

Python

Aula 13 - Numpy

Prof. Eduardo Campos (CEFET-MG)

Slides do prof. João Silva da UFRJ

NumPy é uma biblioteca para computação científica em Python.

- Ela torna mais fácil a criação e operação de arrays multidimensionais. Python é mais lento que linguagens como C e Fortran.
- Numpy fornece uma interface com códigos compilados (e otimizados) em C e Fortran, permitindo que as operações relativas a arrays sejam computadas de maneira mais rápida.
- Manual de referência: <https://docs.scipy.org/doc/numpy/reference/>

NumPy é uma biblioteca para computação científica em Python.

- O tipo de dado *array* definido no Numpy é formalmente chamado **numpy.ndarray**, que representam *arrays homogêneos multidimensionais* (*homogeneous multidimensional array*).
- São containers N-dimensionais (i.e., multidimensionais) de tamanho fixo, compostos de itens (normalmente numéricos) de mesmo tipo (i.e., homogêneos) e indexado por tuplas de inteiros positivos.

Numpy - Criando Arrays

- Importando a biblioteca Numpy:

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

- Criando arrays:

```
1 In [2]: x = np.array([1, 2, 3]) # criando o numpy.ndarray x com numeros inteiros 1,
2      2 e 3
3 In [3]: x
4 Out[3]:
5 array([1, 2, 3])
6
7 In [4]: y = np.array([[ 1., 0., 0.] , [ 0., 1., 2.]]) # criando o numpy.ndarray y
8      com numeros float [1., 0., 0.] e [0., 1., 2.]
9 In [5]: y
10 Out[5]:
11 array([[ 1.,  0.,  0.],
12        [ 0.,  1.,  2.]])
13
14 In [6]: type(x)
15 Out[6]: numpy.ndarray # x e do tipo numpy.ndarray
```

- Cada dimensão em Numpy é chamada de eixo (axes).
- Array x : [1, 2, 3] tem dimensão (rank) 1 e seu eixo tem comprimento 3. Pode ser visto como um vetor.
- Array y : [[1., 0., 0.], [0., 1., 2.]] tem dimensão (rank) 2. O primeiro eixo tem comprimento 2, e o segundo tem comprimento 3. Pode ser visto como uma matriz 2x3

Numpy - Criando Arrays

- Criando arrays:

```
1 In [4]: y = np.array([[ 1, 2, 3], [ 5, 6, 7]])
2
3 In [5]: y
4 Out[5]:
5 array([[1, 2, 3],
6        [5, 6, 7]])
7
8 In [6]: y[0]      # primeiro eixo
9 Out[6]: array([1, 2, 3])
10
11 In [7]: y[1]      # primeiro eixo
12 Out[7]: array([5, 6, 7])
13
14 In [8]: y[0][1]    # segundo eixo
15 Out[8]: 2
16
17 In [9]: y[0,1]     # outra forma de acessar elemento
18 Out[9]: 2
```

- Cada dimensão em Numpy é chamada de *eixo* (axes).
- Array $y = np.array([[1, 2, 3], [5, 6, 7]])$ tem dimensão (rank) 2. O primeiro eixo tem comprimento 2, e o segundo tem comprimento 3. Pode ser visto como uma matriz 2x3

Numpy - Shape

Podemos usar o comando *shape* para saber a forma de um array, assim como para reformatá-lo:

```
1 In [1]: x = np.array([1, 2, 3, 4])
2
3 In [2]: x.shape # forma do array
4 Out[2]: (4,)
5
6 In [3]: y = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]],
7                       [[13,14,15,16],[17,18,19,20],[21,22,23,24]] )
8
9 In [4]: y.shape
10 Out[4]: (2, 3, 4) # primeiro eixo = 2, segundo eixo = 3, terceiro eixo = 4
11
12 In [5]: y.shape = (3, 8) # reformatando o array
13
14 In [6]: y
15 Out[6]:
16 array([[ 1,  2,  3,  4,  5,  6,  7,  8],
17        [ 9, 10, 11, 12, 13, 14, 15, 16],
18        [17, 18, 19, 20, 21, 22, 23, 24]])
```

Numpy - Shape

Podemos usar o comando *shape* para saber a forma de um array, assim como para reformatá-lo:

```
1 In [6]: y
2 Out [6]:
3 array([[ 1,  2,  3,  4,  5,  6,  7,  8],
4        [ 9, 10, 11, 12, 13, 14, 15, 16],
5        [17, 18, 19, 20, 21, 22, 23, 24]])
6
7 In [7]: y.shape = (2,3,4)
8 Out [7]:
9 array([[[ 1,  2,  3,  4],
10         [ 5,  6,  7,  8],
11         [ 9, 10, 11, 12]],
12       [[13, 14, 15, 16],
13        [17, 18, 19, 20],
14        [21, 22, 23, 24]]])
15
16 In [8]: y.shape=(3,6)
17
18 Traceback (most recent call last):
19   File "<ipython-input-76-397625fb3221>", line 1, in <module>
20     y.shape=(3,6)
21
22 ValueError: cannot reshape array of size 24 into shape (3,6)
```

Numpy - Shape

Exercício: dado um array qualquer, faça uma função que retorne a média de todos os números do array.

Numpy - Shape

Exercício: dado um array qualquer, faça uma função que retorne a média de todos os números do array.

```
1 def mediaArray(dado):
2     """Funcao que calcula a media dos elementos de um array
3     Param. de Entrada: array
4     valor de retorno: float"""
5     formato = dado.shape
6     numeroElementos = 1
7     for i in formato:
8         numeroElementos = numeroElementos * i
9     dado.shape = (numeroElementos)
10    media = sum(dado)/numeroElementos
11    return media
```

Numpy - Shape

Exercício: dado um array qualquer, faça uma função que retorne a média de todos os números do array.

```
1 def mediaArray(dado):
2     """Funcao que calcula a media dos elementos de um array
3     Param. de Entrada: array
4     valor de retorno: float"""
5     formato = dado.shape
6     numeroElementos = 1
7     for i in formato:
8         numeroElementos = numeroElementos * i
9     dado.shape = (numeroElementos)
10    media = sum(dado)/numeroElementos
11    return media
```

Testando:

```
1 In [1]: y = np.array([[ [1,2,3,4] , [5,6,7,8] , [9,10,11,12] ] , [ [13 ,14 ,15 ,16] , [17
   ,18 ,19 ,20] , [21 ,22 ,23 ,24]]])
2
3 In [2]: mediaArray(y)
4 Out[2]: 12.5
5
6 In [3]: y
7 Out[3]:
8 array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
9         18, 19, 20, 21, 22, 23, 24]])
```

Note que mudamos o formato original do array. Como podemos recuperá-lo?

Numpy - Shape

Exercício: dado um array qualquer, faça uma função que retorne a média de todos os números do array.

```
1 def mediaArray(dado):
2     """Funcao que calcula a media dos elementos de um array
3     Param. de Entrada: array
4     valor de retorno: float"""
5     formato = dado.shape
6     numeroElementos = 1
7     for i in formato:
8         numeroElementos = numeroElementos * i
9     dado.shape = (numeroElementos)
10    media = sum(dado)/numeroElementos
11    dado.shape = formato # restaura o formato original
12    return media
```

Testando:

```
1 In [1]: y = np.array([[ [1,2,3,4] , [5,6,7,8] , [9,10,11,12] ] , [ [13 ,14 ,15 ,16] , [17
2     ,18 ,19 ,20] , [21 ,22 ,23 ,24]]])
3
4 In [2]: mediaArray(y)
5 Out[2]: 12.5
6
7 In [3]: y
8 Out[3]:
9 array([[ [1,2,3,4] , [5,6,7,8] , [9,10,11,12] ] , [ [13 ,14 ,15 ,16] , [17 ,18 ,19 ,20]
10    , [21 ,22 ,23 ,24]]])
```

Numpy - dtypes

- **float64**: número de ponto flutuante de 64 bits
- **int64**: inteiro de 64 bits
- **bool**: True or False (8 bits)
- Outros tipos: **complexfloat**, **signedinteger**,...

O tipo de dado pode ser definido explicitamente quando da criação do array:

```
1 In [12]: x1 = np.array([[ 1, 2, 3] , [ 4, 5, 6]], dtype='int32')
2
3 In [13]: x1
4 Out[13]:
5 array([[1, 2, 3],
6        [4, 5, 6]], dtype=int32)
7
8 In [14]: x1.dtype
9 Out[14]: dtype('int32')
10
11 In [15]: x2 = np.array([[ 1, 2, 3] , [ 4, 5, 6]], dtype=np.float64) # outra forma de dtype
12
13 In [16]: x2
14 Out[16]:
15 array([[ 1.,  2.,  3.],
16        [ 4.,  5.,  6.]])
17
18 In [17]: x2.dtype
19 Out[17]: dtype('float64')
```

Numpy - Criando Arrays

Lembra como criamos matrizes ...

Numpy - Criando Arrays

Lembra como criamos matrizes ...

Faça uma função que construa uma matriz 4x3 com valores iguais a zero.
Retorne a matriz.

```
1 def constroiMatriz4():
2
3     """ Funcao que constroi uma matriz 4x3 de 0's
4     Parametro de entrada: nao tem
5     Valor de retorno: list """
6
7     matriz = [ ]
8     for i in range(4):
9         list.append(matriz,[0] * 3 )
10    return matriz
```

Numpy - Criando Arrays

- **np.zeros**: cria um array com todos os elementos sendo 0.
- **np.ones**: cria um array com todos os elementos sendo 1.
- **np.empty**: cria um array cujo conteúdo inicial é aleatório.

Podemos criar arrays *flat*:

```
1 In [2]: a = np.zeros(10)
2 In [3]: a
3 Out[3]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
4
5 In [4]: b = np.ones(10)
6 In [5]: b
7 Out[5]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
8
9 In [6]: c = np.empty(10)
10 In [7]: c
11 Out[7]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]) # resultado pode variar
```

Numpy - Criando Arrays

Podemos criar arrays *flat*:

```
1 In [2]: a = np.zeros(10)
2 In [3]: a
3 Out[3]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
4
5 In [4]: b = np.ones(10)
6 In [5]: b
7 Out[5]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
8
9 In [6]: c = np.empty(10)
10 In [7]: c
11 Out[7]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]) # resultado pode variar
```

e depois usar **shape** para formatá-lo:

Numpy - Criando Arrays

Podemos criar arrays *flat*:

```
1 In [2]: a = np.zeros(10)
2 In [3]: a
3 Out[3]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
4
5 In [4]: b = np.ones(10)
6 In [5]: b
7 Out[5]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
8
9 In [6]: c = np.empty(10)
10 In [7]: c
11 Out[7]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]) # resultado pode variar
```

e depois usar **shape** para formatá-lo:

```
1 In [8]: a.shape=(2,5)
2 In [9]: a
3 Out[9]:
4 array([[ 0.,  0.,  0.,  0.,  0.],
5        [ 0.,  0.,  0.,  0.,  0.]])
```

Numpy - Criando Arrays

Podemos criar arrays *flat*:

```
1 In [2]: a = np.zeros(10)
2 In [3]: a
3 Out[3]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
4
5 In [4]: b = np.ones(10)
6 In [5]: b
7 Out[5]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
8
9 In [6]: c = np.empty(10)
10 In [7]: c
11 Out[7]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]) # resultado pode variar
```

e depois usar **shape** para formatá-lo:

```
1 In [8]: a.shape=(2,5)
2 In [9]: a
3 Out[9]:
4 array([[ 0.,  0.,  0.,  0.,  0.],
5        [ 0.,  0.,  0.,  0.,  0.]])
```

Atenção para o formato escolhido:

```
1 In [8]: a.shape=(3,4)
2 Traceback (most recent call last):
3   File "<ipython-input-10-510a1dfd21dc>", line 1, in <module>
4     a.shape=(3,4)
5 ValueError: cannot reshape array of size 10 into shape (3,4)
```

Numpy - Criando Arrays

Podemos definir o formato e o tipo de dados desejado na hora da criação do array:

```
1 In [2]: np.zeros((3,4)) # o formato e (3,4)
2 Out[2]:
3 array([[ 0.,  0.,  0.,  0.],
4        [ 0.,  0.,  0.,  0.],
5        [ 0.,  0.,  0.,  0.]])
6
7 In [3]: np.ones((2,3,4), dtype='int16') # dtype pode ser especificado
8 Out[3]:
9 array([[[ 1,  1,  1,  1],
10         [ 1,  1,  1,  1],
11         [ 1,  1,  1,  1]],
12        [[ 1,  1,  1,  1],
13         [ 1,  1,  1,  1],
14         [ 1,  1,  1,  1]]], dtype=int16)
15
16 In [4]: np.empty((2,3)) # a saída pode variar
17 Out[4]:
18 array([[ 1.,  2.,  3.],
19        [ 4.,  5.,  6.]])
```

Exercício: Crie um array de inteiros de 32 bits de formato (3,2,5), onde todos os elementos são iguais a 3.

Numpy - Criando Arrays

Podemos definir o formato e o tipo de dados desejado na hora da criação do array:

```
1 In [2]: np.zeros((3,4)) # o formato e (3,4)
2 Out[2]:
3 array([[ 0.,  0.,  0.,  0.],
4        [ 0.,  0.,  0.,  0.],
5        [ 0.,  0.,  0.,  0.]])
6
7 In [3]: np.ones((2,3,4), dtype='int16') # dtype pode ser especificado
8 Out[3]:
9 array([[[ 1,  1,  1,  1],
10         [ 1,  1,  1,  1],
11         [ 1,  1,  1,  1]],
12        [[ 1,  1,  1,  1],
13         [ 1,  1,  1,  1],
14         [ 1,  1,  1,  1]]], dtype=int16)
15
16 In [4]: np.empty((2,3)) # a saída pode variar
17 Out[4]:
18 array([[ 1.,  2.,  3.],
19        [ 4.,  5.,  6.]])
```

Exercício: Crie um array de inteiros de 32 bits de formato (3,2,5), onde todos os elementos são iguais a 3.

```
3*np.ones((3,2,5),dtype='int32')
```

Numpy - Operações

Os operadores $+$, $-$, $*$, $/$ e $**$ são aplicados em cada par de elementos dos arrays:

```
1 In [2]: a = np.array([1, 2, 3, 4])
2 In [3]: b = np.array([5, 6, 7, 8])
3
4 In [4]: a + b
5 Out[4]: array([ 6,  8, 10, 12])
6
7 In [5]: a * b
8 Out[5]: array([ 5, 12, 21, 32])
9
10 In [6]: a + 10
11 Out[6]: array([11, 12, 13, 14])
12
13 In [7]: a * 10
14 Out[7]: array([10, 20, 30, 40])
```

Numpy - Operações

Os operadores $+$, $-$, $*$, $/$ e $**$ são aplicados em cada par de elementos dos arrays:

```
1 In [8]: A = np.ones((2, 2))
2 In [9]: B = np.ones((2, 2))
3
4 In [10]: A + B
5 Out[10]:
6 array([[ 2.,  2.],
7        [ 2.,  2.]])
8
9 In [11]: A + 10
10 Out[11]:
11 array([[ 11.,  11.],
12        [ 11.,  11.]])
13
14 In [12]: A * B
15 Out[12]:
16 array([[ 1.,  1.],
17        [ 1.,  1.]])
```

Numpy - Multiplicação de Matrizes

O operador @ pode ser usado para multiplicar matrizes:

```
1 In [2]: A = np.array( [[1,1],[0,1]] )
2 In [3]: B = np.array( [[2,0],[3,4]] )
3
4 In [4]: A @ B
5 Out[4]:
6 array([[5, 4],
7        [3, 4]])
```

Para o produto interno de dois arrays flats:

```
1 In [2]: A = np.array([1, 2])
2 In [3]: B = np.array([10, 20])
3
4 In [4]: A @ B
5 Out[4]: 50
```

Para multiplicar um array por uma lista ou tupla:

```
1 In [2]: A = np.array([[1, 2], [3, 4]])
2 In [3]: A
3 Out[3]:
4 array([[1, 2],
5        [3, 4]])
6
7 In [4]: A @ [0, 1]
8 Out[4]: array([2, 4])
```

Numpy - Multiplicação de Matrizes

Cuidado com a dimensão das matrizes que você está multiplicando:

```
1 In [2]: x = np.array([1,2,3,4,5,6])
2 In [3]: y = np.array([9,7,5,3,1,5])
3 In [4]: x.shape=(3,2)    # 3 linhas e 2 colunas
4 In [5]: y.shape=(2,3)    # 2 linhas e 3 colunas
5
6 In [6]: x
7 Out[6]:
8 array([[1, 2],
9        [3, 4],
10       [5, 6]])
11
12 In [7]: y
13 Out[7]:
14 array([[9, 7, 5],
15        [3, 1, 5]])
16
17 In [8]: x@y
18 Out[8]:
19 array([[15, 9, 15],      # matriz resultant 3x3
20        [39, 25, 35],
21        [63, 41, 55]])
```


Numpy - Multiplicação de Matrizes

Cuidado com a dimensão das matrizes que você está multiplicando:

```
1 In [2]: x = np.array([1,2,3,4,5,6])
2 In [3]: y = array([9, 7, 5, 3, 1, 5, 2, 1, 4])
3 In [4]: x.shape=(3,2)      # 3 linhas e 2 colunas
4 In [5]: y.shape=(3,3)      # 3 linhas e 3 colunas
5
6 In [6]: x
7 Out[6]:
8 array([[1, 2],
9        [3, 4],
10       [5, 6]])
11
12 In [7]: y
13 Out[7]:
14 array([[9, 7, 5],
15        [3, 1, 5],
16        [2, 1, 4]])
17
18 In [8]: x@y
19 Traceback (most recent call last):
20
21   File "<ipython-input-99-3fa8ae35169b>", line 1, in <module>
22     x@y
23
24 ValueError: shapes (3,2) and (3,3) not aligned: 2 (dim 1) != 3 (dim 0)
```

Numpy - Multiplicação de Matrizes

Podemos usar também *dot*:

```
1 In [2]: A = np.array( [[1,1],[0,1]] )
2 In [3]: B = np.array( [[2,0],[3,4]] )
3
4 In [4]: A.dot(B)
5 Out[4]:
6 array([[5, 4],
7        [3, 4]])
```

```
1 In [2]: A = np.array([1, 2])
2 In [3]: B = np.array([10, 20])
3
4 In [4]: A.dot(B)
5 Out[4]: 50
```

```
1 In [2]: A = np.array([[1, 2], [3, 4]])
2 In [3]: A
3 Out[3]:
4 array([[1, 2],
5        [3, 4]])
6
7 In [4]: A.dot([0, 1])
8 Out[4]: array([2, 4])
```

Numpy - Criando sequência de números

Função análoga a função **arange** que retorna arrays ao invés de listas.

- **np.arange([start],stop,[step],dtype=None)**: retorna um *array* de valores no intervalo *[start, end)* com o tamanho do passo igual a *step*.

```
1 In [2]: np.arange( 10, 30, 5 )
2 Out[2]: array([10, 15, 20, 25])
3
4 In [3]: np.arange( 0, 2, 0.3 )           # aceita argumentos do tipo float
5 Out[3]: array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

Quando os argumentos de *arange* são de ponto flutuante (float), geralmente não é possível prever o número de elementos obtidos (precisão finita). Neste caso, o melhor é usar a função **linspace** que recebe como argumento o número de elementos que queremos ao invés do tamanho do passo:

```
1 In [4]: np.linspace(0, 2, 9)           # 9 numeros de 0 a 2
2 Out[4]: array([ 0. ,  0.25,  0.5 ,  0.75,  1. ,  1.25,  1.5 ,  1.75,  2.  ])
```

Numpy - Exercício

Faça uma função que dado um intervalo fechado $[a, b]$ e um número de pontos n , retorna o array x , formado pelos n pontos obtidos a partir do intervalo $[a, b]$ e y é o array onde $y_i = x_i^2 - 9$.

Exemplo

```
1 In [1]: f([-10,10],10)
2 Out[1]:
3 (array([-10.          ,  -7.77777778,  -5.55555556,  -3.33333333,
4         -1.11111111,   1.11111111,   3.33333333,   5.55555556,
5          7.77777778,  10.          ]),
6  array([ 91.          ,  51.49382716,  21.86419753,   2.11111111,
7         -7.7654321 ,  -7.7654321 ,   2.11111111,  21.86419753,
8         51.49382716,  91.          ]))
```

Numpy - Exercício

Faça uma função `gera_matriz` que recebe como parâmetros de entrada (a, b, n, m) e que tem como retorno um array Numpy de formato $m \times m$, onde os elementos são os n números igualmente espaçados obtidos ao se dividir o intervalo $[a, b]$. Caso não seja possível construir tal array, a função deve retornar a mensagem *'Nao e possivel construir'*.

Exemplo

```
1 In [1]: gera_matriz(0,10,16,4)
2 Out[1]
3 array([[ 0.,          0.66666667,  1.33333333,  2.          ],
4        [ 2.66666667,  3.33333333,  4.          ,  4.66666667],
5        [ 5.33333333,  6.          ,  6.66666667,  7.33333333],
6        [ 8.          ,  8.66666667,  9.33333333, 10.          ]])
7
8 In [2]: gera_matriz(0,10,10,4)
9 Out[2]: 'Nao e possivel construir'
```

Numpy - Índices

Arrays flat:

```
1 In [2]: a = np.linspace(1, 2, 5)
2
3 In [3]: a
4 Out[3]: array([ 1.   ,  1.25,  1.5   ,  1.75,  2.   ])
5
6 In [4]: a[0]
7 Out[4]: 1.0
8
9 In [5]: a[0:2] # retornar 2 elementos a partir do índice 0
10 Out[5]: array([ 1.   ,  1.25])
11
12 In [6]: a[-1]
13 Out[6]: 2.0
```

Arrays 2D:

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

e assim por diante.

Numpy - Extraindo Elementos

Podemos extrair linhas e colunas de uma matriz:

```
1 In [8]: b
2 Out[8]:
3 array([[1, 2],
4        [3, 4]])
5
6 In [9]: b[0,:]
7 Out[9]: array([1, 2]) # extraindo linha
8
9 In [10]: b[:,1]
10 Out[10]: array([2, 4]) # extraindo coluna
11
12 In [11]: c
13 Out[11]: array([[[ 1,  2,  3,  4],
14                 [ 5,  6,  7,  8]],
15                [[ 9, 10, 11, 12],
16                 [13, 14, 15, 16]]])
17
18 In [12]: c.shape
19 Out[12]: (2, 2, 4)
20 In [13]: c[1,1,2]
21 Out[13]: 15
22
23 In [14]: c[:,1,2]
24 Out[14]: array([ 7, 15])
25
26 In [15]: c[:, :, 2]
27 Out[15]: array([[ 3,  7],
28                 [11, 15]])
```

Numpy - Extraindo Elementos

Podemos extrair elementos usando arrays de inteiros:

```
1 In [11]: c = np.linspace(2, 4, 5)
2 In [12]: c
3 Out[12]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])
4
5 In [13]: indices = np.array([0, 2, 3]) # indices que estamos interessados sao 0, 2 e 3
6 In [14]: c[indices]
7 Out[14]: array([ 2. ,  3. ,  3.5]) # array com os elementos do array que queremos
```


Numpy - Extraindo Elementos - Array bool

Um array dtype bool podem ser usados para extrair os elementos de um array:

```
1 In [11]: c = np.linspace(2, 4, 5)
2 In [12]: c
3 Out[12]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])
4
5 In [13]: indices = np.array([0, 1, 1, 0, 0], dtype=bool)
6 In [14]: indices
7 Out[14]: array([False,  True,  True, False, False], dtype=bool)
8
9 In [15]: c[indices]
10 Out[14]: array([ 2.5,  3. ])
```

Numpy - Exercício

Faça duas funções, *linha* e *coluna*, cada uma recebendo como parâmetro de entrada um array M e um número inteiro i . O formato (n, m) do array M não é passado como parâmetro de entrada. As funções devem retornar, respectivamente, a linha e a coluna i de M . Se M não possui linha (coluna) i , a mensagem "Nao existe esta linha (coluna)" deve ser retornada. Caso M não tenha o formato (n, m) , a mensagem "Isto nao e uma matriz." deve ser retornada.

Exemplo

```
1 In [1]: z
2 Out[1]:
3 array([[ 0.,          1.11111111,  2.22222222,  3.33333333,  4.44444444],
4        [ 5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.          ]])
5
6 In [2]: w
7 Out[2]:
8 array([[[ 0.,          1.03846154,  2.07692308],
9         [ 3.11538462,  4.15384615,  5.19230769],
10        [ 6.23076923,  7.26923077,  8.30769231]],
11
12        [[ 9.34615385, 10.38461538, 11.42307692],
13         [12.46153846, 13.5         , 14.53846154],
14         [15.57692308, 16.61538462, 17.65384615]],
15
16        [[18.69230769, 19.73076923, 20.76923077],
17         [21.80769231, 22.84615385, 23.88461538],
18         [24.92307692, 25.96153846, 27.          ]]])
19
20 In [3]: coluna(z,5)
21 Out[3]: array([ 4.44444444, 10.          ])
```

Numpy - Exercício

Faça duas funções, *linha* e *coluna*, cada uma recebendo como parâmetro de entrada um array M e um número inteiro i . O formato (n, m) do array M não é passado como parâmetro de entrada. As funções devem retornar, respectivamente, a linha e a coluna i de M . Se M não possui linha (coluna) i , a mensagem "Nao existe esta linha (coluna)" deve ser retornada. Caso M não tenha o formato (n, m) , a mensagem "Isto nao e uma matriz." deve ser retornada.

Exemplo

```
1 In [1]: z
2 Out[1]:
3 array([[ 0.,          1.11111111,  2.22222222,  3.33333333,  4.44444444],
4        [ 5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.          ]])
5
6 In [2]: w
7 Out[2]:
8 array([[[ 0.,          1.03846154,  2.07692308],
9         [ 3.11538462,  4.15384615,  5.19230769],
10        [ 6.23076923,  7.26923077,  8.30769231]],
11       [[ 9.34615385, 10.38461538, 11.42307692],
12        [12.46153846, 13.5         , 14.53846154],
13        [15.57692308, 16.61538462, 17.65384615]],
14       [[18.69230769, 19.73076923, 20.76923077],
15        [21.80769231, 22.84615385, 23.88461538],
16        [24.92307692, 25.96153846, 27.          ]]])
17
18 In [4]: linha(z,5)
19 Out[4]: 'Nao existe esta linha'
```

Numpy - Exercício

Faça duas funções, *linha* e *coluna*, cada uma recebendo como parâmetro de entrada um array M e um número inteiro i . O formato (n, m) do array M não é passado como parâmetro de entrada. As funções devem retornar, respectivamente, a linha e a coluna i de M . Se M não possui linha (coluna) i , a mensagem "Nao existe esta linha (coluna)" deve ser retornada. Caso M não tenha o formato (n, m) , a mensagem "Isto nao e uma matriz." deve ser retornada.

Exemplo

```
1 In [1]: z
2 Out[1]:
3 array([[ 0.,          1.11111111,  2.22222222,  3.33333333,  4.44444444],
4        [ 5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.          ]])
5
6 In [2]: w
7 Out[2]:
8 array([[[ 0.,          1.03846154,  2.07692308],
9         [ 3.11538462,  4.15384615,  5.19230769],
10        [ 6.23076923,  7.26923077,  8.30769231]],
11
12        [[ 9.34615385, 10.38461538, 11.42307692],
13         [12.46153846, 13.5         , 14.53846154],
14         [15.57692308, 16.61538462, 17.65384615]],
15
16        [[18.69230769, 19.73076923, 20.76923077],
17         [21.80769231, 22.84615385, 23.88461538],
18         [24.92307692, 25.96153846, 27.          ]]])
19
20 In [5]: linha(w,3)
21 Out[5]: 'Isto nao e uma matriz.'
```

Numpy - Exercício

Faça duas funções, *linha* e *coluna*, cada uma recebendo como parâmetro de entrada um array M e um número inteiro i . O formato (n, m) do array M não é passado como parâmetro de entrada. As funções devem retornar, respectivamente, a linha e a coluna i de M . Se M não possui linha (coluna) i , a mensagem "Nao existe esta linha (coluna)" deve ser retornada. Caso M não tenha o formato (n, m) , a mensagem "Isto nao e uma matriz." deve ser retornada.

Exemplo

```
1 In [1]: z
2 Out[1]:
3 array([[ 0.,          1.11111111,  2.22222222,  3.33333333,  4.44444444],
4        [ 5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.          ]])
5
6 In [2]: w
7 Out[2]:
8 array([[[ 0.,          1.03846154,  2.07692308],
9         [ 3.11538462,  4.15384615,  5.19230769],
10        [ 6.23076923,  7.26923077,  8.30769231]],
11
12        [[ 9.34615385, 10.38461538, 11.42307692],
13         [12.46153846, 13.5         , 14.53846154],
14         [15.57692308, 16.61538462, 17.65384615]],
15
16        [[18.69230769, 19.73076923, 20.76923077],
17         [21.80769231, 22.84615385, 23.88461538],
18         [24.92307692, 25.96153846, 27.          ]]])
19
20 In [6]: coluna(w,4)
21 Out[6]: 'Isto nao e uma matriz.'
```

Numpy - Criando Matriz Identidade

Criadas através de **np.identity** ou **np.eye**

```
1 In [2]: z = np.identity(2)
2 In [3]:
3 Out[3]:
4 array([[ 1.,  0.],
5        [ 0.,  1.]])
6
7 In [4]: w = np.eye(2, dtype='int32')
8 In [5]: w
9 Out[5]:
10 array([[1, 0],
11        [0, 1]], dtype=int32)
```

Cuidado, tais funções não são equivalentes:

- **np.identity(n, dtype=None)**: retorna um array com valores 1 na diagonal principal.
- **np.eye(N,M=None,k=0,dtype=<type 'float'>)**: retorna um array com N linha, M colunas (opcional), com os elementos da diagonal *k* iguais a 1.

```
1 In [2]: np.eye(3, k=1)
2 array([[ 0.,  1.,  0.],
3        [ 0.,  0.,  1.],
4        [ 0.,  0.,  0.]])
```

Numpy - Funções

```
1 In [2]: A = np.array([4, 3, 2, 1])
2 In [3]: A
3 Out[3]: array([4, 3, 2, 1])
```

A.sort()	array([1, 2, 3, 4])
A.sum()	10
A.mean()	2.5
A.max()	4
A.argmax()	3

A.cumsum()	array([1, 3, 6, 10])
A.cumprod()	array([1, 2, 6, 24])
A.var()	1.25
A.std()	1.1180339887498949
A.shape = (2, 2) A.T	array([[1, 3],[2, 4]])

Numpy - Copiando Arrays

Arrays são tipos de dados *mutáveis*:

```
1 In [2]: a = np.array([1, 2])
2 In [3]: a
3 Out[3]: array([1, 2])
4
5 In [4]: a[0] = 3 # Mudamos o primeiro elemento
6 In [5]: a
7 Out[5]: array([3, 2])
```

CUIDADO:

```
1 In [2]: a = np.array([1, 2, 3])
2 In [3]: a
3 Out[3]: array([1, 2, 3])
4
5 In [4]: b = a
6 In [5]: b
7 Out[5]: array([1, 2, 3])
8
9 In [6]: b[0] = 4 # Mudamos o primeiro elemento
10 In [7]: b
11 Out[7]: array([4, 2, 3])
12
13 In [8]: a
14 Out[8]: array([4, 2, 3]) # array a também é afetado pela mudança
```


Numpy - Copiando Arrays

Fazendo cópias independentes:

```
1 In [2]: a = np.array([1, 2, 3])
2 In [3]: a
3 Out[3]: array([1, 2, 3])
4
5 In [4]: b = np.empty_like(a) # array vazio com o mesmo formato de a
6 In [5]: b
7 Out[5]: array([4607182418800017408, 4611686018427387904, 4613937818241073152])
8
9 In [6]: np.copyto(b, a) # copia a para b
10 In [7]: b
11 Out[7]: array([1, 2, 3])
12
13 In [8]: b[0] = 4 # Mudamos o primeiro elemento
14 In [9]: b
15 Out[9]: array([4, 2, 3])
16
17 In [10]: a
18 Out[10]: array([1, 2, 3]) # array a nao e afetado pela mudanca
19
20 In [11]: b[:] = 1
21 In [12]: b
22 Out[12]: array([ 1,  1,  1])
```

Numpy - Universal Functions

Podemos aplicar funções como *log*, *exp*, *sin*, etc diretamente em arrays:

```
1 In [2]: z = np.array([1, 2, 3])
2         n = len(z)
3         y = np.empty(n)
4         for i in range(n):
5             y[i] = np.sin(z[i])
6 In [3]: y
7 Out[3]: array([ 0.84147098,  0.90929743,  0.14112001])
```

Podemos simplificar:

```
1 In [2]: z = np.array([1, 2, 3])
2
3 In [3]: np.sin(z)
4 Out[3]: array([ 0.84147098,  0.90929743,  0.14112001])
```

É possível construir funções mais complexas:

```
1 In [2]: z = np.array([1, 2, 3])
2 In [3]: z
3 Out[3]: array([1, 2, 3])
4
5 In [3]: (1 / np.sqrt(2 * np.pi)) * np.exp(- 0.5 * z**2)
6 Out[3]: array([ 0.24197072,  0.05399097,  0.00443185])
```

Tais funções também são chamadas de *funções vetorizadas*.

Numpy - Vetorizando funções

Funções definidas pelo usuário não são vetorizadas (por default):

```
1 import numpy as np
2
3 def f(x):
4     if x > 0:
5         valor = 1
6     else:
7         valor = 0
8     return valor
```

```
1 In [2]: x = np.array([1, -2, 3])
2 In [3]: x
3 Out[3]: array([1, -2, 3])
4
5 In [4]: f(x)
6 Traceback (most recent call last):
7   File "<ipython-input-48-963bfc488912>", line 1, in <module> f(x)
8   File "/home/joao/exercicio.py", line 6, in f
9     if x > 0:
10 ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

Numpy - Vetorizando funções

Funções definidas pelo usuário não são vetorizadas (por default):

```
1 import numpy as np
2
3 def f(x):
4     if x > 0:
5         valor = 1
6     else:
7         valor = 0
8     return valor
```

Precisamos vetorizar a função f :

```
1 In [2]: x = np.array([1, -2, 3])
2 In [3]: x
3 Out[3]: array([1, -2, 3])
4
5 In [4]: f = np.vectorize(f)
6 In [5]: f(x)
7 Out[5]: array([1, 0, 1])
```

Nem sempre a vetorização usada desta forma é eficiente (tempo).

Numpy - Comparações

Podemos comparar arrays (elemento por elemento) usando `==`, `!=`, `>`, `<`, `>=`, `<=`:

```
1 In [2]: a = np.array([2, 3])
2 In [3]: b = np.array([2, 3])
3 In [4]: a == b
4 Out[4]: array([ True,  True], dtype=bool)
5
6 In [5]: b[0] = 5
7 In [6]: a == b
8 Out[6]: array([ False,  True], dtype=bool)
9
10 In [7]: a != b
11 Out[7]: array([ True,  False], dtype=bool)
```

E podemos comparar com escalares:

```
1 In [2]: z = np.linspace(0, 10, 5)
2 In [3]: z
3 Out[3]: array([ 0. ,  2.5,  5. ,  7.5, 10.])
4
5 In [4]: z > 3
6 Out[4]: array([ False,  False,  True,  True,  True], dtype=bool)
```

Computação II - Python

Aula 7 - Numpy

João C. P. da Silva

Carla A. D. M. Delgado

Dept. Ciência da Computação - UFRJ