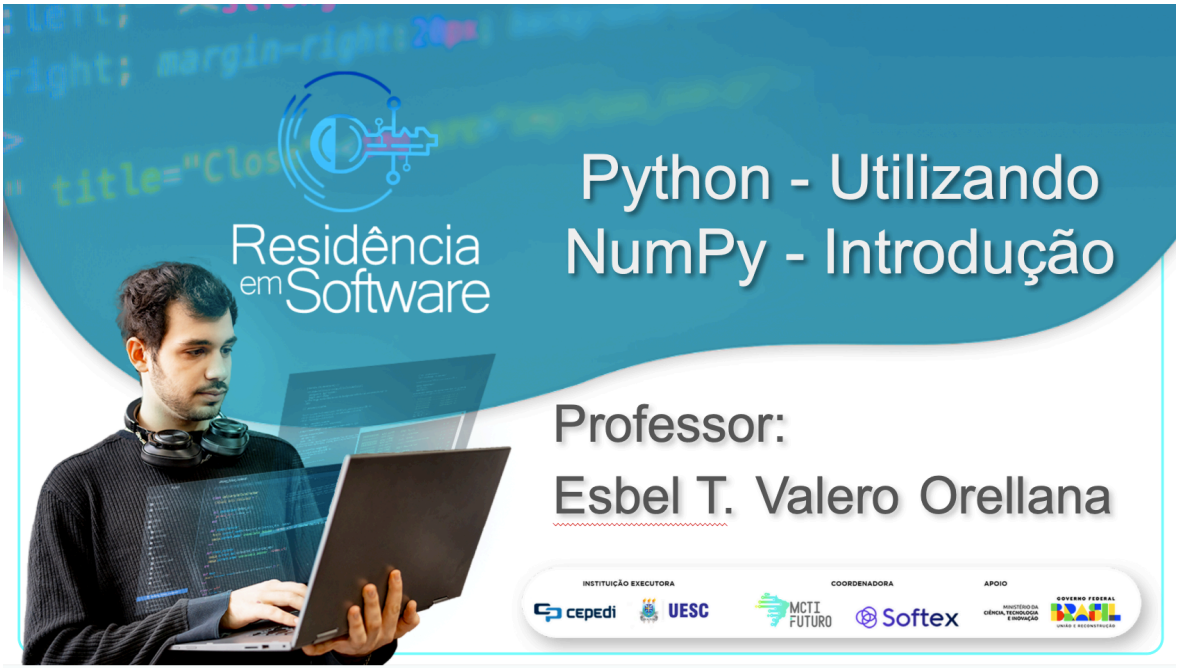
Módulo de Programação Python

Trilha Python - Aula 9: Utilizando NumPy - Introdução



Objetivo: Trabalhar com pacotes e módulos disponíveis em python: **Numpy**. Trabalhar os atributos e métodos disponibilizados pelas classes do **NumPy**. Apresentar as diferentes formas de acessar matrizes e vetores criados com **NumPy**.

A manipulação de grandes volumes de dados em **Python** é quase sinônimo de manipulação de **Numpy** arrays. Até mesmo ferramentas mais recentes, como **Pandas** que será abordado em próximas aulas, são construídas em torno dos *ndarrys*.

Na aula de hoje discutiremos vários exemplos de uso da manipulação de NumPy arrays para acessar dados e submatrizes e para dividir, remodelar e unir as matrizes.

Embora os tipos de operações mostrados aqui possam parecer um pouco pouco úteis, eles constituem os blocos de construção de muitos outros exemplos que usaremos nas próximas aulas.

Abordaremos alguns tópicos importantes nesta aula:

- Atributos de arrays: determinação do tamanho, formato, consumo de memória e tipos de dados de arrays;
- Indexação de arrays: obtendo e definindo os valores de elementos individuais do array;
- Slicing de matrizes: obtendo e configurando submatrizes menores dentro de uma matriz maior;
- Remodelagem de arrays: alterando a forma de um determinado array

Atributos dos ndarray

As funções utilizadas para criar os *ndarray*, que utilizamos até aqui criam objetos da classe e permitem que exploremos alguns atributos dos *ndarray*. Veja estes exemplos

In [1]:
```
import numpy as np
#Podemos definir a semente do gerador de números aleatórios
np.random.seed(123456789)
x_1 = np.random.randint(10, size=8) # array 1D
x_2 = np.random.randint(10, size=(2,4)) # array 2D
x_3 = np.random.randint(10, size=(2,2,2)) # array 3D
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
```

Os três arrays que foram criados, apesar de terem formas diferentes, tem a mesma quantidade de elementos: 8. Veja que o parâmetro `size` espera uma tupla que pode conter apenas um valor (8), uma dupla (`(2,4)`) ou uma tripla (`(2,2,2)`). Este parâmetro então está definindo a "forma" do *ndarray*. Veja como estas características de cada array estão disponíveis na forma de atributos dos objetos.

In [2]:
```
#Podemos avaliar a dimensão, a forma e o tamanho de cada um
print("x_1 ndim: ", x_1.ndim)
print("x_1 shape: ", x_1.shape)
print("x_1 size: ", x_1.size)
```

```
x_1 ndim:  1
x_1 shape:  (8,)
x_1 size:  8
```

In [3]:
```
print("x_2 ndim: ", x_2.ndim)
print("x_2 shape: ", x_2.shape)
print("x_2 size: ", x_2.size)
```

```
x_2 ndim:  2
x_2 shape:  (2, 4)
x_2 size:  8
```

In [4]:
```
print("x_3 ndim: ", x_3.ndim)
print("x_3 shape: ", x_3.shape)
print("x_3 size: ", x_3.size)
```

```
x_3 ndim:  3
x_3 shape:  (2, 2, 2)
x_3 size:  8
```

Com os atributos `ndim` , dimensão do *ndarray*, `shape` , forma do *ndarray*, e `size` , quantidade de elementos, podemos ter uma descrição das principais características dos arrys que estão sendo gerados.

Cada *ndarray* possui ainda outros dois atributos importantes:

In [5]:
```
# tamanho em bytes de cada elemento
print("x_3 itemsize: ", x_3.itemsize, "bytes")
# tamanho em bytes do array
print("x_3 nbytes: ", x_3.nbytes, "bytes")
```

```
x_3 itemsize:  8 bytes
x_3 nbytes:  64 bytes
```

Indexação dos *ndarray*

Para acessar os elementos de um *ndarray* utilizamos a mesma sintaxes de indexação das listas.

- Os índices começando em 0 e indo até tamanho menos um;
- Podendo também utilizar índices negativos para acessar o array começando pelo final;
- Os array multidimensionais podem ser acessados utilizando tuplas de índices;

In [6]:
```
x = np.array([1,2,3,4,5,6,7,8,9,10])
for i in range(len(x)):
    print(x[i], end=' ')
print()
# ou
for i in x:
    print(i, end=' ')
print()

print("x[-1]: ", x[-1])
print(x[::-1])  # inverte o array

matA = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]

print("matA[0][0]: ", matA[0][0])
```

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
x[-1]:  10
[10  9  8  7  6  5  4  3  2  1]
matA[0][0]:  1.0
```

Os *ndarray* são tipos mutáveis pelo que seus elementos podem ser modificados quando acessados pelo seu índice. Deve-se levar em consideração que, ao contrário das listas, os *ndarray* tem um tipo fixo.

In [7]:
```
x[0] = 11
print(x)
x[1] = 3.14  # conversão automática de tipos, será truncado!
print(x)
matA[0][0] = 3.14
print(matA)
#ou
#matA[1, 0] = 3.14
#print(matA)
try:
    x[0] = "Hello"  # erro de tipo!
except Exception as e:
    print(e)
```

```
[11  2  3  4  5  6  7  8  9 10]
[11  3  3  4  5  6  7  8  9 10]
[[3.14, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
invalid literal for int() with base 10: 'Hello'
```

Slicing de matrizes

O mecanismo de *slicing*, utilizado em listas e outros tipos indexáveis, também pode ser utilizado com os *ndarray*. Um subconjunto de elementos pode ser acessado utilizando x[inicio : fim : passo]. Se algum dos elementos não for especificado, seu valor padrão é: inicio = 0, fim = tamanho, passo = 1.

In [8]:
```
matA = np.random.randint(10, size=(5,5))
print(matA)
print("matA[1:3,1:4]")
print(matA[1:3,1:4])
print("matA[1:3,:]")
print(matA[1:3,:])
```

```
[[3 8 3 9 0]
 [6 9 2 7 2]
 [8 8 7 4 2]
 [4 0 6 9 5]
 [3 5 8 1 7]]
matA[1:3,1:4]
[[9 2 7]
 [8 7 4]]
matA[1:3,:]
[[6 9 2 7 2]
 [8 8 7 4 2]]
```

O exemplo anterior mostra que quando trabalhamos com conjuntos de dados muito grandes, podemos acessar e processar pequenos subconjuntos do mesmo sem necessidade de criarmos uma cópia. Veja que o subconjunto especificado não cria uma nova matriz.

In [9]:

```
print("matA\n", matA)
subA = matA[:2,:2]
print("subA\n", subA)
subA[0,0] = 99
print("subA\n", subA)
print("matA\n", matA)
```

```
matA
 [[3 8 3 9 0]
 [6 9 2 7 2]
 [8 8 7 4 2]
 [4 0 6 9 5]
 [3 5 8 1 7]]
subA
 [[3 8]
 [6 9]]
subA
 [[99  8]
 [ 6  9]]
matA
 [[99  8  3  9  0]
 [ 6  9  2  7  2]
 [ 8  8  7  4  2]
 [ 4  0  6  9  5]
 [ 3  5  8  1  7]]
```

Se estivermos interessados em trabalhar com uma cópia da matriz original, ou de uma submatriz extraída dela, preservando conjunto original de dados, podemos utilizar o método `copy()` .

In [10]:
```
print("matA\n", matA)
subA = matA[:2,:2].copy()
print("subA\n", subA)
subA[0,0] = 0
print("subA\n", subA)
print("matA\n", matA)
```

```
matA
 [[99  8  3  9  0]
 [ 6  9  2  7  2]
 [ 8  8  7  4  2]
 [ 4  0  6  9  5]
 [ 3  5  8  1  7]]
subA
 [[99  8]
 [ 6  9]]
subA
 [[0 8]
 [6 9]]
matA
 [[99  8  3  9  0]
 [ 6  9  2  7  2]
 [ 8  8  7  4  2]
 [ 4  0  6  9  5]
 [ 3  5  8  1  7]]
```

Uma rotina comumente necessária quando se trabalha com matrizes é acessar uma linha ou coluna específica. Isso pode ser feito combinando indexação e *slicing*, usando uma slice vazia marcada por dois pontos (:).

In [11]:
```
print("matA\n", matA)
print("Primeira linha de matA: ")
print(matA[0,:])
print("Segunda coluna de matA: ")
print(matA[:,1])

#  no caso das linhas também é possível utilizar indexação simples

print("Primeira linha de matA: ")
print(matA[0])
```

```
matA
 [[99  8  3  9  0]
 [ 6  9  2  7  2]
 [ 8  8  7  4  2]
 [ 4  0  6  9  5]
 [ 3  5  8  1  7]]
Primeira linha de matA:
[99  8  3  9  0]
Segunda coluna de matA:
[8 9 8 0 5]
Primeira linha de matA:
[99  8  3  9  0]
```

Remodelando de matrizes: *reshape*

Outro tipo útil de operação é a remodelagem de arrays, que pode ser feita com o método `reshape` . Por exemplo, se quiser colocar os números de 1 a 9 em uma grade 3×3, você pode fazer o seguinte:

In [12]:
```
grid = np.arange(1, 10).reshape(3, 3)
print(grid)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Vejamos novamente aquele primeiro exemplo da aula

In [13]:
```
x_1 = np.random.randint(10, size=8) # array 1D
print(x_1)
```

```
[7 0 3 7 6 3 5 0]
```

Ele pode ser remodelado para uma matriz de 2×4 ...

In [14]:
```
x_2 = x_1.reshape(2, 4) # x_2 é uma outra forma de ver x_1
print(x_1)              # x_1 continua sendo o mesmo
print(x_2)              # x_2 é uma nova matriz
x_2[0,0] = 99           # mas se ateramos x_2
print(x_2)
print(x_1)              # x_1 também é alterado!
```

```
[7 0 3 7 6 3 5 0]
[[7 0 3 7]
 [6 3 5 0]]
[[99  0  3  7]
 [ 6  3  5  0]]
[99  0  3  7  6  3  5  0]
```

In [15]:
```
# veja outras formas de ver a mesma matriz
print("4x2")
print(x_1.reshape(4, 2))
print("2x2x2")
print(x_1.reshape(2, 2, 2))
```

```
4x2
[[99  0]
 [ 3  7]
 [ 6  3]
 [ 5  0]]
2x2x2
[[[99  0]
  [ 3  7]]

 [[ 6  3]
  [ 5  0]]]
```

Para finalizar vamos retomar um exemplo que começamos a testar com listas na aula anterior: Multiplicação de Matrizes. Veja que agora podemos criar as matrizes como *ndarrays* e forma simples.

In [16]:
```
#matA = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
matA = np.arange(1,10).reshape(3,3)
#matB = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
matB = np.eye(3)
#matC = [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0., 0., 0.0]]
matC = np.zeros((3,3))

def GEMM(alpha, A, B, beta, C):
    for i in range(len(A)):
        for j in range(len(B[0])):
            C[i][j] *= beta
            for k in range(len(A[0])):
                C[i][j] += alpha * A[i][k] * B[k][j]
    return C

matC = GEMM(1.0, matA, matB, 0.0, matC)
print(matC)
```

```
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

In [17]:
```
matSize = 512
#matA = [[random() for i in range(matSize)] for j in range(matSize)
matA = np.random.random((matSize, matSize))
#matB = [[random() for i in range(matSize)] for j in range(matSize)
matB = np.random.random((matSize, matSize))
#matC = [[random() for i in range(matSize)] for j in range(matSize)
matC = np.random.random((matSize, matSize))
print(len(matA), len(matA[0]))
print(matA.shape, matB.shape, matC.shape)
print(matA.dtype, matB.dtype, matC.dtype)
print(matA.size, matB.size, matC.size)
```

```
512 512
(512, 512) (512, 512) (512, 512)
float64 float64 float64
262144 262144 262144
```

In [18]:
```
%timeit GEMM(2.0, matA, matB, 0.5, matC)
```

```
1min 33s ± 453 ms per loop (mean ± std. dev. of 7 runs, 1 loop eac
h)
```

In [ ]:
```

```