Ada Tech [DS-PY-004] Técnicas de Programação I (PY)

Apresentação

Rudiney Casali:

- Doutorado em Física (UFSC)
- Especialista em Dados (Mackenzie)
- Lead Senior Data Scientist (Kyndryl)
- linkedin.com/in/rcasali/

Requerimentos

- Todos aqui estão familiarizados com a ferramenta Jupyter Notebook?

- Você pode editar seu JN no browser, e usando anaconda. Usaremos o editor de texto Visual Studio Code (NNF).

- Os encontros com a língua inglesa serão inevitáveis daqui pra frente.

Conteúdos

- Aula 1 (Expositiva e Prática): Revisão GIT;
- Aulas 2 e 3 (Expositiva e Prática): Numpy;
- Aulas 4 e 5 (Expositiva e Prática): Pandas;
- Aulas 6, 7 e 8 (Prática): Trabalho em EDA;
- Aula 9 (Prática): Apresentação do Trabalho em EDA e Rubrica.

Aulas 2 e 3 | Numpy

O problema dos arranjos

Imagine que você precisa trabalhar com dados numéricos em arranjos multidimensionais, realizando uma variedade de rotinas para operações matemática em matrizes, incluindo, aritmética, lógica, manipulação de formas, classificação, seleção, transformadas discretas de Fourier, álgebra linear básica, operações estatísticas básicas e simulações aleatórias. Como organizar e tratar um volume grande de dados?

Numpy

Uma biblioteca de código aberto para a linguagem de programação Python, adicionando suporte para matrizes e arrays grandes e multidimensionais, juntamente com uma grande coleção de funções matemáticas de alto nível para operar nessas matrizes.

Numpy

- Objeto de matriz multidimensional,
- Objetos derivados (matrizes/matrizes mascaradas)
- Variedade de rotinas para operações rápidas em matrizes:
    - Matemática,
    - Lógica,
    - Manipulação de formas,
    - Classificação,
    - Seleção,
    - Transformadas discretas de Fourier,
    - Álgebra linear básica ,

  Operações estatísticas básicas,

  Simulação aleatória.

Numpy

O Objeto **ndarray** (nd-arranjo) contempla matrizes n-dimensionais de tipos de dados homogêneos, com muitas operações sendo executadas em código compilado para desempenho

- Tamanho fixo na criação, ao contrário das listas Python, que podem crescer dinamicamente.

1D

2D

1	5	2	0
8	3	6	1
1	7	2	9

Numpy

O Objeto **ndarray** contempla matrizes n-dimensionais de tipos de dados homogêneos, com muitas operações sendo executadas em código compilado para desempenho

- Tamanho fixo na criação, ao contrário das listas Python, que podem crescer dinamicamente.
- Alterar o tamanho de um ndarray criará um novo arranjo (**array**) e excluirá o original.

Numpy

O Objeto **ndarray** contempla matrizes n-dimensionais de tipos de dados homogêneos, com muitas operações sendo executadas em código compilado para desempenho

- Tamanho fixo na criação, ao contrário das listas Python, que podem crescer dinamicamente.
- Alterar o tamanho de um ndarray criará um novo arranjo (**array**) e excluirá o original.
- Elementos devem ser do mesmo tipo de dados, ocupando o mesmo espaço em memória.

Numpy

O Objeto **ndarray** contempla matrizes n-dimensionais de tipos de dados homogêneos, com muitas operações sendo executadas em código compilado para desempenho

- Tamanho fixo na criação, ao contrário das listas Python, que podem crescer dinamicamente.
- Alterar o tamanho de um ndarray criará um novo arranjo (**array**) e excluirá o original.
- Elementos devem ser do mesmo tipo de dados, ocupando o mesmo espaço em memória.
- A exceção: pode-se ter matrizes de objetos (Python, incluindo NumPy), permitindo assim matrizes de elementos de tamanhos diferentes.

Instalação do Numpy

- A instalação ocorre através do Conda ou pip:

```
$ conda install numpy
$ pip install numpy
```

Pré-requisitos para o Numpy

- Para se trabalhar melhor com os exemplos da biblioteca Numpy, é comum realizar a instalação da biblioteca de visualização de gráficos Matplotlib

Pré-requisitos para o Numpy

- Para se trabalhar melhor com os exemplos da biblioteca Numpy, é comum realizar a instalação da biblioteca de visualização de gráficos Matplotlib
- Faremos aqui um breve estudo dos arranjos (**arrays**) n-dimensionais ($n \geqslant 2$) no NumPy, demonstrando como matrizes são representadas e podem ser manipuladas.

O básico do Numpy

- O arranjo multidimensional homogêneo é o principal objeto do Numpy (ndarray)

O básico do Numpy

- O arranjo multidimensional homogêneo é o principal objeto do Numpy (ndarray)
- Uma tabela de elementos, geralmente números, todos do mesmo tipo, indexados por uma tupla de inteiros não negativos.

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

O básico do Numpy

- O arranjo multidimensional homogêneo é o principal objeto do Numpy (ndarray)
- Uma tabela de elementos, geralmente números, todos do mesmo tipo, indexados por uma tupla de inteiros não negativos.

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

- Em NumPy, as dimensões são chamadas de eixos.

O básico do Numpy

- O arranjo multidimensional homogêneo é o principal objeto do Numpy (ndarray)
- Uma tabela de elementos, geralmente números, todos do mesmo tipo, indexados por uma tupla de inteiros não negativos.

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

- Em NumPy, as dimensões são chamadas de eixos.

- A classe de arranjo (**array**) do NumPy é chamada **ndarray**.

O objeto ndarray em Numpy

- Alguns atributos do ndarray são:
  - .ndim: O número de eixos (dimensões) da matriz.
  - .shape: A forma (linhas, colunas) da matriz.
  - .size: O número total de elementos da matriz (linhas x colunas).

O objeto ndarray em Numpy

Primeiro importamos a biblioteca :

```
import numpy as np
```

```
a = np.arange(15).reshape(3, 5)
a
```

Então checamos o número de eixos:

```
a.ndim
```

A forma:

```
a.shape
```

E o númeo total de elementos:

```
a.size
```

O objeto ndarray em Numpy

- Alguns atributos do ndarray são:
    - .dtype: O tipo dos elementos da matriz (int16, int32 float64).
    - .itemsize: O tamanho em bytes de cada elemento da matriz (float64: 8 (=64/8), complex32:4 (=32/8)).
    - .data: O buffer que contém os elementos reais da matriz (<memory at 0×14adf0380>).

O objeto ndarray em Numpy

Primeiro importamos a biblioteca e usamos a função .arange(), para gerar um arranjo de valores uniformemente espaçados dentro de um determinado intervalo e também a função .reshape(), que dá uma nova forma ao arranjo,sem alterar os dados:

```
import numpy as np

a = np.arange(15).reshape(3, 5)
a
```

Então checamos o número de eixos:

```
a.ndim
```

A forma:

```
a.shape
```

E o númeo total de elementos:

```
a.size
```

Checamos o tipo dos elementos:

```
a.dtype
```

O tamanho do arranjo em bytes:

```
a.itemsize
```

E o endereço na memória:

```
a.data
```

Criando um arranjo

Para a criação de um arranjo unidimensional usaremos a função np.array(), O arranjo_1 é formado de números inteiros:

```
arranjo_1 = np.array([1, 2])
Arranjo_1
```

Checamos o tipo dos elementos no arranjo:

```
arranjo_1.dtype
```

O arranjo_2 é composto por flutuantes:

```
arranjo_2 = np.array([4.2, 9.2, 3.3])
arranjo_2.dtype
```

Criando um arranjo

Para a criação de um arranjo unidimensional usaremos a função np.array(), O arranjo_1 é formado de números inteiros:

```
arranjo_1 = np.array([1, 2])
Arranjo_1
```

Checamos o tipo dos elementos no arranjo:

```
arranjo_1.dtype
```

O arranjo_2 é composto por flutuantes:

```
arranjo_2 = np.array([4.2, 9.2, 3.3])
arranjo_2.dtype
```

Um arranjo bi-dimensional terá a forma:

```
arranjo_3 = np.array([(1.5, 2, 3), (4, 5, 6)])
Arranjo_3
```

É possível explicitar o tipo dos elementos em um arranjo:

```
arranjo_complexo = np.array(
[[1 + 3.j, 2], [3, 4.j]],
dtype = complex
)
arranjo_complexo
```

Criando um arranjo

É possível criar arranjos completamente zerados, com a função np.zeros():

```
np.zeros((10, 5))
```

É possível também criar arranjos repletos de uns com a função np.ones():

```
np.ones((2), dtype = np.int16)
```

Criando um arranjo

É possível criar arranjos completamente zerados, com a função np.zeros():

```
np.zeros((10, 5))
```

É possível também criar arranjos repletos de uns com a função np.ones():

```
np.ones((2), dtype = np.int16)
```

É possível ainda criar um arranjo de valores não inicializados (arbitrário), com a função np.empty():

```
np.empty(
    [2, 2],
    dtype = int
    )
```

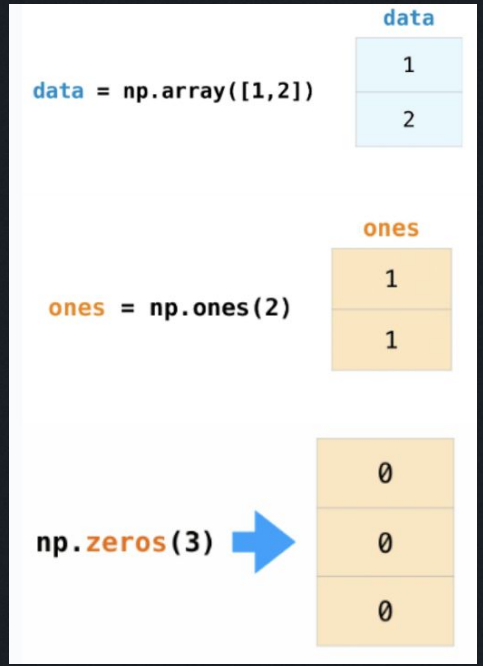
A função np.arange() retorna valores igualmente espaçados dentro de um intervalo:

```
np.arange(0, 100, 5)
```

E a função np.linspace() números igualmente espaçados sobre um intervalo específico:

```
np.linspace(0, 2, 10)
```

Criando um arranjo



Imprimindo um arranjo

Ao criar um arranjo, NumPy o exibe como listas aninhadas:

```
aranjo_1 = np.arange(6)
print(aranjo_1)

aranjo_2 = np.arange(12).reshape(4, 3)
print(aranjo_2)

aranjo_3 = np.arange(24).reshape(2, 3, 4)
print(aranjo_3)
```

A função `.reshape()` dá uma nova forma a arranjo, sem alterar seus dados.

Operações básicas com arranjos

Definindo dois arranjos:

```
arranjo_1 = np.array([10, 20, 30, 40])
arranjo_1
```

```
arranjo_2 = np.arange(4)
arranjo_2
```

Podemos subtrair seus elementos:

```
arranjo_3 = arranjo_1 - arranjo_2
Arranjo_3
```

Ou tomar o quadrado de arranjo_2:

```
arranjo_2 ** 2
```

Operações básicas com arranjos

Definindo dois arranjos:

```
arranjo_1 = np.array([10, 20, 30, 40])
arranjo_1
```

```
arranjo_2 = np.arange(4)
arranjo_2
```

Podemos subtrair seus elementos:

```
arranjo_3 = arranjo_1 - arranjo_2
Arranjo_3
```

Ou tomar o quadrado de arranjo_2:

```
arranjo_2 ** 2
```

É Possível também realizar operações trigonométricas com np.sin():

```
10 * np.sin(arranjo_1)
```

Ou testar condições lógicas:

```
arranjo_3 < 35
```

Operações básicas com arranjos

Considerando as matrizes:

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

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

print(mat_B)
```

É possível realizar o produto das mesmas, elemento a elemento:

```
mat_A * mat_B
```

Operações básicas com arranjos

Considerando as matrizes:

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

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

print(mat_B)
```

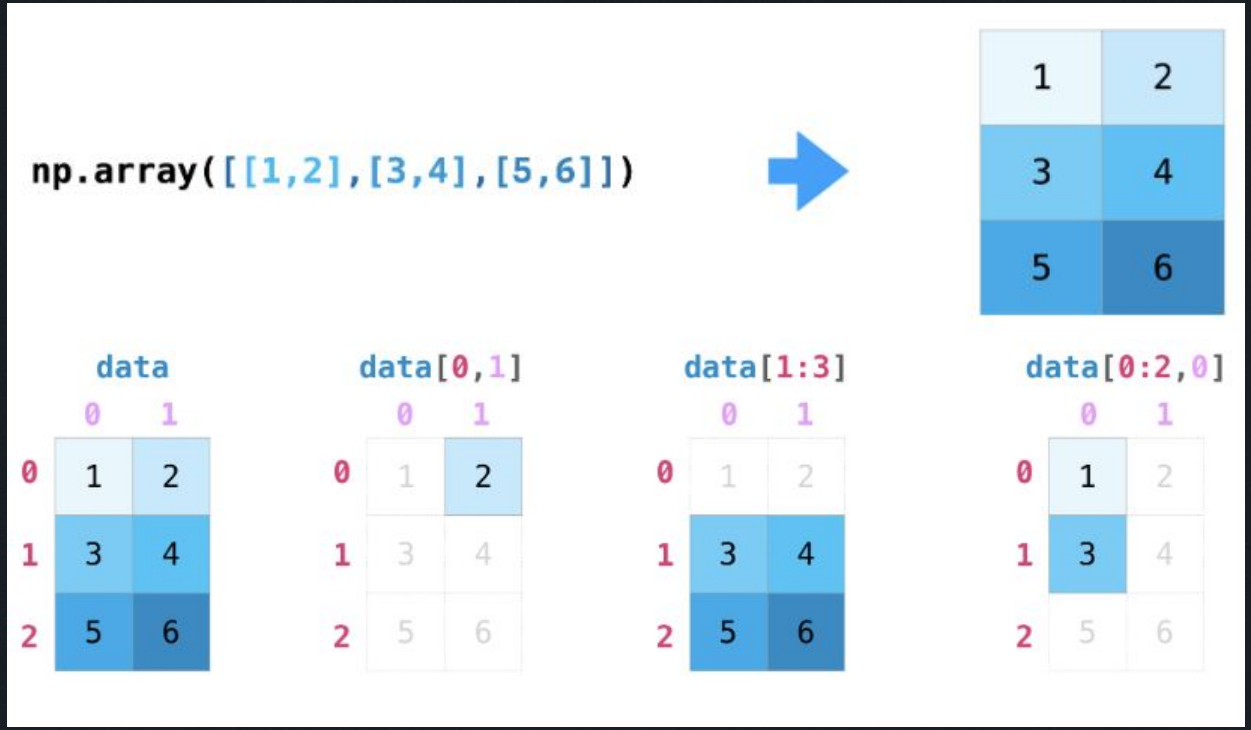
É possível realizar o produto das mesmas, elemento a elemento:

```
mat_A * mat_B
```

Também é possível realizar o produto matricial ou produto escalar:

```
mat_A @ mat_B
mat_A.dot(mat_B)
np.dot(mat_A, mat_B)
```

Operações básicas com arranjos



Operações básicas com arranjos

Definindo dois arranjos:

```
arranjo_1 = np.array([10, 20, 30, 40])
arranjo_1
```

```
arranjo_2 = np.arange(4)
arranjo_2
```

Podemos subtrair seus elementos:

```
arranjo_3 = arranjo_1 - arranjo_2
Arranjo_3
```

Ou tomar o quadrado de arranjo_2:

```
arranjo_2 ** 2
```

Operações básicas com arranjos

Definindo dois arranjos:

```
arranjo_1 = np.array([10, 20, 30, 40])
arranjo_1
```

```
arranjo_2 = np.arange(4)
arranjo_2
```

Podemos subtrair seus elementos:

```
arranjo_3 = arranjo_1 - arranjo_2
Arranjo_3
```

Ou tomar o quadrado de arranjo_2:

```
arranjo_2 ** 2
```

É Possível também realizar operações trigonométricas com np.sin():

```
10 * np.sin(arranjo_1)
```

Ou testar condições lógicas:

```
arranjo_3 < 35
```

Funções universais

Definimos dois arranjos:

```
arranjo_1 = np.arange(5)
print(arranjo_1)
arranjo_2 = np.array([1., 2., 3., 4., 5.])
print(arranjo_2)
```

Tomamos o valor exponencial de cada elemento com a função np.exp():

```
np.exp(arranjo_1)
```

E a raíz quadrado dos mesmos, com np.sqrt():

```
np.sqrt(arranjo_1)
```

Depois adicionamos os dois arranjos, com np.add():

```
np.add(arranjo_1, arranjo_2)
```

Depois subtrair os dois arranjos, com np.subtract():

```
np.subtract(arranjo_1, arranjo_2)
```

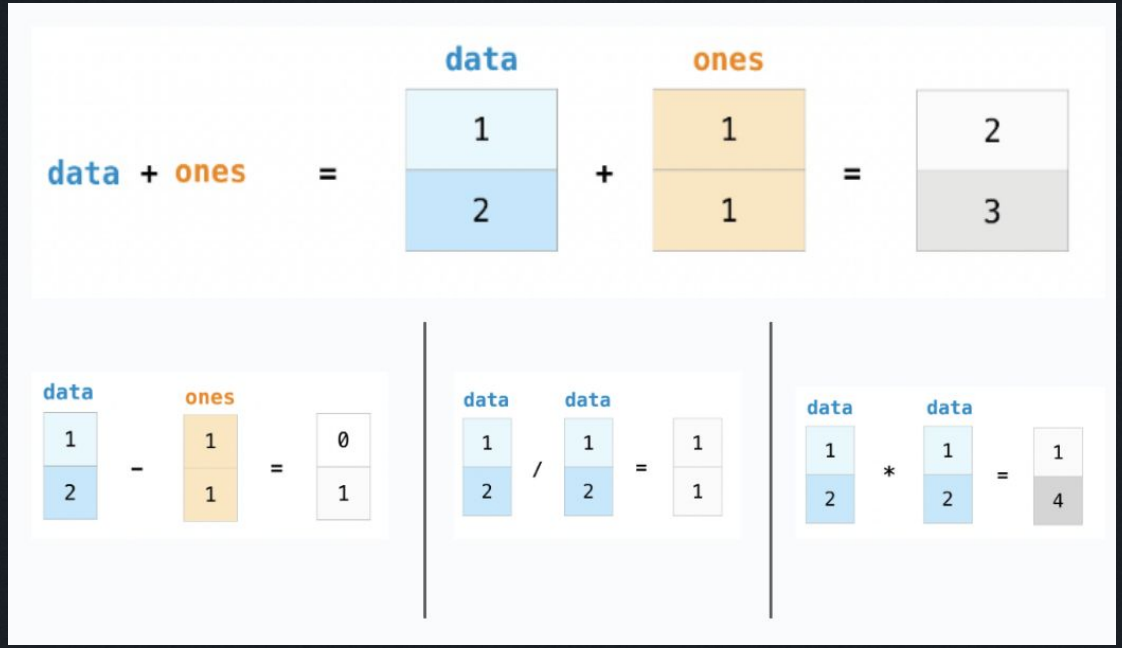
Também podemos dividir os dois arranjos, com np.divide():

```
np.divide(arranjo_1, arranjo_2)
```

E ainda multiplicar os dois arranjos, com np.multiply():

```
np.multiply(arranjo_1, arranjo_2)
```

Funções universais



Indexação (Indexing), fatiamento (Slicing) e iteração (Iterating)

Criamos um arranjo:

```
arranjo_1 = np.arange(10) ** 3
arranjo_1
```

Investigamos o valor no índice específico:

```
arranjo_1[2]
```

Realizamos o fatiamento (slice) em um intervalo:

```
arranjo_1[2 : 5]
```

Atualizamos valores com [start : end : step]

```
arranjo_1[ : 6 : 2] = 1000
```

Indexação (Indexing), fatiamento (Slicing) e iteração (Iterating)

Criamos um arranjo:

```
arranjo_1 = np.arange(10) ** 3
arranjo_1
```

Investigamos o valor no índice específico:

```
arranjo_1[2]
```

Realizamos o fatiamento (slice) em um intervalo:

```
arranjo_1[2 : 5]
```

Atualizamos valores com [start : end : step]

```
arranjo_1[ : 6 : 2] = 1000
```

Invertemos a ordem de leitura dos elementos:

```
arranjo_1[ : : -1]
```

E varremos o arranjo em um laço (loop):

```
for i in arranjo_1:
    print(i ** (1 / 2.))
```

Também é possível definir uma função, que retorna valores a partir de parâmetros de entrada (inputs):

```
def f(x, y):

    func_1 = 10 * x + y
    func_2 = np.sin(x) + y ** 3

    return func_1 - func_2
```

Indexação (Indexing), fatiamento (Slicing) e iteração (Iterating)

E podemos chamar essa função ao criarmos um arranjo:

```
arranjo_2 = np.fromfunction(f, (5, 4), dtype = int)
arranjo_2
```

Que também podemos fatiar:

```
arranjo_2[2, 3]
```

E apreciar apenas as cinco primeiras linhas da coluna um:

```
arranjo_2[0 : 5, 1]
```

Indexação (Indexing), fatiamento (Slicing) e iteração (Iterating)

E podemos chamar essa função ao criarmos um arranjo:

```
arranjo_2 = np.fromfunction(f, (5, 4), dtype = int)
arranjo_2
```

Que também podemos fatiar:

```
arranjo_2[2, 3]
```

E apreciar apenas as cinco primeiras linhas da coluna um:

```
arranjo_2[0 : 5, 1]
```

Ou todas as linhas da última coluna:

```
arranjo_2[ :, -1]
```

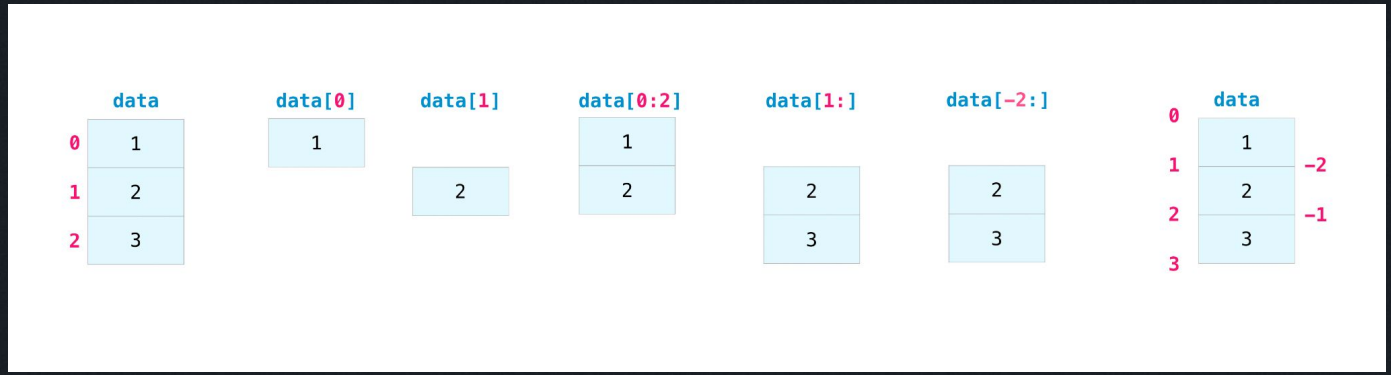
Ou as linhas 1 e 2 de todas as colunas:

```
arranjo_2[1 : 3, :]
```

Usado um iterador `.flat` que achata (flat) o arranjo, podemos iterar sobre o arranjo como se esse fosse 1D:

```
for element in arranjo_2.flat:
    print(element)
```

Indexação (Indexing), fatiamento (Slicing) e iteração (Iterating)



Manipulação de forma

Criamos um arranjo:

```
arranjo_1 = np.floor(10 * np.random.rand(3, 4))
arranjo_1
```

A função `.ravel()` retorna uma matriz achatada (flattened) contígua.

```
arranjo_1.ravel()
```

Tomamos a transposição do arranjo:

```
arranjo_1.T
```

E comparamos as duas formas:

```
print('arranjo_1.shape:', arranjo_1.shape)
print('arranjo_1.T.shape:', arranjo_1.T.shape)
```

Manipulação de forma

Criamos um arranjo:

```
arranjo_1 = np.floor(10 * np.random.rand(3, 4))
arranjo_1
```

A função `.ravel()` retorna uma matriz achatada (flattened) contígua.

```
arranjo_1.ravel()
```

Tomamos a transposição do arranjo:

```
arranjo_1.T
```

E comparamos as duas formas:

```
print('arranjo_1.shape:', arranjo_1.shape)
print('arranjo_1.T.shape:', arranjo_1.T.shape)
```

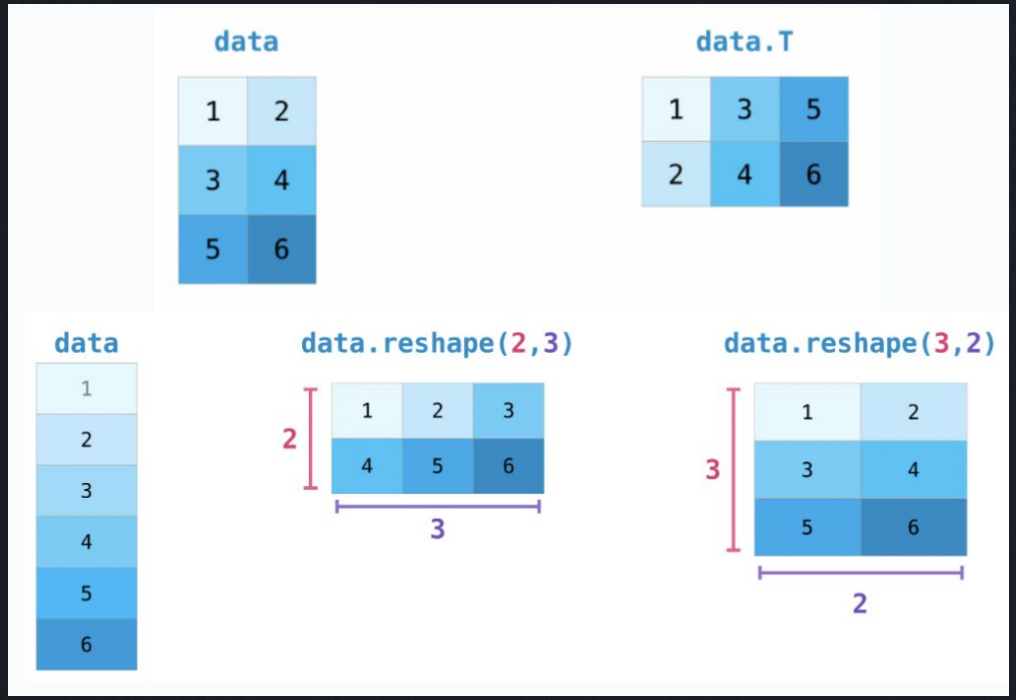
Podemos também re-definir o tamanho do arranjo de acordo com uma forma especificada:

```
arranjo_1.resize((2, 6))
arranjo_1
```

E Depois usamos `.reshape()`, que dá uma nova forma a um array sem alterar seus dados.

```
arranjo_1.reshape(3, -1)
```

Manipulação de forma



Manipulação de forma

Depois usamos .reshape(), que dá uma nova forma a um array sem alterar seus dados.

```
arranjo_1.reshape(3, -1)
```

Outra possibilidade é a de empilhar arranjos diferentes:

```
arranjo_1 = np.floor(10 * np.random.rand(3, 4))
arranjo_1
```

```
arranjo_2 = np.floor(10 * np.random.rand(3, 4))
arranjo_2
```

Manipulação de forma

Depois usamos .reshape(), que dá uma nova forma a um array sem alterar seus dados.

```
arranjo_1.reshape(3, -1)
```

Outra possibilidade é a de empilhar arranjos diferentes:

```
arranjo_1 = np.floor(10 * np.random.rand(3, 4))
arranjo_1
```

```
arranjo_2 = np.floor(10 * np.random.rand(3, 4))
arranjo_2
```

As funções np.vstack() e np.hstack empilham arranjos em sequência verticalmente (linha) e verticalmente (linha) e horizontalmente (coluna):

```
np.vstack((arranjo_1, arranjo_2))
```

```
np.hstack((arranjo_1, arranjo_2))
```

Dividindo um arranjo em vários menores

Criamos um arranjo

```
arranjo_1 = np.floor(10 * np.random.rand(2, 12))
Arranjo_1
```

E usamos a função np.hsplit() para dividí-lo em múltiplos sub-arranjos horizontalmente (colunas):

```
np.hsplit(arranjo_1, 3)
```

```
np.hsplit(arranjo_1, (3, 6))
```

```
np.hsplit(arranjo_1, (3, 6), 2)
```

Cópias e visualizações

Podemos comparar dois arranjos:

```
arranjo_1 = np.array([
    [ 0,  1,  2,  3],
    [ 4,  5,  6,  7],
    [ 8,  9, 10, 11]])

arranjo_2 = arranjo_1
arranjo_2 is arranjo_1
```

O método `.copy()` retorna uma cópia do arranjo do objeto fornecido.

```
arranjo_3 = arranjo_1.copy()

print('arranjo_3 is arranjo_1:',
      arranjo_3 is arranjo_1
      )
print(arranjo_3)
```

Cópias e visualizações

Podemos comparar dois arranjos:

```
arranjo_1 = np.array([
    [ 0,  1,  2,  3],
    [ 4,  5,  6,  7],
    [ 8,  9, 10, 11]])

arranjo_2 = arranjo_1
arranjo_2 is arranjo_1
```

O método `.copy()` retorna uma cópia do arranjo do objeto fornecido.

```
arranjo_3 = arranjo_1.copy()

print('arranjo_3 is arranjo_1:',
      arranjo_3 is arranjo_1
      )
print(arranjo_3)
```

É possível realizar o fatiamento de uma porção reduzida de um arranjo:

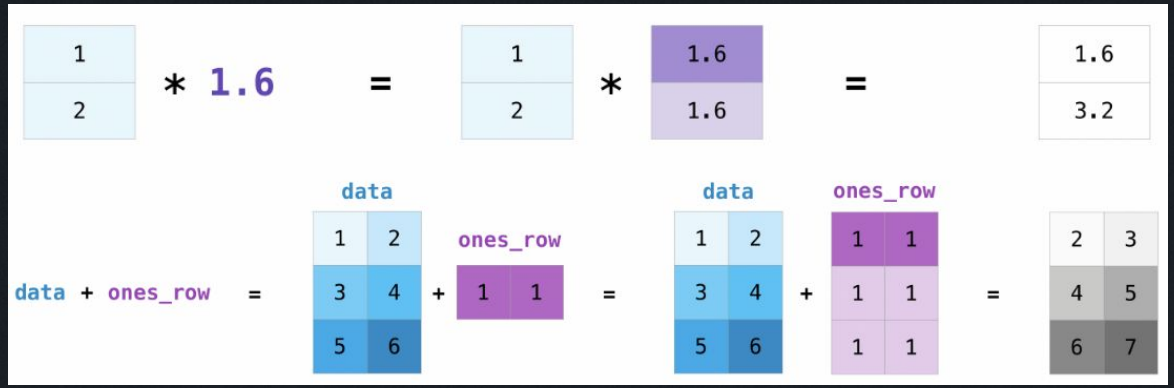
```
arranjo_1 = np.arange(int(1e3))
arranjo_2 = arranjo_1[ : 100].copy()

del arranjo_1

print('arranjo_2:', arranjo_2)
```

Regras de transmissão (broadcasting)

- A transmissão permite que funções universais lidem de maneira significativa com entradas que não possuem exatamente o mesmo formato.



Regras de transmissão (broadcasting)

- A transmissão permite que funções universais lidem de maneira significativa com entradas que não possuem exatamente o mesmo formato.

  1 - Se todos os arranjos de entrada não tiverem o mesmo número de dimensões, um “1” será repetidamente acrescentado às formas dos arranjos menores até que todos os arranjos tenham o mesmo número de dimensões.

Regras de transmissão (broadcasting)

- A transmissão permite que funções universais lidem de maneira significativa com entradas que não possuem exatamente o mesmo formato.

1 - Se todos os arranjos de entrada não tiverem o mesmo número de dimensões, um “1” será repetidamente acrescentado às formas dos arranjos menores até que todos os arranjos tenham o mesmo número de dimensões.

2 - Arranjos com tamanho 1 ao longo de uma dimensão específica atuam como se tivessem o tamanho do arranjo com o maior formato ao longo daquela dimensão. O valor do elemento do arranjo é considerado o mesmo ao longo dessa dimensão para o arranjo de “transmissão”.

Regras de transmissão (broadcasting)

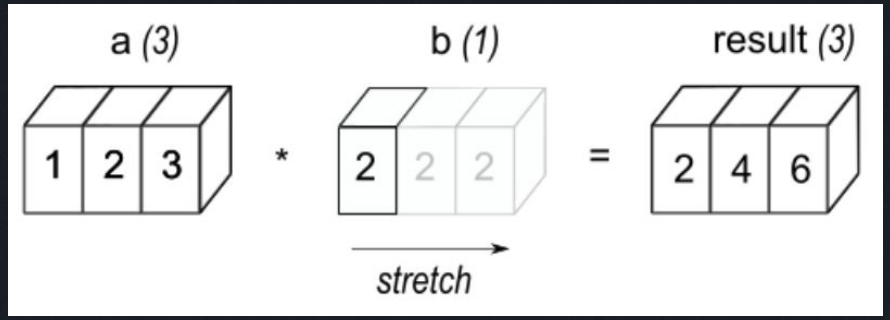
- A transmissão permite que funções universais lidem de maneira significativa com entradas que não possuem exatamente o mesmo formato.

1 - Se todos os arranjos de entrada não tiverem o mesmo número de dimensões, um “1” será repetidamente acrescentado às formas dos arranjos menores até que todos os arranjos tenham o mesmo número de dimensões.

2 - Arranjos com tamanho 1 ao longo de uma dimensão específica atuam como se tivessem o tamanho do arranjo com o maior formato ao longo daquela dimensão. O valor do elemento do arranjo é considerado o mesmo ao longo dessa dimensão para o arranjo de “transmissão”.

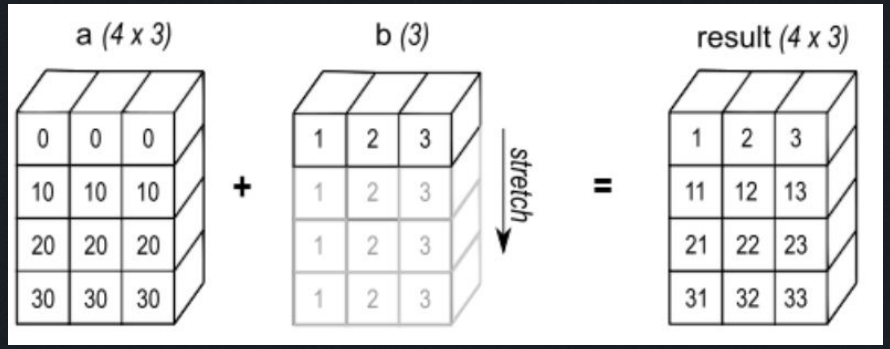
3 - Após a aplicação das regras de transmissão, os tamanhos de todos os arranjos devem corresponder.

Regras de transmissão (broadcasting)



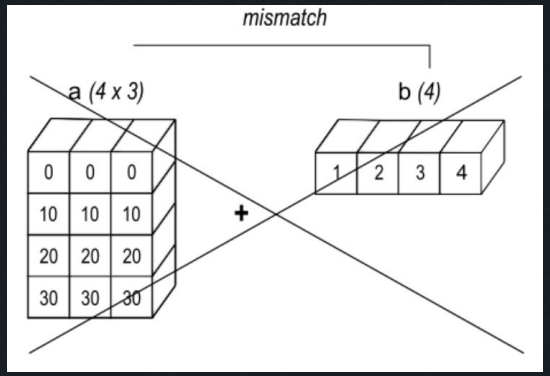
No exemplo mais simples de transmissão, o escalar b é esticado para se tornar uma matriz com o mesmo formato de a, de modo que as formas sejam compatíveis para multiplicação elemento por elemento.

Regras de transmissão (broadcasting)



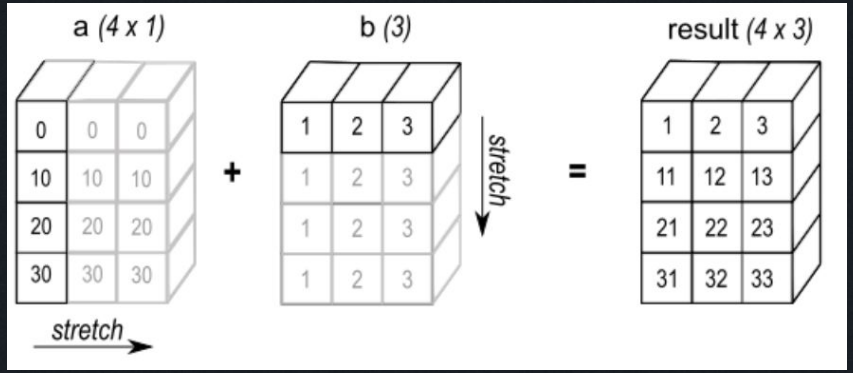
Uma matriz unidimensional adicionada a uma matriz bidimensional resulta em transmissão se o número de elementos da matriz 1-d corresponder ao número de colunas da matriz 2-d.

Regras de transmissão (broadcasting)



Quando as dimensões finais das matrizes são desiguais, a transmissão falha porque é impossível alinhar os valores nas linhas da 1ª matriz com os elementos das 2ª matrizes para adição elemento por elemento.

Regras de transmissão (broadcasting)



Em alguns casos, a transmissão estende ambas as matrizes para formar uma matriz de saída maior do que qualquer uma das matrizes iniciais.

Indexando com arranjos de índices

Podemos realizar a indexação com um arranjo de índices. Definimos um arranjo:

```
arranjo_1 = np.arange(12) ** 2
arranjo_1
```

E criamos um arranjo unidimensional de índices :

```
i = np.array([1, 1, 3, 8, 5])
print(arranjo_1[i])
```

Ou um arranjo bidimensional de índices:

```
j = np.array([
    [3, 4],
    [9, 7]
    ])
print(arranjo_1[j])
```

Indexando com arranjos de índices

Podemos realizar a indexação com um arranjo de índices. Definimos um arranjo:

```
arranjo_1 = np.arange(12) ** 2
arranjo_1
```

E criamos um arranjo unidimensional de índices :

```
i = np.array([1, 1, 3, 8, 5])
print(arranjo_1[i])
```

Ou um arranjo bidimensional de índices:

```
j = np.array([
    [3, 4],
    [9, 7]
    ])
print(arranjo_1[j])
```

Podemos alterar a forma do arranjo criado com `.reshape()`:

```
arranjo_2 = arranjo_1.reshape(3, 4)
print(arranjo_2)
```

Podemos usar um arranjo para indexar outro arranjo:

```
arranjo_1 = np.linspace(20, 145, 5)
arranjo_1
arranjo_2 = np.sin(np.arange(20)).reshape(5, 4)
arranjo_2
```

A função `.argmax()` é um exemplo de uso:

```
ind = arranjo_2.argmax(axis = 0)
ind
arranjo_1_max = arranjo_1[ind]
arranjo_1_max
```

Indexando com arranjos de índices

Usamos np.linspace() para Criar um arranjo com números com espaçamento uniforme em um intervalo especificado:

```
arranjo_1 = np.linspace(20, 145, 5)
arranjo_1
```

A função trigonométrica mp.sin() para gerar um segundo arranjo:

```
arranjo_2 = np.sin(np.arange(20)).reshape(5, 4)
arranjo_2
```

Indexando com arranjos de índices

Usamos np.linspace() para Criar um arranjo com números com espaçamento uniforme em um intervalo especificado:

```
arranjo_1 = np.linspace(20, 145, 5)
arranjo_1
```

A função trigonométrica mp.sin() para gerar um segundo arranjo:

```
arranjo_2 = np.sin(np.arange(20)).reshape(5, 4)
arranjo_2
```

Com a função .argmax() para obter os índices dos valores máximos ao longo do arranjo.

```
ind = arranjo_2.argmax(axis = 0)
ind
```

E então aplicamos o filtro ind como uma máscara para o arranjo inicial:

```
arranjo_1_max = arranjo_1[ind]
arranjo_1_max
```

Indexando com arranjos booleanos

Primeiramente, criamos um arranjo e depois o submetemos a uma condição:

```
arranjo_1 = np.arange(12).reshape(3, 4)
print('arranjo_1:', arranjo_1)
arranjo_2 = arranjo_1 > 4
print('arranjo_2:', arranjo_2)
```

Então aplicamos a máscara criada:

```
Arranjo_1[arranjo_2]
```

Depois, podemos alterar os elementos que atendam à condição:

```
arranjo_1[arranjo_2] = 0
print('arranjo_1:', arranjo_1)
```

Indexando com arranjos booleanos

Primeiramente, criamos um arranjo e depois o submetemos a uma condição:

```
arranjo_1 = np.arange(12).reshape(3, 4)
print('arranjo_1:', arranjo_1)
arranjo_2 = arranjo_1 > 4
print('arranjo_2:', arranjo_2)
```

Então aplicamos a máscara criada:

```
Arranjo_1[arranjo_2]
```

Depois, podemos alterar os elementos que atendam à condição:

```
arranjo_1[arranjo_2] = 0
print('arranjo_1:', arranjo_1)
```

E podemos alterar os valores que filtrados:

```
arranjo_1[arranjo_2] = 0
print('arranjo_1:', arranjo_1)

arranjo_1 = np.arange(12).reshape(3, 4)
print('arranjo_1:', arranjo_1)

arranjo_2 = np.array([False, True, True])
arranjo_3 = np.array([True, False, True, False])
```

Selecionando por linhas:

```
arranjo_1[arranjo_2, :]
arranjo_1[arranjo_2]
```

Ou por colunas:

```
arranjo_1[:, arranjo_3]
arranjo_1[arranjo_2, arranjo_3]
```

Histogramas em Numpy

Importamos a biblioteca matplotlib e criamos um arranjo aleatório com a função np.random.default_rng():

```
import matplotlib.pyplot as plt

np.random.default_rng()
rg = np.random.default_rng(1)
print('rg:', rg)
```

com a função .normal() obtemos uma distribuição normal (Gaussiana).

```
mu, sigma = 2, 0.5
v = rg.normal(mu, sigma, 10000)
```

Histogramas em Numpy

Importamos a biblioteca matplotlib e criamos um arranjo aleatório com a função np.random.default_rng():

```
import matplotlib.pyplot as plt

np.random.default_rng()
rg = np.random.default_rng(1)
print('rg:', rg)
```

com a função .normal() obtemos uma distribuição normal (Gaussiana).

```
mu, sigma = 2, 0.5
v = rg.normal(mu, sigma, 10000)
```

E plotamos a distribuição com a função .hist():

```
plt.hist(
    v,
    bins = 50,
    density = True
    )

(n, bins) = np.histogram(
    v,
    bins = 50,
    density = True
    )
plt.plot(.5 * (bins[1 : ] + bins[ : -1]), n);
```

E calculamos o histograma do conjunto de dados com a função np.histogram().

Exercício proposto:

- Estude e realize os exercícios dos notebooks:
  - ADA_TECH_2_numpy_enunciado_exercicio_1.ipynb
  - ADA_TECH_3_numpy_enunciado_exercicio_2.ipynb
  - ADA_TECH_6_numpy_enunciado_exercicio_3.ipynb

De volta ao problema dos arranjos

O numpy é uma biblioteca Python de código aberto usada em quase todos os campos da ciência e da engenharia. É o padrão universal para trabalhar com dados numéricos em Python e está no centro dos ecossistemas científicos Python e PyData

Referências

- Numpy
- NumPy for Data Science: Part 1
- The N-dimensional array (ndarray)
- NumPy Tutorial
- Transpose
- The Python Tutorial
- Basics of Linear Algebra
- Linear algebra (numpy.linalg)
- How do I use the slice notation in Python?