THƯ VIỆN NUMPY – PYTHON

# TÌM HIỂU THƯ VIỆN NUMPY

* + Cài đặt: Mở Command Prompt và gõ lệnh: pip install numpy
  + Thao tác:

## Khai báo thư viện

import numpy as np

## Khởi tạo mảng

* 1. **Khởi tạo mảng 1 chiều:**

# Khởi tạo mảng 1 chiều với kiểu dữ liệu các phần tử là Integer arr = np.array([1, 3, 4, 5, 6], dtype = int)

# Khởi tạo mảng 1 chiều với kiểu dữ liệu mặc định arr = np.array([1, 3, 4, 5, 6])

print(arr) # Output: [1 3 4 5 6]

## Khởi tạo mảng 2 chiều:

arr1 = np.array([(4, 5, 6), (1, 2, 3)], dtype = int)

print(arr1) # Output: [ [4 5 6]

[1 2 3] ]

## Khởi tạo mảng 3 chiều:

arr2 = np.array(( [(2, 4, 0, 6), (4, 7, 5, 6)],

[(0, 3, 2, 1), (9, 4, 5, 6)],

[(5, 8, 6, 4), (1, 4, 6, 8)] ), dtype = int)

print(arr2) # Output: [ [ [2 4 0 6]

[4 7 5 6] ]

[ [0 3 2 1]

[9 4 5 6] ]

[ [5 8 6 4]

[1 4 6 8] ] ]

## Khởi tạo với các hàm có sẵn:

* np.zeros((3, 4), dtype = int): tạo mảng 2 chiều với các phần tử 0 với kích thước 3x4
* np.ones((2, 3, 4), dtype = int): tạo mảng 3 chiều với các phần tử 1 với kích thước 2x3x4
* np.arange(1, 7, 2): tạo mảng với các phần tử từ 1 – 6 với bước nhảy là 2
* np.full((2, 3), 5): tạo mảng 2 chiều các phần tử 5 với kích thước 2x3
* np.eye(4, dtype = int): tạo ma trận đơn vị với kích thước là 4x4
* np.random.random((2, 3)): tạo ma trận các phần tử ngẫu nhiên thuộc [0.0, 1.0) kích thước 2x3

## Thao tác với mảng

* dtype: kiểu dữ liệu của phần tử trong mảng print(arr2.dtype) # int32
* shape: kích thước của mảng print(arr.shape) # (5, ) print(arr2.shape) # (3, 2, 4)
* size: số phần tử trong mảng print(arr2.size) # 24
* ndim: số chiều của mảng print(arr2.ndim) # 3

**Truy cập phần tử trong mảng:** các phần tử trong mảng được đánh số từ 0 trở đi

* arr[i]: phần tử thứ i của mảng 1 chiều
* arr1[i, j]: phần tử hàng i, cột j của mảng 2 chiều
* arr2[n, i, j]: truy cập tới phần tử chiều n, hàng i, cột j của mảng 3 chiều
* arr[a : b]: các phần tử từ a đến b – 1 trong mảng 1 chiều
* arr1[:, : i]: các phần tử từ cột 0 đến cột i – 1 của tất cả các hàng trong mảng 2 chiều print(arr[2]) # 4

print(arr1[1, 2]) # 3

print(arr2[1, 1, 3]) # 6

print(arr[0 : 3]) # [1 3 4]

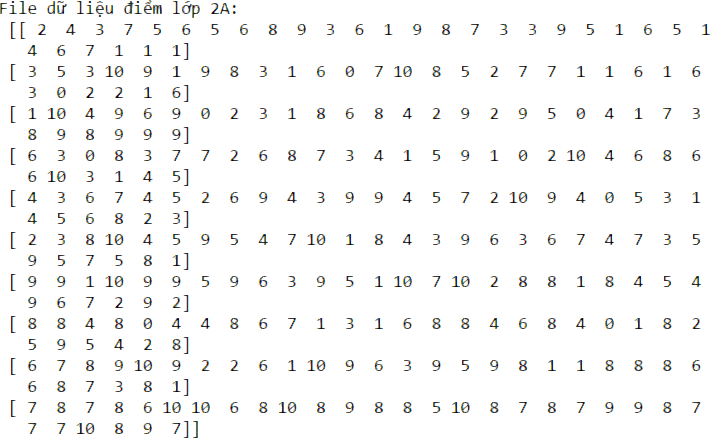
print(arr1[:, : 2]) # [ [4 5]

[1 2] ]

## Đọc mảng từ file .txt:

****

diem\_2a = np.loadtxt(‘Diem\_2A.txt’, dtype = int, delimiter = ‘,’) print("File dữ liệu điểm lớp 2A:\n", diem\_2a)



## Các hàm thống kê:

* arr.max() hoặc np.max(arr): lấy giá trị lớn nhất của mảng arr
* arr.min() hoặc np.min(arr): lấy giá trị nhỏ nhất của mảng arr
* arr.sum() hoặc np.sum(arr): tổng tất cả các phần tử trong mảng arr
* arr.mean() hoặc np.mean(arr): trung bình cộng của tất cả các phần tử trong mảng arr.
* np.median(arr): trả về giá trị trung vị của mảng arr print(np.max(arr)) # 6

print(np.min(arr)) # 1

print(np.sum(arr)) # 19

print(np.mean(arr)) # 3.8

print(np.median(arr)) # 4.0

## Toán tử trong Numpy Array:

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

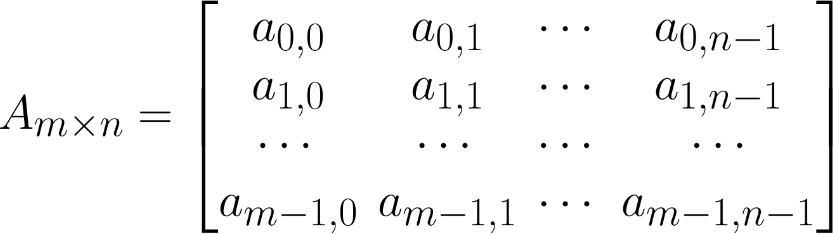
|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Ví dụ** | **Kết quả** |
| (+) một số với mảng | 3 + arr  arr + 3 | [4 6 7 8 9] |
| (+) mảng với mảng | arr + a a + arr | [3 4 7 9 11] |
| (-) một số với mảng | arr – 3  3 - arr | [-2 0 1 2 3]  [2 0 -1 -2 -3] |
| (-) mảng với mảng | arr – a a - arr | [-1 2 1 1 1]  [1 -2 -1 -1 -1] |
| (\*) một số với mảng | arr \* 3  3 \* arr | [3 9 12 15 18] |
| (\*) mảng với mảng | arr \* a a \* arr | [2 3 12 20 30] |
| (//) một số với mảng | arr // 3  3 // arr | [0 1 1 1 2]  [3 1 0 0 0] |
| (//) mảng với mảng | arr // a a // arr | [0 3 1 1 1]  [2 0 0 0 0] |

# MA TRẬN CƠ BẢN VỚI NUMPY

## Ma trận trong Numpy

Ma trận A là một bảng dữ liệu gồm m hàng và n cột. Trong Numpy, người ta dùng một

mảng **ndarray** hai chiều để biểu diễn một ma trận. Hay nói cách khác, ma trận trong Numpy là mảng của các vector một chiều thông thường.



Trong Numpy, ta dùng cú pháp như sau để khai báo một ma trận:

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

print(arr) # Output: [ [1 2 3]

[4 5 6]

[7 8 9] ]

Để truy cập vào từng phần tử của ma trận, ta chỉ cần dựa vào chỉ số của phần tử đó như với mảng trong các ngôn ngữ lập trình khác thôi:

print(arr[1, 2]) # Output: 6

Để trả về kích cỡ của ma trận, ta dùng hàm **shape.** Cần chú ý rằng kiểu dữ liệu trả về ở đây là **tuples**:

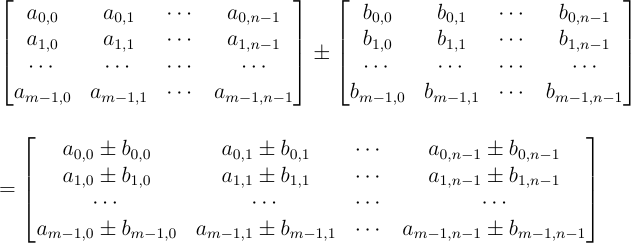
arr = np.array([ [1, 2, 3, 4], [4, 5, 6, 7], [7, 8, 9, 10] ])

print(arr.shape) # Output: (3, 4)

## Các phép toán với ma trận

* 1. **Phép cộng/trừ hai ma trận:**

Để cộng hay trừ hai ma trận cho nhau, ta tiến hành cộng hay trừ các phần tử tương ứng của hai ma trận đó. Nhớ rằng các ma trận phải có cùng kích cỡ mới có thể thực hiện được, khác kích cỡ thì kết quả báo lỗi.



Trong Python, ta có thể tiến hành cộng trừ hai ma trận như sau:

A = np.array([ [1, 3, 4], [-2, 6, 0], [-5, 7, 2] ])

B = np.array([ [2, 3, 4], [-1, -2, -3], [0, 4, -4] ])

print(A + B) # Output: [ [ 3 6 8]

[-3 4 -3]

[-5 11 -2] ]

|  |  |  |
| --- | --- | --- |
| print(A – B) # Output: [ [-1 | 0 | 0] |
| [-1 | 8 | 3] |
| [-5 | 3 | 6] ] |

## Phép nhân/chia ma trận với một số:

Để nhân/chia ma trận với một số, ta chỉ cần nhân/chia từng phần tử của ma trận với số đó thôi. Trong Python có thể thực hiện như sau:

A = np.array([ [1, 3, 4], [-2, 6, 0], [-5, 7, 2] ])

print(A \* 3) # [ [ 3 9 12]

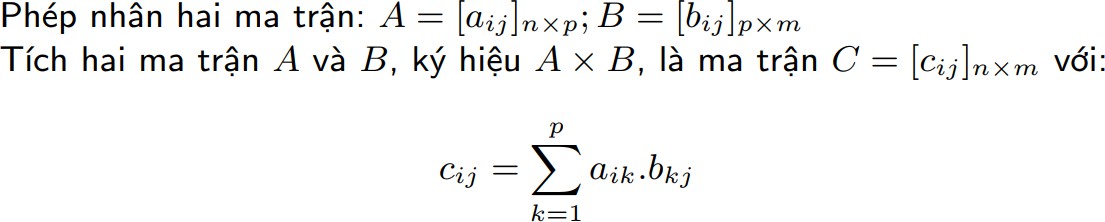
[ -6 18 0 ]

[-15 21 6 ] ]

|  |  |
| --- | --- |
| print(A / 4) # [ [0.25 0.75 | 1. ] |
| [-0.5 1.5 | 0. ] |
| [-1.25 1.75 | 0.5] ] |

## Phép nhân hai ma trận:

Phép nhân hai ma trận không giống như nhân ma trận với một số, không phải là lấy phần tử của ma trận này nhân với phần tử tương ứng của ma trận kia. Một cách tổng quát nhất, phép nhân hai ma trận được thể hiện như sau:



Ta cần chú ý rằng thứ tự của phép nhân hai ma trận rất quan trọng, chỉ có thể thực hiện được nếu số cột của ma trận thứ nhất bằng số hàng của ma trận thứ hai, và ma trận trả về có số hàng bằng

số hàng của ma trận thứ nhất và số cột bằng số cột của ma trận thứ hai. Phép nhân hai ma trận cũng không có tính chất giao hoán, nghĩa là **A.B** không bằng **B.A**.

Trong Numpy, ta có thể dùng toán tử **@** tiến hành nhân hai ma trận như sau:

A = np.array([ [1, 3, 4], [-2, 6, 0], [-5, 7, 2] ])

B = np.array([ [2, 3, 4], [-1, -2, -3], [0, 4, -4] ])

print("A \* B = \n", A @ B) print("B \* A = \n", B @ A)

Ngoài toán tử **@,** ta có thể nhân hai ma trận bằng cách như sau:

print("A \* B = \n", A.dot(B))

print("B \* A = \n", B.dot(A)) Kết quả thu được đều sẽ là:

A \* B =

[ [ -1 13 -21]

[-10 -18 -26]

[-17 -21 -49] ] B \* A =

[ [-24 52 16]

[ 18 -36 -10]

[ 12 -4 -8] ]

Tuy vậy, Python vẫn hỗ trợ nếu như ta muốn lấy phần tử của ma trận này nhân với phần tử tương ứng của ma trận kia qua toán tử **\*** thông thường.

A = np.array([ [1, 3, 4], [-2, 6, 0], [-5, 7, 2] ])

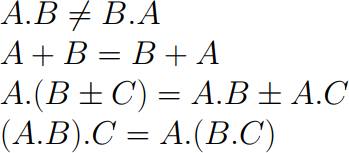
B = np.array([ [2, 3, 4], [-1, -2, -3], [0, 4, -4] ])

print(A \* B) # [ [2 9 16]

[2 -12 0]

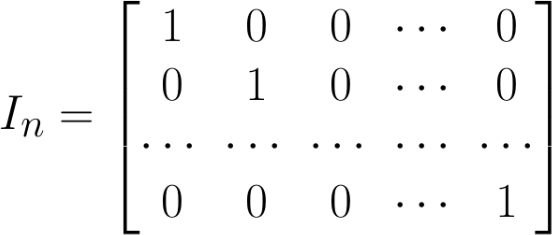
[0 28 -8] ]

Với các phép toán đã đề cập ở trên, ta cần chú ý một vài tính chất sau đây để tránh nhầm lẫn:



## Ma trận đơn vị

Giống như trong tập số thực ta có phần tử đơn vị là 1, thì với ma trận, ta cũng có ma trận đơn vị. Ma trận đơn vị là ma trận vuông (ma trận có số hàng bằng số cột) có các phần tử trên đường chéo chính bằng 1, các phần tử còn lại bằng 0. Như vậy, ma trận đơn vị cấp n sẽ bao gồm n hàng và n cột.



Ma trận đơn vị có tính chất:



Ta thử khai báo một ma trận đơn vị cấp 3 trong Numpy như sau:

arr = np.eye(3) print(arr)

Kết quả thu được là:

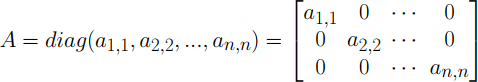
[ [1. 0. 0.]

[0. 1. 0.]

[0. 0. 1.] ]

## Ma trận đường chéo

Ma trận đường chéo là ma trận vuông cấp n thỏa mãn mọi phần tử không nằm trên đường chéo chính đều bằng 0.



Ta có thể khai báo một ma trận đường chéo theo cách sau đây:

A = np.diag([1, 2, 3]) print(A)

Kết quả thu được là:

[ [1 0 0]

[0 2 0]

[0 0 3] ]

Ngoài ra, ta cũng có thể lấy ra các phần tử nằm trên đường chéo chính của ma trận bằng cách sau:

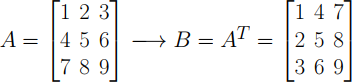
A = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ])

print(np.diag(A)) # [1 5 9]

## Ma trận chuyển vị

Hiểu một cách đơn giản, thì ma trận B được gọi là ma trận chuyển vị của ma trận A, nếu các hàng của ma trận B là các cột của ma trận A (hoặc các cột của ma trận B là các hàng của ma trận A). Như vậy, một ma trận có **m** hàng và **n** cột, khi chuyển vị sẽ thành ma trận mới có **n** hàng

và **m** cột. Ta có thể xem qua ví dụ dưới đây để hiểu rõ hơn:



Tất nhiên, ma trận nào cũng có thể chuyển vị được. Chuyển vị của ma trận đơn vị sẽ bằng chính nó.

Trong Python, ta có 2 cách sau để khởi tạo một ma trận chuyển vị: A = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ])

print(A.T) print(np.transpose(A))

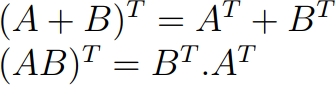
Cả 2 cách đều ra cùng một kết quả:

[ [1 4 7]

[2 5 8]

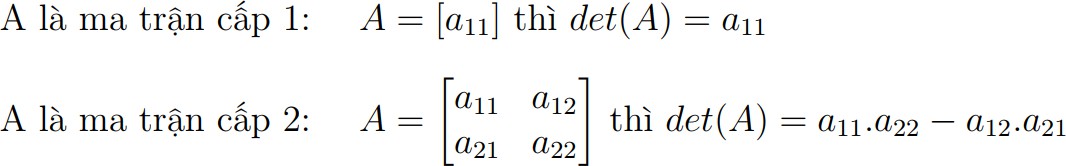
[3 6 9] ]

Một vài tính chất của ma trận chuyển vị như sau:

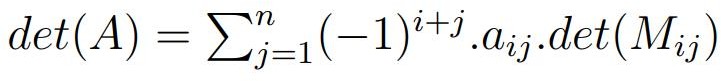


## Định thức của ma trận vuông

Định thức của ma trận vuông A, kí hiệu là det(A) (định thức là một số thực cụ thể chứ không phải là ma trận), được định nghĩa dần dần như sau:



Với A là ma trận cấp n, ta có công thức tính det(A) như sau:



Với M là ma trận thu được bằng cách bỏ đi hàng i và cột j của ma trận A. Như vậy, ta có thể thấy, bản chất của công thức tính định thức chính là việc quy nạp tính các định thức của ma trận có cấp nhỏ hơn. Nếu định thức của một ma trận bằng 0, người ta gọi ma trận này là ma trận suy biến. Và ngược lại, nếu định thức khác 0 thì ta sẽ có ma trận không suy biến.

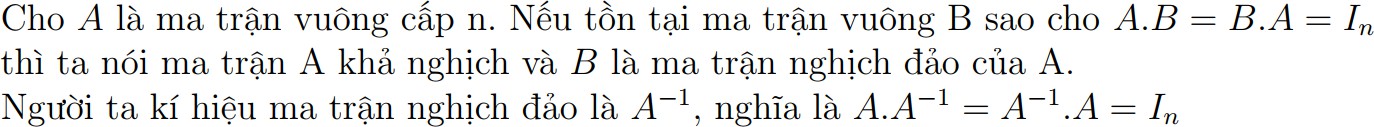
Numpy có hỗ trợ một hàm tính gần đúng định thức, rất tiện phải không nào, ta không phải tính một cách loằng ngoằng như trên nữa.

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

print(np.linalg.det(A))

Kết quả thu được là: 40.000000000000014

## Ma trận nghịch đảo

****

Cần phải nhớ rằng, điều kiện để một ma trận vuông A có ma trận nghịch đảo, là định thức của nó phải khác 0, hay nói cách khác là ma trận A phải là ma trận không suy biến.

Có nhiều con đường để tìm ra ma trận nghịch đảo. Các bạn có thể tìm hiểu thêm về các phương pháp Gauss, Gauss - Jordan, phương pháp viền quanh hay các phương pháp lặp, Tuy nhiên,

các phương pháp kể trên khá phức tạp, mất thời gian để code. Thật may là, trong Numpy cũng hỗ trợ một hàm tính gần đúng ma trận nghịch đảo, là hàm **np.linalg.inv()**. Ta hãy thử xem nó hoạt động ra sao nhé:

A = np.array([ [1, 2, 3], [2, 5, 3], [1, 0, 8] ])

print(np.linalg.inv(A)) Kết quả thu được là:

[ [-40. 16. 9.]

[ 13. -5. -3.]

[ 5. -2. -1.] ]

# MA TRẬN NÂNG CAO VỚI NUMPY

## Thay đổi kích thước ma trận với Reshape

Khi làm việc với ma trận, chúng ta thường xuyên phải sử dụng các thao tác thay đổi kích thước của chúng. Trong Numpy, ta có thể thực hiện qua hàm **.reshape**.

## Chuyển từ mảng 1 chiều thành ma trận 2 chiều:

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

print(A.reshape(3, 3))

Trong đoạn code trên, ta thử chuyển một mảng một chiều có 9 phần tử sang một ma trận hai chiều có 3 hàng và 3 cột. Kết quả thu được như sau:

[ [1 2 3]

[4 5 6]

[7 8 9] ]

Dĩ nhiên, khi chuyển từ mảng một chiều thành hai chiều, số lượng phần tử được giữ nguyên, do vậy ta cần chú ý đến số hàng và số cột của ma trận hai chiều để tránh lỗi nha.

## Chuyển từ ma trận 2 chiều về mảng 1 chiều:

Ngược lại, ta có thể chuyển một ma trận hai chiều về mảng một chiều bằng phương thức **.reshape(-1)**.

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

print(A.reshape(-1))

Kết quả thu được là: [1 2 3 4 5 6 7 8 9]

## Thay đổi số hàng, số cột của ma trận:

A = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12] ])

print(A.reshape(3, 4))

Trong đoạn code trên, mình thử chuyển ma trận A gồm 4 hàng và 3 cột thành 3 hàng và 4 cột. Cùng xem kết quả ra sao nhé:

[ [ 1 2 3 4]

[ 5 6 7 8]

[ 9 10 11 12] ]

## Hạng của ma trận

Hạng của ma trận, được định nghĩa là cấp cao nhất của định thức con khác 0 của ma trận đó. Người ta còn định nghĩa hạng của ma trận là số vector độc lập tuyến tính tối đa khác 0 có trong

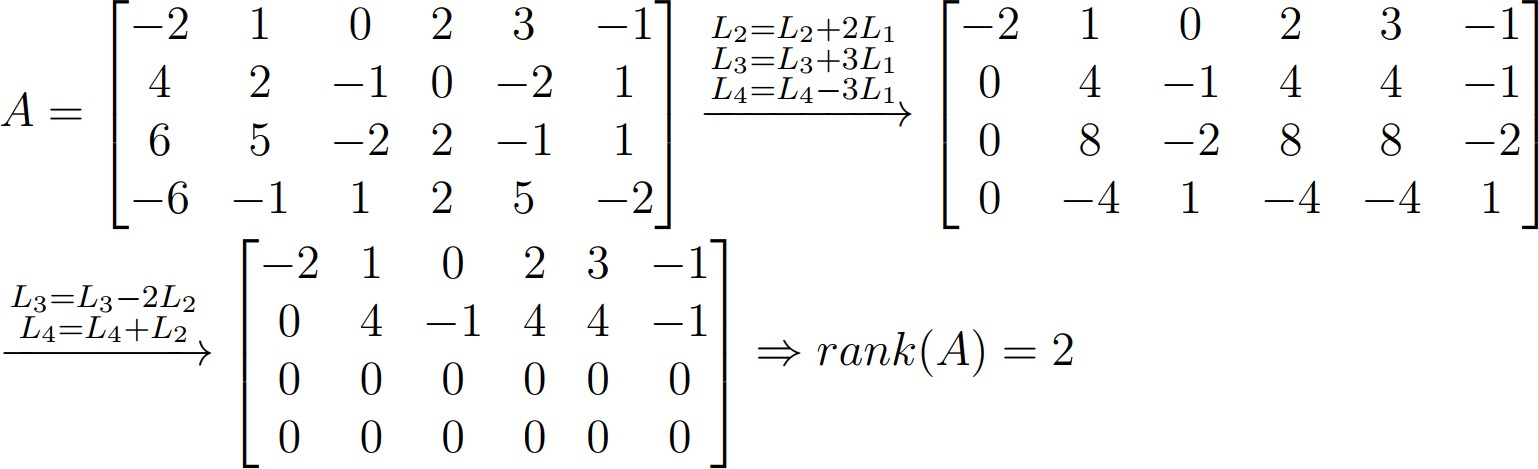
A. Tuy nhiên, mình thấy cách định nghĩa này hơi khó tiếp cận đối với những bạn chưa được học về Đại số tuyến tính.

Ký hiệu hạng của ma trận A là rank(A).

Có nhiều phương pháp để xác định hạng của ma trận. Ta có thể thực hiện các phép biến đổi sơ cấp để đưa ma trận ban đầu về một ma trận hình thang. Các phép biến đổi sơ cấp thường dùng đó là:

* Đổi 2 hàng hoặc 2 cột cho nhau.
* Nhân các phần tử của cùng 1 hàng hoặc cột với một số thực khác 0.
* Cộng vào các phần tử của 1 hàng hoặc cột các phần tử tương ứng của hàng khác hoặc cột khác cùng nhân với một số.

Để mọi người dễ hình dung, mình sẽ thử biến đổi sơ cấp để tính hạng của ma trận sau:



Đến đây thì mọi người đã có cái nhìn rõ hơn về hạng của ma trận rồi nhỉ. Tất nhiên là có nhiều phương pháp để xác định hạng của ma trận, ngoài phương pháp biến đổi sơ cấp như trên.

Trong Numpy, ta có thể sử dụng hàm **.linalg.matrix\_rank()** để tìm hạng của ma trận như sau: A = np.array([ [-2, 1, 0, 2, 3, -1], [4, 2, -1, 0, -2, 1], [6, 5, -2, 2, -1, 1], [-6, -1, 1, 2, 5, -2] ])

print(np.linalg.matrix\_rank(A)) # 2

## Vết của ma trận

Vết (trace) của một ma trận vuông, được định nghĩa bằng tổng các phần tử trên đường chéo chính của ma trận đó.



Trong Numpy, ta có thể sử dụng 2 cách sau để tính vết của ma trận:

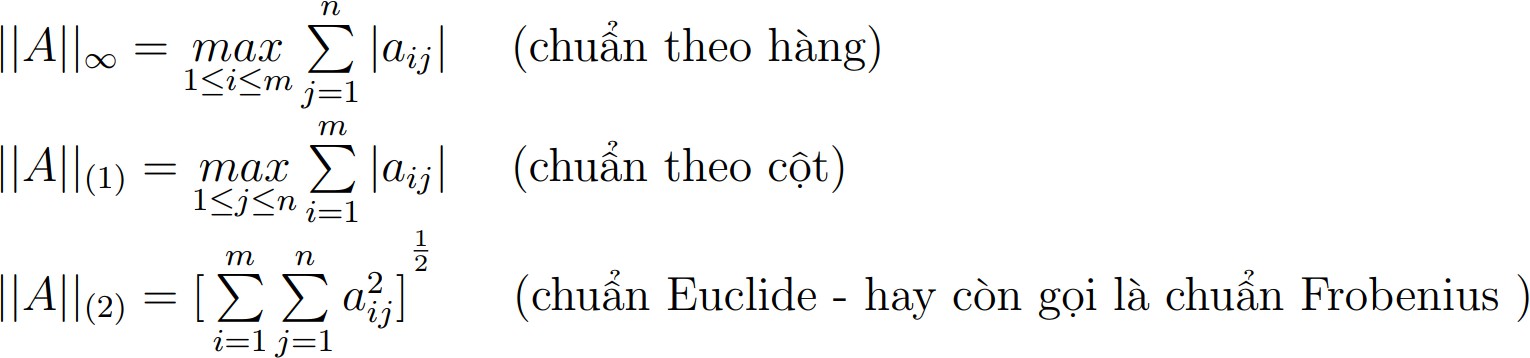
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(np.trace(A)) # 15

print(A.trace()) # 15

## Chuẩn của ma trận

Trong tọa độ Oxy, chúng ta thường sử dụng khoảng cách Euclide (thông qua định lý Pythagoras quen thuộc) để tính khoảng cách giữa 2 điểm, hay là tính độ dài của một vector,nhằm xem điểm này gần với điểm nào, 2 điểm có mối tương quan như nào với nhau. Mở rộng ra cho không gian nhiều chiều, tức là vector nhiều chiều, chúng ta cũng cần có một công thức để tính "khoảng cách" giữa chúng. Điều này dẫn tới sự ra đời của khái niệm về chuẩn (norm). Có nhiều chuẩn khác nhau ứng với các không gian khác nhau, tuy nhiên, trong bài viết này, mình sẽ đề cập tới ba chuẩn thường dùng đối với một ma trận m hàng n cột bất kì:



Các bạn thấy chuẩn Frobenius có quen thuộc không nào. Công thức tính chuẩn Frobenius của một ma trận khá tương đồng với các công thức tính khoảng cách giữa 2 điểm trong hệ tọa độ Oxy, hay độ dài của 1 vector mà mình đã học trong trương trình THPT. Đối với chuẩn này, mình còn có thể sử dụng vết của ma trận để tính như sau:



Đối với chuẩn hàng, ta dùng câu lệnh như sau:

A = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12] ])

print(np.linalg.norm(A, ord = np.inf)) # 33.0 Đối với chuẩn cột, ta dùng câu lệnh sau:

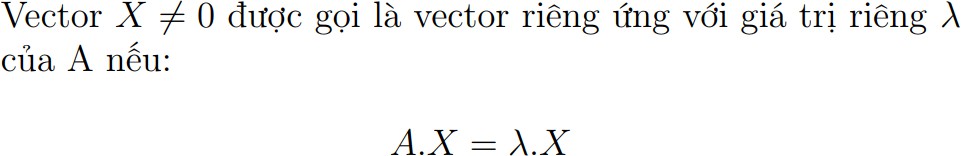
print(np.linalg.norm(A, ord = 1)) # 30.0

Đối với chuẩn Frobenius, ta tính theo cách như sau: print(np.linalg.norm(A)) # 25.495097567963924 print(mt.sqrt(np.trace(A.T @ A))) # 25.495097567963924

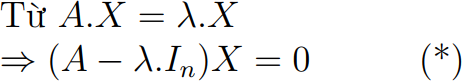
Chuẩn của ma trận được sử dụng rất nhiều để đánh giá sai số trong các thuật toán giải hệ phương trình, tìm ma trận nghịch đảo,... cũng như đánh giá tính đúng đắn trong các thuật toán Machine learning, đặc biệt là chuẩn Frobenius. Cách tính cũng rất đơn giản phải không nào, vậy nên các bạn hãy nắm chắc nha.

## Trị riêng và vector riêng của ma trận

Cho ma trận A vuông cấp n. Ta có định nghĩa sau về trị riêng và vector riêng:



Từ công thức kia, ta có một cách biến đổi quen thuộc trong Đại số tuyến tính như sau:

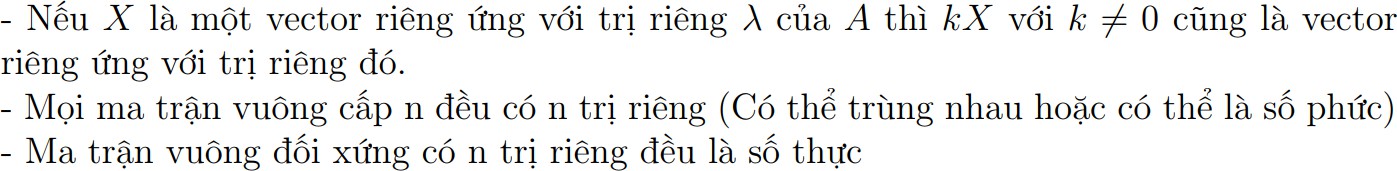


Như vậy, để tồn tại các vector X khác 0, ta có điều kiện sau:



Giải phương trình (\*\*) ta thu được các trị riêng, và ứng với đó, thay vào phương trình (\*) ta được các vector riêng tương ứng. Tất nhiên, cách giải thủ công này chỉ có thể áp dụng được với các ma trận cỡ nhỏ thôi, vì cấp của ma trận càng lớn thì khối lượng phép tính cũng sẽ tăng lên rất nhiều. Có một vài phương pháp giúp tìm trị riêng và vector riêng mà các bạn có thể tham khảo như phương pháp Danilevsky, phương pháp lũy thừa hay phương pháp xuống thang, ... Việc tìm trị riêng và vector riêng có ý nghĩa vô cùng lớn, đặc biệt là ứng dụng trong việc nén hình ảnh, giảm dung lượng của hình ảnh (Các bạn học machine learning chắc sẽ gặp bài toán khai triển SVD - Singular Value Decomposition).

Có một vài tính chất sau mà ta cần lưu ý:



Đặc biệt hơn, ta có mối liên hệ giữa định thức và trị riêng như sau:



Trong Numpy, ta có thể tính trị riêng và vector riêng theo cách như sau: A = np.array([ [2, 1, 0], [1, 3, 1], [0, 1, 2] ])

w, v = np.linalg.eig(A) print("Eigenvalues: \n", w) # trị riêng print("Eigenvectors: \n", v) # vector riêng Kết quả trả về sẽ là:

Eigenvalues: [4. 2. 1.]

Eigenvectors:

[ [-4.08248290e-01 7.07106781e-01 5.77350269e-01] [-8.16496581e-01 -3.45767522e-16 -5.77350269e-01]

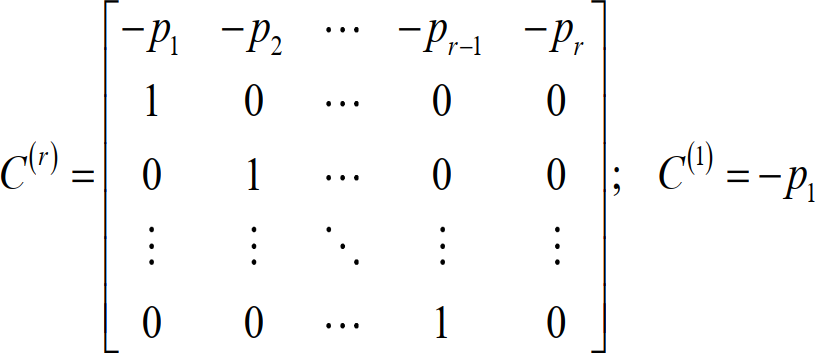
[-4.08248290e-01 -7.07106781e-01 5.77350269e-01] ]

## Ma trận đồng dạng

Cho hai ma trận vuông A và B cùng cấp n. Ta nói ma trận A đồn dạng với ma trận B (A ~ B) nếu tồn tại một ma trận T không suy biến (có định thức khác 0) sao cho:



Một tính chất quan trọng mà chúng ta cần biết, đó là hai ma trận đồng dạng có cùng trị riêng, do đó chúng có cùng các vector riêng tương ứng với các trị riêng đó. Do vậy, thay vì đi tìm trị riêng của ma trận ban đầu (vốn dĩ có thể vô cùng phức tạp), chúng ta có thể tìm một ma trận đơn giản hơn và đồng dạng với nó. Các bạn có thể đọc thêm về phương pháp Danilevsky, ở đây người ta sẽ biến đổi về một ma trận đặc biệt, gọi là ma trận Frobenius.



## Một vài thao tác khác với ma trận

* 1. **Hàm np.sum():**

Để tính tổng tất cả các phần tử của ma trận, ta dùng hàm **np.sum()** như sau: A = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ])

print(np.sum(A)) # 45

Để tính tổng các phần tử trên từng cột, ta làm như sau: print(np.sum(A, axis = 0)) # [12 15 18]

Tương tự như vậy, ta có thể tính tổng các phần tử trên từng hàng: print(np.sum(A, axis = 1)) # [6 15 24]

## Hàm np.min():

Hàm **np.min()** để tính phần tử nhỏ nhất của một ma trận như sau: print(np.min(A)) # 1

Tương tự, ta có thể thêm vào **axis = 0** để tính phần tử nhỏ nhất của từng cột, và **axis = 1** để tính phần tử nhỏ nhất của từng hàng.

## Hàm np.max():

Chúng ta có thể đoán được tác dụng của hàm này rồi nhỉ, ngược với hàm **np.min()** trả về giá trị nhỏ nhất, thì hàm **np.max()** trả về phần tử lớn nhất trên cả ma trận, trên từng hàng hay từng cột.

## Hàm np.mean():

Hàm **np.mean()** sẽ trả về trung bình cộng các phần tử của toàn bộ ma trận, trung bình cộng của từng hàng hay từng cột:

print(np.mean(A)) # 5.0

print(np.mean(A, axis = 0)) # [4. 5. 6.]

print(np.mean(A, axis = 1)) # [2. 5. 8.]

## Hàm np.transpose():

Chuyển vị (transpose) ma trận là hoán đổi các trục trong ma trận theo axes. Nếu không chỉ định axes thì hoán đổi trục mặc định (tức là đảo ngược thứ tự các trục). Ví dụ mảng 2D thì luôn luôn là (0, 1) thành (1, 0) (trục 0 thành trục 1, trục 1 thành trục 0), mảng 3D thì mặc định là (0, 1, 2)

thành (2, 1, 0), nếu chỉnh axes thì có thể thành (1, 0, 2), (0, 2, 1),… Trong Numpy, có 3 cách để transpose ma trận:

* **A.T**: áp dụng cho mảng 2 chiều và mảng nhiều chiều (hoán đổi trục mặc định), không cho phép tuỳ chỉnh hoán đổi trục (axes).
* **np.transpose(A)**: áp dụng cho mọi trường hợp với axes tuỳ chỉnh. Nếu không chỉnh axes thì hoán đổi trục mặc định.
* **A.transpose()**: tương tự np.transpose(A) mà ít dùng hơn. A = np.arange(12).reshape((2, 2, 3))

print(A) # [ [ [ 0 1 2]

[ 3 4 5] ]

[ [ 6 7 8 ]

[ 9 10 11] ] ]

print(A.transpose()) # Đảo trục (0, 1, 2) thành (2, 1, 0) nên kích thước thu được là (3, 2, 2)

# Cách làm tay: tại (0, 0, 0) trong A, giá trị là 0 thì kết quả 0 sẽ nằm tại (0, 0, 0)

tại (0, 0, 1) trong A, giá trị là 1 thì kết quả 1 sẽ nằm tại (1, 0, 0)

tại (1, 0, 2) trong A, giá trị là 8 thì kết quả 8 sẽ nằm tại (2, 0, 1) tương tự cho phần còn lại

# Ta sẽ thu được kết quả: [ [ [ 0 6]

[ 3 9] ]

[ [ 1 7]

[ 4 10] ]

[ [ 2 8]

[ 5 11] ] ]

## Strides:

Strides là một thuộc tính quan trọng giúp xác định cách các phần tử trong mảng được lưu trữ trong bộ nhớ và được truy cập. Stride được biểu diễn dưới dạng một tuple các số nguyên, trong đó mỗi số xác định số **byte** cần nhảy qua trong bộ nhớ để di chuyển từ phần tử này đến phần tử tiếp theo dọc theo một trục cụ thể.

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

print(arr.shape) # (2, 3)

print(arr.strides) # (24, 8)

Mảng arr có shape (2, 3) và kiểu dữ liệu int64 (8 byte mỗi phần tử), strides (24, 8) có nghĩa:

* Di chuyển từ hàng này sang hàng khác: nhảy 24 byte (3 cột x 8 byte)
* Di chuyển từ cột này sang cột khác trong cùng một hàng: nhảy 8 byte Khi cắt lát (slicing), stride sẽ thay đổi nhưng dữ liệu gốc không đổi. subarray = arr[: , : : 2] # lấy cột 1 và 3

print(subarray.strides) # (24, 16)

Khi chuyển vị (transpose), strides thay đổi thứ tự mà không thay đổi dữ liệu. transposed = arr.T

print(transposed.strides) # (8, 24)

Đối với mảng 3 chiều, strides chứa 3 giá trị đại diện cho các bước nhảy (số byte) cần thiết để di chuyển giữa các phần tử trong từng chiều: chiều sâu (depth), chiều hàng (row), và chiều cột (column).

arr = np.zeros((3, 4, 5), dtype = np.int32) # 4 byte mỗi phần tử print(arr.shape) # (3, 4, 5)

print(arr.strides) # (80, 20, 4)

* Di chuyển từ mặt phẳng này sang mặt phẳng khác (**axis = 0**) cần nhảy 80 byte.
* Di chuyển từ hàng này sang hàng khác (**axis = 1**) cần nhảy 20 byte.
* Di chuyển từ cột này sang cột khác (**axis = 2**) cần nhảy 4 byte.

## Hàm np.swapaxes():

Hàm **np.swapaxes()** trong Numpy được sử dụng để hoán đổi hai trục của một mảng. Hàm này không thay đổi dữ liệu mà chỉ thay đổi thứ tự của các trục.

Cú pháp: np.swapaxes(arr, axis1, axis2) # hoán đổi trục axis1 với trục axis2 arr = np.array([ [1, 2], [3, 4] ])

result = np.swapaxes(arr, 0, 1) # hoán đổi trục 0 với trục 1

print(result) # [ [1 3]

[2 4] ]

arr\_3d = np.array([ [ [1, 2], [3, 4] ],

[ [5, 6], [7, 8] ] ])

result\_3d = np.swapaxes(arr\_3d, 0, 2)

print(result\_3d) # [ [ [1 5]

[3 7] ]

[ [2 6]

[4 8] ] ]

# Hoặc print(arr\_3d.swapaxes(0, 2))

## Hàm np.concatenate():

Hàm này nối các mảng dọc theo một trục (axis) được chỉ định, mà không làm tăng số chiều của mảng. Các mảng mà bạn muốn nối phải có cùng kích thước trên tất cả các trục (axes) ngoài trục mà bạn muốn nối, còn không là báo lỗi.

Cú pháp: np.concatenate((mảng\_1, mảng\_2,…), axis = trục)

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

b = np.array([ [ [9, 10], [11, 12] ], [ [13, 14], [15, 16] ] ])

result\_axis0 = np.concatenate((a, b), axis = 0) result\_axis1 = np.concatenate((a, b), axis = 1) result\_axis2 = np.concatenate((a, b), axis = 2) print("Nối theo trục 0:", result\_axis0) # nối theo hàng print("Nối theo trục 1:", result\_axis1) # nối theo cột

print("Nối theo trục 2:", result\_axis2) # nối theo mặt phẳng Kết quả thu được:

|  |  |  |
| --- | --- | --- |
| Nối theo trục 0: | [ [ [ 1 | 2 ] |
|  | [ 3 | 4 ] ] |
|  | [ [ 5 | 6 ] |
|  | [ 7 | 8 ] ] |
|  | [ [ 9 | 10] |

[11 12] ]

[ [13 14]

[15 16] ] ]

Nối theo trục 1: [ [ [ 1 2 ]

[ 3 4 ]

[ 9 10]

[11 12] ]

[ [ 5 6 ]

[ 7 8 ]

[13 14]

[15 16] ] ]

Nối theo trục 2: [ [ [ 1 2 9 10]

[ 3 4 11 12] ]

[ [ 5 6 13 14]

[ 7 8 15 16] ] ]

## Hàm np.repeat():

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

result = np.repeat(a, 2) # lặp lại mỗi phần tử 2 lần print(result) # [1 1 2 2 3 3]

result1 = np.repeat(a, [1, 2, 3]) # lần lượt lặp: 1 lần, 2 lần, 3 lần

print(result1) # [1 2 2 3 3 3]

b = np.array([ [1, 2], [3, 4] ])

result2 = np.repeat(a, 2, axis = 0) # lặp lại dọc theo hàng print(result2) # [ [1 2]

[1 2]

[3 4]

[3 4] ]

result3 = np.repeat(a, 2, axis = 1) # lặp lại dọc theo cột print(result3) # [ [1 1 2 2]

[3 3 4 4] ]

Khi sử dụng **np.repeat()** trên mảng đa chiều như mảng 3D, Numpy mặc định sẽ hoạt động trên phiên bản “flatten” (mảng phẳng) của mảng nếu axis không được chỉ định.

a = np.arange(4).reshape((1, 2, 2))

print(a) # [ [ [0 1]

[2 3] ] ]

b = a.repeat(2) # không chỉ định axis print(b) # [0 0 1 1 2 2 3 3]

c = a.repeat(2, axis = 1) # chỉ định lặp dọc theo hàng print(c) # [ [ [0 1]

[0 1]

[2 3]

[2 3] ] ]

## Hàm np.tile():

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

b = np.tile(a, 2)

c = a.repeat(2)

print(b) # [1 2 3 4 5 1 2 3 4 5]

print(c) # [1 1 2 2 3 3 4 4 5 5]

Hàm **np.tile()** khác với **np.repeat()**. Trong khi **np.tile()** lặp toàn bộ mảng, **np.repeat()** lặp lại từng phần tử trong mảng theo cách định rõ.

A = np.array([ [1, 2], [3, 4] ])

result = np.tile(A, (2, 3))

print(result) # [ [1 2 1 2 1 2]

[3 4 3 4 3 4]

[1 2 1 2 1 2]

[3 4 3 4 3 4] ]

## Hàm np.random.randn():

Hàm **np.random.randn()** trong thư viện Numpy được sử dụng để tạo các số ngẫu nhiên từ phân phối chuẩn (normal distribution, N(0, 1)) với trung bình (mean) là 0 và độ lệch chuẩn (standard deviation) là 1.

Cú pháp: np.random.randn(d0, d1,..., dn)

d0, d1,..., dn: Các kích thước của mảng (array) đầu ra. Nếu không cung cấp kích thước, hàm sẽ trả về một số thực ngẫu nhiên.

x = np.random.randn()

print(x) # ví dụ: 0.4793768452914568 arr = np.random.randn(5)

print(arr) # ví dụ: [ 1.255 -0.983 0.776 -1.437 0.219]

arr = np.random.randn(3, 4)

print(arr) # ví dụ: [ [ 0.856 -0.393 0.672 1.326]

[-0.489 1.124 0.432 -0.918]

[-1.387 -0.653 0.782 0.556] ]

arr1 = np.random.randn(2, 3, 4)

print(arr1.shape) # (2, 3, 4)

## Views và Copies:

* Views thay đổi thì ảnh hưởng trực tiếp đến dữ liệu gốc và ngược lại.
* Copies thay đổi thì không ảnh hưởng đến dữ liệu gốc và ngược lại. **Ví dụ 1:**

arr = np.arange(10) # [0 1 2 3 4 5 6 7 8 9]

arr[5 : 8] = 12 # [0 1 2 3 4 12 12 12 8 9]

**Ví dụ 2:**

a = np.array([ [ [1, 2, 3], [4, 5, 6] ], [ [7, 8, 9], [10, 11, 12] ] ]) # shape: (2, 2, 3)

temp = a[0] # temp là một view của a[0], mọi thay đổi trên temp sẽ ảnh hưởng trực tiếp đến a[0] temp[ : ] = 0 # a thay đổi

|  |  |  |
| --- | --- | --- |
| print(a) # [ [ [ 0 | 0 | 0 ] |
| [ 0 | 0 | 0 ] ] |
| [ [ 7 | 8 | 9 ] |

[10 11 12] ]

**Ví dụ 3:**

a = np.array([ [[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]] ]) # shape: (2, 2, 3)

temp = a[0] # temp là một view của a[0], mọi thay đổi trên temp sẽ ảnh hưởng trực tiếp đến a[0] temp = 0 # biến temp không còn là view của a[0] nữa

lúc này temp được gán một giá trị hoàn toàn mới (một số nguyên 0) và không còn liên quan tới a[0] nên a không thay đổi

**Ví dụ 4:**

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

b = a[0] # b là view print(b) # [1 2 3]

a[0][0] = 0

print(b) # [0 2 3]

**Ví dụ 5:**

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

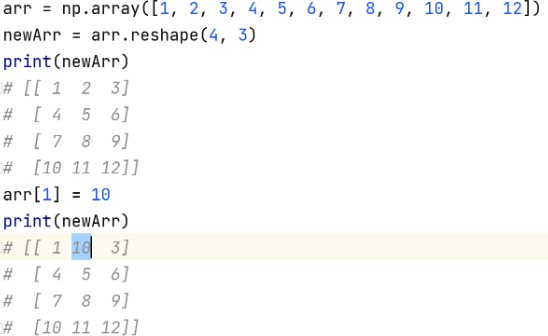
b = a[0].copy() # b là copy print(b) # [1 2 3]

a[0][0] = 0

print(a[0]) # [0 2 3]

print(b) # [1 2 3]

**Ví dụ 6:**



## Fancy Indexing:

**Ví dụ 1:**

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

indices = np.array([2, 3, 4]) # chỉ định các vị trí muốn lấy print(a[indices]) # [3 4 5]

**Ví dụ 2:**

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

row\_indices = [0, 1, 2] # chọn các hàng 0, 1, 2

col\_indices = [2, 1, 0] # chọn các cột 2, 1, 0 result = a[row\_indices, col\_indices]

print(result) # Output: [3 5 7] (phần tử tại các vị trí (0,2), (1,1), (2,0))

**Ví dụ 3:**

a = np.array([ [[1, 2], [3, 4]], [[5, 6], [7, 8]] ]) # truy cập các phần tử cụ thể

result = a[[0, 1], [1, 0], [0, 1]] # các vị trí: (0,1,0), (1,0,1)

print(result) # Output: [3 6]

**Ví dụ 4:**

a = np.array([10, 20, 30, 40, 50])

indices = [1, 3] # chỉ định vị trí muốn thay đổi a[indices] = [99, 88] # gán giá trị mới

print(a) # Output: [10 99 30 88 50]

**Ví dụ 5:**

a = np.array([10, 20, 30, 40])

b = a[[0, 2]] # fancy indexing tạo copy, không phải view b[0] = 99

print(a) # Output: [10 20 30 40] (mảng gốc không đổi)

**Ví dụ 6:**

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

result = a[ :2, [0, 2] ] # lấy 2 hàng đầu và các cột 0, 2

print(result) # Output: [ [1 3]

[4 6] ]

**Ví dụ 7:**

arr = np.array( [ [0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15], [16, 17, 18, 19],

[20, 21, 22, 23], [24, 25, 26, 27], [28, 29, 30, 31] ] )

print(arr[[1, 4, 7]]) # [ [ 4 5 6 7 ]

[16 17 18 19]

[28 29 30 31] ]

print(arr[ [1, 4, 7], [0, 0, 1] ]) # [4 16 29]

print(arr[[1, 5, 7, 2]][ : , [0, 3, 1, 2]]) # [ [ 4 7 5 6]

[20 23 21 22]

[28 31 29 30]

[ 8 11 9 10] ]

## Boolean Indexing:

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

condition = arr > 3

print(condition) # [False False False True True] result = arr[condition]

print(result) # [4 5]

arr1 = np.array([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ])

condition1 = arr1 > 5

print(condition1) # [ [False False False]

[False False True ] [True True True ] ]

result1 = arr1[condition1] print(result1) # [6 7 8 9]

## Hàm np.where():

* Loại đầu tiên: **np.where(condition)** trả về các chỉ số (indexes) của các phần tử trong mảng mà điều kiện đó đúng.

arr = np.array([10, 15, 20, 25, 30])

indices = np.where(arr > 20) # lấy chỉ số của các phần tử > 20 print(indices) # Output: (array([3, 4]),)

print(arr[indices]) # Output: [25 30]

* Loại thứ hai: **np.where(condition, x, y)**

**+** condition: điều kiện cần kiểm tra

+ x: giá trị được chọn nếu điều kiện đúng

+ y: giá trị được chọn nếu điều kiện sai arr = np.array([10, 15, 20, 25, 30])

result = np.where(arr > 20, 1, 0) # đánh dấu 1 nếu > 20, ngược lại là 0

print(result) # Output: [0 0 0 1 1]

modified\_arr = np.where(arr < 20, 0, arr) # thay thế các giá trị nhỏ hơn 20 bằng 0 print(modified\_arr) # Output: [0 0 20 25 30]

## So sánh hai mảng:

* So sánh từng phần tử (Comparisons):

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

b = np.array([4, 2, 2, 4])

print(a == b) # [False True False True] print(a > b) # [False False True False]

Nếu a và b khác kích thước thì sẽ báo lỗi ValueError: operands could not be broadcast together.

* So sánh toàn bộ mảng (Array-wise Comparisons): print(np.array\_equal(a, b)) # False print(np.array\_equal(a, a)) # True

Nếu a và b khác kích thước thì ra kết quả là False.

## Broadcasting:

* Cách xử lý khi tính toán hai mảng không tương thích kích thước: a = np.arange(12).reshape(4, 3) # tạo mảng 2D kích thước (4, 3) # a = [ [ 0 1 2 ]

[ 3 4 5 ]

[ 6 7 8 ]

[ 9 10 11] ]

b = np.array([1, 2, 3, 4]) # tạo mảng 1D kích thước (4,)

# b = [1 2 3 4]

# print(a - b) sẽ lỗi do kích thước không tương thích

print(a - b[:, np.newaxis]) # điều chỉnh kích thước của b để tương thích # b[:, np.newaxis] = [ [1]

[2]

[3]

[4] ]

# [ [ 0-1 1-1 2-1 ] dòng đầu tiên của a trừ dòng đầu tiên của b [ 3-2 4-2 5-2 ] dòng thứ hai của a trừ dòng thứ hai của b

[ 6-3 7-3 8-3 ] tương tự

[ 9-4 10-4 11-4] ]

Kết quả thu được: [ [-1 0 1 ]

[ 1 2 3 ]

[ 3 4 5 ]

[ 5 6 7 ] ]

* Hàm **np.reshape(x, y, -1)** hoặc **np.reshape(x, -1)** trong Numpy được sử dụng để thay đổi kích thước của mảng mà không thay đổi số lượng phần tử. Dưới đây là các tham số của hàm này:
  + **x**: Số lượng hàng mới mà bạn muốn mảng có sau khi reshape.
  + **y**: Số lượng cột mới mà bạn muốn mảng có sau khi reshape.

**-1**: Được sử dụng để tự động tính toán kích thước còn lại sao cho tổng số phần tử của mảng không thay đổi.