Tutorial_Numpy

April 9, 2018

1 Numpy de forma Prática e fácil

1.0.1 Parte I - O básico

O objeto principal do NumPy é o array multidimensional homogêneo. É uma tabela de elementos (geralmente números), todos do mesmo tipo, indexados por uma tupla de inteiros positivos. No NumPy, as dimensões são chamadas de eixos. O número de eixos é rank.

Por exemplo, as coordenadas de um ponto no espaço 3D [1, 2, 1] é uma matriz de classificação 1, porque tem um eixo. Esse eixo tem um comprimento de 3. No exemplo abaixo, o array tem o rank 2 (é bidimensional). A primeira dimensão (eixo) tem um comprimento de 2, a segunda dimensão tem um comprimento de 3.

```
In [1]: [[ 1., 0., 0.],
         [ 0., 1., 2.]]
```

```
Out[1]: [[1.0, 0.0, 0.0], [0.0, 1.0, 2.0]]
```

A classe de array do NumPy é chamada de ndarray. Também é conhecido pelo array alias. Observe que numpy.array não é o mesmo que a classe array.array da Standard Python Library, que manipula apenas matrizes unidimensionais e oferece menos funcionalidade.

1.0.2 Exemplos

1 - importação da classe.

```
In [2]: import numpy as np
```

ndarray.arange

cria uma sequencia de numeros começando em 0 até o número definido,por padrão, ou pode ser definido a sequencia, expecificando onde começa e onde termina, onde essa função retorna array em vez de listas.

ndarray.reshape

reshape define a dimenção do array e o rank do proprio.

```
In [3]: a = np.arange(15).reshape(3, 5)
        a
```

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

```
In [4]: type(a)

Out[4]: numpy.ndarray
```

ndarray.shape

as dimensões da matriz. Esta é uma tupla de inteiros indicando o tamanho da matriz em cada dimensão. Para uma matriz com n linhas e m colunas, a forma será (n, m). O comprimento da tupla de forma é, portanto, a classificação, ou o número de dimensões, ndim.

```
In [5]: a.shape

Out[5]: (3, 5)
```

ndarray.ndim

o número de eixos (dimensões) da matriz. No mundo do Python, o número de dimensões é chamado de classificação.

```
In [6]: a.ndim

Out[6]: 2
```

ndarray.size

o número total de elementos da matriz. Isso é igual ao produto dos elementos da forma.

```
In [7]: a.size

Out[7]: 15
```

ndarray.dtype

um objeto que descreve o tipo dos elementos na matriz. É possível criar ou especificar dtipos usando tipos padrão do Python. Além disso, o NumPy fornece seus próprios tipos. numpy.int32, numpy.int16 e numpy.float64 são alguns exemplos.

```
In [8]: a.dtype.name

Out[8]: 'int64'
```

ndarray.itemsize

o tamanho em bytes de cada elemento da matriz. Por exemplo, uma matriz de elementos do tipo float64 possui tamanho de item 8 (= 64/8), enquanto um tipo de complexo32 possui tamanho de item 4 (= 32/8). É equivalente ndarray.dtype.itemsize.

```
In [9]: a.itemsize

Out[9]: 8
```

ndarray.data

o buffer contendo os elementos reais da matriz. Normalmente, não precisaremos usar esse atributo porque acessaremos os elementos em uma matriz usando recursos de indexação.

```
In [10]: a.data

Out[10]: <memory at 0x7f4885a56558>

In [11]: b = np.array([6, 7, 8])

In [12]: type(b)

Out[12]: numpy.ndarray
```

1.0.3 Parte II - criação do array

forma mais simples e mais utilizada,o array transforma sequências de sequências em matrizes bidimensionais, sequências de sequências de sequências em matrizes tridimensionais e assim por diante.

```
In [13]: #a = np.array(1,2,3,4)    # Forma errada.
         a = np.array([1,2,3,4])
         b = np.array([(1.5,2,3), (4,5,6)])
```

O tipo da matriz também pode ser explicitamente especificado no momento da criação.

```
In [14]: c = np.array( [ [1,2], [3,4] ], dtype=complex )
```

Frequentemente, os elementos de uma matriz são originalmente desconhecidos, mas seu tamanho é conhecido. Portanto, o NumPy oferece várias funções para criar matrizes com conteúdo inicial de espaço reservado. Isso minimiza a necessidade de cultivar matrizes, uma operação cara.

A função zeros cria um array cheio de zeros, a função ones cria um array cheio de uns e a função empty cria um array cujo conteúdo inicial é aleatório e depende do estado da memória. Por padrão, o dtype da matriz criada é float64.

```
In [15]: d = np.zeros( (3,4) )
         d

Out[15]: array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]])

In [16]: e = np.ones( (2,3,4)) #o 2 adicionado nas entradas da função define a quantidade de arr
         e

Out[16]: array([[[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]],

                [[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]]])

In [17]: f = np.empty( (2,3) )
         f

Out[17]: array([[6.91444587e-310, 4.65827208e-310, 0.00000000e+000],
                [0.00000000e+000, 0.00000000e+000, 0.00000000e+000]])
```

também pode ser definido uma sequencia de números apartir da função arange() onde também pode definido as regras para a criação da sequencia, onde essas regras podem ser aplicadas tanto com números flutuantes quanto inteiros.

```
In [18]: np.arange( 10, 30, 5 ) # De 10 a 30, de 5 em 5 números.
```

```
Out[18]: array([10, 15, 20, 25])

In [19]: np.arange( 0, 2, 0.3 ) # De 0 a 2, de 0.3 e 0.3 números.

Out[19]: array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

Quando arange é usado com argumentos de ponto flutuante, geralmente não é possível prever o número de elementos obtidos, devido à precisão do ponto flutuante finito. Por esse motivo, geralmente é melhor usar a função linspace que recebe como argumento o número de elementos que queremos.

```
In [20]: np.linspace( 0, 2, 9 )                 # 9 números de 0 a 2

Out[20]: array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
```

1.0.4 Parte III - Operações Básicas

Operadores aritméticos em matrizes aplicam-se elementarmente, Assim uma nova matriz é criada e preenchida com o resultado.

Exemplos:

```
In [21]: a = np.array( [20,30,40,50] )
         b = np.arange( 4 )
         a,b

Out[21]: (array([20, 30, 40, 50]), array([0, 1, 2, 3]))

In [22]: a-b

Out[22]: array([20, 29, 38, 47])

In [23]: b**2

Out[23]: array([0, 1, 4, 9])

In [24]: 10*np.sin(a)

Out[24]: array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])

In [25]: a<35

Out[25]: array([ True,  True, False, False])
```

Ao contrário de muitas linguagens matriciais, o operador do produto * opera de maneira elementar nos arrays NumPy. O produto da matriz pode ser executado usando a função de ponto ou método:

```
In [26]: A = np.array( [[1,1],[0,1]] )
         B = np.array( [[2,0],[3,4]] )

In [27]: A*B # produto elementar
```

```
Out[27]: array([[2, 0],
                [0, 4]])

In [28]:  A.dot(B)  # produto matricial

Out[28]: array([[5, 4],
                [3, 4]])

In [29]: np.dot(A, B) # produto matricial

Out[29]: array([[5, 4],
                [3, 4]])
```

Algumas operações, como + = e * =, atuam para modificar uma matriz existente em vez de criar uma nova.

```
In [30]: A *= 3
         A

Out[30]: array([[3, 3],
                [0, 3]])

In [31]: A += 3
         A

Out[31]: array([[6, 6],
                [3, 6]])

In [32]: A += B
         A

Out[32]: array([[ 8,  6],
                [ 6, 10]])
```

Ao operar com matrizes de tipos diferentes, o tipo do array resultante corresponde ao array mais geral ou preciso (um comportamento conhecido como upcasting). Além de que muitas operações unárias, como calcular a soma de todos os elementos na matriz, são implementadas como métodos da classe ndarray.

```
In [33]: A.sum(),A.min(),A.max()

Out[33]: (30, 6, 10)
```

A expressão entre parêntesis em b [i] é tratada como um i seguido por tantos exemplos de: conforme necessário para representar os eixos restantes. O NumPy também permite que você escreva isso usando pontos como b [i, ...].

Os pontos (...) representam o número de dois pontos necessários para produzir uma tupla de indexação completa. Por exemplo, se x é uma matriz de classificação 5 (ou seja, tem 5 eixos), então

- x [1,2, ...] é equivalente a x [1,2,:,:,:]

- x [..., 3] para x [:,:,:,: 3] e
- x [4, ..., 5 ,:] para x [4,:,:, 5 ,:].

```
In [34]: A[1:,1], A[1,:], A[:,1]

Out[34]: (array([10]), array([ 6, 10]), array([ 6, 10]))
```

Usando o hsplit, você pode dividir um array ao longo de seu eixo horizontal, especificando o número de matrizes de forma igual a retornar ou especificando as colunas após as quais a divisão deve ocorrer:

```
In [35]: a = np.arange(24).reshape(2,12)
         np.hsplit(a,3)

Out[35]: [array([[ 0,  1,  2,  3],
                 [12, 13, 14, 15]]), array([[ 4,  5,  6,  7],
                 [16, 17, 18, 19]]), array([[ 8,  9, 10, 11],
                 [20, 21, 22, 23]])]

In [36]: A

Out[36]: array([[ 8,  6],
                [ 6, 10]])

In [37]: np.transpose(A)

Out[37]: array([[ 8,  6],
                [ 6, 10]])
```

Uma das melhores coisas na biblioteca é a otimização para uso matricial, se comparar uma função np.sum com um for normal para fazer uma somatoria em uma matriz de elementos do python podemos ver a diferença.

```
In [38]: V = np.arange(1500).reshape(3, 500)
         V

Out[38]: array([[   0,    1,    2, ...,  497,  498,  499],
                [ 500,  501,  502, ...,  997,  998,  999],
                [1000, 1001, 1002, ..., 1497, 1498, 1499]])

In [39]: import time

         inicio = time.time()
         np.sum(V)
         fim = time.time()
         print(fim - inicio)

0.00018167495727539062
```

In [40]:
```
import time

inicio = time.time()
result = 0
for line in V:
    for elem in line:
        result += elem
fim = time.time()
print(fim - inicio)
```

```
0.0008997917175292969
```