

NumPy

Introdução

NumPy

O que é o NumPy

O NumPy (Numerical Python - Python Numérico), é um dos pacotes básicos mais importantes para processamento numérico em Python. A maioria dos pacotes de processamento com funcionalidades científicas utiliza objetos array do NumPy para troca de dados.

Veja alguns recursos que você encontrará no NumPy:

- Narray: Um array multidimensional eficaz que oferece operações matemáticas rápidas, orientadas a arrays, e recursos flexíveis de broadcasting.
- Funções matemáticas para operações rápidas em arrays de dados inteiros, sem que seja necessário escrever laços.
- Ferramentas para ler/escrever dados de array em disco e trabalhar com arquivos mapeados em memória.
- Recursos para álgebra linear, geração de números aleatórios e transformadas de Fourier.
- Uma API C para conectar o NumPy a bibliotecas escritas em C, C++ ou FORTRAN.

NumPy

Array NumPy - ndarray

Um dos principais recursos do NumPy é seu objeto array N-dimensional (um array de N dimensões), ou ndarray, que é um contêiner rápido e flexível para conjuntos de dados grandes em Python.

Um ndarray é um contêiner genérico multidimensional para dados homogêneos. Isso significa que todos os elementos devem ser do mesmo tipo.

NumPy

Array NumPy - ndarray

Um vetor (array uni-dimensional) é uma variável que armazena vários valores do mesmo tipo. Para acessar um dos valores usamos o índice, começando por zero:

`a = x[1]`

Neste caso o valor de “a” é 20.

x
10
20
30
40

Lista com Python

```
>>> x = [10, 20, 30, 40]
>>> a = x[1]
>>> a
20
```

Array com NumPy

```
In [1]: import numpy as np
In [2]: x = np.array([10, 20, 30, 40])
In [3]: a = x[1]
In [4]: a
Out[4]: 20
```

import numpy as np

Devemos usar um alias para que não haja conflito entre funções do Python e do Numpy. Você pode usar qualquer alias/apelido, porém, é melhor usar a convenção.

Uma Matriz (array bi-dimensional) é um vetor de vetores:

`a = x[1][2]`

Neste caso o valor de “a” é 60.

	Eixo 1			
		0	1	2
E i x o 0	0	10 (0,0)	20 (0,1)	30 (0,2)
	1	40 (1,0)	50 (1,1)	60 (1,2)
	2	70 (2,0)	80 (2,1)	90 (2,2)
	3	100 (3,0)	110 (3,1)	120 (3,2)

```
In [18]: x = np.array([[10, 20, 30],[40, 50, 60],[70, 80, 90],[100, 110, 120]])
In [19]: a = x[1,2]
In [20]: a
Out[20]: 60
```



NumPy

Array NumPy - ndarray

Uma maneira fácil de criar um array é usando a função **array**. Ela aceita qualquer objeto do tipo sequência (incluindo outros arrays) e gera um novo array NumPy contendo os dados recebidos.

Todo array tem um **shape**, ou seja, uma tupla que indica o tamanho de cada dimensão, e um **dtype**, que é um objeto que descreve o tipo de dado do array. A menos que seja explicitamente especificado, `np.array` tentará inferir um bom tipo de dado para o array que ele criar. Esse tipo de dado é armazenado em `dtype` (conjunto de metadados).

```
In [30]: myList = [0, 1, 2, 3, 4, 5]
myArray = np.array(myList)
print(myArray)

[0 1 2 3 4 5]
```

```
In [31]: print(myArray.shape) # (6,)
print(myArray.dtype) # int32

(6,)
int32
```

NumPy

Array NumPy - ndarray

Podemos inicializar arrays NumPy de listas Python aninhadas e acessar elementos usando colchetes.

Exemplo de uma lista Python aninhada:

`[[1, 2, 3], [4, 5, 6]]`

Veja uma matriz de
2 linhas e 6 colunas.

```
In [60]: myList = [[0,1,2,3,4,5],[10,20,30,40,50,60]]
```

```
In [61]: myArray = np.array(myList)
```

```
In [62]: myArray
```

```
Out[62]:
```

```
array([[ 0,  1,  2,  3,  4,  5],  
       [10, 20, 30, 40, 50, 60]])
```

```
In [63]: myArray.shape
```

```
Out[63]: (2, 6)
```

NumPy

Array NumPy - ndarray

Veja mais alguns exemplos:

```
import numpy as np

myList = [[1, 2, 3], [4, 5, 6]]

# indice
#[      0      ,      1      ]
#[ [0][1][2], [0][1][2] ]

# Posicionamento
#[ [0,0][0,1][0,2],
#  [1,0][1,1][1,2] ]

# Valores
#[ [1][2][3], [4][5][6] ]
```

```
print(type(myList)) # <class 'list'>
myArray = np.array(myList)
print(type(myArray)) # <class 'numpy.ndarray'>
print(myArray.shape) # (2, 3)
print(myArray.dtype) # dtype('int32')
print(myArray[0][0], myArray[0][1], myArray[0][2]) # 1 2 3
print(myArray[1][0], myArray[1][1], myArray[1][2]) # 4 5 6
myArray[0][0] = 10 # Alterando o conteúdo da posição [0][0] que era "1" para "10"
print(myArray) # [[10  2  3] [ 4  5  6]]
a = myArray[0,0]
print(a) # 10
a = myArray[1,2]
print(a) # 6
```

Python tem um tipo inteiro, um tipo float e um tipo complexo. Contudo, isto não é o suficiente para computação científica e, por esse motivo, o NumPy tem muito mais tipos de dados com diferentes precisões, dependentes dos requisitos de memória.

Na prática, precisamos de mais tipos com precisão variável e, portanto, de diferentes tamanhos.

Tipos numéricos NumPy

A maioria dos tipos numéricos NumPy termina com um número. Este número indica o número de bits associados ao tipo. A tabela a seguir fornece uma visão geral dos tipos numéricos do NumPy:

Type	Description
<code>bool</code>	Boolean (True or False) stored as a bit
<code>inti</code>	Platform integer (normally either <code>int32</code> or <code>int64</code>)
<code>int8</code>	Byte (-128 to 127)
<code>int16</code>	Integer (-32768 to 32767)
<code>int32</code>	Integer (-2^{31} to $2^{31}-1$)
<code>int64</code>	Integer (-2^{63} to $2^{63}-1$)
<code>uint8</code>	Unsigned integer (0 to 255)
<code>uint16</code>	Unsigned integer (0 to 65535)
<code>uint32</code>	Unsigned integer (0 to $2^{32}-1$)
<code>uint64</code>	Unsigned integer (0 to $2^{64}-1$)
<code>float16</code>	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
<code>float32</code>	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
<code>float64</code> or <code>float</code>	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
<code>complex64</code>	Complex number, represented by two 32-bit floats (real and imaginary components)
<code>complex128</code> or <code>complex</code>	Complex number, represented by two 64-bit floats (real and imaginary components)

NumPy

Comparando velocidade de listas Python com arrays NumPy

Os algoritmos baseados no NumPy são de 10 a 100 vezes mais rápidos (ou mais) do que suas contrapartidas em Python puro, além de utilizarem significativamente menos memória.

Vamos fazer um teste usando a função mágica %time para conferirmos a velocidade da função soma do Python sobre uma lista e da função soma do NumPy sobre um array.

```
In [6]: myList = list(range(9999999))
        myArray = np.arange(9999999)

        %time sum(myList)
        %time np.sum(myArray)

        Wall time: 332 ms
        Wall time: 5.01 ms
```

Funções para criação de arrays

O NumPy também fornece funções para criar arrays com valores preenchidos.

- `ones`, `ones_like`: gera um array preenchido com uns (1s) com o formato e dtype especificados. `ones_like` aceita outro array e gera um array de uns com o mesmo formato e dtype.
- `zeros`, `zeros_like`: similar a `ones` e `ones_like`, porém, gera arrays com zeros (0s).

```
In [5]: a = np.ones((3, 2)) # Cria um array 3x2 preenchido com números 1.  
print(a)  
  
[[1. 1.]  
 [1. 1.]  
 [1. 1.]]
```

```
In [4]: a = np.zeros((3, 2)) # Cria um array 3x2 preenchido com zeros.  
print(a)  
  
[[0. 0.]  
 [0. 0.]  
 [0. 0.]]
```

Funções para criação de arrays

- `full`, `full_like`: gera um array com o formato e o dtype especificados, preenchendo com o valor informado, `full_like` aceita outro array e gera um array preenchido com o mesmo formato e dtype.
- `eye`, `identity`: cria uma matriz-identidade quadrada $N \times N$ (1s na diagonal e 0s nas demais posições).

```
In [6]: a = np.full((3, 2), 7) # Cria um array 3x2 preenchido com números 7.  
print(a)  
  
[[7 7]  
 [7 7]  
 [7 7]]
```

```
In [7]: a = np.eye(4) # Cria um array 4x4 2d com números "1" na diagonal e o restante com zero.  
print(a)  
  
[[1. 0. 0. 0.]  
 [0. 1. 0. 0.]  
 [0. 0. 1. 0.]  
 [0. 0. 0. 1.]]
```

NumPy

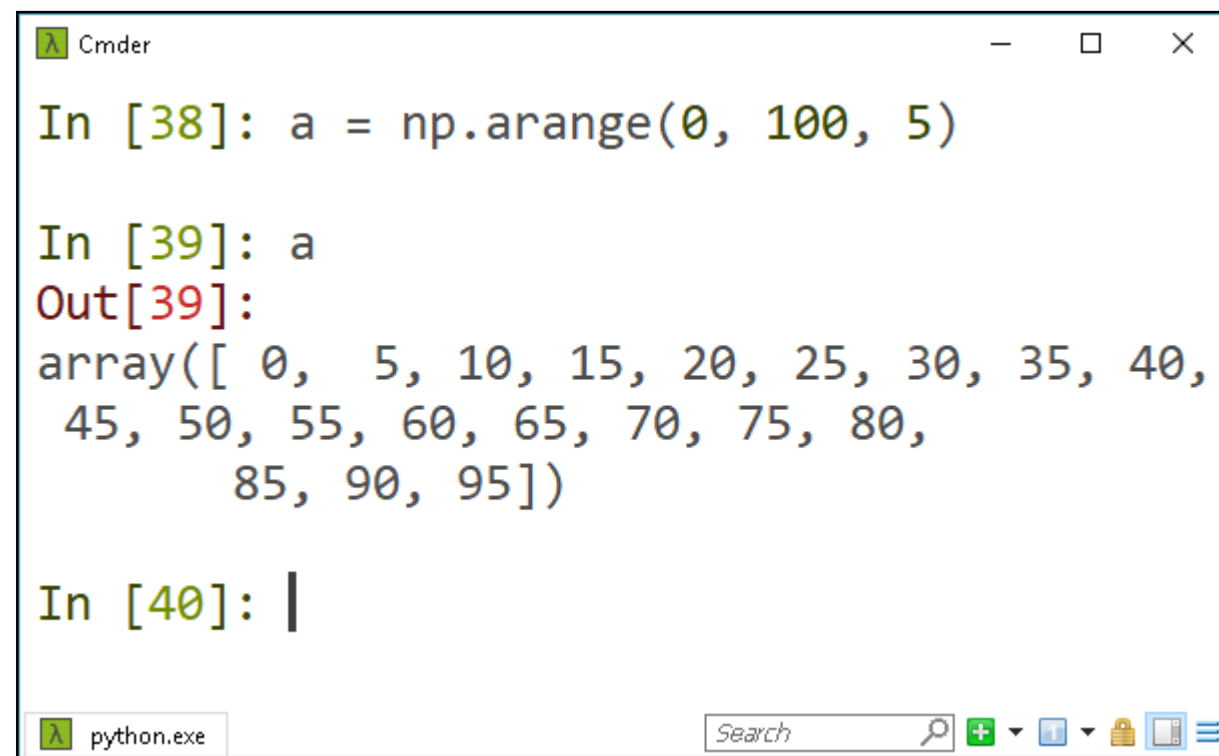
Funções para criação de arrays

- arange: é uma versão da função range de Python com valor de array.

```
In [5]: a = np.arange(10)
        print(a)

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

Usando início, fim e incremento (passos):



```
Cmder

In [38]: a = np.arange(0, 100, 5)

In [39]: a
Out[39]:
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40,
        45, 50, 55, 60, 65, 70, 75, 80,
        85, 90, 95])

In [40]: |

python.exe Search
```

CONTINUA...