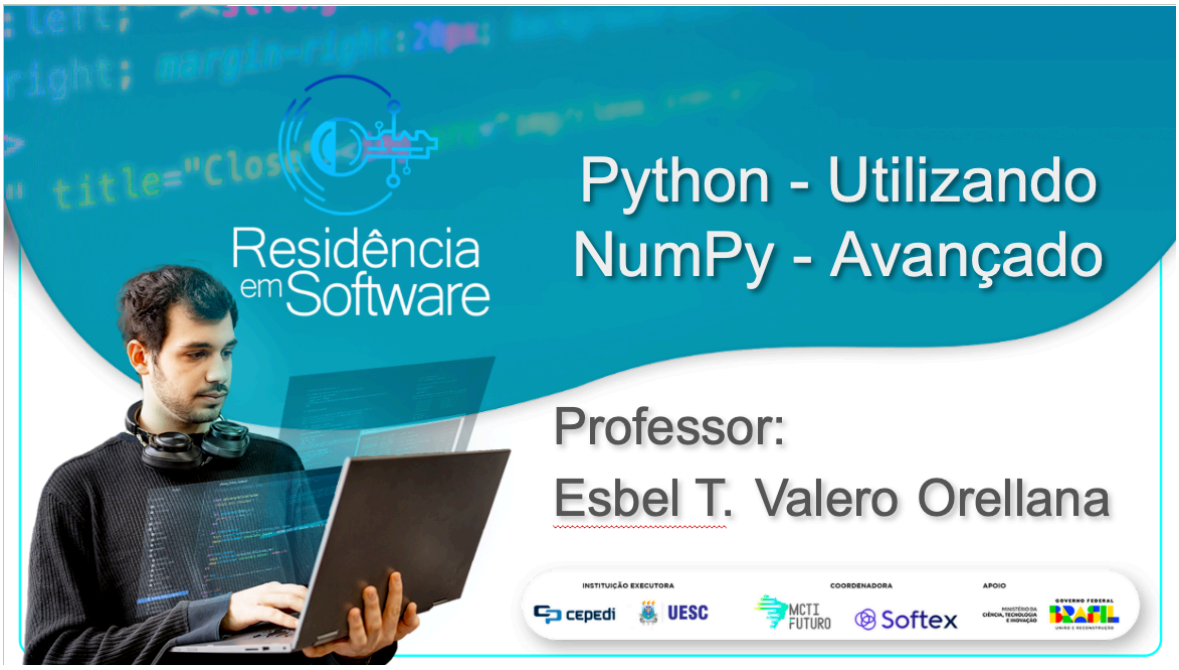
Módulo de Programação Python

Trilha Python - Aula 13: Utilizando NumPy - Avançado



Objetivo: Trabalhar com pacotes e módulos disponíveis em python: **Numpy**. Introduzir recursos avançados da **NumPy** para trabalhar com tipos diferentes de dados, subconjuntos e operações aritméticas e lógicas.

Broadcasting

Até aqui vimos como as *ufunc* do **NumPy** podem ser usadas para vetorizar operações e, assim, substituir o uso de estruturas de repetição lentas.

Outro meio de vetorizar operações é usar a funcionalidade de broadcasting do **NumPy**.

Broadcasting é simplesmente um conjunto de regras para aplicar *ufuncs* binários (por exemplo, adição, subtração, multiplicação, etc.) em matrizes de tamanhos diferentes.

Já vimos que as *unfuncs* binárias podem operar com facilidade e eficiência em *ndarrays* do mesmo tamanho.

Podemos reproduzir um dos exemplos da aula anterior.

In [1]:
```
import numpy as np
```

```
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.
```

In [2]:
```
# Adição de arrays
x = np.array([1,2,3])
y = np.array([4,5,6])
z = x + y
print(z)
```

```
[5 7 9]
```

Também vimos que este tipo de operação pode ser utilizada para somar um escalar com um *ndarray*.

In [3]:
```
# Adição de array e escalar
z = x + 10
print(z)
```

```
[11 12 13]
```

Podemos pensar este exemplo como uma operação que trata o escalar como um *ndarray* do tipo `[10, 10, 10]`, e adiciona com o outro operando, neste caso outro *ndarray* do mesmo tamanho.

A vantagem de utilizar broadcasting do **NumPy** é que, de fato, o tal *ndarray* com elementos repetidos não é criado, simplificando o procedimento e otimizando o uso de memória.

O mecanismo de broadcasting pode ser aplicado em para *ndarrays* de dimensão superior.

In [4]:
```
A = np.array([[1,2,3],[4,5,6]])
print("A = \n", A)
print("A.shape = ", A.shape)
print("x = \n", x)
print("x.shape = ", x.shape)
B = A + x
print("B = A + x = \n", B)
```

```
A =
 [[1 2 3]
 [4 5 6]]
A.shape =  (2, 3)
x =
 [1 2 3]
x.shape =  (3,)
B = A + x =
 [[2 4 6]
 [5 7 9]]
```

Vejamos um exemplo mais complexo

In [5]:
```
#a = np.array([0, 1, 2])
a = np.arange(3)
#b = np.array([[0], [1], [2]])
b = np.arange(3).reshape((3,1))
print(a)
print(b)
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

In [6]:
```
print("a + b = \n", a + b)
```

```
a + b =
 [[0 1 2]
 [1 2 3]
 [2 3 4]]
```

Para entender melhor como o broadcasting funciona podemos colocar as operações em função de um conjunto de regras que definem como acontece a interação entre as duas matrizes:

Regra 1: Se as duas matrizes diferirem no número de dimensões, a forma (*shape*) daquela com menos dimensões será preenchida com uns no lado esquerdo até igualar o número de dimensões das duas matrizes.

In [7]:
```
A = np.ones((2,3))
x = np.arange(3)
```

In [8]:
```
print("A.shape = ", A.shape)
print("x.shape = ", x.shape)
```

```
A.shape =  (2, 3)
x.shape =  (3,)
```

In [9]:
```
print("A = \n", A)
print("x = \n", x)
```

```
A =
 [[1. 1. 1.]
 [1. 1. 1.]]
x =
 [0 1 2]
```

No exemplo anterior o array `x` tem apenas uma dimensão enquanto a matriz `A` tem duas. Neste caso se adiciona uma dimensão à esquerda de `x`, ou seja, se cria um array com uma linha e 3 colunas.

Regra 2: Se a forma dos dois arrays não corresponder em alguma dimensão, o array que na sua forma (*shape*) tiver dimensão igual a 1, naquela dimensão, é esticado para corresponder à forma do outro array.

Veja no exemplo anterior que, após aplicar a **Regra 1** o array `x` agora tem *shape* `(1,3)`, enquanto que a matriz `A` tem *shape* `(2,3)`. Neste caso o se cria mais uma linha em `x`, repetindo a primeira para que os dois arrays fiquem do mesmo tamanho. O resultado será então.

In [10]:
```
print("A + x = \n", A + x)
```

```
A + x =
 [[1. 2. 3.]
 [1. 2. 3.]]
```

Veja outro exemplo.

In [11]:
```
A = np.arange(3).reshape((3, 1))
x = np.arange(3)
```

In [12]:
```
print("A.shape = ", A.shape)
print("x.shape = ", x.shape)
```

```
A.shape =  (3, 1)
x.shape =  (3,)
```

Veja que neste caso, novamente as dimensões das matrizes não coincidem. Novamente se adiciona uma dimensão a `x` que pasa a ter *shape* `(1,3)`. Como a *shape* de `A` é `(3,1)`, se *esticam* as colunas de `x` apara preencher as duas novas linhas e se se esticam as linhas de `A` para preencher duas novas colunas. Desta forma se opera com duas matrizes de `(3,3)`.

In [13]:
```
print("A = \n", A)
print("x = \n", x)
print("A + x = \n", A + x)
```

```
A =
 [[0]
 [1]
 [2]]
x =
 [0 1 2]
A + x =
 [[0 1 2]
 [1 2 3]
 [2 3 4]]
```

Regra 3: Se em qualquer dimensão os tamanhos discordam e nenhum deles for igual a 1, um erro é lançado.

Ou seja:

In [14]:
```
A = np.ones((3, 2))
x = np.arange(3)
```

In [15]:
```
print("A.shape = ", A.shape)
print("x.shape = ", x.shape)
```

```
A.shape =  (3, 2)
x.shape =  (3,)
```

In [16]:
```
print("A = \n", A)
print("x = \n", x)
try:
    print("A + x = \n", A + x)
except ValueError as e:
    print("ValueError:", e)
```

```
A =
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
x =
 [0 1 2]
ValueError: operands could not be broadcast together with shapes (
3,2) (3,)
```

Outras *unfuc* : Operações de comparação.

NumPy também implementa operadores de comparação como `<` (menor que) e `>` (maior que) elemento a elemento, como *ufuncs* . O resultado desses operadores de comparação é sempre um array com tipo de dados *booleano*. Todas as seis operações de comparação padrão estão disponíveis:

Operador	*ufunc*
==	np.equal
<	np.less
>	np.greater
!=	np.not_equal
<=	np.less_equal
>=	np.greater_equal

Podemos então fazer a seguinte análise:

- Suponha que obtemos, de alguma fonte, as notas em cada uma das três avaliações de um curso, assim como a media final, de uma turma de alunos.

In [17]:
```
# Vamos gerar as notas de forma aleatória para 30 alunos
notas = np.zeros((30, 4))
notas[:, :3] = np.round(np.random.uniform(4, 10, size=(30, 3)),1)
notas[:, 3] = np.round(np.mean(notas[:, :3], axis=1)) # axis=1 indi
```

In [18]:
```
print(notas[:5, :])
print(" ... ")
print(notas[-5:, :])
```

```
[[4.8 6.4 8.  6. ]
 [5.9 8.  5.2 6. ]
 [6.6 4.4 7.  6. ]
 [7.  4.1 7.3 6. ]
 [4.3 9.5 4.5 6. ]]
 ...
[[8.4 7.6 5.4 7. ]
 [5.5 6.1 5.2 6. ]
 [5.  5.8 5.8 6. ]
 [7.2 4.  6.1 6. ]
 [7.2 4.2 8.5 7. ]]
```

Agora queremos saber quantos alunos tiveram nota maior ou igual a 7,0 na primeira avaliação.

In [19]:
```
aprov1raAva = notas[:,0] >= 7
print("aprov1raAva = \n", aprov1raAva)
```

```
aprov1raAva =
 [False False False  True False False False  True  True  True Fals
e  True
 False False  True  True  True False False False False  True  True
True
  True  True False False  True  True]
```

In [20]:
```
#nAprov = np.sum(aprov1raAva)
#ou
#nAprov = np.count_nonzero(aprov1raAva)
#ou
nAprov = aprov1raAva.sum()
print(nAprov, "alunos foram aprovados na primeira avaliação")
```

```
15 alunos foram aprovados na primeira avaliação
```

Podemos identificar quantos alunos estão abaixo da média da turma na segunda avaliação.

In [21]:
```
mean2daAval = np.mean(notas[:,1])
print("A média da segunda avaliação foi", mean2daAval)
nAbaiMedia = np.sum(notas[:,1] < mean2daAval)
print(nAbaiMedia, "alunos ficaram abaixo da média na segunda avalia
```

```
A média da segunda avaliação foi 7.216666666666666
13 alunos ficaram abaixo da média na segunda avaliação
```

Podemos identificar quantas notas, nas três avaliações, foram maiores que 9,0

In [22]:
```
notasMaiores = notas[:,:3] > 9
print(notasMaiores[:5, :])
print("...")
print(notasMaiores[-5:, :])
```

```
[[False False False]
 [False False False]
 [False False False]
 [False False False]
 [False  True False]]
...
[[False False False]
 [False False False]
 [False False False]
 [False False False]
 [False False False]]
```

In [23]:
```
quantMaiores = np.sum(notasMaiores)
print(quantMaiores, "notas maiores que 9")
```

```
16 notas maiores que 9
```

Ainda podemos perguntar se tem alguma nota, na média final, menor que 5,0

In [24]:
```
np.any(notas[:, 3] < 5)
```

Out[24]:
```
False
```

Ou se todas as notas da média final são menores que 9,0

In [25]:
```
np.all(notas[:, 3] < 9)
```

Out[25]:
```
False
```

Ou se um aluno específico teve alguma nota abaixo de 6,0

In [26]:
```
np.any(notas[:, 3] < 5, axis=0)
```

Out[26]:
```
False
```

Operadores lógicos

Até aqui conseguimos contar a quantidade de alunos com nota menor ou maior ed um determinado valor. Os operadores relacionais entretanto podem ser utilizados, quando combinados com operadores lógicos, poara determinar, por exemplo, quantos alunos estão de prova final (media menor que 7,0) mas já tem media maior que 5. No módulo **NumPy** são impementados, via *unfunc* os operadores lógicos bit a bit do Python, &, |, ^ e ~. Veja a lista a seguir

Operador	*ufunc*
&	np.bitwise_and
^	np.bitwise_xor
|	np.bitwise_or
~	np.bitwise_not

In [27]:
```
notasInter = (notas[:, 3] < 7) & (notas[:, 3] > 5)
quantInter = np.count_nonzero(notasInter)
print(quantInter, "notas estão entre 5 e 7")
```

```
9 notas estão entre 5 e 7
```

Repare que foram utilizados parênteses ma expressão. Devido às regras de precedência de operadores, que discutimos no módulo anterior, a expressão geraria um erro.

Você sabe explica por que?

Utilizando mascaras

Os operadores relacionais geram um arrays de booleans. Estes arrays podem ser utilizados como máscaras para extrair subconjuntos de um arrays.

Por exemplo, se quisermos extrair quais as notas abaixo de 7 na terceira avaliação podemos utilizar uma expressão como a seguinte:

In [28]:
```
print(notas[:,2])
x = notas[:,2]
#notas3ra = notas[:, 2] [notas[:, 2] < 7]
notas3ra = x [x < 7]
print(notas3ra)
```

```
[8.  5.2 7.  7.3 4.5 9.9 9.2 9.9 6.2 8.9 6.2 8.5 8.6 9.1 7.2 8.2
8.4 9.7
 7.7 6.5 6.9 5.9 4.1 6.2 7.  5.4 5.2 5.8 6.1 8.5]
[5.2 4.5 6.2 6.2 6.5 6.9 5.9 4.1 6.2 5.4 5.2 5.8 6.1]
```

A expressão anterior retorna um array unidimensional preenchido com todos os valores que atendem à condição posta; em outras palavras, todos os valores nas posições nas quais o array de máscaras é `True`.

Aqui se fz necessário parar para discutir a diferença entre os operadores lógicos `and` e `or` e os operadores bit a bit `&` e `|`. Em que momento usar um ou outro?

Os operadores lógicos avaliam o objeto como um todo e não cada um dos valores que ele armazena. Já os operadores bit a bit avaliam a relação entre o contudo dos objetos.

Quando você usa operadores lógicos, significa que você quer que **Python** trate o objeto como uma única entidade booleana.

Tem outro recurso de indexação importante que permite utilizar um array de índice como se fosse uma máscara. Esta forma sofisticada de indexação consiste em passar um array de índices para acessar vários elementos do array de uma só vez.

Suponha que, na turma de alunos um grupo apresentou um trabalho extra, valendo 3 pontos adicionais na última nota. Temos a lista de índices dos alunos que apresentaram o trabalho.

In [29]:
```
trabalhoExtra = np.array([3, 6, 12, 17, 21, 23, 25, 28])
print(notas[:,2])
```

```
[8.  5.2 7.  7.3 4.5 9.9 9.2 9.9 6.2 8.9 6.2 8.5 8.6 9.1 7.2 8.2
8.4 9.7
 7.7 6.5 6.9 5.9 4.1 6.2 7.  5.4 5.2 5.8 6.1 8.5]
```

In [30]:
```
print(notas[trabalhoExtra,2])
```

```
[7.3 9.2 8.6 9.7 5.9 6.2 5.4 6.1]
```

In [31]:
```
notas[trabalhoExtra,2] += 3
nota3 = notas[:,2]
nota3[nota3 > 10] = 10
print(notas[:,2])
```

```
[ 8.   5.2  7.  10.   4.5  9.9 10.   9.9  6.2  8.9  6.2  8.5 10.
9.1
  7.2  8.2  8.4 10.   7.7  6.5  6.9  8.9  4.1  9.2  7.   8.4  5.2
5.8
  9.1  8.5]
```

Ordenação de *ndarrays*

Um dos recursos importantes que destacamos na implementação de listas em **Python** foi a implementação de funções que ordenam a lista, sempre que ela esteja formado por elementos que sejam comparáveis uns com os outros.

Os *ndarrys* são formados por tipos numéricos e, em muitos casos, pode ser necessário ordenar os elementos em ordem crescente ou decrescente.

Você provavelmente já pesquisou algoritmos de ordenação para resolver algumas tarefa, sobre tudo do módulo de Programação Imperativa.

insertion sorts, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*,

Algoritmos como o de **ordenação por inserção (insertion sorts)**, **ordenação por seleção (selection sorts)**, **ordenação por mesclagem (merge sorts)** , **ordenação quick sorts**, **ordenação da bolha(bubble sorts)**, são tradicionalmente estudados em cursos de estruturas de dados e algoritmos.

O algoritmos de **selection sorts**, por exemplo, procura repetidamente o valor mínimo de uma lista e faz trocas até que a lista seja classificada.

In [32]:
```
def selection_sort(x):
    for i in range(len(x) - 1):
        iTroca = i + np.argmin(x[i:])
        (x[i], x[iTroca]) = (x[iTroca], x[i])
    return x
```

In [33]:
```
x = np.random.randint(0, 30, 10)
print(x)
selection_sort(x)
print(x)
```

```
[29 15 27 28 28 27 19  8 12 13]
[ 8 12 13 15 19 27 27 28 28 29]
```

A ordenação por seleção não é dos algoritmos mais eficientes de ordenação, mas não vamos entrar em detalhes sobre complexidade de algoritmos neste curso,

Felizmente, **Python** disponibiliza algoritmos de ordenação que são muito mais eficientes. No **NumPy**, por exemplo, temos a função `sort`

In [34]:
```
x = np.random.randint(0, 30, 10)
print(x)
# Se se deseja preservar o array original sem ordenar
y = np.sort(x)
print(y)
print(x)
# Já se queremos ordenar o array original
x.sort()
print(x)
```

```
[ 5  9 13 26  0 12 16  7 21 12]
[ 0  5  7  9 12 12 13 16 21 26]
[ 5  9 13 26  0 12 16  7 21 12]
[ 0  5  7  9 12 12 13 16 21 26]
```

Uma variação do `sort` é o `argsort`, que retorna um array de índices que, quando usado como máscara, retorna o array ordenado.

In [35]:
```
x = np.random.randint(0, 30, 10)
print(x)
iOrd = np.argsort(x)
print(iOrd)
print(x[iOrd])
```

```
[ 8 15 10 13 23 12 23 21  7  3]
[9 8 0 2 5 3 1 7 4 6]
[ 3  7  8 10 12 13 15 21 23 23]
```