = np.min(arr2d, axis=0)
print("Giá trị nhỏ nhất theo hàng trong mả
```

```
    ng
2 chiều:", min_value_row)
print("Giá trị nhỏ nhất theo cột trong
mảng 2 chiều:", min_value_col)
```

```
================ Output ===============
Giá trị lớn nhất theo hàng trong mảng 2
    chiều: [3 6]
Giá trị lớn nhất theo cột trong mảng 2
chiều: [4 5 6]
Giá trị nhỏ nhất theo hàng trong mảng 2
    chiều: [1 4]
Giá trị nhỏ nhất theo cột trong mảng 2
chiều: [1 2 3]
=======================================
```

6.3 Tìm chỉ số của giá trị lớn nhất và nhỏ nhất

Hàm np.argmax và np.argmin được sử dụng để tìm chỉ số của giá trị lớn nhất và nhỏ nhất trong mảng. Cú pháp:

```
np.argmax(array, axis)
np.argmin(array, axis)
```

- array: mảng đầu vào
- axis: Chiều thực hiện tìm kiếm

Để hiểu rõ hơn về cách sử dụng np.argmax và np.argmin, chúng ta sẽ tạo một mảng và tìm chỉ số của giá trị lớn nhất và nhỏ nhất trong mảng đó và tìm kiếm theo các chiều khác nhau.

```python
#aivietnam
import numpy as np

# Tạo mảng 2 chiều
arr2d = np.array([[1, 2, 3], [4, 5, 6]])

# Tìm chỉ số của giá trị lớn nhất trong toàn bộ mảng
index_max2d = np.argmax(arr2d)
print("Chỉ số của giá trị lớn nhất trong mảng 2 chiều:", index_max2d)

# Tìm chỉ số của giá trị nhỏ nhất trong toàn bộ mảng
index_min2d = np.argmin(arr2d)
print("Chỉ số của giá trị nhỏ nhất trong mảng 2 chiều:", index_min2d)

# Tìm chỉ số của giá trị lớn nhất theo từng hàng
index_max_row = np.argmax(arr2d, axis=1)
print("Chỉ số của giá trị lớn nhất theo từng hàng trong mảng 2 chiều:", index_max_row)

# Tìm chỉ số của giá trị lớn nhất theo từng cột
index_max_col = np.argmax(arr2d, axis=0)
print("Chỉ số của giá trị lớn nhất theo từng cột trong mảng 2 chiều:", index_max_col)
```

```
================ Output ================
Mảng 2 chiều ban đầu:
 [[1 2 3]
 [4 5 6]]

Chỉ số của giá trị lớn nhất trong mảng 2 chiều: 5

Chỉ số của giá trị nhỏ nhất trong mảng 2 chiều: 0

Chỉ số của giá trị lớn nhất theo từng hàng trong mảng 2 chiều: [2 2]

Chỉ số của giá trị lớn nhất theo từng cột trong mảng 2 chiều: [1 1 1]

========================================
```

6.4 Sắp xếp

Hàm np.sort được sử dụng để sắp xếp các phần tử của mảng theo thứ tự tăng dần. Cú pháp:

```python
np.sort(array, axis)
```

- array: mảng cần sắp xếp
- axis: Chiều được chọn để sắp xếp, mặc định là -1, tức là sắp xếp theo chiều cuối cùng của mảng.

Ví dụ dưới đây sẽ thực hiện sắp xếp mảng theo các chiều khác nhau.

```python
#aivietnam
import numpy as np

# Tạo mảng 2 chiều
arr2d = np.array([[3, 1, 2], [6, 5, 4]])
# Sắp xếp theo mặc định
sorted_arr2d = np.sort(arr2d)
print("Sắp xếp mảng theo mặc định:\n",
    sorted_arr2d)
# Sắp xếp theo từng hàng
sorted_arr2d_row = np.sort(arr2d, axis=1)
print("Sắp xếp theo từng hàng:\n",
    sorted_arr2d_row)
# Sắp xếp theo từng cột
sorted_arr2d_col = np.sort(arr2d, axis=0)
print("Sắp xếp theo từng cột:\n",
    sorted_arr2d_col)
```

```
================ Output ===============
Sắp xếp mảng theo mặc định:
 [[1 2 3]
 [4 5 6]]
Sắp xếp theo từng hàng:
 [[1 2 3]
 [4 5 6]]
Sắp xếp theo từng cột:
 [[3 1 2]
 [6 5 4]]




=========================================
```

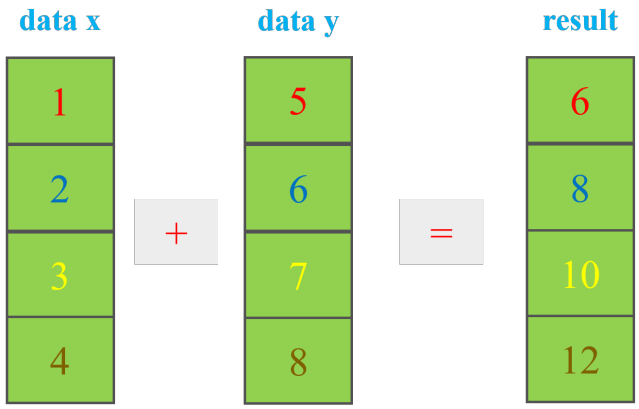
7 Các phép toán trên mảng

NumPy cung cấp nhiều hàm và phép toán để thực hiện các phép toán học phức tạp trên các mảng. Hiểu cách chuyển đổi các công thức toán học thành mã NumPy là rất quan trọng để tận dụng sức mạnh của thư viện này

7.1 Phép cộng - Addition

Phép cộng hai mảng được thực hiện theo cùng một nguyên tắc cơ bản, là phép cộng element-wise (theo từng phần tử). Phép cộng này được định nghĩa như sau: Cho hai array (hoặc tensor) A và B cùng kích thước, phép cộng giữa chúng $(A + B)$ tạo ra một array mới C, với mỗi phần tử C_i được tính bằng cách cộng phần tử A_i với phần tử B_i.

$$C_i = A_i + B_i$$



Hình 13: Minh họa Addition

Trong ví dụ dưới đây, ta thực hiện phép cộng các array bằng cách sử dụng toán tử "+"hoặc hàm add().

```python
#aivietnam
import numpy as np

# Tạo hai array 1D
arr_1 = np.array([1, 2, 3, 4])
arr_2 = np.array([5, 6, 7, 8])

# Thực hiện phép cộng hai array
# Cách 1: Sử dụng toán tử '+'
arr_add_1 = arr_1 + arr_2
# Cách 2: Sử dụng hàm np.add()
arr_add_2 = np.add(arr_1, arr_2)
# In ra màn hình
print(f"arr_1 = \n{arr_1}")
print(f"arr_2 = \n{arr_2}")
print("arr_1 + arr_2")
print(f"arr_add_1 = \n{arr_add_1}")
print(f"arr_add_2 = \n{arr_add_2}")
```

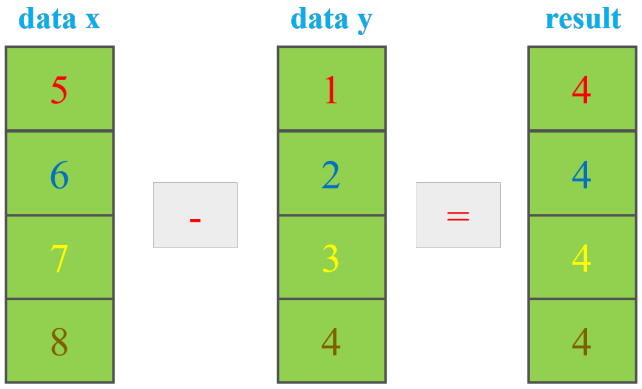
```
================ Output ===============
arr_1 =
[1 2 3 4]

arr_2 =
[5 6 7 8]

arr_1 + arr_2

arr_add_1 =
[ 6  8 10 12]

arr_add_2 =
[ 6  8 10 12]




=======================================
```

Kết quả thực hiện phép cộng sẽ là một array mới là [6, 8, 10, 12] với mỗi phần tử là tổng của các phần tử tương ứng trong arr1 và arr2.

7.2 Phép trừ - Subtraction

Tương tự như phép cộng array, tensor, phép trừ cũng là một phép toán element-wise, tức là mỗi phần tử của array kết quả được tính bằng cách trừ phần tử tương ứng của array thứ nhất cho phần tử tương ứng của array thứ hai. Định nghĩa: Cho hai array cùng kích thước A và B, phép trừ giữa chúng $(A - B)$ tạo ra một array mới C, với mỗi phần tử C_i được tính bằng cách trừ phần tử B_i khỏi phần tử A_i.

$$C_i = A_i - B_i$$



Hình 14: Minh họa Subtraction

Ví dụ phép trừ hai array trong Numpy, chúng ta có thể sử dụng toán tử "-"hoặc hàm **np.subtract()**

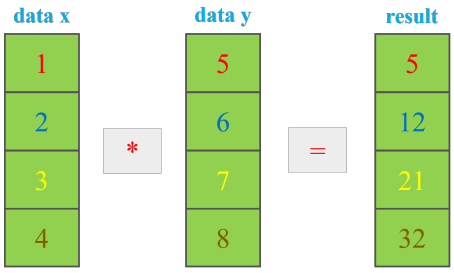
```python
#aivietnam
import numpy as np
# Tạo hai array 1D
arr_1 = np.array([1, 2, 3, 4])
arr_2 = np.array([5, 6, 7, 8])
print("arr_1:", arr_1)
print("arr_2:", arr_2)
# Hiệu hai array
# Cách 1: Sử dụng toán tử '-'
array_sub_1 = arr_1 - arr_2
# Cách 2: Sử dụng hàm np.subtract()
array_sub_2 = np.subtract(arr_1, arr_2)
# In ra màn hình
print("Cách 1\n", array_sub_1)
print("Cách 2\n", array_sub_2)
```

```
================ Output ===============
arr_1: [1 2 3 4]

arr_2: [5 6 7 8]

Cách 1
 [-4 -4 -4 -4]

Cách 2
 [-4 -4 -4 -4]

=======================================
```

7.3 Phép nhân - Multiplication

Phép nhân array được thực hiện theo nguyên tắc element-wise: Cho hai array cùng kích thước A và B, phép nhân giữa chúng $(A \cdot B)$ tạo ra một array mới C, với mỗi phần tử C_{ij} được tính bằng cách nhân phần tử A_{ij} với phần tử B_{ij}.

$$C_{ij} = A_{ij} \cdot B_{ij}$$



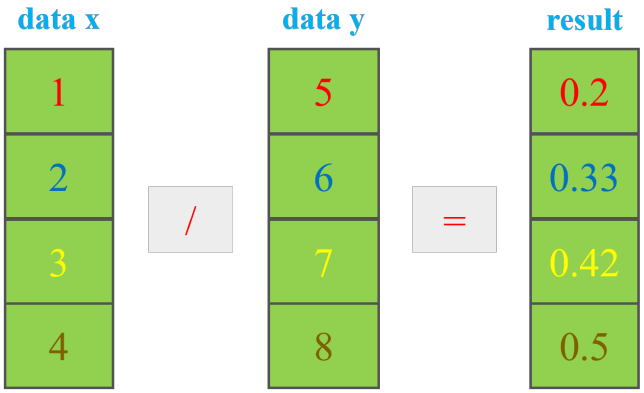
Hình 15: Minh họa Multiplication

Để thực hiện multiplication, ta sử dụng toán tử "*"hoặc hàm **multiply**.

```python
#aivietnam
import numpy as np

# Tạo hai array 1D
arr_1 = np.array([1, 2, 3, 4])
arr_2 = np.array([5, 6, 7, 8])
# In ra giá trị của hai array
print("arr_1:\n", arr_1)
print("arr_2:\n", arr_2)
# Tích hai array
# Cách 1: Sử dụng toán tử '*'
result_mul_1 = arr_1 * arr_2
# Cách 2: Sử dụng hàm np.multiply()
result_mul_2 = np.multiply(arr_1, arr_2)
# In ra màn hình
print("Cách 1:\n", result_mul_1)
print("Cách 2:\n", result_mul_2)
```

```
================ Output ===============
arr_1:
 [1 2 3 4]

arr_2:
 [5 6 7 8]

Cách 1:
 [ 5 12 21 32]

Cách 2:
 [ 5 12 21 32]

=======================================
```

7.4 Phép chia - Division



Hình 16: Minh họa Division

Phép chia mảng được thực hiện theo nguyên tắc element-wise. Cho hai mảng cùng kích thước A và B, phép chia giữa chúng $(A \div B)$ tạo ra một mảng (hoặc tensor) mới C, với mỗi phần tử C_{ij} được tính bằng cách chia phần tử A_{ij} cho phần tử B_{ij}.

$$C_{ij} = \frac{A_{ij}}{B_{ij}}$$

Để thực hiện phép chia mảng ta có thể sử dụng toán tử "/"hoặc hàm divide().

```python
#aivietnam
import numpy as np

# Tạo hai array 1D
arr_1 = np.array([1, 2, 3, 4])
arr_2 = np.array([5, 6, 7, 8])
# In ra giá trị của hai array
print("arr_1:\n", arr_1)
print("arr_2:\n", arr_2)
# Chia hai array
# Cách 1: Sử dụng toán tử '/'
result_div_1 = arr_1 / arr_2
# Cách 2: Sử dụng hàm np.divide()
result_div_2 = np.divide(arr_1, arr_2)
# In ra màn hình
print("Cách 1:\n", result_div_1)
print("Cách 2:\n", result_div_2)
```

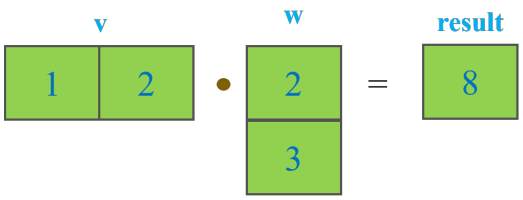
```
================= Output ================
arr_1:
 [1 2 3 4]

arr_2:
 [5 6 7 8]

Cách 1:
 [0.2  0.33333333 0.42857143 0.5]

Cách 2:
 [0.2  0.33333333 0.42857143 0.5]

=========================================
```

7.5 Inner product

Phép tính tích vô hướng (inner product), còn được biết đến với tên gọi khác là dot product. Phép tích vô hướng giữa hai vector có thể được tính để đo lường sự tương quan hoặc hướng của chúng.
Cho hai vectơ $\mathbf{a} = [a_1, a_2, \ldots, a_n]$ và $\mathbf{b} = [b_1, b_2, \ldots, b_n]$ trong không gian Euclidean $\mathbb{R}^n$, phép tích vô hướng của chúng được xác định bởi công thức:

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

Trong thư viện Numpy, hàm **np.dot()** được sử dụng để thực hiện phép nhân ma trận, và nó cũng hỗ trợ tích vô hướng cho các vectơ(array 1D). Cú pháp:



Hình 17: Minh họa inner

```
np.dot(a, b)
```

Trong đó a, b là các array để thực hiện phép nhân. Các chiều của a và b phải thích hợp để thực hiện phép nhân (ví dụ, số cột của a phải bằng số hàng của b).

```
#aivietnam
import numpy as np

# Tạo hai array 1D
arr_1 = np.array([1, 2])
arr_2 = np.array([2, 3])
# In ra giá trị của hai array
print("arr_1:\n", arr_1)
print("arr_2:\n", arr_2)
# Sử dụng hàm np.dot()
result_dot = np.dot(arr_1, arr_2)
# In ra màn hình
print("Kết quả tích vô hướng arr_1, arr_2
    :\n", result_dot)
```

```
================ Output ===============
arr_1:
 [1 2]

arr_2:
 [2 3]

Kết quả tích vô hướng arr_1, arr_2:
 8


=======================================
```

7.6 Matrix-matrix multiplication

Trong toán học, phép nhân ma trận (matrix multiplication) thường được biểu diễn bằng dấu chấm ($\cdot$) hoặc dấu nhân ($\times$). Giả sử A là một ma trận có kích thước $m \times n$ (m hàng, n cột), và B là một ma trận có kích thước $n \times p$ (n hàng, p cột), thì ma trận kết quả C sẽ có kích thước $m \times p$. Quy tắc này còn được biết đến như là quy tắc hàng-cột, vì mỗi phần tử của ma trận kết quả C được tính bằng cách lấy tổng của tích từng phần tử của hàng tương ứng của ma trận A và cột tương ứng của ma trận B. Phép nhân ma trận $C = A \times B$ có thể được mô tả như sau:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

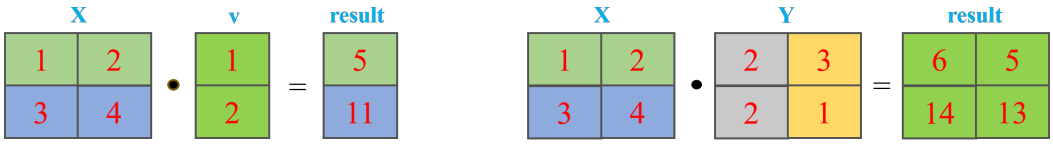
Trong đó:

- C_{ij} là phần tử ở hàng thứ i và cột thứ j của ma trận C.
- A_{ik} là phần tử ở hàng thứ i và cột thứ k của ma trận A.
- B_{kj} là phần tử ở hàng thứ k và cột thứ j của ma trận B.
- Phép toán $\sum_{k=1}^{n}$ biểu diễn việc tính tổng qua tất cả các giá trị của k từ 1 đến n.

Ví dụ:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Trong trường hợp này, $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$, $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$, và tương tự cho các phần tử còn lại của ma trận C.



Hình 18: Minh họa phép nhân ma trận

Trong thư viện Numpy, hàm **np.matmul()** được sử dụng để thực hiện phép nhân ma trận. Hàm này cung cấp một cách linh hoạt để thực hiện nhân ma trận giữa các mảng (hoặc tensor) Numpy. Sử dụng cú pháp:

```
np.matmul(a, b, out=None)
```

Trong đó các tham số được định nghĩa như sau:

- a, b: Các ma trận(mảng) để thực hiện phép nhân. Số cột của ma trận a phải bằng số hàng của ma trận b.
- out: Mảng kết quả. Nếu được chỉ định, kết quả sẽ được lưu trữ vào mảng này.

```
#aivietnam
import numpy as np

# Tạo 2 array 2D
arr_1 = np.array([[1, 2], [3, 4]])
arr_2 = np.array([[5, 6], [7, 8]])

# In ra giá trị của hai array
print("arr_1:\n", arr_1)
print("arr_2:\n", arr_2)

# Sử dụng hàm np.matmul()
result_matmul = np.matmul(arr_1, arr_2)

# In ra màn hình
print("Kết quả tích arr_1, arr_2:\n",
    result_matmul)
```

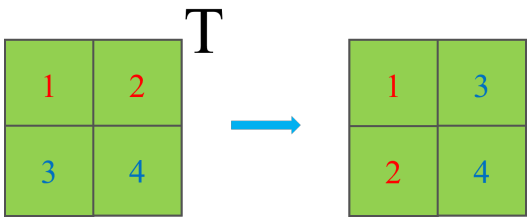
```
================= Output ================
arr_1:
 [[1 2]
 [3 4]]

arr_2:
 [[5 6]
 [7 8]]

Kết quả tích arr_1, arr_2:
 [[19 22]
 [43 50]]

=========================================
```

7.7 Chuyển vị - Transpose

Chuyển vị (Transpose) là một phép toán quan trọng trong đại số tuyến tính, nó thực hiện thay đổi vị trí của các hàng thành cột và ngược lại trong một ma trận. Kí hiệu của phép toán chuyển vị thường được biểu diễn bằng kí hiệu ^{T} hoặc là $^\intercal$. Nếu A là một ma trận với các phần tử a_{ij}, thì chuyển vị của A, ký hiệu là A^T hay $A^\intercal$, có kích thước là số cột của A trở thành số hàng của A và ngược lại. Cụ thể, nếu A có kích thước $m \times n$, thì A^T có kích thước $n \times m$.

$$(A^T)_{ij} = A_{ji}$$

Ví dụ:
Giả sử có ma trận A như sau:



Hình 19: Minh họa transpose

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

Chuyển vị của A, ký hiệu là A^T, sẽ là:

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

Một số tính chất của phép chuyển vị:

- Chuyển vị của một ma trận chuyển vị lại sẽ cho ra ma trận ban đầu: $(A^T)^T = A$.
- Chuyển vị của tổng hai ma trận bằng tổng chuyển vị của từng ma trận: $(A + B)^T = A^T + B^T$.
- Chuyển vị của tích hai ma trận bằng tích đảo ngược vị trí của chuyển vị từng ma trận: $(AB)^T = B^T A^T$.

Trong Numpy, chuyển vị của một mảng (array) có thể được thực hiện bằng cách sử dụng hàm **np.transpose()** hoặc toán tử chuyển vị **.T**. Chuyển vị thay đổi vị trí của các hàng thành cột và ngược lại trong array.

```python
#aivietnam
import numpy as np

#Tạo array 2d
arr_1 = np.array([[1, 2], [3, 4]])
# Chuyển vị array
# Cách 1: Sử dụng hàm np.transpose()
arr_transposed_1 = np.transpose(arr_1)
# Cách 2: Sử dụng toán tử T
arr_transposed_2 = arr_1.T
# In ra màn hình
print("array 1:\n", arr_1)
print("array 1 sau khi chuyển vị, cách 1:\
    n", arr_transposed_1)
print("array 1 sau khi chuyển vị, cách 2:\
    n", arr_transposed_2)
```

```
================ Output ===============
array 1:
 [[1 2]
 [3 4]]

array 1 sau khi chuyển vị, cách 1:
 [[1 3]
 [2 4]]

array 1 sau khi chuyển vị, cách 2:
 [[1 3]
 [2 4]]



=======================================
```

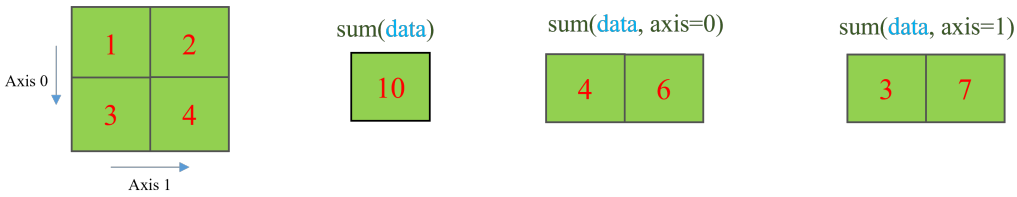
7.8 Summation

Hàm **np.sum()** được sử dụng để tính tổng của các phần tử trong mảng (array). Hàm này có thể được áp dụng trên các mảng 1D, 2D hoặc có số chiều cao hơn. Cú pháp:

```
np.sum(a, axis=None, dtype=None, keepdims=False, initial=0, where=True)
```

Trong đó:

- a: Mảng đầu vào.
- axis: (Tùy chọn) Chiều hoặc các chiều trên đó tổng sẽ được thực hiện. Mặc định là None, tức là tổng của tất cả các phần tử trong mảng.
- dtype: (Tùy chọn) Kiểu dữ liệu của kết quả.
- keepdims: (Tùy chọn) Nếu là True, giữ chiều của mảng kết quả (nếu có) giống với chiều của mảng đầu vào.
- initial: (Tùy chọn) Giá trị khởi tạo cho tổng.
- where: (Tùy chọn) Một mảng Boolean chỉ định vị trí các phần tử được sử dụng trong phép toán.



Hình 20: Minh họa sumation

Trong ví dụ sau, **np.sum()** được sử dụng để tính tổng của mảng array. Tham số sử dụng ở đây là mặc định khi tính tổng các phần tử trong toàn bộ array, sử dụng tham số axis để tính tổng theo cột hoặc hàng. Kết quả được in ra màn hình bao gồm tổng của tất cả các phần tử, tổng theo cột, và tổng theo hàng của mảng.

```
#aivietnam
import numpy as np

# Tạo mảng 2D
arr = np.array([[1, 2, 3],
                [4, 5, 6]])

# Tính tổng của tất cả các phần tử
total_sum = np.sum(arr)

# Tính tổng theo cột
column_sum = np.sum(arr, axis=0)

# Tính tổng theo hàng
row_sum = np.sum(arr, axis=1)

# In ra màn hình
print("Mảng:\n", arr)
print("Tổng của tất cả các phần tử trong m
    ảng:\n", total_sum)
print("Tổng theo cột:\n", column_sum)
print("Tổng theo hàng:\n", row_sum)
```

```
================= Output ================
Mảng:
 [[1 2 3]
 [4 5 6]]

Tổng của tất cả các phần tử trong mảng:
 21

Tổng theo cột:
 [5 7 9]

Tổng theo hàng:
 [ 6 15]

=========================================
```

7.9 Một số hàm thông dụng khác

a) Tính Trung Bình Cộng Các Phần Tử Trong Mảng

$$\text{Trung bình} = \frac{1}{n}\sum_{i=1}^{n} a_i$$

Để tính trung bình cộng các phần tử trong một mảng, ta sẽ sử dụng hàm np.mean.

```python
#aivietnam
import numpy as np

# Tạo mảng 1 chiều
arr = np.array([1, 2, 3, 4, 5])

# Tính trung bình cộng các phần tử mảng
mean_value = np.mean(arr)
print("Trung bình cộng các phần tử trong m
    ảng:", mean_value)
```

```
================ Output ================
Trung bình cộng các phần tử trong mảng:
    3.0

========================================
```

b) Tính Chuẩn Euclid của Vector

$$\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Để tính chuẩn Euclid (độ dài) của một vector, ta sẽ sử dụng hàm np.linalg.norm.

```python
#aivietnam
import numpy as np

# Tạo vector
vector = np.array([1, 2, 3])

# Tính chuẩn Euclid của vector
norm = np.linalg.norm(vector)
print("Chuẩn Euclid của vector:", norm)
```

```
================ Output ================
Chuẩn Euclid của vector:
    3.7416573867739413

========================================
```

c) Giải hệ phương trình tuyến tính

$$A\mathbf{x} = \mathbf{b}$$

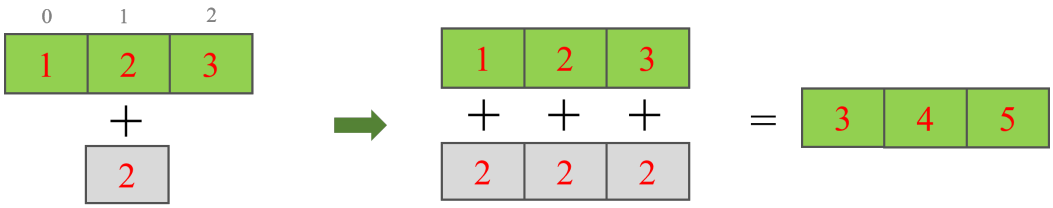
Để giải hệ phương trình tuyến tính $A\mathbf{x} = \mathbf{b}$, ta sẽ sử dụng hàm np.linalg.solve.

```python
#aivietnam
import numpy as np

# Tạo ma trận hệ số A
A = np.array([[3, 1], [1, 2]])
# Tạo vector hằng số b
b = np.array([9, 8])
# Giải hệ phương trình tuyến tính
x = np.linalg.solve(A, b)
print("Nghiệm của hệ phương trình:", x)
```

```
================ Output ================
Nghiệm của hệ phương trình: [2. 3.]

========================================
```

8 Broadcasting, function

8.1 Cơ chế Broadcasting



Hình 21: Minh họa Broadcasting

Broadcasting cho phép thực hiện các phép toán trên các mảng có kích thước khác nhau. Khi thực hiện các phép toán, NumPy tự động mở rộng các mảng nhỏ hơn để chúng có cùng kích thước với mảng lớn hơn. Điều này giúp tiết kiệm bộ nhớ và tối ưu hóa hiệu suất.
Một số quy tắc khi thực hiện broadcasting:

- Nếu các mảng không có cùng số chiều, mảng có số chiều nhỏ hơn sẽ được mở rộng với các chiều có kích thước 1 ở đầu.
- Nếu kích thước của các chiều không khớp, mảng có kích thước 1 trong chiều đó sẽ được mở rộng để khớp với kích thước của mảng khác.
- Nếu kích thước của bất kỳ chiều nào không khớp và không phải là 1, sẽ xảy ra lỗi.

Để hiểu rõ hơn về cơ chế broadcasting, chúng ta thực hiện một số ví dụ dưới đây:
Ví dụ 1: Cộng một số vô hướng vào mảng

```python
#aivietnam
import numpy as np

# Tạo mảng 1 chiều
arr = np.array([1, 2, 3])
# Cộng số 2 vào từng phần tử của mảng
result = arr + 2
print("Kết quả:", result)
```

```
================= Output ================
Kết quả: [3, 4, 5]


=========================================
```

Ví dụ 2: Cộng hai mảng có kích thước khác nhau

```python
#aivietnam
import numpy as np

# Tạo mảng 2 chiều
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
# Tạo mảng 1 chiều
arr1d = np.array([1, 2, 3])
# Cộng hai mảng
result = arr2d + arr1d
print("array 2d:\n", arr2d)
print("array 1d:\n", arr1d)
print("Kết quả:\n", result)
```

```
================= Output ================
array 2d:
 [[1 2 3]
 [4 5 6]]
array 1d:
 [1 2 3]
Kết quả:
 [[2 4 6]
 [5 7 9]]


=========================================
```

Ví dụ 3: Nhân hai mảng có kích thước khác nhau

```python
#aivietnam
import numpy as np

# Tạo mảng 3x1
arr1 = np.array([[1], [2], [3]])
# Tạo mảng 1x3
arr2 = np.array([10, 20, 30])
# Nhân hai mảng
result = arr1 * arr2
print("array 1:\n", arr1)
print("array 2:\n", arr2)
print("Kết quả:\n", result)
```

```
================ Output ===============
array 1:
 [[1]
 [2]
 [3]]
array 2:
 [10 20 30]
Kết quả:
 [[10 20 30]
 [20 40 60]
 [30 60 90]]
=======================================
```

Ví dụ 4: Broadcasting với mảng nhiều chiều

```python
#aivietnam
import numpy as np

# Tạo mảng 3 chiều
arr3d = np.array([[[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]]])

# Tạo mảng 1 chiều
arr1d = np.array([1, 2, 3])

# Cộng hai mảng
result = arr3d + arr1d

print("array 3d:\n", arr3d)
print("array 1d:\n", arr1d)
print("Kết quả:\n", result)
```

```
================ Output ===============
array 3d:
 [[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
array 1d:
 [1 2 3]
Kết quả:
 [[[ 2  4  6]
  [ 5  7  9]]

 [[ 8 10 12]
  [11 13 15]]]
=======================================
```

8.2 Tạo hàm

a) apply_along_axis()

np.apply_along_axis là hàm cho phép ta áp dụng một hàm tự định nghĩa dọc theo một trục cụ thể của mảng. Cú pháp:

```python
np.apply_along_axis(func, axis, arr)
```

- func: Hàm sẽ được áp dụng lên các phần tử của mảng.
- axis: Trục mà bạn muốn áp dụng hàm.
- arr: Mảng muốn áp dụng hàm.

Để hiểu rõ hơn về cách sử dụng np.apply_along_axis, chúng ta sẽ viết một hàm tính tổng các phần tử trong mỗi hàng của mảng 2 chiều và sau đó áp dụng hàm này lên mảng.

```python
#aivietnam
import numpy as np

# Định nghĩa hàm tính tổng
def sum_of_elements(arr):
    return np.sum(arr)
# Tạo mảng 2 chiều
arr2d = np.array([[1, 2, 3], [4, 5, 6],
    [7, 8, 9]])
# Áp dụng hàm tính tổng lên từng hàng của
    mảng
result = np.apply_along_axis(
    sum_of_elements, axis=1, arr=arr2d)
print("Tổng các phần tử trong mỗi hàng:",
    result)
```

```
================ Output ================
Tổng các phần tử trong mỗi hàng: [ 6 15
    24]
========================================
```

b) vectorize

np.vectorize là một hàm cho phép vector hóa một hàm Python thuần túy, tức là biến nó thành một hàm có thể áp dụng trên từng phần tử của mảng. Cú pháp:

```python
np.vectorize(func)
```

- func: Hàm sẽ được vector hóa.

Ví dụ, chúng ta sẽ viết một hàm chuyển đổi nhiệt độ từ Celsius sang Fahrenheit và sau đó vector hóa hàm này.

```python
#aivietnam
import numpy as np

# Định nghĩa hàm chuyển đổi nhiệt độ
def celsius_to_fahrenheit(c):
    return (c * 9/5) + 32
# Vector hóa hàm celsius_to_fahrenheit
vectorized_conversion = np.vectorize(
    celsius_to_fahrenheit)
# Tạo mảng nhiệt độ Celsius
celsius = np.array([0, 20, 37, 100])
# Áp dụng hàm vector hóa lên mảng
fahrenheit = vectorized_conversion(celsius
    )
print("Chuyển đổi từ Celsius sang
    Fahrenheit:", fahrenheit)
```

```
================ Output ================
Chuyển đổi từ Celsius sang Fahrenheit: [
    32.   68.   98.6 212. ]
========================================
```

9 Kết Luận

Qua bài viết này, chúng ta đã làm quen với cách sử dụng NumPy, một thư viện mạnh mẽ và linh hoạt trong Python, giúp dễ dàng thao tác và tính toán với các mảng số. Dưới đây là những kiến thức cơ bản nhưng quan trọng mà bạn cần nắm được:

- Tạo Mảng: Chúng ta đã học cách tạo mảng khác nhau. Điều này giúp bạn có được các mảng dữ liệu cần thiết cho các phép tính toán sau này.
- Truy Cập Mảng: Bằng cách sử dụng các phương pháp indexing và slicing, bạn có thể truy cập và thao tác với từng phần tử hoặc một dãy phần tử trong mảng một cách hiệu quả. Đây là kỹ năng cơ bản nhưng rất quan trọng để làm việc với dữ liệu trong NumPy.
- Tìm Kiếm và Sắp Xếp: Chúng ta đã tìm hiểu các phương pháp tìm kiếm và sắp xếp mảng, giúp bạn có thể xử lý và tổ chức dữ liệu một cách có hệ thống và dễ dàng hơn.
- Các phương pháp nối mảng, thay đổi chiều của mảng
- Broadcasting: Hiểu cơ chế broadcasting giúp bạn thao tác với các mảng có kích thước khác nhau một cách dễ dàng và trực quan, mà không cần phải lặp lại các phép toán thủ công.
- Hàm Vector hóa và áp dụng trên các trục: Sử dụng các hàm np.vectorize và np.apply_along_axis giúp bạn tối ưu hóa và thực hiện các phép toán phức tạp trên mảng một cách nhanh chóng và hiệu quả.
- Các phép toán trên mảng: Chúng ta đã học các hàm hỗ trợ tính toán hiệu quả khi làm việc với mảng.

Những kiến thức này là nền tảng để bạn có thể tiếp tục tìm hiểu và ứng dụng NumPy trong các bài toán phức tạp hơn, trong các lĩnh vực như khoa học dữ liệu, trí tuệ nhân tạo và học máy. Hy vọng bài viết đã cung cấp cho bạn một cái nhìn tổng quan và các công cụ cần thiết để bắt đầu với NumPy. Hãy thực hành, thực hành và thực hành để thành thạo hơn bạn nhé!