

Python para dados

Numpy



Introdução a Numpy

Numpy e o ecossistema Python



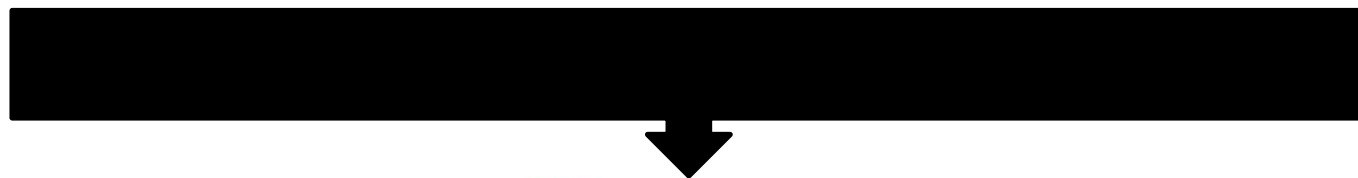
pandas



matplotlib



TensorFlow





Array

1	2	3	4
---	---	---	---

1	2	3	3
4	5	6	6
7	8	9	9

					20	2	3	5
				1	2	3	3	
			20	2	3	5		8
		4	11	8	8		6	12
		7	8	9	12		9	



Importando Numpy

```
import numpy as np
```



Importando Numpy

```
import numpy as np
```

```
1 l = [4, 8, 2, 12, 5, 8, 0]
```

```
1 arr = np.array(l)  
2 arr
```

```
array([ 4,  8,  2, 12,  5,  8,  0])
```



Criando um numpy array do zero

```
np.zeros()  
np.arange()  
np.random.random()
```



Por que usar numpy e não listas?

- Arrays NumPy aceita que todos os elementos sejam de diferentes tipo de dados, contudo o array sempre terá apenas um tipo de dados*. Contudo a normal dentro da análise de dados que usamos **apenas um tipo de dados dentro de um array**.
- Um único tipo de dados também resulta em arrays NumPy ocupando **menos espaço na memória** em comparação com listas.
- Quando precisamos de uma **estrutura multidimensional** para armazenar os dados, optamos por arrays em vez de listas, pois as listas podem ser unidimensionais apenas.
- Se precisarmos de um comprimento fixo e alocação estática, usamos arrays em vez de listas.
- Quando é necessária uma processamento de dados mais rápido, preferimos arrays em vez de listas.
- Tipos de dados primitivos podem ser armazenados diretamente em arrays, mas não em listas.



Arrays e mais dimensões

Array 3D

```
arr1_2d = np.array([[1, 2], [3, 4]])  
arr2_2d = np.array([[5, 6], [7, 8]])  
arr3_2d = np.array([[9, 2], [3, 4]])  
  
arr_3d = np.array([arr1_2d, arr1_2d, arr3_2d])
```



Array 3D

```
arr1_2d = np.array([[1, 2], [3, 4]])  
arr2_2d = np.array([[5, 6], [7, 8]])  
arr3_2d = np.array([[9, 2], [3, 4]])  
  
arr_3d = np.array([arr1_2d, arr1_2d, arr3_2d])
```

1 arr_3d

```
array([[[1, 2],  
        [3, 4]],  
       [[1, 2],  
        [3, 4]],  
       [[9, 2],  
        [3, 4]]])
```

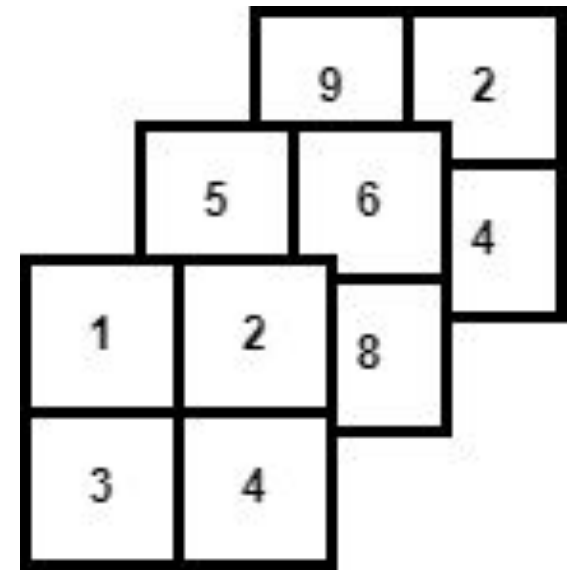


Array 3D

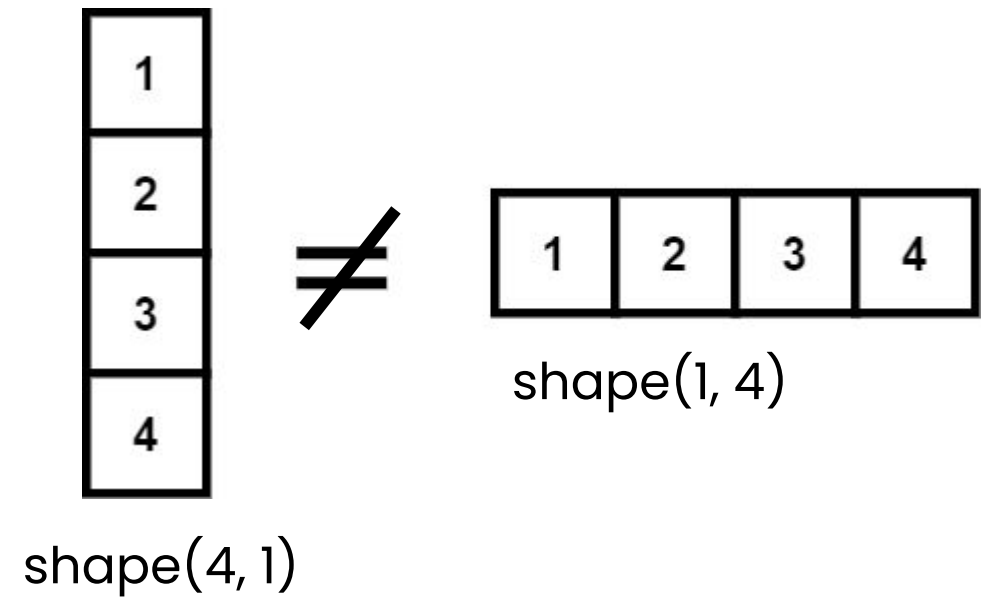
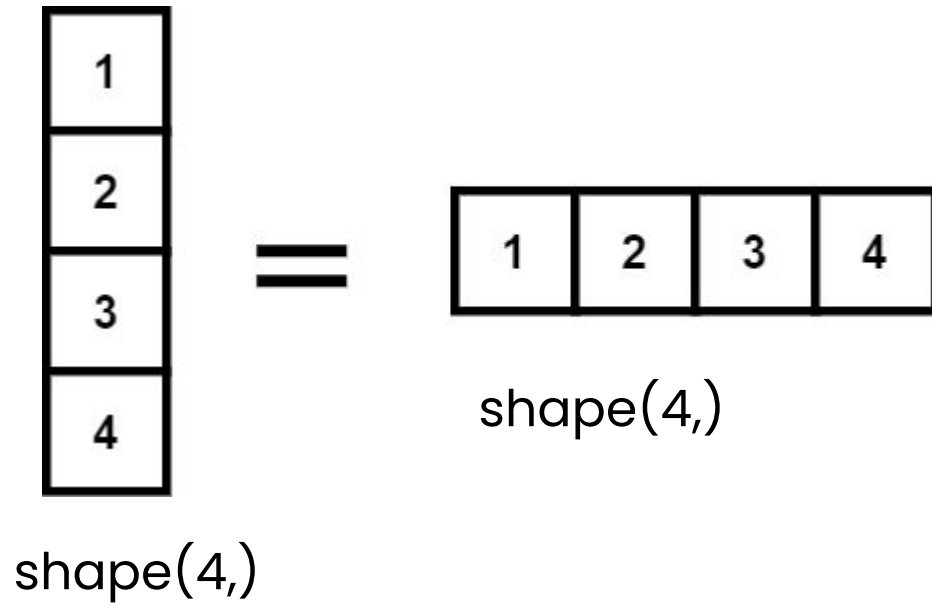
```
arr1_2d = np.array([[1, 2], [3, 4]])  
arr2_2d = np.array([[5, 6], [7, 8]])  
arr3_2d = np.array([[9, 2], [3, 4]])  
  
arr_3d = np.array([arr1_2d, arr1_2d, arr3_2d])
```

1 arr_3d

```
array([[[1, 2],  
        [3, 4]],  
       [[1, 2],  
        [3, 4]],  
       [[9, 2],  
        [3, 4]]])
```



Array 3D



Matriz

2 dimensões

1	2	3	3
4	5	6	6
7	8	9	9



Matriz

2 dimensões

1	2	3	3
4	5	6	6
7	8	9	9

Tensor

mais de 3 dimensões

				20	2	3	5
			1	2	3	3	
20	2	3	5				8
4	11	8	8			6	
7	8	9	12			9	12



Dimensões do array (atributos e métodos)

Atributo:

`.shape`

Método:

`.flatten()`
`.reshape()`



Quantas dimensões tem o array (shape)

```
1 arr = np.zeros((2, 4))  
2 arr
```

```
array([[0., 0., 0., 0.],  
       [0., 0., 0., 0.]])
```

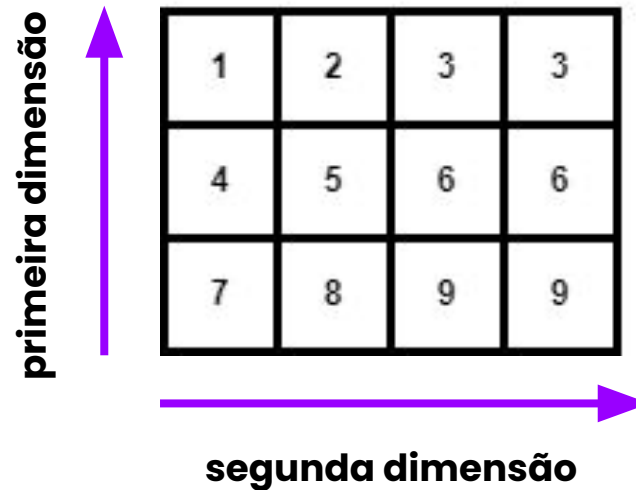
```
1 arr.shape
```

```
(2, 4)
```



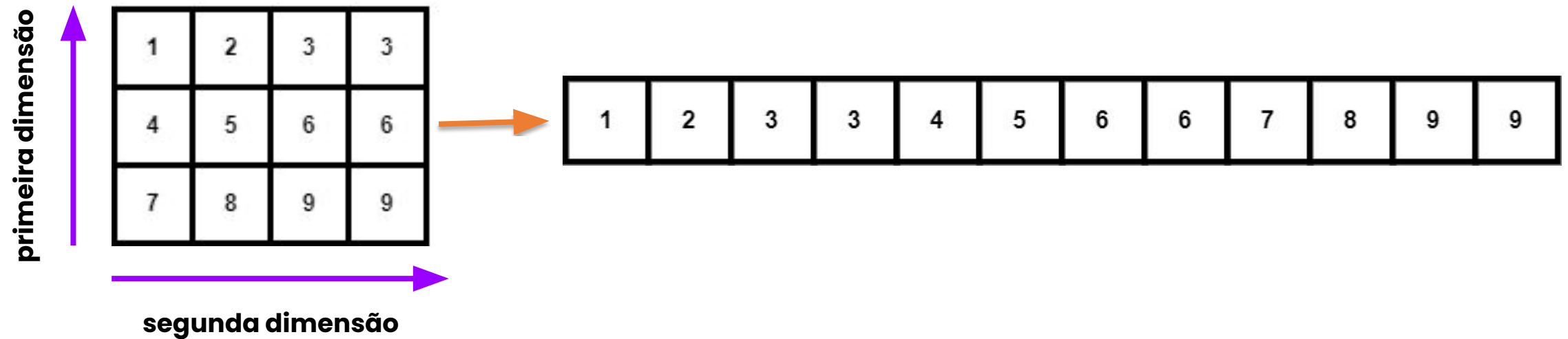
Linhas e colunas em arrays 2D

- Linhas (rows) são a primeira dimensão
- Colunas (columns) são a segunda dimensão



“Achatando” um array

Coloca o array em apenas uma dimensão



Reshaping – mudando as dimensões

```
1 arr = np.array([[3, 4, 2], [8, 11, 5]])
2 arr.reshape((3, 2))
3
```

```
array([[ 3,  4],
       [ 2,  8],
       [11,  5]])
```

```
1 arr.reshape((3, 3))
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-65-a5ab0f3d8792> in <cell line: 1>()
----> 1 arr.reshape((3, 3))
```

```
ValueError: cannot reshape array of size 6 into shape (3,3)
```



HORA DE PRATICAR



Tipo de dados em um numpy array

Tipos em Numpy vs. tipo em Python

Python

- `Int`
- `float`

Numpy

- `np.int64`
- `np.int32`
- `np.float64`
- `np.float32`



Bits e bytes

000001111101000



Bits e bytes

2024

000001111101000



Bits e bytes

2024

0000011111101000

8 bits = 1 byte

8 bits = 1 byte

np.int32 armazena 4.294.967.296 em valor de inteiro = 2^{32}

2.147.488.648

2.147.488.648

4.294.967.296 = 2^{32}



Atributo .dtype e tipo default

```
1 np.array([2.14, 6.25, 160.87, 8.4]).dtype  
dtype('float64')
```

```
1 np.array([2, 6, 160, 8]).dtype  
dtype('int64')
```



String

```
1 np.array(["Hello", "Coder", "Girls"]).dtype  
dtype('<U5')
```



Conversão e coerção

dtype como argumento

```
1 arr_float32 = np.array([2.14, 6.25, 160.87], dtype=np.float32)
2 arr_float32
```

```
array([ 2.14,  6.25, 160.87], dtype=float32)
```

Conversão

```
1 arr_vf = np.array([[True, False], [True, True]], dtype=np.bool8)
2 arr_vf.astype(np.int32)
```

```
array([[1, 0],
       [1, 1]], dtype=int32)
```

Coerção

```
1 np.array(["pedra", False, 42, 42.42])
```

```
array(['pedra', 'False', '42', '42.42'], dtype='<U32')
```



Hierarquia de coerção

Adicionar um float a um array de int mudará todos os valores para float.

```
1 np.array([0, 10]).dtype  
dtype('int64')
```

```
1 np.array([0, 10, 1.]).dtype  
dtype('float64')
```

Adicionar um inteiro em um array de booleanos transformará todo o array em inteiros

```
1 np.array([True, False]).dtype  
dtype('bool')
```

```
1 np.array([True, False, 12]).dtype  
dtype('int64')
```



HORA DE PRATICAR



Manipulando array

Manipulando arrays

Index e slices



Index em um array de 1 dimensão

```
1  arr = np.array([2, 4, 6, 8])  
2  arr[2]
```

6



Index em um array de 2 dimensão

16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19

```
1  cartela_bing[1, 3]
```

27



Index em um array de 2 dimensão

16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19

```
1 cartela_bing[0]
```

```
array([16, 10, 3, 15])
```



Index em um array de 2 dimensão

16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19

```
1 cartela_bing[:, 1]  
  
array([10, 23, 19,  4])
```



Fatiar (slicing) um array de 1 dimensão



```
1  arr = np.array([2, 4, 6, 8, 10])  
2  arr[2:4]
```

```
array([6, 8])
```



Fatiar (slicing) um array de 2 dimensão

16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19

```
1  cartela_bingo[1:3, 0:2]
```

```
array([[14, 23],  
       [ 6, 19]])
```



Ordenar (sorting) um array

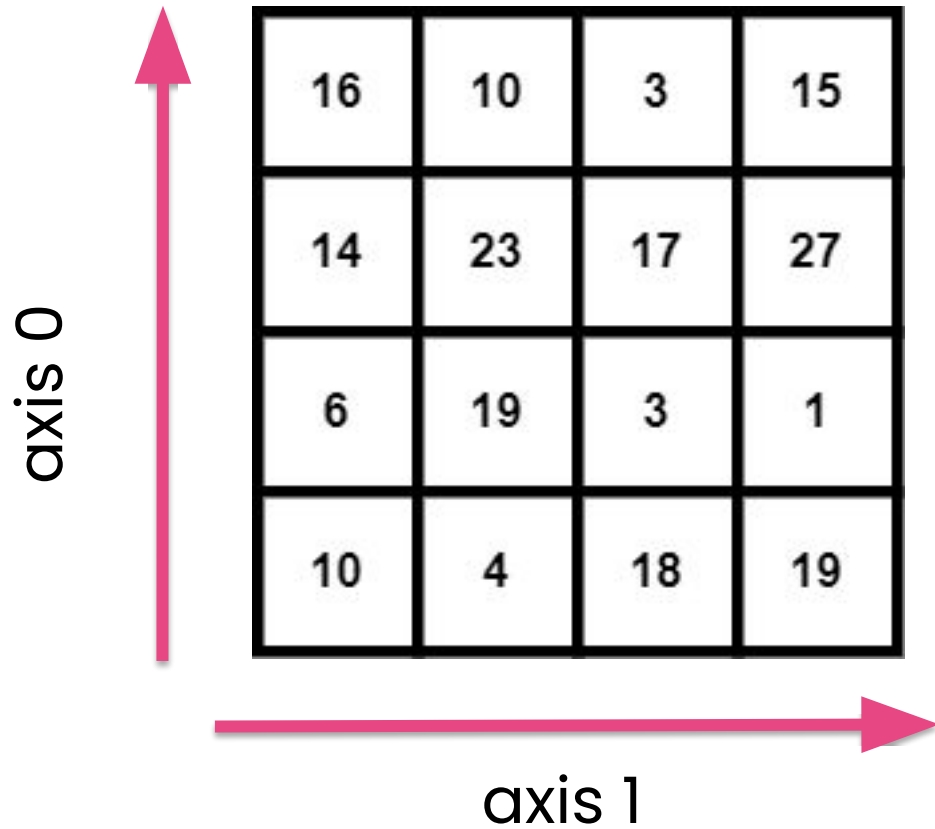
16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19

```
1 np.sort(cartela_bingo)
```

```
array([[ 3, 10, 15, 16],  
       [14, 17, 23, 27],  
       [ 1,  3,  6, 19],  
       [ 4, 10, 18, 19]])
```



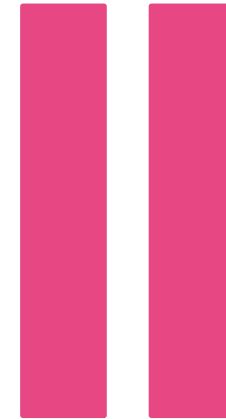
Eixos (axis)



axis 0



axis 1



Eixos (axis)

axis 0

16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19

axis 1

```
1 np.sort(cartela_bingo, axis=0)
```

```
array([[ 6,  4,  3,  1],  
       [10, 10,  3, 15],  
       [14, 19, 17, 19],  
       [16, 23, 18, 27]])
```

```
1 np.sort(cartela_bingo, axis=1)
```

```
array([[ 3, 10, 15, 16],  
       [14, 17, 23, 27],  
       [ 1,  3,  6, 19],  
       [ 4, 10, 18, 19]])
```



HORA DE PRATICAR



Filtrando arrays

Indexação sofisticada (fancy indexing) e máscara (mask) em array 2D

```
1 pessoas_id_idade = np.array([[1, 22], [2, 21], [3, 27], [4, 26]])
```

```
1 pessoas_id_idade[:, 1] % 2 == 0
```

```
array([ True, False, False,  True])
```



Indexação sofisticada (fancy indexing) e máscara (mask)

Boolean mask

- Retorna um array de verdadeiros e falsos

```
1 arr1 = np.array([1, 2, 3, 4, 5])
2 mask = arr1 % 2 == 0
3 mask
```

```
array([False,  True, False,  True, False])
```

Fancy indexing

- Retorna os valores do array quando passado a máscara

```
1 arr1[mask]
```

```
array([2, 4])
```



Fancy indexing e np.where()

Fancy indexing

- Retorna os valores do array quando passado a máscara

```
1 cartela_bingo
```

```
array([[16, 10,  3, 15],  
       [14, 23, 17, 27],  
       [ 6, 19,  3,  1],  
       [10,  4, 18, 19]])
```

```
1 np.where(cartela_bingo % 3 == 0)
```

```
(array([0, 0, 1, 2, 2, 3]), array([2, 3, 3, 0, 2, 2]))
```

np.where()

- Retorna um array de índices
- Cria uma matriz com base em se os elementos correspondem ou não a uma condição.

```
1 pessoas_id_idade = np.array([[1, 22], [2, 21], [3, 27], [4, 26]])
```

```
1 np.where(pessoas_id_idade[:, 1] % 2 == 0)
```

```
(array([0, 3]),)
```



Buscar e substituir

```
1 np.where(cartela_bingo % 3 == 0, "", cartela_bingo)
2
```

```
array([[ '16', '10', '', ''],
       [ '14', '23', '17', ''],
       [ '', '19', '', '1'],
       [ '10', '4', '', '19']], dtype='<U21')
```



HORA DE PRATICAR



Adicionando e removendo dados do array

Concatenando linhas

16	10	3	15
14	23	17	27
6	19	3	1



10	4	18	19
----	---	----	----



16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19



Concatenando linhas

```
1 arrv1 = np.random.randint(10, size=(3,2))
2 arrv1
```

```
array([[7, 6],
       [4, 1],
       [1, 1]])
```

```
1 arrv2 = np.array([[ "Peras", "Morango"]])
2 arrv2
```

```
array([[ 'Peras', 'Morango']], dtype='<U7')
```

```
1 np.concatenate((arrv1, arrv2))
```

```
array([[ '7', '6'],
       [ '4', '1'],
       [ '1', '1'],
       [ 'Peras', 'Morango']], dtype='<U21')
```



Concatenando columnas

16	10	3
14	23	17
6	19	3
10	4	18



15
27
1
19



16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19



Concatenando columnas

```
1 arrv1
```

```
array([[7, 6],  
       [4, 1],  
       [1, 1]])
```

```
1 arro1 = np.array(["Uva", "Abacaxí", "Laranja"]).reshape((3, 1))  
2 arro1
```

```
array([[ 'Uva'],  
       ['Abacaxí'],  
       ['Laranja']], dtype='<U7')
```

```
1 np.concatenate((arrv1, arro1), axis=1)
```

```
array([[ '7', '6', 'Uva'],  
       [ '4', '1', 'Abacaxí'],  
       [ '1', '1', 'Laranja']], dtype='<U21')
```



Concatenando columnas

16	10	3
14	23	17
6	19	3
10	4	18



27
1



ERRO



Concatenando columnas

16	10	3
14	23	17
6	19	3
10	4	18

Shape(4,3)



15
27
1
19

Shape(4,)



ERRO

16	10	3
14	23	17
6	19	3
10	4	18

Shape(4,3)



15
27
1
19

Shape(4,1)



16	10	3	15
14	23	17	27
6	19	3	1
10	4	18	19



Concatenando columnas

```
1 arrv1
```

```
array([[7, 6],  
       [4, 1],  
       [1, 1]])
```

```
1 arro1 = np.array(["Uva", "Abacaxí", "Laranja"]).reshape((3, 1))  
2 arro1
```

```
array([[ 'Uva'],  
       ['Abacaxí'],  
       ['Laranja']], dtype='<U7')
```

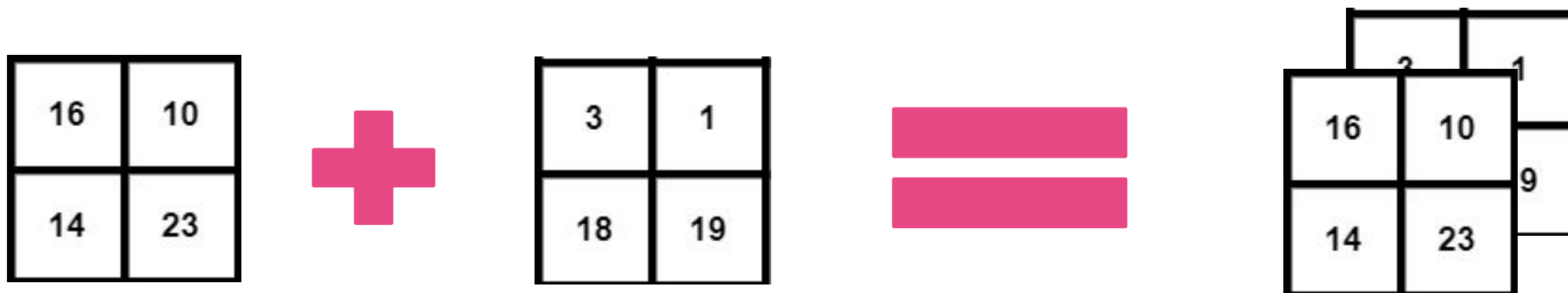
```
1 np.concatenate((arrv1, arro1), axis=1)
```

```
array([[ '7', '6', 'Uva'],  
       [ '4', '1', 'Abacaxí'],  
       [ '1', '1', 'Laranja']], dtype='<U21')
```



Concatenando colunas

- É possível adicionar arrays para criar novas dimensões
- Mas o método `np.concatenate` não permite
- É compatível apenas para as dimensões já existentes no array



Não com `np.concatenate()`



Deletando com np.delete()

Deletando com axis = 0

```
5 sala_espera  
  
array([[ '5', '30', '1', 'Alice'],  
      [ '6', '29', '1', 'Bob'],  
      [ '7', '35', '3', 'Cristina'],  
      [ '8', '34', '3', 'Luiz']], dtype='<U8')
```

```
1 np.delete(sala_espera, 2, axis=0)  
  
array([[ '5', '30', '1', 'Alice'],  
      [ '6', '29', '1', 'Bob'],  
      [ '8', '34', '3', 'Luiz']], dtype='<U8')
```

Deletando com axis = 1

```
1 sala_espera  
  
array([[ '5', '30', '1', 'Alice'],  
      [ '6', '29', '1', 'Bob'],  
      [ '7', '35', '3', 'Cristina'],  
      [ '8', '34', '3', 'Luiz']], dtype='<U8')
```

```
1 np.delete(sala_espera, 1, axis=1)  
  
array([[ '5', '1', 'Alice'],  
      [ '6', '1', 'Bob'],  
      [ '7', '3', 'Cristina'],  
      [ '8', '3', 'Luiz']], dtype='<U8')
```



Deletando com np.delete()

Deletando sem axis

```
1 sala_espera
```

```
array([[ '5', '30', '1', 'Alice'],  
       [ '6', '29', '1', 'Bob'],  
       [ '7', '35', '3', 'Cristina'],  
       [ '8', '34', '3', 'Luiz']], dtype='<U8')
```

```
1 np.delete(sala_espera, 1)
```

```
array([ '5', '1', 'Alice', '6', '29', '1', 'Bob', '7', '35', '3',  
       'Cristina', '8', '34', '3', 'Luiz'], dtype='<U8')
```



Cálculos com Array

Métodos de agregação de dados

Python

- `.sum()`
- `.min()`
- `.max()`
- `.mean()`
- `.cumsum()`



Dados

1 accidentes

```
array([[1, 3, 2],  
       [0, 1, 0],  
       [2, 1, 4],  
       [0, 0, 0],  
       [1, 1, 0]])
```

cliente 1	cliente 2	cliente 3	
1	3	2	ano 1
0	1	1	ano 2
2	1	4	ano 3
0	0	0	ano 4
1	1	0	ano 5



Somar

```
1 acidentes.sum()
```

16

cliente 1	cliente 2	cliente 3	
1	3	2	ano 1
0	1	1	ano 2
2	1	4	ano 3
0	0	0	ano 4
1	1	0	ano 5



Somar linhas

```
1 acidentes.sum(axis=0)
```

```
array([4, 6, 6])
```


cliente1	cliente2	cliente3	
1	3	2	ano 1
0	1	1	ano 2
2	1	4	ano 3
0	0	0	ano 4
1	1	0	ano 5



Somar columnas

```
1 accidentes.sum(axis=1)
```

```
array([6, 1, 7, 0, 2])
```



	cliente 1	cliente 2	cliente 3	
1	3	2		ano 1
0	1	1		ano 2
2	1	4		ano 3
0	0	0		ano 4
1	1	0		ano 5



Somar columnas

cliente 1	cliente 2	cliente 3	
1	3	2	ano 1
0	1	1	ano 2
2	1	4	ano 3
0	0	0	ano 4
1	1	0	ano 5

```
1 accidentes.sum(axis=1)  
array([6, 1, 7, 0, 2])
```



Todos os clientes	
6	ano 1
1	ano 2
7	ano 3
0	ano 4
2	ano 5



Mínimos e máximos

```
1 acidentes.min()
```

0

```
1 acidentes.max()
```

4

```
1 acidentes.max(axis=0)
```

array([2, 3, 4])

cliente 1	cliente 2	cliente 3	
1	3	2	ano 1
0	1	1	ano 2
2	1	4	ano 3
0	0	0	ano 4
1	1	0	ano 5



Média

```
1 acidentes.mean()
```

```
1.0666666666666667
```

```
1 acidentes.mean(axis=0)
```

```
array([0.8, 1.2, 1.2])
```

```
1 acidentes.mean(axis=1)
```

```
array([2.          , 0.33333333, 2.33333333, 0.          , 0.66666667])
```

cliente 1	cliente 2	cliente 3	
1	3	2	ano 1
0	1	1	ano 2
2	1	4	ano 3
0	0	0	ano 4
1	1	0	ano 5



Keepdims - manter as mesmas dimensões

```
1 acidentes.mean(axis=0)
```

```
array([0.8, 1.2, 1.2])
```

```
1 acidentes.mean(axis=0, keepdims=True)
```

```
array([[0.8, 1.2, 1.2]])
```

```
1 acidentes.mean(axis=1)
```

```
array([2.          , 0.33333333, 2.33333333, 0.          , 0.66666667])
```

```
1 acidentes.mean(axis=1, keepdims=True)
```

```
array([[2.          ],  
       [0.33333333],  
       [2.33333333],  
       [0.          ],  
       [0.66666667]])
```

	cliente 1	cliente 2	cliente 3	
1	3	2	ano 1	
0	1	1	ano 2	
2	1	4	ano 3	
0	0	0	ano 4	
1	1	0	ano 5	



Soma cumulativa - cumsum

```
1 acidentes.cumsum(axis=0)
```

```
array([[1, 3, 2],  
       [1, 4, 2],  
       [3, 5, 6],  
       [3, 5, 6],  
       [4, 6, 6]])
```

```
1 acidentes.cumsum(axis=1)
```

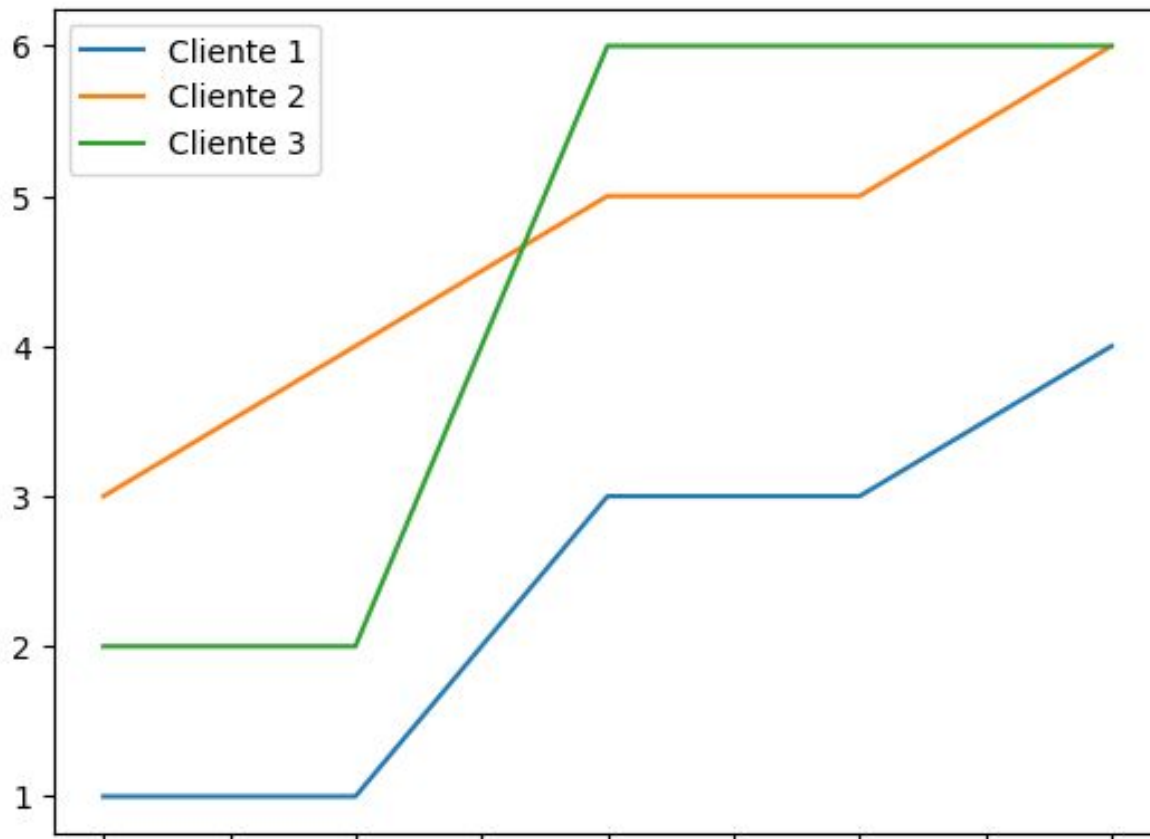
```
array([[1, 4, 6],  
       [0, 1, 1],  
       [2, 3, 7],  
       [0, 0, 0],  
       [1, 2, 2]])
```

	cliente 1	cliente 2	cliente 3	
1	3	2	ano 1	
0	1	1	ano 2	
2	1	4	ano 3	
0	0	0	ano 4	
1	1	0	ano 5	



Soma cumulativa - cumsum

```
1 plt.plot(np.arange(1, 6), acidentes.cumsum(axis=0), label=['Cliente 1', 'Cliente 2', 'Cliente 3'])  
2 plt.legend()  
3 plt.show()
```



HORA DE PRATICAR



Operações com vetorização

Comparação operações - Python vs Numpy

Ao realizar a soma de elementos em um array utilizando o NumPy, a operação não ocorre de forma individual para cada elemento; em vez disso, ela é realizada de maneira simultânea para todos os elementos do array.

```
1  arr = np.array([[1, 2], [4, 5]])
2
3  for linha in range(0, arr.shape[0]):
4      for coluna in range(0, arr.shape[1]):
5          arr[linha][coluna] += 3
6
7  arr
```

```
array([[4, 5],
       [7, 8]])
```

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

```
array([[4, 5],
       [7, 8]])
```

velocidade



Sintax Numpy

No NumPy, a **sintaxe é otimizada**: empregamos um sinal de adição e indicamos ao NumPy o número único que desejamos adicionar a todos os elementos do array. Na linguagem matemática, esse único número é comumente denominado **escalar**. Podemos utilizar uma sintaxe semelhante para multiplicar um array por um escalar.

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

```
array([[4, 5],
       [7, 8]])
```

```
1  arr2 = np.array([[1, 2], [4, 5]])
2
3  arr2 * 5
```

```
array([[ 5, 10],
       [20, 25]])
```



Adicionando 2 array

O NumPy permite operações vetorizadas entre arrays do mesmo formato. Na adição de dois arrays com o mesmo formato, o NumPy realiza a operação elemento por elemento, somando cada elemento do primeiro array ao elemento correspondente no segundo array.

```
1 a = np.array([[1, 2, 3], [4, 5, 6]])
2 b = np.array([[2, 2, 2], [0, 0, 1]])
3 a + b
```

```
array([[3, 4, 5],
       [4, 5, 7]])
```

Multiplicando 2 arrays

O mesmo conceito se aplica à multiplicação, subtração e divisão de dois arrays no NumPy. Nessas operações, os elementos nas posições correspondentes de cada array são multiplicados, subtraídos ou divididos entre si.

```
1 a = np.array([[1, 2, 3], [4, 5, 6]])
2 b = np.array([[2, 2, 2], [0, 0, 1]])
3 a * b
```

```
array([[2, 4, 6],
       [0, 0, 6]])
```



Operação booleanas

Embora as operações vetorizadas se destaquem em operações matemáticas, seu aproveitamento se estende por todo o NumPy. Anteriormente, exploramos o uso dessas operações para criar máscaras booleanas e filtrar arrays.

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

```
array([[False, False, False],  
       [ True,  True,  True]])
```



Vetorizando código Python (np.vectorize)

`np.vectorize` transforma funções Python regulares em funções vetorizadas. Isso significa que podemos aplicar eficientemente uma função a cada elemento de um array NumPy sem nos preocuparmos com loops explícitos. É muito útil quando precisamos operar em arrays, mas temos funções projetadas para operar em valores únicos.

```
1 arr = np.array(["Hello", "meninas", "coders"])
2 len(arr) > 5
```

False

```
1 vetorizar_len = np.vectorize(len)
2
3 vetorizar_len(arr) > 5
```

array([False, True, True])



Vetorizando código Python (np.vectorize)

`np.vectorize` transforma funções Python regulares em funções vetorizadas. Isso significa que podemos aplicar eficientemente uma função a cada elemento de um array NumPy sem nos preocuparmos com loops explícitos. É muito útil quando precisamos operar em arrays, mas temos funções projetadas para operar em valores únicos.

```
1  def minha_funcao(x):  
2      return x**2 + 3*x + 1  
3  
4  vetorizada_funcao = np.vectorize(minha_funcao)  
5  
6  array_original = np.array([1, 2, 3, 4, 5])  
7  
8  resultado = vetorizada_funcao(array_original)  
9  
10 resultado
```

```
array([ 5, 11, 19, 29, 41])
```



Broadcasting

Broadcasting (transmissão)

Broadcasting (transmissão) refere-se à capacidade do NumPy de lidar com operações entre arrays de diferentes formas e tamanhos durante operações aritméticas. O Broadcasting permite que o NumPy execute essas operações mesmo quando as dimensões dos arrays não são as mesmas, seguindo um conjunto de regras específicas.

```
1 arr2 = np.array([[1, 2], [4, 5]])
2
3 arr2 * 5
```

```
array([[ 5, 10],
       [20, 25]])
```

```
1 arr_3x3 = np.array([[1, 2],
2                     [4, 5],
3                     [7, 8]])
4 arr_3x3
```

```
array([[1, 2],
       [4, 5],
       [7, 8]])
```

```
1 arr_3x1 = np.array([1, 0, -1]).reshape((3,1))
2 arr_3x1
```

```
array([[ 1],
       [ 0],
       [-1]])
```

```
1 arr_3x3 + arr_3x1
```

```
array([[2, 3],
       [4, 5],
       [6, 7]])
```



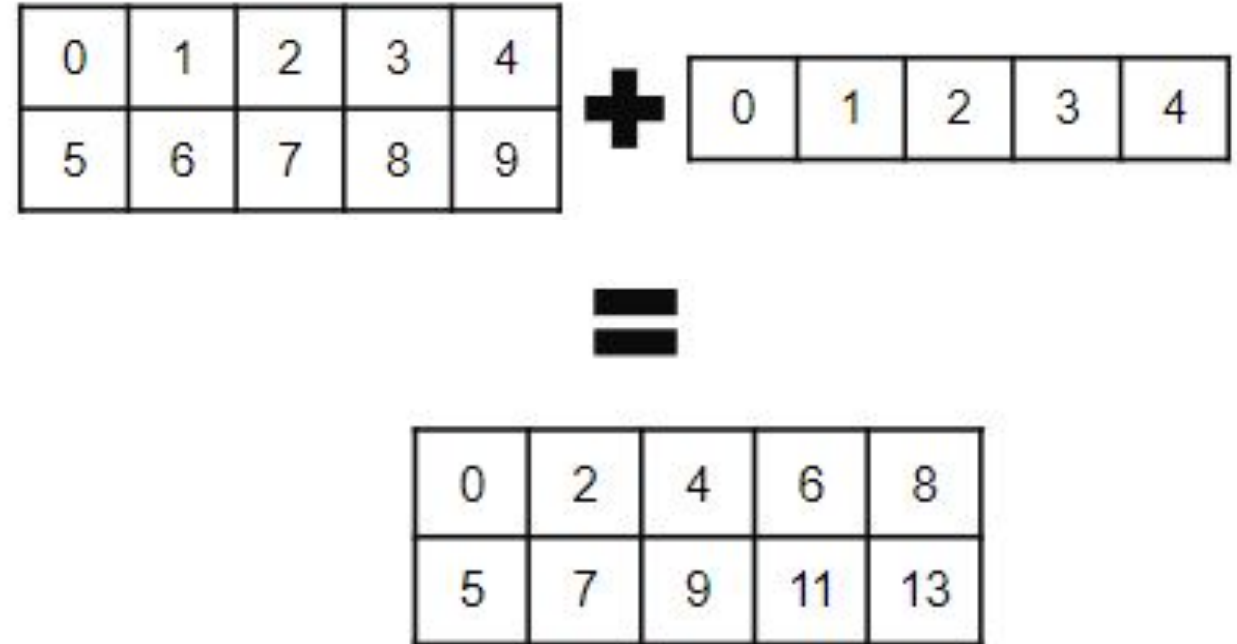
Linhas

```
1 arr10 = np.arange(10).reshape((2, 5))
2 arr5 = np.array([0, 1, 2, 3, 4])
3 arr10.shape, arr5.shape
```

```
((2, 5), (5,))
```

```
1 arr10 + arr5
```

```
array([[ 0,  2,  4,  6,  8],
       [ 5,  7,  9, 11, 13]])
```



Colunas

0	1	2	3	4
5	6	7	8	9

 $+$

0
1

 $=$

0	1	2	3	4
6	7	8	9	10

```
1 arr10 = np.arange(10).reshape((2, 5))
2 arr2 = np.array([0, 1]).reshape(2, 1)
3 arr10.shape, arr2.shape
```

```
((2, 5), (2, 1))
```

```
1 arr10 + arr2
```

```
array([[ 0,  1,  2,  3,  4],
       [ 6,  7,  8,  9, 10]])
```



Outras operações

A aplicação da mesma lógica estende-se à **multiplicação, subtração e divisão**! Ao multiplicar por uma coluna, cada elemento dessa coluna é multiplicado pelos elementos correspondentes em ambas as colunas do array maior. Da mesma forma, ao subtrair uma linha, cada elemento dessa linha é subtraído dos elementos correspondentes em ambas as linhas do array maior.

```
1 arr10 = np.arange(10).reshape((2, 5))
2 arr2 = np.array([2, 1]).reshape(2, 1)
3 arr10.shape, arr2.shape
```

```
((2, 5), (2, 1))
```

```
1 arr10 / arr2
```

```
array([[0. , 0.5, 1. , 1.5, 2. ],
       [5. , 6. , 7. , 8. , 9. ]])
```

```
1 arr10 = np.arange(10).reshape((2, 5))
2 arr2 = np.array([2, 1]).reshape(2, 1)
3 arr10.shape, arr2.shape
```

```
((2, 5), (2, 1))
```

```
1 arr10 * arr2
```

```
array([[0, 2, 4, 6, 8],
       [5, 6, 7, 8, 9]])
```



Salvando arquivos numpy

Por quê usar arquivos .npy

Dados numpy podem ser salvos em diferentes formatos:

- .npy (NumPy Binary)
- .txt (Text/CSV)
- .h5 (HDF5 - Hierarchical Data Format)
- .json (JSON)
- .csv (Comma-Separated Values)
- .parquet (Apache Parquet)



Por quê usar arquivos .npy

1. **Eficiência de Armazenamento:** O formato binário utilizado é altamente eficiente em termos de armazenamento, sendo especialmente útil para grandes conjuntos de dados, reduzindo os requisitos de espaço e melhorando a velocidade de operações de leitura e escrita.
2. **Preservação de Informações:** Ao salvar um array como .npy, o formato inclui informações cruciais sobre o tipo de dados e a forma do array. Isso garante que, ao carregá-lo posteriormente, a estrutura original seja recriada com precisão, evitando perda de informações.
3. **Carregamento Rápido:** O carregamento de dados a partir de um arquivo .npy é geralmente mais rápido do que em formatos de texto. Isso se deve à eficiente serialização e deserialização dos dados, resultando em tempos de carregamento mais curtos.
4. **Compatibilidade com NumPy:** O formato .npy é específico do NumPy, proporcionando compatibilidade sólida com outras ferramentas e bibliotecas baseadas em NumPy. Isso estabelece uma maneira padronizada e confiável de armazenar e compartilhar arrays entre diferentes projetos e colaboradores.



Carregando

```
1 arr = np.load('arr10.npy')
2 arr
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Salvando

```
1 arr10 = np.arange(10).reshape((2, 5))
2 arr10
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

```
1 np.save('arr10.npy', arr10)
```



HORA DE PRATICAR

