

NumPy

Muitos trabalhos aplicados em empresa envolvem cálculos matemáticos avançados, e por isso precisam se apoiar em alguma biblioteca. Quer trabalhemos com estatística, aprendizado de máquina, processamento de imagem, de texto, de áudio, com otimização, ou até simulação de sistemas reais, uma boa biblioteca matemática é fundamental.

No Python, a biblioteca **NumPy** (abreviação para *Numerical Python* [2]) faz esse papel. Ela é uma poderosíssima biblioteca matemática, e é um dos cerne da computação científica e ciência de dados em Python.

Os cálculos realizados pelo NumPy são baseados em operações com os chamados Arrays Multidimensionais (em matemática, chamamos de vetores, matrizes ou tensores). Esses objetos fundamentais da biblioteca são basicamente "listas de listas de listas de ...". O NumPy implementa operações nesses objetos de forma eficiente, em uma linguagem de mais baixo nível (linguagem C), o que confere a eles enorme velocidade e eficiência. Nós aproveitamos essa eficiência através da interface amigável que a biblioteca possui em Python.

Quando organizamos nosso código para aproveitar as operações do NumPy, é comum dizermos que nós "vetorizamos" o nosso código. Isso significa que evitamos loops explícitos no código Python, e reescrevemos tudo usando os arrays do NumPy. Assim, conseguimos aproveitar toda a velocidade das operações "vetorizadas", acelerando muito o nosso código.

A esmagadora maioria das bibliotecas de Python para computação científica, processamento de dados e ciência de dados (como por exemplo, SciPy, pandas, scikit-learn e statsmodels) se apoiam nas funcionalidades do NumPy [2].

1. Instalação
2. Introdução ao NumPy
 - i. Cuidados iniciais: Como copiar arrays
 - ii. Propriedades de ndarrays
 - iii. Acessando e modificando elementos (Indexing & Slicing)
 - iv. Máscara Booleana e Seleção Avançada
 - v. Métodos built-in de criação de arrays O argumento é o formato do ndarray resultante. O argumento é o formato do array resultante.
3. Matemática com NumPy
 - i. Broadcasting
 - ii. Funções matemáticas
4. Álgebra Linear e Estatística básica
 - i. Álgebra Linear
 - ii. Estatística

1. Instalação

Como falamos antes, o NumPy nada mais é que uma biblioteca do Python. Dito isso, para instalarmos ele, fazemos de forma semelhante a qualquer outra biblioteca Python.

Entre as diversas formas de fazer isso, a mais simples é instalar o pacote Anaconda (<https://www.anaconda.com/distribution/>). Ele já vem com o Python e diversas bibliotecas científicas e de ciência de dados instaladas. Se preferir reinstalar depois, o anaconda provê também o gerenciador de pacotes Conda, e assim é possível instalar o NumPy através do terminal.

```
$ conda install numpy
```

Caso prefira, você também pode instalar via o gerenciador de pacotes nativo do Python, o `pip`. Para isso, utilizamos o comando abaixo via terminal.

```
$ pip install numpy
```

Para quem for usuário com maior conhecimento de programação e computação científica, recomenda-se uma lida nas [instruções de instalação mais detalhadas do NumPy](#).

O maior motivo para isso é o fato de que, por trás dos panos, o NumPy utiliza as chamadas "bibliotecas de álgebra linear aceleradas", que pode ser tanto a biblioteca Intel MKL quanto a biblioteca OpenBLAS. Em algumas situações, é importante escolher qual das duas será usada pela sua instalação do NumPy.

Embora existam questões de performance computacionais para essa escolha, existem também motivos de negócio. A biblioteca Intel MKL não é de código fonte aberto, e sua licença pode não permitir alguns tipos de distribuição do código que você esteja desenvolvendo com o NumPy.

2. Introdução ao NumPy

A primeira coisa que precisamos fazer para usar o NumPy em um código, é importar a biblioteca. A importação funciona como qualquer outra biblioteca Python, porém o alias `np` para a biblioteca é amplamente utilizado pela comunidade de usuários do Python. A importação então é sempre feita da forma abaixo.

```
import numpy as np
```

Com esse alias, o nosso código fica bem menor, facilitando para outras pessoas entenderem ele depois. Como o alias é comum na comunidade, ele também não gera confusões, e permite sabermos sempre quando alguma operação feita é nativa do NumPy.

Na introdução, comentamos que a estrutura de dados básica do NumPy é um "array multidimensional". Esse objeto do NumPy se chama **ndarray**. Um mnemônico para ajudar a lembrar é lembrar pensar em "N Dimensões ARRAY", ou "NDARRAY". Basta lembrar que o "nd" significa "n-dimensional", pois também podemos ter tabelas multidimensionais.

Essa estrutura é semelhante aos arrays de outras linguagens de programação. Pode ser uma lista de valores, uma tabela, ou uma tabela de tabelas. O importante é que todos os valores são do mesmo "tipo". Se um deles é um número inteiro, todos serão. Se um deles for um float, todos serão. Esse tipo é chamado de **dtype**.

O mais comum é usarmos ndarrays como listas ou tabelas de valores.

No caso de uma lista, nós temos um ndarray de uma dimensão. Matematicamente, esse objeto é equivalente a um vetor.

O método mais comum de criar um ndarray é usando a função `np.array`. Assim, para criar nosso "vetor", usamos o comando abaixo.

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

Se olharmos o tipo da variável `vetor`, veremos que ela é do tipo `NumPy.ndarray`.

```
>>> print(type(vetor))
<class 'numpy.ndarray'>
```

Podemos criar também uma tabela com duas dimensões. Nesse caso, temos uma matriz.

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

É possível, também, aumentar o número de dimensões. Nesse caso, se tivermos 3 dimensões, por exemplo, teríamos uma lista de tabelas. Se forem 4 dimensões, então temos uma tabela cujos elementos são tabelas. Assim, quanto mais dimensões, nós vamos empilhando em uma lista objetos com menos dimensões. Se forem 5 dimensões, temos uma lista de ndarrays de 4 dimensões (ou uma lista de tabelas de tabelas).

Matematicamente, quando temos 3 dimensões ou mais, nós chamamos esse objeto de um tensor.

```
>>> tensor = np.array([[[1, 2], [3, 4]], [[1, 0], [0, 1]]])
>>> print(tensor)
[[[1 2]
  [3 4]]

 [[1 0]
  [0 1]]]
```

Note que a variável `tensor` nada mais é que duas tabelas.

Uma representação visual de ndarrays podem ajudar a entender melhor o que está acontecendo.

<img src=<https://s3-sa-east-1.amazonaws.com/lcpi/b50bd331-732b-48a4-9d46-01f20c15ab29.png> width=400 height="450">

(Fonte: Imagem adaptada da original, disponível pela ABRACD [5])

O `ndarray` é a estrutura de dados básica do NumPy, e a que usaremos sempre. A partir dela que toda a "mágica" acontece. Todo o poder do NumPy vem da implementação desse tipo de array e as operações relacionadas, que são mais eficientes que usar variáveis do tipo `list` em Python (embora ndarrays sejam menos flexíveis).

Esses objetos também são muito comuns em matemática, pois representam basicamente vetores e matrizes. Assim, é bem direto "vetorizar" nosso código escrevendo-o de forma bem parecida com como faríamos em notação matemática.

Obs: Falamos antes também de tensores. Não é importante entender o que eles são para poder usar o NumPy. Apenas cálculos muito específicos se utilizam da matemática de tensores. Para a maioria dos casos, basta ter a visão que demos acima, sobre como eles são basicamente "matrizes com 3 ou mais

dimensões". Isso significa que eles podem ser uma lista de tabelas, uma tabela de tabelas, uma lista de tabelas de tabelas, uma tabela de tabelas de tabelas, e por aí vai...

2.1. Cuidados iniciais: Como copiar arrays

Antes de começar a ver as propriedades básicas de um ndarray, é importante ter uma questão muito importante em mente. Assim como os arrays nativos do Python, devido à forma como o NumPy trabalha com ndarrays na memória do computador, eles se comportam da forma mostrada abaixo.

```
>>> 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])
```

Note que no bloco de código, a variável `a` foi modificada depois que alteramos a variável `b`! Isso acontece porque ambas as variáveis apontam para o mesmo endereço de memória do computador. Quando fazemos `b[0] = 100`, estamos alterando o valor do primeiro elemento de `b` na memória. Como `a[0]` aponta para o mesmo lugar na memória, ele vai passar a ver o valor novo. Logo, teremos que seu valor será `100`.

Por este motivo, sempre que queremos copiar um array de uma variável para outra, usamos o método `copy()`.

```
>>> 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])
```

O método `copy()` copia todos os valores do array para um novo endereço de memória. Assim, a variável `b` passa a apontar para esse novo endereço, e evitamos o problema anterior.

2.2. Propriedades de ndarrays

Dimensão e formato

Como vimos antes, é possível criar um ndarray com diferentes dimensões e formatos. Cada ndarray tem os atributos `ndim` e `shape` que guardam estas informações. Logo, para sabermos como é o nosso array, basta acessar esses atributos.

```

>>> print(vetor.ndim)
>>> print(vetor.shape)
1
(3,)
>>> print(matriz.ndim)
>>> print(matriz.shape)
2
(2,2)
>>> print(tensor.ndim)
>>> print(tensor.shape)
3
(2,2,2)

```

Isso é extremamente importante para nos ajudar quando temos um ndarray muito grande, e precisamos lembrar o quão grande. Além disso, esses atributos também nos ajudam quando queremos criar funções genéricas que atuem em ndarrays.

A partir do formato do array, podemos obter também o número de elementos. Porém, objetos ndarray já têm o atributo **size** para facilitar.

```

>>> print(vetor.size)
3
>>> print(matriz.size)
4
>>> print(tensor.size)
8

```

Muitas vezes queremos mudar o formato de um array. Por exemplo, podemos querer transformar um vetor de 4 elementos em uma matriz 2x2. Para isso, utilizamos a função **reshape**.

```

>>> before = np.array([[1,2,3,4],[5,6,7,8]])
>>> print(before.shape)
(2, 4)
>>> after = before.reshape((8)) # o novo array deve ter a mesma quantidade de elementos
>>> after
array([1, 2, 3, 4, 5, 6, 7, 8])
>>> after = before.reshape((8, 1))
>>> after
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7],
       [8]])
>>> after = before.reshape(2, 2, 2)

```

```
>>> after
array([[1, 2],
       [3, 4],

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

Outra forma de obter arrays com dimensões ou formatos diferentes é juntar e empilhar arrays. Podemos tanto empilhar um array em cima do outro (um "empilhamento vertical") usando a função `np.vstack`, quanto tentar colocar um ao lado do outro (um "empilhamento horizontal") usando a função `np.hstack`.

```
>>> 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]])

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

É comum falarmos de cada dimensão de um ndarray como um "eixo". Assim, se um ndarray tem 3 dimensões, nós podemos falar da "1ª dimensão do array" (o eixo 0 do array), da "2ª dimensão do array" (o eixo 1 do array) e da "3ª dimensão do array" (eixo 2 do array). Assim, um ndarray com formato `(3, 3, 2)` seria uma lista ao longo do seu primeiro eixo (eixo 0). Essa lista seria composta por 3 matrizes de tamanho 3 x 2. De forma totalmente equivalente, a gente poderia pensar que ele é uma lista com 2 matrizes 3 x 3 ao longo do seu terceiro eixo (eixo 2).

```
>>> a = np.array(range(18))
>>> b = a.reshape((3, 3, 2)).copy()
>>> b
array([[[ 0, 1],
        [ 2, 3],
        [ 4, 5]],

       [[ 6, 7],
        [ 8, 9],
        [10, 11]],

       [[12, 13],
        [14, 15],
        [16, 17]])
```

```
[[12, 13],
 [14, 15],
 [16, 17]])
```

A variável `b` vai ser um "cubo" de valores, invés de uma tabela. Assim, o que muda é só qual eixo a gente está olhando.

<img src=<https://s3-sa-east-1.amazonaws.com/lcpi/1e527c6a-ea64-46e3-bd31-ffba06749000.png> width=600 height="450">

(Fonte: Imagem original Ada)

Tipos de dados e tamanho em memória

Um array do NumPy tem a propriedade `dtype`, que nos dá o tipo dos seus elementos mais básicos.

```
>>> vetor = np.array([1, 2, 3, 4, 5, 6, 7])
>>> vetor.dtype
dtype('int64')
```

Quando nós construímos o ndarray, se não passarmos explicitamente qual é o `dtype`, ele infere a partir dos valores que passamos.

```
>>> vetor = np.array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0])
>>> vetor.dtype
dtype('float64')
```

Para passar esse parâmetro explicitamente, basta utilizar um segundo argumento na função `np.array`.

```
>>> vetor = np.array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0], dtype=np.int32)
>>> vetor.dtype
dtype('int32')
```

Para esses arrays, nós dizemos que a consistência dos dados é **forte**. Isso significa que, se criarmos um vetor de números inteiros, por exemplo, então todos os elementos que colocarmos no nosso vetor **vai ser inteiro**.

Se tentarmos colocar um valor numérico de ponto flutuante, o NumPy vai automaticamente converter para um valor inteiro.

```
>>> a = np.array([1, 2, 3])
>>> a[0] = 10
>>> a
array([10, 2, 3])
>>> a[0] = 1.23
>>> a
array([1, 2, 3])
```

O valor padrão de *dtype* depende do Python instalado. Se o Python instalado for de 64 bits, o padrão será tipos `int` ou `float` de 64 bits de memória (`int64` ou `float64`). Caso seu Python seja 32 bits, o padrão será `int32` e `float32`.

Essa diferença afeta a precisão dos cálculos. Quanto mais espaço em memória utilizado (maior o número de bits), maior a precisão.

Como o NumPy é feito para cálculos numéricos, ele precisa de grande controle sobre essa precisão. De fato, ele precisa de mais controle sobre isso do que o Python básico.

Devido a isso, os tipos de dados numéricos básicos do NumPy parecem com os do Python, mas em geral têm um número à direita. Esse número representa o seu nível de precisão computacional.

Alguns exemplos de tipos de dados do NumPy (e provavelmente os mais comuns) são:

- `np.int32`
- `np.int64`
- `np.float32`
- `np.float64`

Como exemplo, podemos 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 mostrar o array na tela.

Para descobrir o espaço que cada elemento ocupa na memória, individualmente, podemos acessar o atributo `itemsize`.

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

Ele retornou 2! Mas a gente não falou que era um tipo inteiro de "16 bits"?

O NumPy nos mostra o espaço em memória em bytes, e não em bits. Cada byte equivale a 8 bits. Logo, 16 bits são o mesmo que 2 bytes.

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

Já se fizemos um array padrão em um Python de 64 bits, temos um `itemsize` de 8.

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')
>>> a.itemsize
8
```

Isso pois o padrão de 64 bits equivale a 8 bytes. $\frac{64}{8} = 8$

A quantidade de elementos multiplicado pelo tamanho de cada elemento em memória nos dará o tamanho total de bytes que o array inteiro ocupa na memória do computador.


```
>>> # quantidade de elementos total
>>> a.size
3
>>> # tamanho em memória do vetor "a"
>>> a.size * a.itemsize
24
```

Não precisamos, porém, ficar calculando esse valor se quisermos saber essa informação. Podemos simplesmente acessar o atributo `nbytes`, que representa o tamanho total em bytes ocupado pelo array.

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

Observação

Geralmente não é necessário 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.

Podemos converter os tipos de dados do array do NumPy depois da sua criação de forma manual, usando a função `astype`.

```
>>> a = np.array([1., 2., 3.])
>>> a.dtype
dtype('float64')
>>> b = a.astype('int32').copy()
>>> b
array([1, 2, 3], dtype=int32)
>>> b.dtype
dtype('int32')
```

2.3. Acessando e modificando elementos (Indexing & Slicing)

Vamos montar duas matrizes, uma usando Python e outra usando NumPy, para podermos comparar como nós acessamos os elementos de cada uma.

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

```
>>> print(py_mat)
[[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14]]
```

Podemos acessar um elemento específico de forma similar a uma lista natural do Python, utilizando a sintaxe de colchetes. Porém, no NumPy, quando temos 2 ou mais dimensões, nós passamos a posição que queremos buscar através de números separados por vírgulas.

Se quisermos então pegar o número 13 nos arrays acima, que está na posição [1,5] (linha 1, coluna 5, lembrando que o Python começa os índices sempre pelo 0), no Python teríamos que usar a sintaxe abaixo.

```
>>> py_mat[1][5]
13
```

Já para um ndarray com duas dimensões (2D), temos uma sintaxe um pouco mais simples.

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

Além disso, todas as regras do Python são aplicáveis no NumPy. Usar números negativos, por exemplo, funciona como uma indexação de trás para frente.

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

Para pegar uma linha inteira, podemos utilizar a sintaxe de `:` na coluna. Essa sintaxe pode ser lida como "todos os elementos deste eixo". Podemos ler então a sintaxe abaixo como "linha zero, todas as colunas".

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

Embora também poderíamos fazer de forma mais fácil essa indexação em específico.

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

Porém, para pegar uma coluna, não tem jeito, precisamos usar `:`. A sintaxe abaixo pode ser lida como "todas as linhas, coluna 2".

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

O operador `:` é também conhecido como slicing (ou "fatiamento", em português), e pode ser usado para pegar intervalos de índices. Para isso, ele precisa de três parâmetros:

- start, o índice inicial do intervalo
- end, o índice final do intervalo
- step, quantos índices nós pulamos em cada passo dado para ir do *start* até o *end*.

Isso é feito no formato `[start:end:step]`. Neste caso, `step` é o tamanho do passo que vamos dar.

Basicamente, ele diz quantos elementos devemos pular de cada vez. Como exemplo, podemos pegar do

elemento 1 ao 6 pulando de 2 em 2 na linha 0.

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

Novamente, ele também funciona com índice negativo.

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

Podemos usar essas indexações para alterar os elementos do array. Basta usar o operador `=` para atribuir um novo valor ao elemento daquela posição.

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

Isso vale até para colunas ou linhas inteiras.

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

Isso acontece por uma característica fundamental do array do NumPy:

Ao alterar o pedaço recortado da matriz (o "slice" da matriz), você altera a matriz original.

Juntando com o problema que vimos antes, de cópia de array, quando queremos copiar um slice de um array, temos que explicitamente usar o método `copy`.

```
>>> a = np.array([1, 2, 3])
>>> b = a[1:].copy()
>>> b
array([2, 3])
>>> c = a[1:]
>>> c
array([2, 3])
>>> c[0] = 5
>>> c
array([5, 3])
>>> a
array([1, 5, 3])
>>> b
array([2, 3])
```

2.4. Máscara Booleana e Seleção Avançada

Uma outra forma de seleção é por meio de uma lista de posições.

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

No caso de arrays multidimensionais, vamos ter que passar uma lista para cada dimensão. O NumPy nos retornará os elementos formados pelos pares (se 2D, se for 3D são trincas, se 4D quadras, etc.) dessas listas.

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

Uma outra forma muito comum de selecionar elementos de um ndarray é usando o que chamamos de "máscara booleana". Ela nada mais é do que um array de elementos booleanos ("True" ou "False"). Tanto faz se for uma lista do Python ou um ndarray do NumPy.

Assim, quando passamos essa máscara para um array do NumPy, ele nos retorna apenas as posições onde temos o valor "True".

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

Note, no exemplo acima, que temos 3 posições onde nossa máscara possui valor "True". A posição 0, a posição 1, e a posição 7, que correspondem aos valores 1, 2, e 8, respectivamente.

Nós podemos gerar essa máscara a partir de comparações elemento a elemento de um ndarray. De fato, ao aplicar qualquer operador booleano, como

- >,
- < ,
- <= ,
- >= ,
- == ,

para comparar um ndarray com algum valor, o NumPy retorna um ndarray de valores booleanos. Esse novo ndarray vai ter o valor **True** nas posições onde a comparação do elemento original e esse valor separado for verdadeira. Teremos o valor **False** caso contrário.

Vamos ver um exemplo.

```
>>> mat = np.array([1, 10, 20, 30]).reshape(2, 2)
>>> mat
array([[ 1, 10],
       [20, 30]])
>>> mat > 10 # retorna um ndarray com valor True onde o elemento original é maior que 10
array([[False, False],
       [ True,  True]])
```

Podemos juntar os dois exemplos anteriores, de uso da máscara booleana como indexação e da criação de uma máscara, em uma única linha.

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

Neste exemplo, temos como retorno um array com os elementos 20 e 30, como era esperado.

Podemos fazer operações semelhantes, mas restritos a seguir "linha a linha" ou "coluna a coluna". No nosso exemplo anterior, pode ser que queiramos todas as linhas onde existe um elemento com valor 10. Outra opção seria pegar todas as colunas com pelo menos 1 elemento abaixo do valor 10.

Para isso, usamos os métodos auxiliares `any` e `all`.

- `any`: se qualquer elemento do meu eixo for `True`, retorna um valor `True`
- `all`: se, e somente se, todos os elementos do meu eixo forem `True`, retorna `True`

Nas explicações, o "eixo" são as dimensões do ndarray, como discutido na sessão 2.2.

Vamos ver alguns exemplos.

O exemplo abaixo pega todas as linhas onde existe pelo menos um elemento com valor 10.

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

Por que usamos o eixo 1 nesse caso? Porque aqui, o eixo define por onde o NumPy vai "traversar" para avaliar o nosso ndarray.

O eixo 0 são as linhas. Assim, quando olhamos *ao longo do eixo 0*, nós estamos caminhando através das linhas. Se usássemos `(mat == 10).any(axis=0)`, o NumPy vai ver se tem elementos que sejam iguais a 10 **ao longo do eixo 0**.

O processo então vai ser o seguinte:

- olha se `1 == 10`. Nesse caso, temos `False`.
- o próximo elemento, **ao longo do eixo 0**, é o `20`.
- `20 == 10` também retorna `False`. Logo, nessa coluna, a resposta será `False`.
- segue para a próxima coluna.

Já quando usamos `(mat == 10).any(axis=1)`, ele olha **ao longo do eixo 1**, ou seja, ao longo das colunas. O processo fica:

- olha se `1 == 10`. Nesse caso, temos `False`.
- o próximo elemento, **ao longo do eixo 1**, é o `10`.
- `10 == 10` retorna `True`. Logo, existe pelo menos um valor `True`. Então, para essa linha, retornamos o valor `True`.
- segue para a próxima linha.

Como um outro exemplo, se quisermos pegar as colunas onde todos os elements sejam maiores que `5`, podemos usar o `all`. A regra de qual eixo usar é igual.

```
>>> mat[:, (mat > 5).all(axis=0)]
array([[10],
       [30]])
```

Perceba que neste exemplo, nós colocamos dois pontos na primeira posição. Isso garante que pegamos todas as linhas, e olhamos a nossa máscara booleana apenas ao longo das colunas. Além disso, colocamos `axis=0`, para olhar ao longo do eixo 0.

Para nos ajudar a lembrar essa questão do parâmetro `axis`, podemos pensar que esse parâmetro define ao longo de qual eixo a função irá "esmagar" o nosso array.

```
>>> new_mat = np.array([1, 2, 4, 5, 7, 8]).reshape(3, 2)
>>> new_mat
array([[1, 2],
       [4, 5],
       [7, 8]])
>>> (new_mat < 8).all(axis=1) # Apenas as 2 primeiras linhas têm todos os elementos < 8
array([ True,  True, False])
>>> (new_mat == 5).any(axis=0) # Apenas a segunda coluna tem um elemento com valor 5
array([False,  True])
```

Múltiplas condições

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

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

Observação: note que os parênteses, além de melhorarem a legibilidade, são necessárias devido a ordem de precedência dos operadores `>`, `&` e `<=`. Se não colocarmos os parênteses, dará um erro.

Além do `&`, também podemos fazer quaisquer operações booleanas básicas.

No caso do `or`, nós temos o operador `|`.

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

No caso do `not`, nós temos o operador `~`.

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

2.5. Métodos built-in de criação de arrays

O NumPy já possui diversos métodos para gerar alguns arrays comuns mais conhecidos. Isso nos ajuda imensamente a criar algumas matrizes padrões para fazer contas.

Primeiro, é possível gerar um array de zeros com qualquer formato.

```
>>> np.zeros((2, 3)) # O argumento é o formato do ndarray resultante.
array([[0., 0., 0.],
       [0., 0., 0.]])

>>> np.zeros((2, 2, 5))
array([[[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]],
      [[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Para criar um array de uns, a sintaxe é semelhante.

```
>>> 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)
```

Se invés de uma matriz apenas de uns, nós quisermos usar outro número, podemos fazer isso aproveitando o `np.ones`. Basta multiplicarmos a matriz pelo número que desejamos.

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

Mas o NumPy já tem uma opção mais direta, o `full`.

```
>>> np.full((2, 2), 99) # O primeiro argumento é o formato, o segundo é o número
array([[99, 99],
       [99, 99]])
```

Também temos o `full_like`, que copia o formato de uma matriz que já tenha sido declarada antes.

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

Outra função para criação de arrays é `np.repeat`. Ela repete um determinado array na direção do eixo escolhido.

Para um vetor de 1D, temos apenas 1 eixo, mas para matrizes, temos 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]
```

Para fazermos uma matriz, mudamos o formato da variável `arr` de (3,) para (1,3).

```
>>> arr = np.array([[1, 2, 3]])
>>> r1 = np.repeat(arr, 3, axis=0)
>>> print(r1)
[[1 2 3]
 [1 2 3]
 [1 2 3]]
>>> r2 = np.repeat(arr, 3, axis=1)
>>> print(r2)
[[1 1 1 2 2 2 3 3 3]]
```

Criando arrays com listas de números

Um caso específico de criação de arrays é quando queremos gerar uma lista sequencial de números. Isso é muito comum, por exemplo, quando queremos calcular uma função matemática em vários pontos de uma única vez.

A primeira função do NumPy para criar essa sequência numérica é `np.arange`. Esta função retorna elementos igualmente espaçados por um step (que por padrão é 1) dentro de um certo intervalo.

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

A função `np.arange` é semelhante à função `range` do Python, porém permite que peguemos valores de ponto flutuante (números com decimal) dentro do nosso intervalo.

```
>>> 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])
```

Outro caso comum que esbarramos para criar uma lista sequencial é quando queremos escolher uma grande quantidade de pontos entre dois números. Por exemplo, pode ser que nós queiramos fazer um mesmo cálculo para 10 valores entre 3 e 7.

Neste caso, usamos a função `np.linspace`.

```
>>> np.linspace(3, 7, num=10)
array([3.         , 3.44444444, 3.88888889, 4.33333333, 4.77777778,
        5.22222222, 5.66666667, 6.11111111, 6.55555556, 7.         ])
```

Números decimais aleatórios

O NumPy tem um sub-módulo chamado `random` (cujas funções são acessadas usando o preâmbulo `np.random`). Nele, existe um conjunto de funções focadas em gerar arrays com números aleatórios.

Por exemplo, a função `np.random.random_sample` retorna valores aleatórios entre 0 e 1.

```
>>> np.random.random_sample((4, 2)) # O argumento é o formato do array resultante.
array([[0.59701743, 0.40627135],
       [0.23940108, 0.58528558],
       [0.04228252, 0.68467361],
       [0.68907347, 0.85636431]])
```

A função `np.random.rand` é um "alias" da função de cima. Isso significa que, por trás dos panos, ela literalmente só executa a função `np.random.random_sample`.

Essa função existe para facilitar a migração de códigos de outras linguagens de computação numérica (especificamente, códigos de uma linguagem chamada MATLAB).

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

Se quisermos criar um array de números inteiros abaixo de um certo valor, escolhidos ao acaso, usamos a função `np.random.randint`.

```
>>> np.random.randint(7, size=(3, 3)) # Números menores que 7, formato 3 x 3
array([[6, 5, 6],
       [0, 6, 2],
       [4, 6, 5]])
```

Os argumentos principais da função `np.random.randint` são `low`, `high` e `size`, que definem o menor inteiro possível, o maior inteiro possível, e o tamanho do array resultante.

```
>>> np.random.randint(5, high=10, size=10)
array([8, 7, 8, 6, 9, 9, 5, 6, 8, 9])
```

Para incluir o 10 entre os valores possíveis, basta trocar o argumento `high` para 11.

3. Matemática com NumPy

No início deste texto, nós introduzimos o NumPy como uma biblioteca muito utilizada para fazer cálculos numéricos. Vamos ver então as operações matemáticas disponíveis no NumPy.

A primeira coisa a se observar é que as operações matemáticas básicas, como usadas no Python, podem ser usadas também para ndarrays. Só temos que lembrar sempre que elas são aplicadas elemento a elemento.

```
>>> vec1 = np.array([2., 4., 6., 8.])
>>> vec2 = np.array([1., 2., 3., 4.])
>>> vec1 + vec2                                # Soma
array([ 3.,  6.,  9., 12.])
>>> vec1 - vec2                                # subtração
array([ 1.,  2.,  3.,  4.])
>>> vec1 * vec2                                # Multiplicação
array([ 2.,  8., 18., 32.])
>>> vec1 / vec2                                # Divisão
array([2., 2., 2., 2.])
>>> vec1 ** vec2                                # exponenciação
array([2.000e+00, 1.600e+01, 2.160e+02, 4.096e+03])
```

A notação com este "e" minúsculo significa que o número está escrito em potências de 10. Assim, `4.096e+03` significa `4096` (pois multiplicamos o número por 1000, que é dez elevado a 3). Ainda nesta

linha, `2.160e+02` significa `216`.

O NumPy também consegue fazer operações entre arrays e números (chamadas operações com escalares).

```
>>> vec2 + 2                # Somar
array([3, 4, 5, 6])
>>> vec2 - 2                # Subtrair
array([-1,  0,  1,  2])
>>> vec2 * 2                # Multiplicar
array([2, 4, 6, 8])
>>> vec2 / 2                # Dividir
array([0.5, 1. , 1.5, 2. ])
>>> a ** 2                  # Potência (elevar a um número)
array([ 1,  4,  9, 16])
>>> vec2 += 2               # Incrementar
>>> vec2
array([3, 4, 5, 6])
```

Agora uma dúvida é: se os nossos ndarrays tiverem formatos diferentes, então o NumPy apenas retorna um erro?

Depende do caso. Ele consegue fazer a operação se a diferença entre os formatos seguirem algumas regras que veremos abaixo. Nestes casos, ele faz o que chamamos de **Broadcasting**.

Se a diferença entre os formatos não seguirem as regrinhas necessárias para realizar broadcasting, então o NumPy retorna de fato um erro.

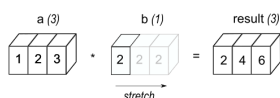
```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 2])
>>> a * b
-----
ValueError                                Traceback (most recent call last)
<ipython-input-113-9bc1a869709f> in <module>
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

3.1. Broadcasting

O que fizemos, nas operações com escalar, é um tipo de broadcasting. Você pode imaginar que o que está acontecendo é o seguinte:

- pegue o seu número,
- crie um ndarray, de mesmo formato que o primeiro, mas cujos valores são o seu número.



(Fonte: Tutoriais oficiais do NumPy [2])

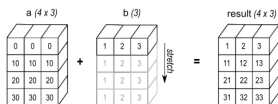
Na imagem acima (que foi tirada dos tutoriais oficiais do NumPy), nota que o valor 2 é "esticado". Ele é repetido ao longo do eixo 0 até ter um ndarray de formato (3,).

Essa ideia pode ser generalizada. Sempre que fizer sentido "esticar" (ou repetir) o array menor até ele ficar com o mesmo formato do maior, então o NumPy consegue realizar uma operação de broadcasting.

```
>>> matrix = np.array([[0, 0, 0],
                        [10, 10, 10],
                        [20, 20, 20],
                        [30, 30, 30]])

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

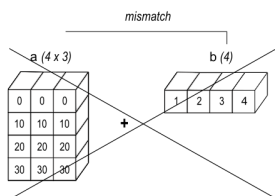
>>> matrix + vec
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```



(Fonte: Tutoriais oficiais do NumPy [2])

A imagem acima mostra o que está acontecendo no código de exemplo que demos.

Mas por que colocamos a expressão "sempre que fizer sentido"? O motivo é que a única coisa que o NumPy aceita fazer é repetir o array menor diversas vezes na direção de uma dimensão. Ele não reajusta o seu array!



(Fonte: Tutoriais oficiais do NumPy [2])

```
>>> matrix = np.array([[0, 0, 0],
                        [10, 10, 10],
                        [20, 20, 20],
                        [30, 30, 30]])

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

>>> matrix + vec
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-17-a18531600448> in <module>
----> 1 matrix + vec
```

```
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

É interessante notar que essa "esticada" para realizar o broadcast só pode ser feita em 1 eixo. Mesmo que fosse possível copiar o array menor até ele ter o mesmo formato do array maior, se para isso precisarmos fazer cópias ao longo de 2 eixos, então o NumPy retorna um erro.

```
>>> a = np.array([[1, 2, 3, 4],
                  [3, 4, 5, 6],
                  [7, 8, 9, 10]])
>>> b = np.array([2, 2])
>>> a * b
-----
ValueError                                Traceback (most recent call last)
<ipython-input-117-9bc1a869709f> in <module>
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (3,4) (2,)
>>> b = np.array([[2, 2]])
>>> a * b
-----
ValueError                                Traceback (most recent call last)
<ipython-input-119-9bc1a869709f> in <module>
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (3,4) (1,2)
```

3.2. Funções matemáticas

Além de fazer operações matemáticas entre arrays, também podemos usar funções matemáticas conhecidas. O NumPy possui milhares de funções já implementadas. Por exemplo, ele possui as funções trigonométricas.

Função seno:

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sin(x)
array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

Função cosseno:

```
>>> x = np.array([1, 2, 3, 4])
>>> np.cos(x)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
```

O NumPy também possui o valor de constantes matemáticas comuns, como o π ("pi").

```
>>> x = np.array([0, np.pi / 2, np.pi, 3 * np.pi / 2, 2 * np.pi])
>>> np.sin(x)
```

```
array([ 0.0000000e+00,  1.0000000e+00,  1.2246468e-16, -1.0000000e+00,
        -2.4492936e-16])
```

Os valores `1.2246468e-16` e `-2.4492936e-16` representam o número zero. O motivo de não estar zerado é que nem todas as operações com números de ponto flutuante (números reais com decimal) conseguem ser perfeitamente representadas pelo computador. Números de ponto flutuante sempre tem um grau de imprecisão. Então dentro da precisão do nosso cálculo, esses valores são basicamente o mesmo que zero.

Depois das funções trigonométricas, vamos ver a função exponencial.

```
>>> x = np.array([1, 2, 3, 4])
>>> np.exp(x)
array([ 2.71828183,  7.3890561 , 20.08553692, 54.59815003])
```

Também existe a constante "e" (número euleriano) no NumPy. De fato, a função exponencial `np.exp(x)` nada mais é do que o número euleriano elevado a x .

Assim, também podemos fazer a mesma conta abaixo.

```
>>> x = np.array([1, 2, 3, 4])
>>> np.e ** x
array([ 2.71828183,  7.3890561 , 20.08553692, 54.59815003])
```

Como falamos, existem milhares de funções matemáticas nativamente no NumPy, e todas são muito úteis para fazermos cálculos matemáticos computacionais. Porém, se tentarmos listar todas, precisaríamos de um texto insanamente maior. Então melhor ficarmos só com uma ou outra mesmo.

Temos falado, ao longo do texto, sobre o poder e eficiência do NumPy. Do ponto de vista de escrever código, uma enorme vantagem do NumPy é que podemos escrever equações matemáticas complicadas para todos os elementos do nosso array da mesma forma que escreveríamos no papel.

Aqui entra em foco a ideia de "operações vetorizadas", que encontramos antes. No fim das contas, conseguimos escrever tudo de forma vetorizada no NumPy.

Imagine a equação abaixo: $e^x + \sin(x)$

Se quisermos achar o valor dela para diversos valores de "x" entre -1 e 2 no Python, teríamos que fazer um loop iterativo. No NumPy, a operação é mais clara e direta.

```
>>> x = np.linspace(-1, 2, 10)
>>> np.exp(x) + np.sin(x)
array([-0.47359154, -0.10495268,  0.38933661,  1.          ,  1.72280712,
        2.56610384,  3.55975281,  4.7656058 ,  6.28989801,  8.29835353])
```

Por fim, o último conceito essencial de se conhecer quanto a operações matemáticas no NumPy é o de número infinito e de NaN (not a number).

O NumPy possui duas constantes, `np.inf` e `np.NaN` que representam um "número infinito" e um erro, respectivamente.

O "número infinito" é um valor de ponto flutuante que é maior que qualquer outro número. Se colocarmos um sinal negativo, ele vira "menos infinito", e passa a ser menor que qualquer outro número.

```
>>> np.inf > 3823748932489
True
>>> -np.inf < -9832743293278423
True
```

Já o NaN é um valor indefinido. Sempre que nossa operação não fizer sentido, o numpy retorna um NaN.

```
>>> np.float64(0) / np.float64(0)
nan
```

O cuidado que devemos tomar com o NaN é que todas as operações entre um número e um NaN passam a ser NaN.

```
>>> np.NaN + 44
nan
```

4. Álgebra Linear e Estatística básica

4.1. Álgebra Linear

Da definição do Wikipédia:

Álgebra linear é um ramo da matemática que surgiu do estudo detalhado de sistemas de equações lineares [...]. A álgebra linear utiliza alguns conceitos e estruturas fundamentais da matemática como vetores,[...] e matrizes.

Independente da definição exata, a questão é que a álgebra linear estuda vetores e matrizes. Como vimos até aqui, o NumPy opera de forma bastante análoga à vetores e matrizes.

O que não vimos ainda é como o NumPy nos permite também executar operações típicas da álgebra linear, como "produto entre matrizes".

De fato, o NumPy tem a capacidade de realizar um "produto escalar" entre vetores, de fazer o produto de uma matriz por um vetor, e de fazer o produto entre matrizes.

Vamos relembrar o que são essas operações. O produto escalar de dois vetores consiste em multiplicar os dois, elemento a elemento, e depois somar os resultados de cada par de elementos.



$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 2 & 2 & 3 \end{bmatrix} = 2 + 4 + 9 = 15$$

(Exemplo de produto escalar, ou "produto interno")
(Fonte: Imagem original Ada)

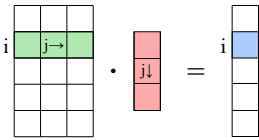
A sintaxe do NumPy é dada abaixo.

```
>>> vec1 = np.array([1, 2, 3])
>>> vec2 = np.array([2, 2, 3])
>>> vec1.dot(vec2)
15
```

Também podemos usar o operador @.

```
>>> vec1 = np.array([1, 2, 3])
>>> vec2 = np.array([2, 2, 3])
>>> vec1 @ vec2
15
```

O produto de matriz e vetor é dado fazendo algo semelhante ao produto escalar. Para cada linha da matriz, a gente faz o produto escalar entre aquela linha e o vetor. O resultado então vai ser o elemento que se encontra na mesma posição da linha.



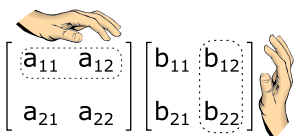
(Fonte: Usuária Claudia4 do site Wikimedia Commons, disponível em https://upload.wikimedia.org/wikipedia/commons/2/22/Matrix_vector_multiplication.svg)

As possibilidades de sintaxe do NumPy seguem abaixo.

```
>>> matrix = np.array([[1, 1, 1],
                        [1, 2, 3],
                        [1, 1, 1]])
>>> vec2 = np.array([2, 2, 3])
>>> matrix.dot(vec2)
array([ 7, 15,  7])
>>> matrix @ vec2
array([ 7, 15,  7])
```

Na multiplicação entre matrizes, nós multiplicamos a linha da matriz da esquerda pela coluna da matriz da direita, elemento a elemento, e somamos o resultado (produto escalar entre a linha da matriz da esquerda com a coluna da matriz da direita) para obter uma nova matriz.

Neste caso, quando fazemos o produto interno da linha N com a coluna M, nós obtemos o elemento que ficará na posição (N,M) na nova matriz.



(Fonte: Usuários Guy vandegrift e Bilou do site Wikimedia Commons, disponíveis em https://upload.wikimedia.org/wikipedia/commons/d/d9/Hands_matrix_multiplication.svg e https://upload.wikimedia.org/wikipedia/commons/2/22/Matrix_vector_multiplication.svg)

Como nos casos anteriores, abaixo temos a sintaxe do NumPy.

```
>>> matrix1 = np.array([[1, 1, 1],
                        [1, 2, 3],
                        [1, 1, 1]])
>>> matrix2 = np.array([[1, 1, 1],
```



```

        [1, 1, 1],
        [1, 1, 1]])
>>> matrix1.dot(matrix2)
array([[3., 3., 3.],
       [6., 6., 6.],
       [3., 3., 3.]])
>>> matrix1 @ matrix2
array([[3., 3., 3.],
       [6., 6., 6.],
       [3., 3., 3.]])

```

No campo da álgebra linear, existem algumas matrizes especiais muito úteis. O NumPy possui métodos de criação para a maioria delas.

A mais importante é a chamada matriz identidade. Ela é uma matriz quadrada (mesmo número de linhas e colunas) com diagonal 1, e os outros valores 0.

```

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

```

Ela se chama identidade pois qualquer matriz multiplicada pela identidade (por produto de matrizes) dá a própria matriz.

```

>>> matrix1 @ np.identity(3)
array([[1., 1., 1.],
       [1., 2., 3.],
       [1., 1., 1.]])

```

Outra operação comum com matrizes é a transposição.

$$\begin{matrix} \mathbf{A} \\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \end{matrix} \longrightarrow \begin{matrix} \mathbf{A}^T \\ \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \end{matrix}$$

(Fonte: Imagem original Ada)

A transposição pode ser feita com a função `np.transpose`.

```

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

```

Outra forma é usando o atributo `T`:

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

Operações mais específicas se encontram no módulo `linalg` do NumPy.

Um exemplo é encontrar o determinante de uma matriz. O determinante é um número característico de uma dada matriz, e que se relaciona com diversas propriedades dela.

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

Para encontrar todas as funções específicas de Álgebra Linear que têm no NumPy, você pode acessar o link <https://numpy.org/doc/stable/reference/routines.linalg.html>.

Alguns exemplos de funções já implementadas na biblioteca:

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

4.2. Estatística

Por fim, o NumPy também possui 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
>>> np.min(stats, axis=1) # Pega o menor valor de cada linha
array([1, 4])
>>> np.max(stats, axis=0) # Pega o maior valor de cada coluna
array([4, 5, 6])
>>> np.sum(stats, axis=0) # Faz a soma dos valores de cada coluna
array([5, 7, 9])
```

```
>>> np.mean(stats) # Valor médio de todos os elementos do array
3.5
>>> np.mean(stats, axis=0) # Valor médio de cada coluna do array
array([2.5, 3.5, 4.5])
>>> np.mean(stats, axis=1) # Valor médio de cada linha do array
array([2., 5.])
```

Referências:

1. Documentação NumPy, disponível em <https://NumPy.org/doc/stable/index.html> ;
2. Tutorial NumPy, disponível em https://NumPy.org/doc/stable/user/absolute_beginners.html ;
3. Funções matemáticas do NumPy, disponível em <https://numpy.org/doc/stable/reference/routines.math.html>
4. Álgebra linear com NumPy, disponível em <https://numpy.org/doc/stable/reference/routines.linalg.html>
5. Tutorial NumPy da ABRACD, disponível em <https://abracd.org/tutorial-numpy-os-primeiros-passos-em-computacao-numerica-e-tratamento-de-dados/>