Fundamentos Matemáticos e Computacionais de Machine Learning

Especialização em Machine Learning e Big Data

UEL

Profa. Dra. Juliana Felix

jufelix16@uel.br

NumPy

Python NumPy

- NumPy, uma abreviação para Numerical Python,
- Esta é uma das principais bibliotecas em Python para computação científica.

Python NumPy

- É importante destacar que o Python não oferece vetores ou matrizes, mas oferece listas, dicionários, conjuntos de tuplas.
- Embora estas estruturas de dados sejam bastante úteis, elas podem exigir alguns "truques" para utilizá-las em computação científica.

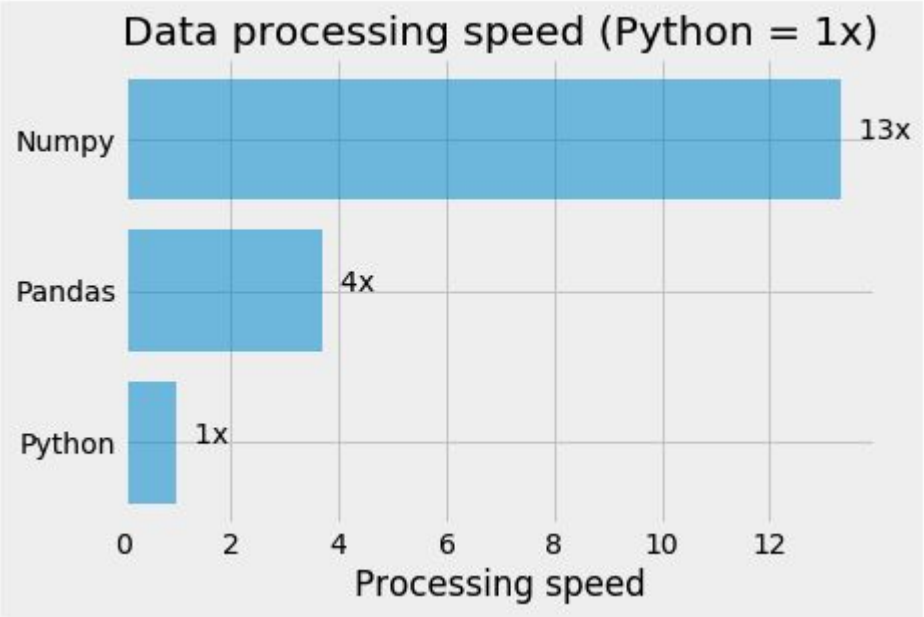
Python NumPy

- Em computação científica, manipular conjuntos de dados em formas básicas como vetores e matrizes pode ser mais prático e eficiente.
- Assim, o NumPy nos fornece um objeto chamado array multidimensional, que é extremamente veloz para cálculos matemáticos e numéricos.

Python NumPy

- NumPy foi escrito (em sua maior parte) em linguagem C, que é uma linguagem de baixo nível, o que torna a biblioteca extremamente veloz, escondendo toda sua complexidade em um módulo Python simples de utilizar.
- O NumPy faz uso da memória de forma diferente, ao contrário das listas em Python, de modo que as funções possam acessá-la e manipulá-la de maneira muito mais eficiente.

Python NumPy



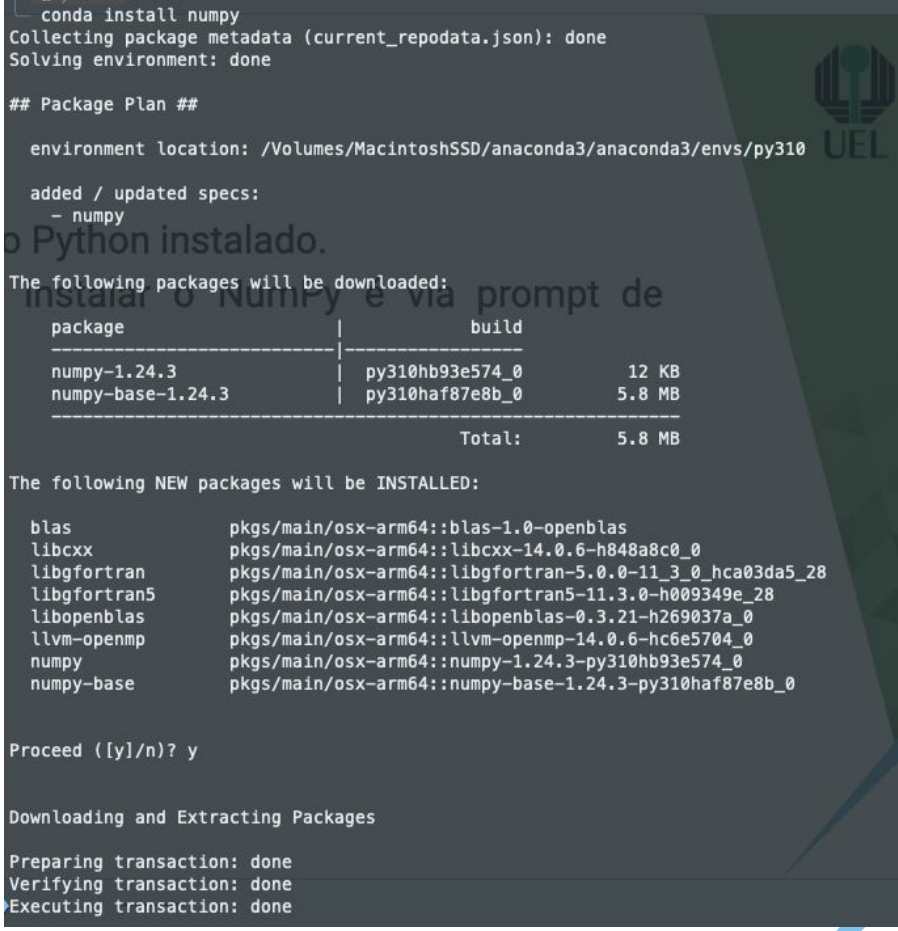
Python NumPy

- Em contrapartida, tal performance tem um custo.
- O NumPy, por esta característica, só trabalha com dados homogêneos (em sua maioria numéricos),
  - Um array só pode conter um tipo de dado
  - Diferente de uma lista em Python, onde pode-se misturar diversos tipos de dados em uma única lista.

Instalando o NumPy

Python NumPy

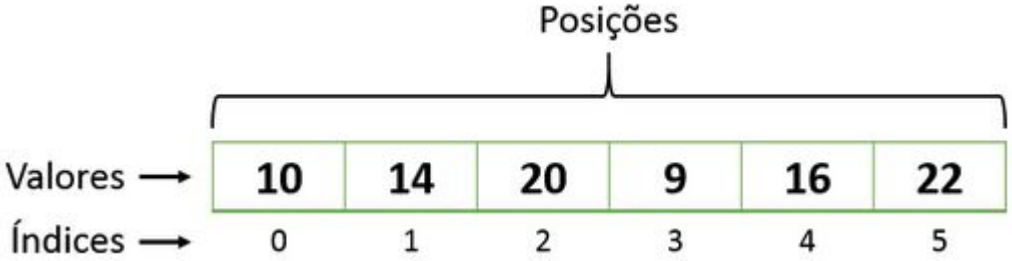
- O único pré-requisito é ter o Python instalado.
- A forma mais rápida de instalar o NumPy é via prompt de comando.
  - conda install numpy; ou
  - pip install numpy



Iniciando no NumPy

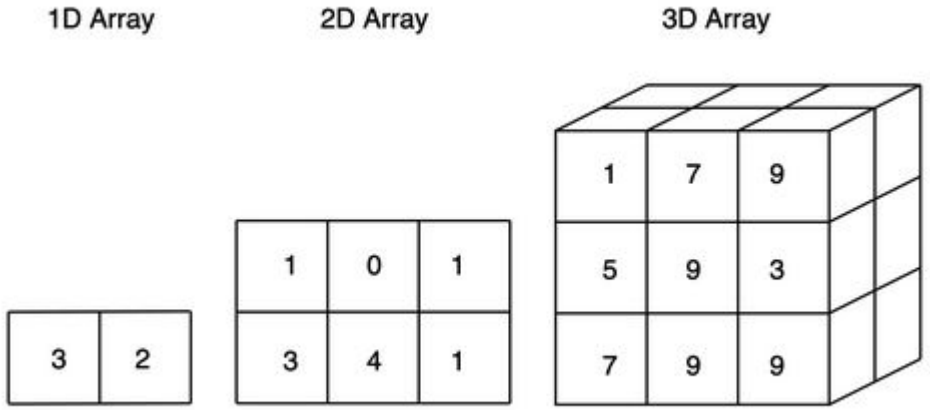
Criando arrays com NumPy

- Arrays são o tipo de objeto básico do NumPy.
- São como listas em Python, que contém valores armazenados em posições.



Criando arrays com NumPy

Um array também pode ser multidimensional, não se limitando apenas à 3D, dimensão máxima que o ser humano consegue compreender.



Criando arrays com NumPy

Um array pode ser criado com a função *array*, que recebe um objeto sequencial (incluindo outros arrays) e gera um novo *array* NumPy.

```
import numpy as np

# Criamos uma lista em Python
data = [1, 2, 3, 4, 5, 6]

# Passamos a lista como parâmetro para a função array do NumPy
arr = np.array(data)

# Vamos nos certificar que arr é do tipo array NumPy?
type(arr)

# E finalmente vamos imprimir o conteúdo de arr
print(arr)
```

Criando arrays com NumPy

- Observe que não é preciso criar uma variável lista para criar a variável do tipo *ndarray*.
- Você pode informar a lista diretamente na criação

```
import numpy as np

# Criamos uma lista e passamos a lista como parâmetro para a função array do NumPy
arr = np.array([1, 2, 3, 4, 5, 6])

# Vamos nos certificar que arr é do tipo array NumPy?
type(arr)

# E finalmente vamos imprimir o conteúdo de arr
print(arr)
```

Criando arrays com NumPy

- Podemos criar arrays multidimensionais com listas aninhadas.

```
import numpy as np

# Criamos uma lista aninhada (lista de listas)
data = [[1, 2, 3], [4, 5, 6]]

# Passamos a lista aninhada como parâmetro
arr_2d = np.array(data)

# O tipo de dado continua sendo um array NumPy
type(arr_2d)

# Imprimimos a variável arr_2d
print(arr_2d)
```

Criando arrays com NumPy

- Com o array multidimensional podemos começar a explorar alguns de seus atributos.

```
# O atributo ndim nos fornece o número de dimensões do array
arr_2d.ndim


# O atributo shape nos diz quantos elementos temos em cada dimensão
arr_2d.shape
```

Criando arrays com NumPy

- Você pode criar alguns arrays úteis com o NumPy
- Você pode criar uma matriz de zeros

```
# Criar um array com todos valores nulos
np.zeros(10)

# Para criar um array multidimensional de zeros
# Vamos passar uma tupla como parâmetro
np.zeros((3, 6))
```

Criando arrays com NumPy

- Você pode criar uma matriz de um's

```
# Podemos usar o ones para criar um array preenchido com 1

# Criar um array com todos valores 1
np.ones(5)

# Para criar um array multidimensional de 1's
# Vamos passar uma tupla como parâmetro
np.ones((4, 5))
```

Criando arrays com NumPy

- Você pode criar uma matriz de valores aleatórios

```
# Criar um array com valores aleatórios de uma distribuição uniforme
numeros = np.random.random(7, 4)
# Criar um array com valores inteiros aleatórios de uma distribuição uniforme
numeros = np.random.randint(7, 4)
# Criar um array com valores inteiros aleatórios de uma distribuição normal
numeros = np.random.randn(7, 4)
```

Criando arrays com NumPy

- Para gerar uma sequência, utilizamos o *arange*, equivalente ao ***range( )*** em Python.

```
# Vamos gerar agora uma array sequencial com 20 elementos?

np.arange(20)

# Saída: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

- Percebam que a função *arange*, assim como a *range*, inicia sua contagem em 0, e o número inserido como parâmetro não é incluso no array.

Tipo de dados em arrays

- O tipo de dado, ou *dtype*, é uma informação de metadados, ou seja, são dados sobre os dados.
- Ele nos informa qual é o tipo de dado armazenado em memória.

```
# Vamos criar um array e informá-lo que desejamos que o dado seja float...
arr1 = np.array([1, 2, 3], dtype = np.float64)

# ... e outro array, onde os mesmos dados serão armazenados como int
arr2 = np.array([1, 2, 3], dtype = np.int32)

# Vamos checar o tipo de dados de ambos os arrays?
arr1.dtype      # saída dtype('float64')
arr2.dtype      # dtype('int32')
```

Alterar o tipo de dado

- Se necessário, é possível alterar o tipo de dado, também conhecido como *casting*.

```
# Perceba que quando não definimos o tipo de dado...
arr = np.array([1, 2, 3])

# ... o NumPy tenta inferir o tipo, baseado nos valores de entrada
arr.dtype           # dtype('int32')

# Podemos alterar o tipo de dado com o método astype()
float_arr = arr.astype(np.float64)

# Que tal conferir se a mudança foi feita?
float_arr.dtype     # dtype('float64')
```

Alterar o tipo de dado

- **Muito cuidado ao alterar o tipo de dado!**
- **Você pode acabar alterando os dados, se não souber o que faz!**

```
# Podemos transformar qualquer tipo de dado, sabendo que...
arr = np.array([1.4, 3.6, -5.1, 9.42, 4.999999])

# ... quando transformamos de float para int...
arr.astype(np.int32)

# ... as casas decimais serão simplesmente ignoradas!
# E não se trata de arredondamento, e sim
# truncamento! As casas decimais serão ELIMINADAS! Atente-se!
arr.astype(np.int32)     # array([ 1,  3, -5,  9,  4])
```

Alterar o tipo de dado

- **E se o casting falhar?**
- **O NumPy nos informará uma bela mensagem de erro ValueError**

```
# Caso seja impossível alterar o tipo de dado...
numeric_str = np.array(['c', 'a', '3', '4'])

# ... o NumPy nos informará qual o primeiro elemento que falhou!
numeric_str.astype(float)
     ValueError: could not convert string to float: 'c'
```

Aritmética com arrays NumPy

Ao realizar operações aritméticas com arrays, não é necessário iterar pelos itens do elemento, como normalmente é feito com Python.

Aritmética com arrays NumPy

Qualquer operação entre arrays de mesmo tamanho faz a operação ser aplicada a todos os elementos.

- Esse conceito é chamado de vetorização.

```
# Vamos criar um array para realizar operações com ele
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Exemplos de operações com arrays
arr + arr
      array([[ 2,  4,  6],
             [ 8, 10, 12]])

arr + 10
      array([[11, 12, 13],
             [14, 15, 16]])

arr * arr
      array([[ 1,  4,  9],
             [16, 25, 36]])

1 / arr
      array([[1.        , 0.5       , 0.33333333],
             [0.25      , 0.2       , 0.16666667]])
```

Aritmética com arrays NumPy

- Lembra-se das funções para criar matrizes de 0's e 1's ?
- E se você quiser criar uma matriz em que todos os valores são iguais a 5?
- Você pode fazer isso de duas formas diferentes!

```
# Criar um array com todos valores 0's e somar 5
np.zeros(10) + 5

# Criar um array com todos valores 1's e multiplicar por 5
np.ones(10) * 5
```

Aritmética com arrays NumPy

- Podemos comparar arrays.
- A comparação nos retornará arrays booleanos.

```
# Criando um array para comparação
arr2 = np.array([[4, 1, 7], [6, 2, -5]])

# Comparando arrays de mesmo tamanho
arr2 > arr
     array([[ True, False,  True],
          [ True, False, False]])
```

Iterando com elementos 1D

- A iteração com elementos 1D em NumPy é feita elemento por elemento

```
import numpy as np

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

for elemento in arr:
     print(elemento)
```

Iterando com elementos 2D

A iteração com elementos 2D em NumPy é feita linha a linha

```
import numpy as np

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

for linha in arr:
     print(linha)
```

Para iterar com os elementos é preciso de 2 laços aninhados

```
import numpy as np

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

for linha in arr:
  for elemento in linha:
     print(elemento)
```

Iterando com elementos 3D

- A iteração com elementos 3D em NumPy é feita matriz por matriz

```
import numpy as np

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

for matriz in arr:
     print(matriz)
```

- Para iterar com os elementos é preciso de 3 laços aninhados

```
import numpy as np

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

for matriz in arr:
  for linha in matriz:
     for elemento in linha:
          print(elemento)
```

Iterando com nditer( )

Você pode iterar com elementos, independentemente da dimensão, como se fosse um vetor 1D

```
import numpy as np

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

for x in np.nditer(arr):
     print(x)
```

Você também pode utilizar *nditer*, ou iteração comum combinados com fatiamento, e simplificar seu acesso à linhas e elementos específicos

```
import numpy as np

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

for x in np.nditer(arr[:, ::2]):
     print(x)
```

Indexação e Fatiamento (Slicing)

- Basicamente trata-se da seleção de uma parte de um array.
  - Mas, lembre-se: A indexação em Python se inicia pelo 0. Assim, o primeiro elemento do array tem índice 0.
- Há diversas formas de se realizar essa operação.

```
# Vamos gerar um array para trabalharmos com fatiamento
arr = np.arange(10)  # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Aqui vamos filtrar apenas o sexto elemento da lista (índice 5)
print(arr[5])   # 5

# Aqui vamos fatiar nosso array para nos retornar os elementos de índice 5, 6 e 7 do
array.
# Na sintaxe seguinte, o elemento à direita dos : é exclusivo, ele não entra no filtro!
# Retorna os elementos de índices 5, 6 e 7. 8 está excluído!
print(arr[5:8])      # array([5, 6, 7])

# Quando atribuímos um valor à um fatiamento de array
# Ele substitui os valores originais, como podemos ver no exemplo abaixo
arr[5:8] = 12   # array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Indexação e Fatiamento (Slicing)

- É importante salientar que toda e qualquer alteração feita no array será refletida no original, e não será feito uma cópia dele, como acontece nas listas em Python.

```
# Vamos pegar uma fatia de arr
arr_slice = arr[5:8]
print(arr_slice)    # array([12, 12, 12])

# Agora vamos fazer uma alteração na fatia que coletamos...
arr_slice[1] = 123456

# ... e percebemos que a alteração se reflete no array original.
print(arr)          # array([0, 1, 2, 3, 4, 12, 123456, 12, 8, 9])
```

Indexação e Fatiamento (Slicing)

- Podemos coletar um array completo com a seguinte sintaxe

```
# Este comando retorna todos os valores do array
arr[:]
```

- Para fazer um fatiamento a partir de um índice específico até o final ou tudo até o índice específico, utilizamos a sintaxe abaixo.

```
# Aqui, selecionamos todos os elementos a partir do índice 2, incluindo o índice 2.
arr[2:]        # array([2, 3, 4, 12, 123456, 12, 8, 9])

# Já para selecionarmos tudo até o índice 4, excluindo o índice...
arr[:4]        # array([0, 1, 2, 3])
```

Indexação e Fatiamento (Slicing)

- Vamos agora para o fatiamento de arrays com mais dimensões.

```
# Vamos criar um array com 2 dimensões
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d)
     array([[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]])

# Podemos acessar cada dimensão da seguinte forma
print(arr2d[2])
     array([7, 8, 9])

# E cada elemento de cada dimensão utilizando ambas as sintaxes abaixo
# Selecionamos o elemento com índice 2, e desse elemento o sub-elemento com índice 1.
print(arr2d[2][1])
     8

# A sintaxe acima é equivalente a esta abaixo
print(arr2d[2,1])
     8
```

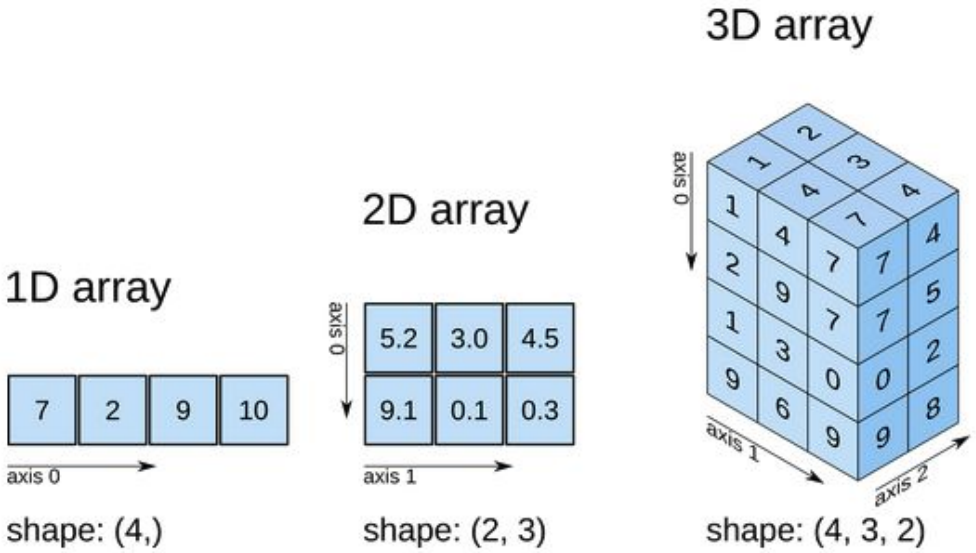
Indexação e Fatiamento (Slicing)

```
# Criando um array com 3 dimensões
arr3d = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
print(arr3d)
     array([[[ 1,  2,  3],
             [ 4,  5,  6]],
            [[ 7,  8,  9],
             [10, 11, 12]]])

# Selecionando o primeiro elemento (índice 0).
print(arr3d[0])
     array([[1, 2, 3],
            [4, 5, 6]])

# Selecionando o elemento com índice 1 e o sub-elemento com índice 0.
print(arr3d[1, 0])
     array([7, 8, 9])

# Da mesma forma, selecionamos o elemento final com os seguintes índices
print(arr3d[1, 0, 2])
     9
```

O fatiamento de arrays com mais de 2 dimensões segue a mesma lógica.

Indexação e Fatiamento (Slicing)

- Para facilitar a visualização, observe as imagens abaixo. As dimensões no NumPy também podem ser chamadas de *eixos (axis)*.



Indexação e Fatiamento (Slicing)

- Abaixo mostro uma sintaxe resumida e completa sobre a indexação.
- Percebam que os dois-pontos separam os índices iniciais e finais de uma dimensão, ou eixo, enquanto que a vírgula separa as dimensões, ou eixos.

```
nome_array[
     indice_inicial_dim_0:indice_final_dim_0,
     indice_inicial_dim_1:indice_final_dim_1,
     ...,
     indice_inicial_dim_N:indice_final_dim_N
]
```

Indexação e Fatiamento (Slicing)

```
nomes = np.array(["Maria", "Joaquim", "José", "João", "Bob", "Bob", "Antônio"])
numeros = np.random.randn(7, 4)

print(nomes)
     array(['Maria', 'Joaquim', 'José', 'João', 'Bob', 'Bob', 'Antônio'], dtype='<U7')

print(numeros)
     array([[ 0.08332437,  0.74508302,  0.29377823, -0.40309144],
            [-0.00657249, -0.37183682,  1.65978568,  0.24354861],
            [ 0.92646306, -1.60207788, -0.70337059,  0.04696397],
            [ 0.64716676, -0.63640023, -0.83317346, -1.1008962 ],
            [ 0.37581831, -1.20021293,  0.73862489, -0.76597305],
            [-1.49527206,  0.72298234,  1.59186498, -1.04811938],
            [ 0.36767802, -0.43538573, -0.87275053, -0.15102263]])

# Vamos gerar um array booleano de acordo com a condição
nomes == "Joaquim"
     array([False,  True, False, False, False, False, False])

# Podemos passar o array booleano resultante para filtar outro array.
# Percebam que somente serão trazidos os resultados
# cujo array booleano acima for igual a True (segundo elemento, índice 1)
numeros[nomes == "Joaquim"]
     array([[-0.00657249, -0.37183682,  1.65978568,  0.24354861]])
```

Também é possível utilizar arrays booleanos para filtrar arrays.

Indexação e Fatiamento (Slicing)

- Podemos inverter um array booleano, ou negar uma condição.

```
# Podemos utilizar o operador !=
nomes != "Bob"
     array([ True,  True,  True,  True, False, False,  True])

# ... ou negar a condição com o símbolo ~
# Ambas operações são equivalentes.
~(nomes == "Bob")
     array([ True,  True,  True,  True, False, False,  True])
```

Indexação e Fatiamento (Slicing)

- Para associarmos várias condições, podemos utilizar os operadores aritméticos booleanos & (E) e | (OU).

```
# O booleano será True caso o nome seja Maria ou Bob.
condicao_dupla_ou = (nomes == "Maria") | (nomes == "Bob")

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

Indexação e Fatiamento (Slicing)

- Usando operações booleanas para efetuar substituições

```
# Para substituirmos todos os valores de um array menores que zero por zero
numeros[numeros < 0] = 0

print(numeros)
    array([[0.08332437,    0.74508302, 0.29377823,    0.          ],
           [0.,            0.,           1.65978568,   0.24354861  ],
           [0.92646306,    0.,           0.,           0.04696397  ],
           [0.64716676,    0.,           0.,           0.          ],
           [0.37581831,    0.,           0.73862489,   0.          ],
           [0.,            0.72298234,   1.59186498,   0.          ],
           [0.36767802,    0.,           0.,           0.          ]])
```

Rearranjo e Transposição dos eixos dos arrays

- O NumPy oferece funções para rearranjar e transpor matrizes

```
# Aqui, criamos um array sequencial de 0 a 14
# e, então, transformamos esse array para uma matriz com 3 linhas e 5 colunas
arr = np.arange(15)
arr = arr.reshape((3, 5))
print(arr)
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
# Vamos agora calcular a matriz transposta desse array com o atributo .T
print(arr.T)
```

```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

Rearranjo e Transposição dos eixos dos arrays

Para arrays de dimensões maiores, há outra função chamada transpose que é mais versátil para as transformações.

```
arr = np.arange(16).reshape((2, 2, 4))
print(arr)
     array([[[ 0,  1,  2,  3],
             [ 4,  5,  6,  7]],
             [[ 8,  9, 10, 11],
             [12, 13, 14, 15]]])

# Vamos rearranjar as linhas, colocando-as na ordem desejada. Aqui, trocamos de posição os eixos 0 e 1 do array. O eixo 2 permaneceu no seu lugar.
arr.transpose((1, 0, 2))
     array([[[ 0,  1,  2,  3],
             [ 8,  9, 10, 11]],
             [[ 4,  5,  6,  7],
             [12, 13, 14, 15]]])
```

Rearranjo e Transposição dos eixos dos arrays

Uma forma mais simples para trocar eixos de lugar é com a função ***swapaxes***

```
print(arr)
     array([[[ 0,  1,  2,  3],
             [ 4,  5,  6,  7]],
             [[ 8,  9, 10, 11],
             [12, 13, 14, 15]]])

# Vamos transpor os eixos 1 e 2 do nosso array.
arr.swapaxes(1, 2)
    array([[[ 0,  4],
             [ 1,  5],
             [ 2,  6],
             [ 3,  7]],
            [[ 8, 12],
             [ 9, 13],
             [10, 14],
             [11, 15]]])
```

Funções úteis

O NumPy oferece uma série de funções úteis que podem ser utilizadas em toda a matriz, como:

- Raiz Quadrada - np.sqrt(array)
  - Função sqrt aplica a raiz quadrada a todos os elementos do array
- Exponencial - np.exp(array)
  - Função exp aplica função exponencial para todos os elementos do array
- Busca - np.where(<condição sobre o array>)
- Funções estatísticas
  - Média - array.mean(<eixo>)
  - Soma - array.sum(<eixo>)
  - Soma Acumulada - array.cumsum(<eixo>)
  - Produto Acumulado - array.cumprod(<eixo>)
- Ordenação - array.sort( )

Lendo arquivos texto e csv

- Para ler arquivos texto e CSV você pode usar o método loadtxt do numpy

```
import numpy as np
data = np.loadtxt("./caminho/arquivo.txt")
```

- Definindo um delimitador

```
import numpy as np
data = np.loadtxt("./caminho/arquivo.txt", delimiter = ",")
```

Lendo arquivos texto e csv

- Também é possível ler arquivos texto e CSV com o método genfromtxt

```
import numpy as np

data = np.genfromtxt("./caminho/arquivo.txt", dtype='str')
```

- Definindo um delimitador

```
import numpy as np

data = np.genfromtxt("./caminho/arquivo.txt", dtype='str', delimiter = ",")
```

Referências

Manual do NumPy: NumPy User Guide