NumPy

O que é o NumPy?

O NumPy é uma poderosa biblioteca Python que é usada principalmente para realizar cálculos em Arrays Multidimensionais. O NumPy fornece um grande conjunto de funções e operações de biblioteca que ajudam os programadores a executar facilmente cálculos numéricos. Esses tipos de cálculos numéricos são amplamente utilizados em tarefas como:

Tarefas matemáticas: NumPy é bastante útil para executar várias tarefas matemáticas como integração numérica, diferenciação, interpolação, extrapolação e muitas outras. O NumPy possui também funções incorporadas para álgebra linear e geração de números aleatórios. É uma biblioteca que pode ser usada em conjuto do SciPy e Matplotlib, substituindo o MATLAB quando se trata de tarefas matemáticas.

Processamento de Imagem e Computação Gráfica: Imagens no computador são representadas como Arrays Multidimensionais de números. NumPy torna-se a escolha mais natural para o mesmo. O NumPy, na verdade, fornece algumas excelentes funções de biblioteca para rápida manipulação de imagens. Alguns exemplos são o espelhamento de uma imagem, a rotação de uma imagem por um determinado ângulo etc.

Modelos de Machine Learning: Ao escrever algoritmos de Machine Learning, supõe-se que se realize vários cálculos numéricos em Array. Por exemplo, multiplicação de Arrays, transposição, adição, etc. O NumPy fornece uma excelente biblioteca para cálculos fáceis (em termos de escrita de código) e rápidos (em termos de velocidade). Os Arrays NumPy são usados para armazenar os dados de treinamento, bem como os parâmetros dos modelos de Machine Learning.

Instalando

Existem diversas formas de instalar o numpy. A mais simples é instalar o pacote Anaconda (https://www.anaconda.com/distribution/) que já vem com o Python e diversas bibliotecas científicas e ciência de dados instaladas.

Outra forma, caso você já tenha o python instalado mas não o numpy, é o utilizar o gerenciador e pacotes `pip`, através do comando no seu terminal:

```
$ pip install numpy
```

Explorando a API do NumPy

Importando numppy com um alias `np`

`np` é uma abreviação amplamente utilizada na comunidade python para o numpy.

```
import numpy as np
```

1D arrays

Array unidimensional, também chamado de vetor ou até mesmo matriz de 1 dimensão:

```
>>> a = np.array([1, 2, 3])
array([1, 2, 3])
```

Checando o tipo da variável a:

```
>>> type(a)
numpy.ndarray
```

o nd significa n-dimensional

Checando o tipo de dados do array

Diversos tipos de dados são possíveis em um array numpy, os mais comuns são os numéricos:

- int32
- int64
- float32
- float64

```
>>> a.dtype
dtype('int64')
```

A consistência do dado é forte...

Se trocarmos um elemento na posição 0 para o valor 10, dará certo:

```
>>> a[0] = 10
>>> a
array([10,  2,  3])
```

Se trocarmos para ponto flutuante, o numpy irá truncar a parte decimal, dado que o array que criamos é inteiro.

```
>>> a[0] = 1.2
>>> a
array([1, 2, 3])
```

2D arrays

Matrizes podem ser consideradas um `array` de 2 dimensões.

Observação

O NumPy possui também uma estrutura, `matrix`, mas não é recomendado utilizá-la pela própria documentação oficial e poderá ser removida no futuro.

Para criar uma matriz, basta aninhar múltiplas listas dentro de uma lista, como o exemplo a seguir:

```
>>> b = np.array([[9.0, 8.0, 7.0],
                  [6.0, 5.0, 4.0]])
>>> b
array([[9., 8., 7.],
       [6., 5., 4.]])
```

Propriedades

Dimensão e formato

Dois conceitos importantes já mencionados acima é o de dimensão e formato.

Para descobrir essas informações, basta acessar os atributos `ndim` e `shape`

```
>>> a.ndim
1


>>> a.shape
(3,)


>>> b.ndim
2


b.shape
(2, 3)
```

Tipo de dado e tamanho

Na sessão **Checando o tipo de dados do array** já foi dito dos tipos de dados, mas agora falaremos da diferença de tamanhos que isso ocupa na memória.

Então, temos as variáveis `a` e `b` criadas anteriormente com os seguintes tipos:

```
>>> a.dtype
dtype('int64')


>>> b.dtype
dtype('float64')
```

Por padrão, se o python instalado é 64 bits, ele irá criar tipos `int` ou `float` de 64 bits. Caso seu python fosse 32 bits, seria `int32` e `float32`.

Vamos criar uma outra array, `a16`, com o tipo inteiro de 16 bits.

```
>>> a16 = np.array([1, 2, 3], dtype=np.int16)
>>> a16
array([1, 2, 3], dtype=int16)
```

Note que por ser um tipo diferente do padrão, ele ressalta ao imprimir.

Para descobrir quanto cada elemento individualmente ocupa na memória, podemos acessar o atributo `itemsize`:

```
>>> a.itemsize #
8
```

Ele retorna 8 e não 64! Isso é porque ele já converteu os bits para bytes. Bytes é o conjunto de 8 bits.

Logo:

$$\frac{64}{8} = 8$$

Já nosso array int16, temos:

```
>>> a16.itemsize
2
```

Dado que:

$$\frac{16}{8} = 2$$

```
# quantidade de elementos total
>>> a.size
3
```

Quantidade de elementos vezes o tamanho de cada elemento nos dará o tamanho total de bytes que o array inteiro ocupa:

```
>>> a.size * a.itemsize
24
```

Mas ao invés de calcular isso, podemos simplesmente acessar o atributo `nbytes`, que já é o tamanho total de bytes ocupado pelo array:

```
>>> a.nbytes
24
```

Observação

Geralmente não é necessário em reduzir o número de bits a não ser que você tenha certeza que um tamanho reduzido vai atender sua necessidade e você quer ser **extremamente** eficiente.

Acessando e modificando elementos (Indexing & Slicing)

Dada a matriz a abaixo:

```
>>> a = np.array([[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14]])
>>> print(a)
[[ 1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14]]
```

Podemos acessar um elemento específico de forma similar a lista, utilzando a sintaxe de colchetes, com a diferença de que podemos separar cada posição com vírgulas.

Para uma array 2D, temos então a sintaxe:

$$[linha, coluna]$$

Exemplo:

```
>>> a[1, 5]
13
```

Podemos (menos comum) fazer dessa forma também:

```
>>> a[1][5]
13
```

Usar números negativsos funciona de trás pra frente a indexação:

```
>>> a[1, -2]
13
```

Pegar uma linha específica, podemos utilizar a sintaxe de `:` que pode ser lida como "todos" daquela dimensão (colunas).

Podemos ler então como: linha zero, todas as colunas

```
>>> a[0, :]
array([1, 2, 3, 4, 5, 6, 7])
```

Também podemos fazer assim simplesmente:

```
>>> a[0]
array([1, 2, 3, 4, 5, 6, 7])
```

Porém, para coluna específica não tem jeito, precisamos usar `:`.

Leia-se: todas as linhas, coluna 2

```
>>> a[:, 2]
array([ 3, 10])
```

O operador `:` também conhecido como slicing, aceita o parâmetro:

- start
- end
- step

No formato

[startindex:endindex:stepsize]

O stepsize basicamente é quantos elementos deve ser pular. Podemos pegar do elemento 1 ao 6 pulando de 2 em 2 por exemplo da linha 0.

```
>>> a[0, 1:6:2]
array([2, 4, 6])
```

Funciona com negativo também:

```
a[0, 1:-1:2]
>>> array([2, 4, 6])
```

Para mudar um elemento específico, basta usar o operador `=`:

```
>>> a[1,5] = 20
>>> print(a)
[[ 1  2  3  4  5  6  7]
 [ 8  9 10 11 12 20 14]]
```

Mudando uma coluna inteira para ser 5:

```
>>> a[:, 2] = 5
>>> print(a)
[[ 1  2  5  4  5  6  7]
 [ 8  9  5 11 12 20 14]]
```

Isso mostra uma característica fundamental do array do NumPy:

Ao alterar o pedaço da matriz recortada, você altera a matriz original

Slicing em listas geram cópias!

```
>>> a = [1, 2, 3]
>>> b = a[1:]
>>> b
[2, 3]
>>> b = [10, 11]
>>> a
[1, 2, 3]
>>> b
[10, 11]
```

Acessando o formato de um slicing:

```
>>> a[:, 2].shape
(2,)
```

```
>>> a[:, 2] = [5, 10]
>>> print(a)
[[ 1  2  5  4  5  6  7]
 [ 8  9 10 11 12 20 14]]
```

Exemplo 3D

```
>>> b = np.array([[[1, 2], [3,4]], [[5, 6], [7, 8]]])
>>> b
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

Checando a dimensão:

```
>>> b.ndim
3
```

Retirando o elemento 4:

```
>>> b[0, 1, 1]
4
```

Pegando todos todos os elementos da posição 1 da dimensão 2:

```
>>> b[:, 1, :]
array([[3, 4],
       [7, 8]])
```

Substituindo:

```
b[:, 1, :] = [[9, 9], [8, 8]]
```

Inicializando arrays usando métodos internos

O NumPy já possui diversos métodos built-in para gerar arrays dos mais diversos tipos

array apenas com zeros

```
>>> np.zeros(5)
array([0., 0., 0., 0., 0.])
```

É possível gerar um array de qualquer formato, basta apenasr passar o formato como uma sequência (lista, tupla geralmente) como argumento

```
>>> np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
>>> np.zeros((2, 2, 3))
array([[[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]]])
```

array apenas com uns

```
>>> np.ones((4, 2, 2), dtype='int32')
array([[[1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1]]], dtype=int32)
```

Um coisa muito comum é usar o `np.ones` para criar uma matriz de qualquer número fazendo a operação, exemplo:

```
>>> np.ones((2, 2)) * 10
array([[10., 10.],
       [10., 10.]])
```

Mas o numpy já tem uma opção mais elegante, o `full`:

```
>>> np.full((2, 2), 99)
array([[99, 99],
       [99, 99]])
```

Qualquer outro número copiando o formato de uma matriz existente

```
>>> np.full_like(a, 4)
array([[4, 4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4, 4]])
```

Números decimais aleatórios

O numpy tem um sub-módulo chamado `random`, que pode ser acessando via `np.random`.

Nele, tem um conjunto de funções para números aleatórios.

```
>>> np.random.random_sample((4, 2))
array([[0.59701743, 0.40627135],
       [0.23940108, 0.58528558],
       [0.04228252, 0.68467361],
       [0.68907347, 0.85636431]])
```

Função igual a de cima, mas padrão do MATLAB (observe que não é uma tupla/lista o formato):

```
>>> np.random.rand(4, 2)
array([[0.13322589, 0.76493887],
       [0.06716679, 0.61834428],
       [0.54098045, 0.09752974],
       [0.6656696 , 0.79337782]])
```

Números inteiros:

```
>>> np.random.randint(7, size=(3, 3))
array([[6, 5, 6],
       [0, 6, 2],
       [4, 6, 5]])
```

Os argumentos principais são `low`, `high` e `size`, exemplo: criando uma matriz de 0 a 99 de 100 elementos:

```
>>> np.random.randint(0, high=100, size=100)
   array([66, 91, 70, 95, 73, 73, 70, 98, 10, 70, 47, 11, 67, 69, 60, 30,
23,
          51, 58, 18, 79, 46,  8, 12, 84, 45, 44, 90, 13, 86, 92, 20,  9,
93,
          41,  7, 56, 39, 11, 58, 36, 94, 80, 65, 94,  9, 20, 47, 74,  5,
90,
          28,  3, 86, 29, 51, 83, 44, 58, 33, 55, 69,  9, 15, 54, 87, 87,
68,
          35,  6, 91, 98, 99, 81, 73, 37, 86, 79, 91, 65, 44, 22, 68,  5,
85,
          28, 49, 70, 64, 23, 23, 73, 94, 56, 76, 11, 12, 21, 14, 48])
```

Para incluir o 100, basta trocar o high por 101

Matriz identidade

Diagonal inteira com 1. É sempre uma matriz quadrada.

```
>>> np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

repeat

Método para repetir uma determinada array na direção do eixo escolhido.

Esse é a primeira função, de várias, que possui o parâmetro `axis`.

Diversas vezes o numpy permite fazer uma operação, nesse caso, `repeat` , no qual é opcional ou necessário dizer qual o eixo da operação.

Para um vetor de 1D, temos apenas 1 eixo, mas para matrizes, tempos dois:

O eixo 0 é linha, o eixo 1 é coluna

```
>>> arr = np.array([1, 2, 3])
>>> r1 = np.repeat(arr, 3)
>>> print(r1)
[1 1 1 2 2 2 3 3 3]
```

Com axis = 0

```
>>> arr = np.array([[1, 2, 3]])
>>> r1 = np.repeat(arr, 3, axis=0)
>>> print(r1)
[[1 2 3]
 [1 2 3]
 [1 2 3]]
```

Com axis = 1

```
>>> arr = np.array([[1, 2, 3]])
>>> r1 = np.repeat(arr, 3, axis=1)
>>> print(r1)
[[1 1 1 2 2 2 3 3 3]]
```

arange

Função que retorna elementos igualmente espaçados num step (por padrão, 1) dentro de um certo intervalo.

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

Step diferente de 1:

```
>>> np.arange(0, 11, 0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
        2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,
        3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4.3,
        4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,  5.4,
        5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3,  6.4,  6.5,
        6.6,  6.7,  6.8,  6.9,  7. ,  7.1,  7.2,  7.3,  7.4,  7.5,  7.6,
```

```
       7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,  8.7,
       8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5,  9.6,  9.7,  9.8,
       9.9, 10. , 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9])
```

linspace

Parecido com o arange, mas você diz quantos pontos você quer e o intervalo e ele define o espaçamento linear

```
>>> np.linspace(0, 100, num=10)
array([  0.        ,  11.11111111,  22.22222222,  33.33333333,
       44.44444444,  55.55555556,  66.66666667,  77.77777778,
       88.88888889, 100.        ])
```

Tenha cuidado ao copiar arrays!

Jeito errado

```
>>> a = np.array([1, 2, 3])
>>> b = a
>>> b
array([1, 2, 3])


>>> b[0] = 100
>>> b
array([100,   2,   3])
>>> a
array([100,   2,   3])
```

a também foi modificado!

Jeito certo (seguro)

```
>>> a = np.array([1, 2, 3])
>>> b = a.copy()
>>> b
array([1, 2, 3])
b[0] = 100
>>> b
array([100,   2,   3])
>>> a
array([1, 2, 3])
```

Matemática

O numpy te fornece um conjunto de funções matemáticas:

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
```

Operação com escalares

Soma:

```
>>> a + 2
array([3, 4, 5, 6])
```

Subtração:

```
>>> a - 2
array([-1,  0,  1,  2])
```

Multiplicação:

```
>>> a * 2
array([2, 4, 6, 8])
```

Divisão:

```
>>> a / 2
array([0.5, 1. , 1.5, 2. ])
```

Incrementar:

```
>>> a += 2
>>> a
array([3, 4, 5, 6])
```

Potência:

```
>>> a ** 2
array([ 1,  4,  9, 16])
```

Operação entre arrays

Tudo que você consegue fazer com escalar, você consegue fazer com arrays elemento-a-elemnto, por exemplo, para soma:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([1, 0, 1, 0])
>>> a + b
array([2, 2, 4, 4])
```

Funções matemáticas

Função seno:

```
>>> np.sin(a)
array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

Função cosseno:

```
>>> np.cos(a)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
```

Álgebra Linear

Da definição do Wikipédia:

> *Álgebra linear é um ramo da matemática que surgiu do estudo detalhado de sistemas de equações lineares, sejam elas algébricas ou diferenciais. A álgebra linear utiliza alguns conceitos e estruturas fundamentais da matemática como vetores, espaços vetoriais, transformações lineares, sistemas de equações lineares e matrizes.*

O numpy nos permite executar diversas diversas operações de álgebra linear, mostradas a seguir:

```
>>> a = np.ones((2, 3))
>>> print(a)
[[1. 1. 1.]
 [1. 1. 1.]]

>>> b = np.full((3, 2), 2)
>>> print(b)
[[2 2]
 [2 2]
 [2 2]]
```

Transposição

A operação de transposição

$$A = \begin{bmatrix} 1 & 5 \\ 3 & 2 \\ 4 & 12 \\ 7 & 9 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 5 & 2 & 12 & 9 \end{bmatrix}$$

Pode ser feita da seguinte forma:

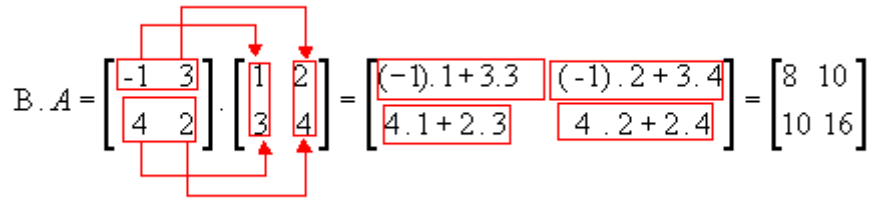
```
>>> np.transpose(a)
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

Ou acessando o atributo `T`:

```
>>> a.T
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

Multiplicação de matrizes

A tradicional multiplicação de matrizes, como mostra a imagem abaixo:



Pode ser feita no numpy simplesmente chamando `matmul`

```
>>> np.matmul(a, b)
array([[6., 6.],
       [6., 6.]])
```

Operador `@` executa a função anterior:

```
>>> a @ b
array([[6., 6.],
       [6., 6.]])
```

Encontrar o determinante

```
>>> c = np.identity(3)
>>> print(c)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>> np.linalg.det(c)
1.0
```

Outras funcções de Álgebra Linear:
https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

- Trace
- Decomposição de vetores
- Autovalor/autovetor
- Norma da Matriz
- Inversa
- Etc...

Estatística

O numpy vem com várias funções básicas de estatística, como mínimo, máximo, média, mediana, etc.

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


>>> np.min(stats)
1


>>> np.max(stats)
6
```

Mínimo por linha:

```
>>> np.min(stats, axis=1)
array([1, 4])
```

Máximo por coluna:

```
>>> np.max(stats, axis=0)
array([4, 5, 6])
```

Soma por coluna:

```
>>> np.sum(stats, axis=0)
array([5, 7, 9])
```

Média:

```
>>> np.average(stats)
3.5
```

Reorganizar Array

Muitas vezes você quer mudar o formato de array, por exemplo, de 4 elementos pra uma matriz 2x2, ou situações similares.

Para isso, você pode utilizar a função `reshape`.

Reshape

```
>>> before = np.array([[1,2,3,4],[5,6,7,8]])
>>> print(before.shape)
(2, 4)
```

```
>>> after = before.reshape((8, 1)) # tem que possuir a mesma quantidade!
>>> after
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7],
       [8]])
```

```
>>> before.reshape(2, 2, 2)
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

Anexar verticalmente os vetores

```
v1 = np.array([1,2,3,4])
v2 = np.array([5,6,7,8])
```

```
>>> np.vstack([v1, v2])
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])


>>> np.vstack([v1, v2, v2, v2])
array([[1, 2, 3, 4],
       [5, 6, 7, 8],
       [5, 6, 7, 8],
       [5, 6, 7, 8]])
```

Apendar horizontalmente os vetores

De forma similar ao anterior:

```
>>> h1 = np.ones((2, 4))
>>> h2 = np.zeros((2, 2))
>>> np.hstack((h1, h2))
array([[1., 1., 1., 1., 0., 0.],
       [1., 1., 1., 1., 0., 0.]])
```

Funcionalidades extras

Carregar dados de um arquivo

Vamos supor que temos um arquivo data.txt com o seguinte conteúdo:

```
1,13,21,11,196,75,4,3,34,6,7,8,0,1,2,3,4,5
3,42,12,33,766,75,4,55,6,4,3,4,5,6,7,0,11,12
1,22,33,11,999,11,2,1,78,0,1,2,9,8,7,1,76,88
```

Podemos gerar uma matriz a partir desse arquivo da seguinte forma:

```
>>> filedata = np.genfromtxt('data.txt', delimiter=',')
>>> filedata
array([[  1.,  13.,  21.,  11., 196.,  75.,   4.,   3.,  34.,   6.,   7.,
          8.,   0.,   1.,   2.,   3.,   4.,   5.],
       [  3.,  42.,  12.,  33., 766.,  75.,   4.,  55.,   6.,   4.,   3.,
          4.,   5.,   6.,   7.,   0.,  11.,  12.],
       [  1.,  22.,  33.,  11., 999.,  11.,   2.,   1.,  78.,   0.,   1.,
          2.,   9.,   8.,   7.,   1.,  76.,  88.]])
```

O primeiro argumento é o nome do arquivo.

Importante ressaltar o segundo key argumento, `delimiter`, no qual você especifica o que separa cada número individualmente no arquivo. Nesse caso, vírgula, mas podería ser `;` por

exemplo, espaços, ou tabs.

Podemos notar também que o numpy converteu para float nossos números, apesar de todos serem inteiros. Ele faz isso como uma medida preventiva dado que ele não sabe ao ler o arquivo qual tipo de dado que é.

Podemos converter manualmente para inteiro usando a função `astype`:

```
>>> filedata.astype('int32')
>>> array([[  1,  13,  21,  11, 196,  75,   4,   3,  34,   6,   7,   8,   0,
          1,   2,   3,   4,   5],
       [  3,  42,  12,  33, 766,  75,   4,  55,   6,   4,   3,   4,   5,
          6,   7,   0,  11,  12],
       [  1,  22,  33,  11, 999,  11,   2,   1,  78,   0,   1,   2,   9,
          8,   7,   1,  76,  88]], dtype=int32)
```

Podemos também salvar uma matriz de uma forma mais otimizada não textual (binária) para uso futuro.

Isso gera um arquivo binário que inclusive salva o tipo de dado, nesse caso, int32.

Quando lido, vai converter corretamnete o tipo daquele dado.

```
np.save('data', filedata.astype('int32'))
```

Igual ao método de cima, porém comprime os dados (economiza espaço em disco, porém é um pouco mais lento pra ler).

```
np.savez_compressed('dataz', filedata)
```

Para ler os dados que acabamos de salvar, basta usar o `np.load`:

```
np.load('data.npy')
array([[  1,  13,  21,  11, 196,  75,   4,   3,  34,   6,   7,   8,   0,
          1,   2,   3,   4,   5],
       [  3,  42,  12,  33, 766,  75,   4,  55,   6,   4,   3,   4,   5,
          6,   7,   0,  11,  12],
       [  1,  22,  33,  11, 999,  11,   2,   1,  78,   0,   1,   2,   9,
          8,   7,   1,  76,  88]], dtype=int32)
```

Máscara Boleana e Seleção Avançada

Conceito super importante no numpy e no pandas é o de máscara booleana.

Ao aplicar qualquer operador booleano

- `>`
- `<`
- `<=`

- `>=`
- `==`
- `in`

o numpy retorna um array de `True` e `False` no qual ele aplicou elemento a elemento aquele operador.

Imagine para a matriz abaixo:

```
>>> mat = np.array([1, 10, 20, 30]).reshape(2, 2)
>>> arr
array([[ 1, 10],
       [20, 30]])
```

Eu quero saber todos os elementos maiores que 10, eu posso aplicar:

```
>>> mat > 10
array([[False, False],
       [ True,  True]])
```

Me é retornado uma matriz de formato (2, 2) assim como com `True` na posição dos elementos que são maiores que 10.

Podemos então fazer essa operação dentro dos colchetes de seleção:

```
>>> mat[mat > 10]
array([20, 30])
```

E será retornado um array com os elementos 20 e 30 como esperado.

Podemos fazer operações linha a linha ou coluna a coluna através de métodos auxiliares como `any` ou `all`:

- any: se qualquer elemento da linha for `True`, retorna `True`
- all: todos os elementos tem que ser `True` para retornar `True`

Exemplos:

Por coluna:

```
>>> np.any(mat > 10, axis=0)
array([ True,  True])
```

Por linha:

```
>>> np.any(mat > 10, axis=1)
array([False,  True])
```

Operador AND

Similar ao `and` do python, podemos usar múltiplas condições para filtrar dados da nossa matriz com o operador `&`.

```
>>> filt = (mat > 10) & (mat <= 20)
>>> mat[filt]
array([20])
```

Observação: note que os colchetes além de melhorarem a legibilidade, são necessárias devido a ordem de precedência dos operadores python. Se não colocarmos os colchetes, dará um erro.

Operador OR

Similar ao `or`, só que devemos utilizar `|`:

```
>>> filt = (mat == 1) | (mat >= 20)
>>> mat[filt]
array([1, 20, 30])
```

Operador NOT

Similar ao `not`, mas devemos utilizar um til `~`

```
>>> filt = (mat == 1) | (mat >= 20)
>>> mat[~filt]
array([10])
```

```
array([[ True,  True,  True,  True,  True, False,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True, False,  True, False,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True,  True,  True,  True, False,
         True,  True,  True,  True,  True,  True,  True, False, False]])
```

Seleção passando listas

Podemos selecionar elementos específicos de um array passando uma lista de posições:

```
>>> a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[[1, 2, 8]]
array([2, 3, 9])
```

Referências

- Documentação oficial