

Biblioteca Numpy

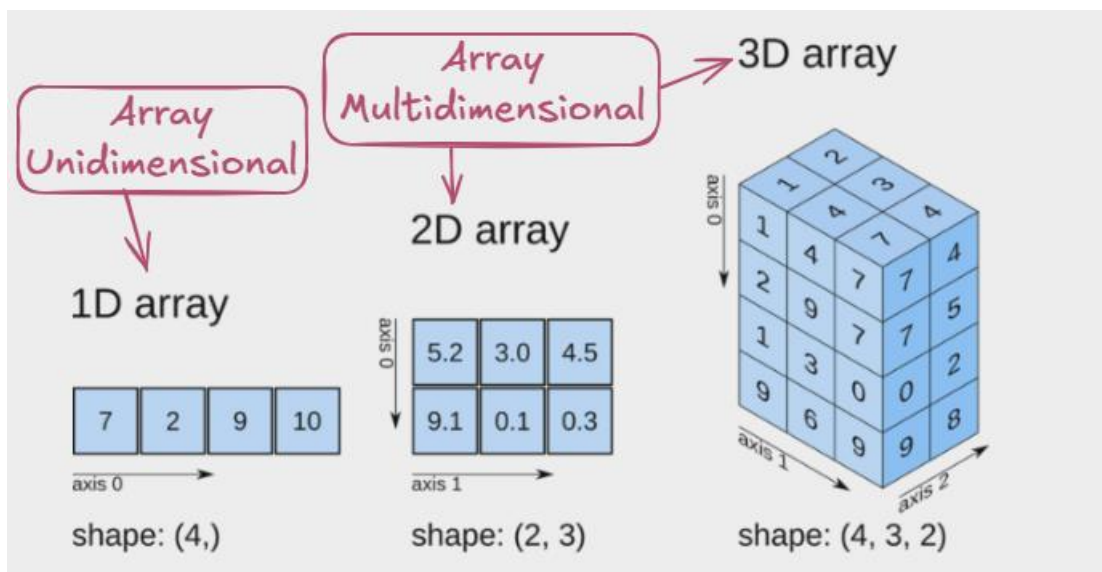
Introdução ao Numpy



NumPy se trata de uma **biblioteca Python** parruda para trabalhar com **arrays unidimensionais e multidimensionais**. Ela oferece várias funções matemáticas de alto nível para operar com esses **arrays** de forma rápida e eficiente. É muito utilizada em conjunto com o **Pandas** e, conseqüentemente, no preparo de dados para aplicações de **Machine Learning**.

Mas como assim, essa biblioteca trabalha com **arrays unidimensionais e multidimensionais**??

Para ilustrar melhor o que seriam esses **tipos de arrays**, segue a imagem a seguir:



- **Array Unidimensional:** Quando trabalhamos com listas de apenas 1 eixo.
- **Array Multidimensional (2D):** Quando trabalhamos com listas de 2 eixos.
- **Array Multidimensional (3D):** Quando trabalhamos com listas de 3 eixo.

Mas se eu for trabalhar apenas com **arrays unidimensionais**, vale a pena utilizar o **Numpy** para isso?? Ou melhor usar o **modulo Array** padrão do **Python** mesmo??

Se for trabalhar com **Data Science**, onde será necessário realizar vários tipos de análises e operações matemáticas, como realizar **somas, multiplicações, médias, etc.** O **Numpy** será **EXTREMAMENTE NECESSARIO!!**

Além disso, quando trabalhamos com milhares de dados, a performance do **Numpy** sempre será amplamente superior!!

Por isso que quando trabalhamos com a biblioteca Pandas para uso de **Data Science**, o uso da biblioteca **Numpy** é altamente recomendada na sua utilização em conjunto!

Criando array com Numpy

Para começar, vamos criar um **array simples** utilizando a biblioteca do **Numpy**, e veja como é simples!

```
import numpy as np
✓ 0.3s

lista_np = np.array([1, 2, 3, 4, 5])
lista_np
✓ 0.0s  🐞 Open 'lista_np' in Data Wrangler

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

Veja que primeiramente importamos a biblioteca do **Numpy**, e com a biblioteca importada, utilizamos a função **array()**, para que seja criado um objeto **ndarray**, onde nesse caso será um **array unidimensional**!

E para demonstrar o ganho de performance quando trabalhamos com a biblioteca **Numby**, segue a criação de **3 listas**, onde foi criado com o tipo de objeto **List, Array e Numpy**. Veja quantos bytes são armazenados em **memória** cada um!

```
[30] sys.getsizeof([1, 2, 3, 4, 5])
✓ 0.0s
... 104 ——— List


[31] sys.getsizeof(array('i', [1, 2, 3, 4, 5]))
✓ 0.0s
... 100 ——— Array

[32] np.array([1, 2, 3, 4, 5]).nbytes
✓ 0.0s
... 20 ——— Numpy
```

Criando arrays em 1D, 2D e 3D


Já vimos anteriormente como criar **arrays 1D**, porém agora além de criar **arrays 1D** criaremos também **arrays de 2D e 3D!!**

```
d1 = np.array([1, 2, 3])
print(d1)
```

✓ 0.0s  Open 'd1' in Data Wrangler


[1 2 3] **1D**

```
d2 = np.array([[1, 2], [2, 3], [3, 4]])
print(d2)
```

✓ 0.0s  Open 'd2' in Data Wrangler

[[1 2]
[2 3]
[3 4]] **2D**

```
d3 = np.array([[[1, 2], [2, 3]], [[3, 4], [4, 5]]])
print(d3)
```

✓ 0.0s  Open 'd3' in Data Wrangler

[[[1 2]
[2 3]]
[[3 4]
[4 5]]] **3D**

Veja que o que diferencia um array **1D**, **2D** ou **3D** é a forma como montamos o nosso **array** na função **array()**!

E para ter certeza qual dimensão que representa o nossos **arrays** criados pelo **Numpy**, podemos utilizar a propriedade **ndim**, onde será retornado o número da dimensão do **array** criado!

```
print(d1.ndim)
print(d2.ndim)
print(d3.ndim)
```

✓ 0.0s

1
2
3

Somando arrays

Um recurso bem interessante quando trabalhamos com **arrays** do **Numpy**, é a possibilidade realizar somas entre os **arrays** de uma forma bem mais simples!!

Pois quando trabalhamos com listas normais do **Python**, ao tentar a realizar a soma com o operador matemático **+**, na verdade está gerando uma nova lista concatenando as listas em si! Como podemos ver a seguir:

```
lista1 = [1, 2, 3]
lista2 = [1, 2, 3]
resultado = lista1 + lista2

print(resultado)
```

✓ 0.0s Open 'resultado' in Data Wrangler

[1, 2, 3, 1, 2, 3]

Porém conseguimos realizar a soma de fato que tanto queremos, porém veja que ficaria um pouco mais complexo, principalmente quando trabalhamos com múltiplas listas!!

```
lista1 = [1, 2, 3]
lista2 = [1, 2, 3]

resultado = [i + j for i, j in zip(lista1, lista2)]
print(resultado)
```

✓ 0.0s Open 'resultado' in Data Wrangler

[2, 4, 6]

Agora veremos como realizamos essa mesma soma, porém utilizando a biblioteca do **Numpy**!! Veja que será muito mais simples, pois agora sim podemos utilizar operadores matemáticos de forma explícita!

```
lista1 = np.array([1, 2, 3])
lista2 = np.array([1, 2, 3])

resultado = lista1 + lista2
print(resultado)
```

✓ 0.0s Open 'resultado' in Data Wrangler

[2 4 6]

Ou podemos utilizar a função **add()** diretamente da biblioteca do **numpy** também passando os **arrays** que queremos como **argumento**!! Porém não seria necessário, o operador matemático já resolveria o nosso problema!

```
lista1 = np.array([1, 2, 3])
lista2 = np.array([1, 2, 3])

resultado = np.add(lista1, lista2)
print(resultado)
```

✓ 0.0s Open 'resultado' in Data Wrangler

[2 4 6]

Buscando itens em arrays

Um recurso essencial quando trabalhamos com **arrays**, seja **unidimensional** ou **multidimensional**, é buscar itens dentro dele!!

E para buscar esses valores é bem simples, basta indicar os indexes do **array**, da mesma forma como já estamos acostumados!!

Para exemplificar, vamos criar um **array 2D** com o **numpy**!

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

✓ 0.0s Open 'lista_np' in Data Wrangler

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

Após a criação do **array**, vamos buscar o valor **3** que está na **segunda linha** e **primeira coluna**!

```
lista_np = np.array([[1, 2, 3], [3, 4, 5]])
print(lista_np[1, 0])
```

✓ 0.0s

```
3
```

Podemos também buscar um **vetor/lista inteira** dentro de uma **matriz**, utilizando o recurso do **slice**, como já conhecemos:

- Buscando a primeira **linha** inteira da **matriz**:

```
print(lista_np[0, :])
```

✓ 0.0s

```
[1 2 3]
```

- Buscando a primeira **coluna** inteira da **matriz**:

```
print(lista_np[:, 0])
```

✓ 0.0s

```
[1 3]
```

Veja como fica fácil de se trabalhar com matrizes utilizando o **Numpy**!! Agora podemos

também trabalhar com “**Mini Tabelas**” em até **3D** com uma performance fora do comum!!

Conclusão

Basicamente se precisamos trabalhar com **listas** ou **matrizes (2D ou 3D)** com uma performance bem considerável, a utilização da biblioteca **Numpy** será indispensável!!

Além disso, quando trabalhamos com **Data Science** juntamente com a biblioteca do **Pandas**, para realizar toda análise e limpeza dos dados, o Pandas acaba utilizando o **Numpy** por debaixo dos panos!!

Até porque quando utilizamos o **DataFrame** ou **Series** do **Pandas**, na verdade estamos utilizando os **arrays** do **Numpy** com uma camada a mais do próprio **Pandas**!!

E claro, podemos também utilizar o Numpy juntamente com o Pandas também, mesmo que o Pandas já utilize o Numpy internamente!! Pois o **Numpy** possui mais opções de funções **matemáticas** e **algébricas**, que pode ser muito útil durante a análise no **Data Science**!!