💯

NumPy

Tutorial de NumPy - Iniciante

O NumPy implementa arrays com controle de bits, sendo mais rápido que as listas do python normal. Ele foi criado para otimizar a utilização de listas e vetores.

O NumPy **permite operações vetorizadas**, enquanto as listas não.

Comandos:

Criar um array/vetor e definir o seu tipo:

```
vetor  = np.array([1, 2, 3, 4], dtype="int8") #Vetor de uma dimensão
matriz = np.array( [ [1, 2, 3, 4], [5, 6, 7, 8] ], dtype="int8") #Matriz (lista de listas)
```

Nesse caso:

- A **lista [1, 2, 3, 4, 5]** é a lista que será convertida em um vetor NumPy.
- O **dtype** é a quantidade de bits reservadas para cada número.

Alguns atributos do array

```
vetor.ndim <<< # Retorna a dimensão desse vetor.
vetor.shape << # Retorna uma tupla com as dimensões (linha, coluna).
vetor.itemsize < # Retorna quantos bytes os elementos do vetor ocupam.
```

Indexação de array

É possível consultar linhas ou colunas específicas por meio da indexação dos vetores/matrizes.

> A fórmula geral de consulta é `vetor[linha, coluna]`, e baseado nisso podemos fazer consultas.

Se um vetor for unidimensional, ele não recebe as colunas.

Omissão de elementos

- A quantidade de colunas pode ser omitida, isso indica que todas as colunas estão selecionadas.
- A quantidade de linhas pode ser omitida, isso indica que todas as linhas estão selecionadas.
    - Linhas não podem ser omitidas se colunas forem informadas.

Se passarmos uma lista de indices para o vetor, exemplo `vetor[[4, 2, 3]]` estaremos retornando um vetor com os valores que ocupam esses índices.

Entendendo a fórmula geral [linha, coluna]

Dada uma matriz 5x5

- Se eu quiser consultar um elemento individual da linha 1 e coluna 4: `matriz[0, 3]`
- Se eu quiser consultar a linha 1 inteira: `matriz[0]`
- Se eu quiser consultar apenas uma coluna, o processo é diferente.

> Para consultar uma coluna, é preciso definir uma linha.

Logo, coisas do tipo matriz[,0] não funcionam, pois elas não têm linhas definidas.

Como a leitura das matrizes é feita de forma linear, para consultar colunas de forma individual, é necessário usar **SLICING** das matrizes.

Slicing

O slicing segue a estrutura `[INICIO:FIM:STEP]`, e *início* e *fim* seguem a estrutura (**linha, coluna)**

`INICIO` pode ser: (1, 2), (2, 3) (3,) mas não (,4)

O `FIM` pode assumir todos esses valores. Estamos fazendo uma consulta de coluna se (,4)

Seleção de colunas

Vimos que para consultar uma coluna, é preciso definir uma linha.

A lógica para selecionar colunas é consultar [**todas as linhas : (até a última, mas da coluna x)**]

Interpretar sem os parênteses. Eles serviram pra indicar a estrutura [linha, coluna] ou [r, c]

Esse é exatamente o código que queremos replicar.

> ✅ matriz[**todas as linhas : até(todas, mas da coluna x)**]

Exemplo prático.

Criando uma matriz aleatória 5x5 de 0 a 100 com a função **randint.**

```
matrix = np.random.randint(low=0, high=100, size=(5,5))
matrix
```

Output:

```
array([[56, 85, 96, 25, 13],
       [23, 11, 70, 83, 51],
       [12,  5, 90, 63, 86],
       [26,  5, 18, 95,  4],
       [21, 55,  8, 61, 43]])
```

- Quero consultar a terceira coluna, por exemplo
- O índice da terceira coluna é 2.

- Pseudocódigo: matriz[**todas as linhas : até(todas, mas da coluna x)**]
- Traduzindo para a coluna 3 → `matrix[:,2]`

Output:

```
array([96, 70, 90, 18,  8])
```

Existe outro método para consultar colunas, mas ele é mais usual para consultar matrizes.

`matriz[slice de linhas, slice de colunas]`

Nesse caso, se eu quisesse consultar a coluna 3, ficaria:

`matriz[primeira linha:até ultima, terceira coluna:até quarta]`

- Seguindo o modelo vetor(linha, coluna), mas é muito mais complicado.

Se eu quiser consultar uma matriz dentro de uma matriz, esse método é o indicado.

```
array([[12, 70,  9, 18, 83],
       [93, 67,  9, 62, 67],
       [ 1, 53,  9, 49, 82],
       [44, 62,  9, 14,  1],
       [93, 91,  9, 29, 54]])
```

Para consultá-la, pegamos o índice da linha de começo e o indice da linha final, assim como o índice da coluna de inicio e o índice da coluna final.

Os indices finais excluem o último valor, então somamos a eles.

Código:

```
matrix[1:4, 1:4]
```

Output:

```
array([[67,  9, 62],
       [53,  9, 49],
       [62,  9, 14]])
```

Inserção

Ao consultar uma elemento/linha/coluna do vetor, podemos substituí-lo por outro se fizermos uma atribuição.

Exemplo:

Dada nossa matriz criada anteriormente, vamos substituir a terceira coluna que conseguimos consultar

```
array([[56, 85, 96, 25, 13],
       [23, 11, 70, 83, 51],
       [12,  5, 90, 63, 86],
       [26,  5, 18, 95,  4],
       [21, 55,  8, 61, 43]])
```

Matriz Antes da substituição

Código para substituir:

```
matrix[:,2] = [9, 9, 9, 9, 9]
```

Output:

```
array([[56, 85,  9, 25, 13],
       [23, 11,  9, 83, 51],
       [12,  5,  9, 63, 86],
       [26,  5,  9, 95,  4],
       [21, 55,  9, 61, 43]])
```

Substituição completa

Criando diferentes tipos de Arrays:

Zeros

`np.zeros(tupla de dimensão)` →Vetor de zeros com a dimensão desejada.

`np.zeros_like(vetor)` → Vetor de zeros com a dimensão do vetor do argumento.

Uns

`np.ones(tupla de dimensão)` → Vetor completo de uns com a dimensão desejada.

`np.ones_like(vetor)` → Vetor completo de uns com a dimensão do vetor do argumento.

Full

`np.full(tupla de dimensão, número)` → Vetor de dimensão desejada e só com o número digitado.

`np.full_like(vetor, número)` → Vetor igual ao de cima mas com a dimensão do vetor desejada.

Identidade

`np.eye(n)` → Matriz identidade quadrada NxN.

Eixos → Axis ou Axes

Os arrays do NumPy possuem eixos, como se fossem eixos do sistema cartesiano.

Linha (eixo horizontal x) ↔

Coluna: (eixo vertical y) ↕

Esses eixos podem ser passados como argumento de algumas funções. `axis=x`

Dependendo de qual eixo for passado na função, ela é aplicada de forma diferente.

Para cada dimensão, existe um eixo, começando de zero.

- Eixos unidimensionais = [0] → Vertical
- Eixos bidimensionais = [0, 1] → Vertical e Horizontal

Assim por diante.

⚠️ O eixo zero sempre é o eixo vertical das linhas.
Mas pra vetores unidimensionais, ele se transforma no eixo horizontal, por conta que os vetores unidimensionais são representados horizontalmente na programação, diferente da matemática.

Métodos de manipulação.

Reshape

`vetor.reshape(tupla de dimensão)` → Transforma as dimensões do vetor para a desejada

Somente se o número de elementos no vetor consiga preencher a nova dimensão.

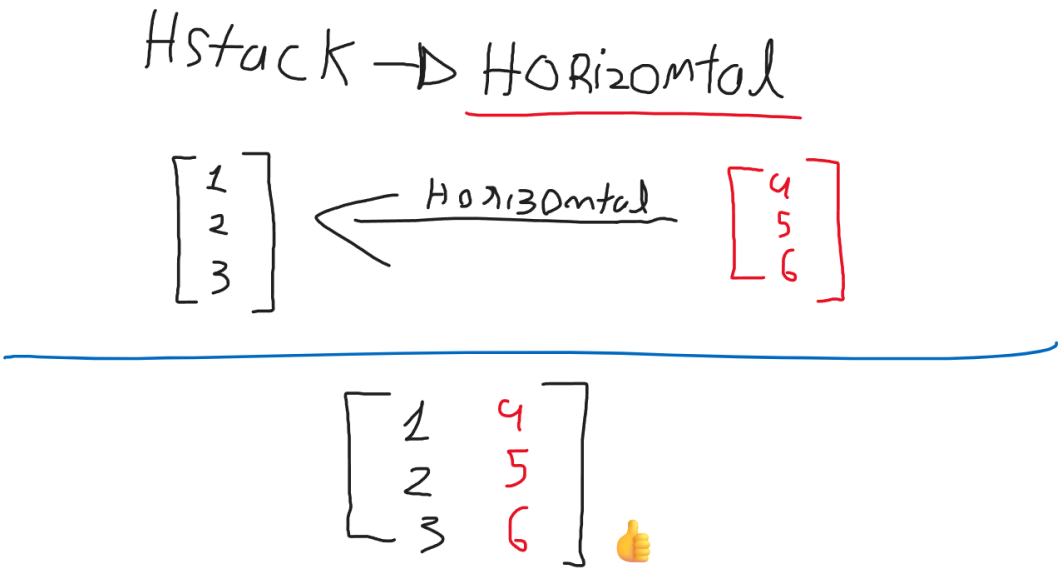
Stacking

Útil para **criar matrizes** ao **empilhar vários vetores** horizontalmente ou verticalmente.

Dado dois vetores:

- A = [1, 2, 3]
- B = [4, 5, 6]

Hstack

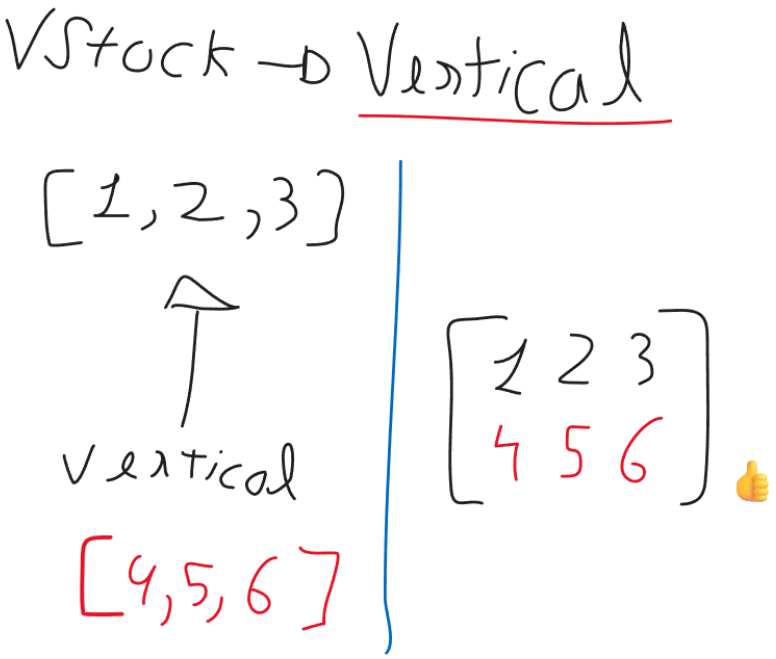


Código:

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

horizontal_stack = np.hstack((A, B))
```

Vstack



Código:

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

vertical_stack = np.vstack((A, B))
```

Concatenate

Uma alternativa para o vstack e o hstack é o `np.concatenate()`

Dada uma lista de vetores e o eixo, o `np.concatenate` junta esses vetores pelo eixo informado.

`np.concatenate(lista, axis=eixo)`

Cópia de vetores

Para fazer uma cópia de um vetor independente do original, é necessário usar o método `.copy`

```
# Caso errado.
A = np.array([1, 2, 3])
B = a
```

```
# Caso correto
A = np.array([1, 2, 3])
B = a.copy()
```

Métodos matemáticos

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

np.matmul(matriz a, matriz b)