Numpy

Esta biblioteca é utilizada para manipulação de listas(arrays/vetores) e matrizes(vetores multidimensionais/array de arrays). Com ela podem aplicar cálculos entre múltiplas listas/matrizes e ainda chamar funções para modificar essas listas/matrizes.

Agora vocês perguntam-se:
-Python já tem listas e nós podemos manipulá-las e aplicar cálculos, porquê usar uma biblioteca!?

Apesar de tudo isso ser verdade, vocês devem ter lido na Introdução (https://github.com/TigaxMT/Apontamentos-Machine-Learning/blob/master/00.%20Introdu%C3%A7%C3%A3o/Introdu%C3%A7%C3%A3o.pdf) que cada feature é uma dimensão, imaginem que temos 100 features, sendo assim temos 100 dimensões (100D). São muitas dimensões para o Python processar, e sendo ela uma linguagem de programação interpretada tornará o processo muito lento!

-Ok ... mas se Numpy é uma biblioteca de Python, não será "lenta" da mesma forma?

Não, e o porquê é simples, ela foi escrita em Linguagem C que é uma linguagem compilada (faz com que tudo seja mais rápido!).

E por fim, outro motivo para aprender Numpy é o facto de ele ser a base de outra biblioteca MUITO importante na análise de dados: o ***Pandas***.

Bora aprender um pouquinho!

Instalação

Antes de tudo precisamos instalar o Numpy! E para isso vocês pode usar o `pip` ou o `conda`, eu pessoalmente utilizo o `pip`, mas irei deixar abaixo o comando de instalação para ambos:

- Pip: `pip install numpy`
- Conda: `conda install numpy`

É só correr um desses comandos no terminal e o numpy ficará instalado!

Nota

- **Primeiro**: Para quem usa alguma distribuição Linux, tenha atenção ao pip! Pois pode haver ainda algumas distribuições a usar Python 2 como padrão e aí o vosso `pip` será do Python 2 (pip2). O melhor é usar `pip3 install numpy`, dessa forma reforçam que querem usar o `pip` do Python 3 (pip3).
  Iremos usar Python 3 pelo simples motivo que o Python 2 foi descontinuado no início de 2020.
- **Segundo**: Se der erro de permissão ao usar o `pip` usem o seguinte comando: `pip install numpy --user`.

Básico

Agora vamos importar o numpy

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

Para não termos de escrever a palavra `numpy` usammos o `as` para atribuir um alias, no caso chamamos ele de `np`.
Dessa forma só temos de escrever 2 letras (np) em vez de 5 (numpy).

Arrays e Matrizes

Comecemos pelo básico, por exemplo vamos criar um array(ou seja, uma lista ou também chamado de vetor) e depois criaremos uma matriz (que não é nada mais que uma lista com outras listas dentro dela).

In [3]:

```
# Criar uma lista/array/vetor
lista = np.array([1,2,3,4,5])

print(lista)
```

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

Como podem ver é uma lista igual ao Python. Vamos ver o tipo

In [4]:

```
print( type(lista) )
```

```
<class 'numpy.ndarray'>
```

Podemos ver que já não é uma *list* do Python e sim um numpy array!

Agora vamos criar uma matriz, que como já disse é uma lista de listas

In [5]:

```
# Criar uma matriz
matriz = np.array([ [1,2,3],
                    [4,5,6] ])

print(matriz)
```

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

E temos aqui uma matriz! Podemos ver que tem 2 linhas e 3 colunas. E como sabemos isso?!
Bem primeiro vejam visualmente, nós temos uma lista representada por *[ ]* e dentro dela temos mais 2 listas:
[1 2 3] e *[4,5,6]*

O número de listas dentro da lista são as **linhas**.

As colunas é fácil, é o **número de elementos que está dentro das listas**. Cada lista tem 3 números, logo temos 3 colunas!

Nota

Vocês podem ter o número de linhas que quiserem, **mas** têm de ter o mesmo número de elementos em todas as listas!!
Vejam este exemplo:

In [6]:

```
# Aqui eu criei uma matriz, contudo a primeira linha tem 3 colunas e a segunda 2 colunas

matriz_errada = np.array([
    [1,2,3],
    [4,5]
])
```

In [7]:

```
print( matriz_errada )
```

```
[list([1, 2, 3]) list([4, 5])]
```

O que acontece aqui é que não é criado um numpy array e sim um numpy array com duas listas de Python dentro!
Tenham atenção a isso!

Agora vamos aprender como ver quantas linhas e colunas tem um numpy array/matriz

In [8]:

```
print(lista.shape)
print(matriz.shape)
```

```
(5,)
(2, 3)
```

O `shape` é um atributo do nosso numpy array, ele retorna uma tupla onde:

No caso de um array/lista/vetor:

- O primeiro e único elemento da tupla, representa o número de elementos

No caso de uma matriz/array de arrays/vetor multidimensional:

- O primeiro elemento é o número de linhas
- O segundo elemento é o número de colunas

Como podem observar o shape da lista é: `(5,)` - o que significa que só está a contar as colunas e não as linhas já que ele tem apenas 1 linha e 5 colunas(números/elementos).
Sempre que virem um shape neste formato `(x,)` é porque o numpy array é realmente uma lista/array e não uma matriz.

Já a nossa matriz retornou: `(2,3)` - o que bate certo 2 linhas e 3 colunas.

Há outro método para criar matrizes: `matrix()` , mas a própria documentação não recomenda o uso dele, logo não vou mostrar aqui.

Métodos

Numpy tem vários métodos que manipulam os nossos arrays e/ou matrizes, vou mostrar alguns aqui

In [9]:

```
"""

    O método arange serve para criar arrays de x a y números.
    Por exemplo um array com números de 0 a 9

"""

# Vamos criar um array de 0-9
lista = np.arange(0,10)
print("Lista de 0 a 9: ", lista)
```

```
Lista de 0 a 9:  [0 1 2 3 4 5 6 7 8 9]
```

In [10]:

```
"""

    Podemos definir os steps,ou seja, pensem num range() do Python, se eu fizer: range(0, 10, 2) - ele vai
contar de 0 a 9
    de 2 em 2: 0 2 4 6 8 - é igual no método arange

"""

# Criar um array de 0-9, só que contando de 2 em 2
lista = np.arange(0,10,2)
print("Lista de 0 a 9, contado de 2 em 2: ", lista)
```

```
Lista de 0 a 9, contado de 2 em 2:  [0 2 4 6 8]
```

In [11]:

```
"""

    No caso de quererem criar arrays ou matrizes preenchidas com zeros ou com uns, há métodos para isso

"""

# Criar uma array com 5 colunas preenchidas com zeros
lista = np.zeros(5)

print("Lista com 0s: ", lista)

# Criar um array com 5 colunas preenchidas com uns

lista = np.ones(5)
print("Lista com 1s: ", lista)

# Criar matriz com 3 linhas e 5 colunas preenchidas com zeros

matriz = np.zeros((3,5))
print("Matriz com 0s:\n", matriz)

# Criar matriz com 3 linhas e 5 colunas preenchidas com uns

matriz = np.ones((3,5))
print("\nMatriz com 1s:\n", matriz)
```

```
Lista com 0s:  [0. 0. 0. 0. 0.]
Lista com 1s:  [1. 1. 1. 1. 1.]
Matriz com 0s:
 [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

Matriz com 1s:
 [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

Isto pode ser útil no caso de quererem inicializar um array/matriz e ainda não terem valores para colocar lá.

In [12]:

```
"""
    Este método cria uma matriz identidade

"""
# Criar matriz identidade de 4 por 4
matriz_identidade = np.eye(4)

print("Com método eye:\n", matriz_identidade)

# Criar a mesma matriz identidade com outro método
matriz_identidade = np.identity(4)
print("\nCom método identity:\n", matriz_identidade)
```

```
Com método eye:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

Com método identity:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

A diferença entre um método e outro é que o método `eye` permite que explicitemos o índice da diagonal, já o `identity` apenas usa a diagonal principal.

In [13]:

```
"""

    Vamos usar o eye novamente mas agora iniciamos a diagonal do índice 1 (coluna 2)

"""
print("Diagonal começada do índice 1:\n", np.eye(4, k=1))
```

```
Diagonal começada do índice 1:
 [[0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 0.]]
```

Deu para ver que a diagonal começou no índice 1 (coluna 2) da primeira linha.
O porquê da criação da matriz identidade, eu explicarei mais à frente.

In [14]:

```
"""

    Vamos novamente criar um array, mas agora vamos fazer que nem o arange só que vamos decidir quantos va
lores queremos
    passando um dado intervalo.

    É meio confuso vejam o exemplo:
    Eu digo que quero um intervalo de 0 a 10, contudo digo que só quero 2 números ... O que ele retorna?

    Retorna: [0, 10]
    Se eu pedisse 3 números ficaria: [0, 5, 10]
    Se pedisse 4 número ficaria: [0, 3.3333333, 6.6666667, 10]

    Ele cria um array num dado intervalo com espaços iguais entre os números.
    Ele conta até X (X é o número máximo do nosso intervalo) com base em quantos números nós queremos.

"""

# Criar array de 0 a 10 em que retorna somente 2 números igualmente espaçados

lista = np.linspace(0, 10, 2)
print("Intervalo de 0 a 10, com 2 número igualmente espaçados: ", lista)

# Criar array de 0 a 10 em que retorna somente 3 números igualmente espaçados

lista = np.linspace(0, 10, 3)
print("Intervalo de 0 a 10, com 3 número igualmente espaçados: ", lista)

# Criar array de 0 a 10 em que retorna somente 4 números igualmente espaçados

lista = np.linspace(0, 10, 4)
print("Intervalo de 0 a 10, com 4 número igualmente espaçados: ", lista)

# Criar array de 0 a 10 em que retorna somente 5 números igualmente espaçados

lista = np.linspace(0, 10, 5)
print("Intervalo de 0 a 10, com 5 número igualmente espaçados: ", lista)
```

```
Intervalo de 0 a 10, com 2 número igualmente espaçados:  [ 0. 10.]
Intervalo de 0 a 10, com 3 número igualmente espaçados:  [ 0.  5. 10.]
Intervalo de 0 a 10, com 4 número igualmente espaçados:  [ 0.          3.33333333  6.66666667
10.        ]
Intervalo de 0 a 10, com 5 número igualmente espaçados:  [ 0.   2.5  5.   7.5 10. ]
```

Espero que tenha dado para entender o que este método faz. Alguma dúvida é só dizer!

In [15]:

```
"""

    Vamos criar alguns arrays/matrizes com números aleatórios


"""

# Criar um array com 10 números ENTRE 0 e 1 aleatórios

lista = np.random.rand(10)
print("Array com 10 números aleatórios[entre 0-1]:\n", lista)
```

```
Array com 10 números aleatórios[entre 0-1]:
 [0.3963623  0.45566789 0.64552633 0.46928107 0.87712831 0.41264246
 0.4734964  0.60565797 0.47325691 0.91364698]
```

Se multiplicarmos por 100 por exemplo obterão resultado entre 0 e 100.

In [16]:

```
# Criar um array com 10 números ENTRE 0 e 100 aleatórios

lista = np.random.rand(10) * 100
print("Array com 10 números aleatórios[entre 0-100]:\n", lista)
```

```
Array com 10 números aleatórios[entre 0-100]:
 [98.88409111 77.08163941 13.39588549 39.31987717 44.05597644 69.86620336
 82.0887926  64.90913717 86.30652569 36.49765819]
```

In [17]:

```
"""

    Também podemos criar matrizes de números aleatórios

"""

# Criar matriz de 10 linhas por 2 colunas (10x2) com número aleatórios de 0 a 1
matriz = np.random.rand(10,2)

print("Matriz 10x2 com número aleatórios:\n", matriz)
```

```
Matriz 10x2 com número aleatórios:
 [[0.28045499 0.04371162]
 [0.06217862 0.10621033]
 [0.41178721 0.4444485 ]
 [0.32144277 0.19412138]
 [0.81610997 0.7750304 ]
 [0.07181731 0.39100106]
 [0.15522744 0.42852564]
 [0.43875585 0.31843559]
 [0.47384322 0.58550299]
 [0.83193642 0.52203983]]
```

In [18]:

```
"""

    Podemos criar arrays/matrizes com números inteiros aleatórios

    Aqui é um pouco diferente, porque temos que passar um intervalo de números, onde o primeiro argumento
é o número mínimo,
    ou seja, de onde parte o intervalo, e o segundo argumento é o número máximo (o fim do intervalo).

    Ainda temos o argumento "size", se especificarem apenas um número, obteremos um array (array de 1 dime
nsão).
    No caso de especificarem uma tupla, definem o número de linhas e colunas (no caso de ser uma matriz 2D
, como nos exemplos
    que tenho mostrado).

    Imaginem que passamos: size=(3,2) - Então teremos uma matriz com 3 linhas e 2 colunas
    Agora se passarmos: size=(3,2,4) - Temos uma matriz, que neste caso é melhor pensarmos que tem um X=3,
um Y=2, e um Z=4,
    ou seja, aqui temos uma matriz de 3 dimensões (3D).

    E assim vai, eu vou ficar pelas 2 Dimensões(Linhas e Colunas) fica muito mais fácil de visualizarem!

"""

# Criar um array com 5 inteiros com inteiros de 0 a 20

lista = np.random.randint(0, 20, size=5)
print("Array de 5 inteiros com inteiros de 0 a 20: ", lista)

# Criar uma matriz com 3 linhas e 2 colunas de 0 a 20

matriz = np.random.randint(0, 20, (3,2))
print("Matriz 3x2 com inteiros de 0 a 20:\n", matriz)
```

```
Array de 5 inteiros com inteiros de 0 a 20:  [ 3 17 17  5  0]
Matriz 3x2 com inteiros de 0 a 20:
 [[13 12]
 [ 2 15]
 [14 12]]
```

Se repararem eu omiti o `size=` no caso da matriz. Isto para vos mostrar que é indiferente, neste caso, especificar explicitamente o `size=`

Nota

Se omitirem o valor máximo, por exemplo: `np.random.randint(5, size=(3,2))` - neste caso os inteiros vão de 0 a 5, e somos obrigados a passar o `size=` explicitamente, **NÃO podemos omitir**.

In [19]:

```
"""

    Outro método extremamente importante é o reshape. Ele permite alterar a estrutura de um array/matriz

"""

# Vamos criar um array de 0 a 15
lista = np.arange(15)

# Agora vamos transformar este array de 1 dimensão em uma matriz de 5 linhas por 3 colunas
# Já que 5 x 3 = 15
print("Array original: ", lista)
print("\nArray transformado em uma matriz 5x3:\n", lista.reshape((5,3)))
```

```
Array original:  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

Array transformado em uma matriz 5x3:
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

Atenção que têm de passar valores válidos para o número de linhas e colunas!!
Para saberem se é um valor válido simplesmente multipliquem ambos, o resultado tem que dar o tamanho do vosso array inicial!

Agora imaginem que uma função está a reclamar porque vocês passaram um array de 1 dimensão e não uma matriz. O que vocês podem fazer??
Bem se fizerem isto abaixo, vocês convertem um array para uma matriz:

In [20]:

```
print( lista.reshape(1, -1) )
```

```
[[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]]
```

Como podemos confirmar ???
Bem, podem comparar o `shape` da lista original com o shape da lista transformada

In [21]:

```
print("Shape original: ", lista.shape)
print("Shape após a lista ser transformada em matriz: ", lista.reshape(1, -1).shape)
```

```
Shape original:  (15,)
Shape após a lista ser transformada em matriz:  (1, 15)
```

Aqui dá para ver que a lista tinha simplesmente 15 elementos dentro dela. Logo depois que foi feito o `reshape` ela passou a ser uma matriz de 1 linha com 15 colunas.

Agora vocês perguntam:
-Certo, então e se eu fizer reshape(-1, 1)? Muda alguma coisa??

Vamos testar!

In [22]:

```
print("reshape(1, -1) = ", lista.reshape(1, -1))
print("reshape(-1, 1) = \n", lista.reshape(-1, 1))
```

```
reshape(1, -1) =  [[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]]
reshape(-1, 1) =
 [[ 0]
 [ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]
 [11]
 [12]
 [13]
 [14]]
```

Dá para ver que a primeira matriz tem simplesmente 1 linha e 15 colunas. Já a segunda tem o contrário, 15 linhas e 1 coluna.
Acho que deu para entender que inverter os valores faz com que o número de linhas troque com o número de colunas.

Este último exemplo fez me lembrar de um atributo dos `numpy.ndarray` (classe de arrays que estamos a usar).
Este atributo é chamado de `T` , que no caso siginfica "Transpose", ou seja, **Transposição**.

Para que serve a transposição? E o que é transposição?

Para que serve é algo que vocês entenderão mais à frente (nas aritméticas). Vamos ver então o que ele faz:

In [23]:

```
"""

    Vou criar 2 matrizes

"""

# Matriz de 1 linha e 5 colunas
matriz1 = np.array([[1,2,3,4,5]])

# Matriz de 5 linhas e 1 coluna
matriz2 = np.array([
    [1],
    [2],
    [3],
    [4],
    [5]
])
```

In [24]:

```
# Agora vamos transpor a primeira matriz

print("matriz1 original: ", matriz1)
print("matriz1 transposta:\n", matriz1.T)
```

```
matriz1 original:  [[1 2 3 4 5]]
matriz1 transposta:
 [[1]
 [2]
 [3]
 [4]
 [5]]
```

Deu para ver?! Passamos de uma matriz de 1 linha e 5 colunas, para uma matriz de 5 linhas e 1 coluna!!!!
Literalmente fizemos o mesmo que `reshape(1, -1)`

In [25]:

```
# Agora vamos transpor a segunda matriz

print("matriz2 original: \n", matriz2)
print("matriz2 transposta: ", matriz2.T)
```

```
matriz2 original:
 [[1]
 [2]
 [3]
 [4]
 [5]]
matriz2 transposta:  [[1 2 3 4 5]]
```

Aconteceu o mesmo novamente, só que na situação oposta! Neste caso fizemos o mesmo que um `reshape(-1, 1)`

Nota

Quando temos uma matriz com 1 linha e N colunas, chamamos essa matriz de, **matriz Linha**(em inglês, row matrix).
Uma matriz com N linhas e 1 coluna, chama-se de **matriz Coluna**(em inglês, column matrix)

É bom saber isto porque é útil quando se faz cálculos entre matrizes.

E agora vamos para os últimos métodos "básicos", estes permitem-nos obter os mínimos e os máximos.

In [26]:

```
"""

    Vamos obter os máximos e os mínimos de uma dada matriz também se aplica a arrays)

"""

matriz = np.array([
    [24, 35],
    [12, 56]
])

print("Máximo da matriz: ", matriz.max())
print("Mínimos da matriz: ", matriz.min())
```

```
Máximo da matriz:  56
Mínimos da matriz:  12
```

In [27]:

```
"""

    Vamos obter os índices dos máximos e mínimos da matriz

"""
print("Índice do máximo da matriz: ", matriz.argmax())
print("Índice do mínimo da matriz: ", matriz.argmin())
```

```
Índice do máximo da matriz:  3
Índice do mínimo da matriz:  2
```

Já mostrei alguns métodos úteis que o numpy disponibiliza e ainda irei mostrar mais.
No entanto, os próximo são mais usados em aritméticas, portanto mostrarei quando estiver nas aritméticas.

Fatiamento e índices de arrays e matrizes

Já sabemos criar arrays/matrizes e aplicar alguns métodos para retirar algumas informações deles. Está a faltar uma coisa:
-*Então e se quisermos apenas os valores de uma parte do array/matriz?*

É isso que vamos aprender aqui, como retirar uma "fatia" do nosso array/matriz

In [28]:

```
# Vamos criar uma matriz com 4 linhas e 4 colunas

matriz = np.array([
    [1,2,3,4],
    [5,6,7,8],
    [9,10,11,12],
    [13,14,15,16]
])

print(matriz)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

Ok, agora que temos a nossa matriz, vamos começar a "fatiá-la".

Vamos começar pelas linhas, eu quero apenas mostrar a linha 2 e 3.

Nota

Lembrem-se que a linha 2 tem índice 1, e a linha 3 tem índice 2, normalmente em programação começamos a contar a partir do 0 e não do 1.

In [29]:

```
# Mostrar apenas as linhas 2 e 3
print(matriz[1:3])
```

```
[[ 5  6  7  8]
 [ 9 10 11 12]]
```

Funcionou!!! Mas secalhar vocês perguntam-se:
Porquê [1:3]?? Não deveria ser [1:2]??

Realmente eu disse acima que o índice da linha 3 era 2, contudo eu coloquei 3 ... Porquê?
Assim como no `range()` o número máximo nunca é contado, ou seja, se o nosso número máximo é 10, ele contará somente até ao 9!
Por essa razão eu passei o índice limite como 3 (que refere-se à linha 4), porque ele vai só contar até ao índice 2 (que refere-se à linha 3).

Recapitulando
Nós queremos ver a linha 2 e 3, então nós passamos o índice 1 que é a linha de partida, e passamos o índice 3 para ver a linha 3 (não esquecendo que a linha 3 tem índice 2 e **NÃO** 3!
Fica assim: `matriz[1:3]`

Agora vamos às colunas. Eu quero ver todas as linhas, mas só quero ver os valores das colunas 1 e 2 (ou seja, índices 0 e 1).

In [30]:

```
# Mostrar todas as linhas, mas apenas as colunas 1 e 2

print( matriz[:, 0:2] )
```

```
[[ 1  2]
 [ 5  6]
 [ 9 10]
 [13 14]]
```

Vamos observar isto bem de perto:

Comecemos pelos primeiros dois pontos -> matriz[***:*** , 0:2] - Passar os dois pontos sem qualquer valor é o mesmo que: `matriz[0:-1, 0:2]` , ou seja, o ponto de partida é no primeiro valor da matriz (índice 0) e o ponto de chegada é no último valor da matriz (índice -1). Logo ele percorre todas as linhas desde da 0 até à última(-1).

Agora vem uma vírgula, essa vírgula separa o fatiamento das linhas e das colunas. Portanto à esquerda da vírgula estamos a definir que linhas queremos ver (no caso queremos ver todas, do índice 0 ao -1), à direita da vírgula vamos definir que colunas queremos ver.

Por fim o nosso: matriz[:, ***0:2***] - este tem a mesma lógica que as linhas. Estou a definir que quero ver a partir da linha 1 (índice 0) até à linha 2 (índice 1, mas passamos 2 porque o python não conta com ele). Uma pergunta pertinente seria:
Então e neste caso, podemos omitir o 0?

E a reposta é ... **CLARO!**, 0 e -1 podem sempre ser omitidos, portanto ficaria assim: `matriz[:, :2]` .

Eu sei, isto por escrito é muito confuso, a melhor forma de entender isto de fatiamento e índices, é colocar na prática e ver os resultados. Vão ver que depois de uns testes vão entender a lógica.

Aritméticas

Por fim chegamos às aritméticas. E é para isso que o numpy serve, para fazer aritméticas em matrizes gigantes! Por essa razão é muito importante saber como aplicá-las e também saber como funcionam.

Soma de arrays/matrizes por números

Caso tenhamos um array/matriz e um número, podemos somá-los facilmente.

In [31]:

```
# Criar um array com 3 elementos

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

# Criar uma matriz de 2 linhas e 3 colunas

matriz = np.array([
    [1,2,3],
    [4,5,6]
])

print( "lista + 5 = ", lista + 5 )
print( "matriz + 5 =\n", matriz + 5 )
```

```
lista + 5 =  [6 7 8]
matriz + 5 =
 [[ 6  7  8]
 [ 9 10 11]]
```

Funcionou certinho! Basicamente o que acontece é que o 5 é somado a cada elemento do array/matriz

Soma de arrays/matrizes por outros arrays/matrizes

Também é possível somar 2 arrays/matrizes ou mesmo 1 array e 1 matriz.

In [32]:

```
# Vou usar a lista e matriz já criada no último exemplo

print( "lista + lista = ", lista + lista )
print( "\nlista + matriz =\n", lista + matriz )
print( "\nmatriz + matriz =\n", matriz + matriz )
```

```
lista + lista =  [2 4 6]

lista + matriz =
 [[2 4 6]
 [5 7 9]]

matriz + matriz =
 [[ 2  4  6]
 [ 8 10 12]]
```

Então *lista + lista* é nada mais nada menos que pegar no valor de índice 0 da 1ª lista e somar ao valor de índice 0 da 2ª lista. E assim sucessivamente

A *matriz + matriz* é a mesma coisa, só que aqui temos múltiplas linhas, pegamos na 1ª linha e 1º valor (índice 0) e somamos à 1ª linha e 1º valor da outra matriz, assim sucessivamente.

A *lista + matriz*, é pegar no array/lista e somar o valor de índice 0 ao valor de índice 0 de cada linha da matriz. Depois o valor de índice 1 ao valor de índice 1 de cada lista ... E assim vai. **Mas espera!**, há uma senão ... O tamanho da lista tem que ser **igual** ao número de colunas, caso contrário vai dar um erro, porque vai sobrar ou faltar valores para somar!
Poderá haver vezes que fazer um `reshape` ao array (para assim tornar-se uma matriz) pode ajudar caso ambas as matrizes fiquem com o mesmo número de colunas.

Subtração de arrays/matrizes por números

É exatamente o mesmo que na soma, portanto é só ler as explicações acima e substituir "soma" por "subtração".

In [33]:

```
print( "lista - 5 = ", lista - 5 )
print( "matriz - 5 =\n", matriz - 5 )
```

```
lista - 5 =  [-4 -3 -2]
matriz - 5 =
 [[-4 -3 -2]
 [-1  0  1]]
```

Subtração de arrays/matrizes por outros arrays/matrizes

Digo o mesmo, é tudo igual à soma.

In [34]:

```
print( "lista - lista = ", lista - lista )
print( "\nlista - matriz =\n", lista - matriz )
print( "\nmatriz - matriz =\n", matriz - matriz )
```

```
lista - lista =  [0 0 0]

lista - matriz =
 [[ 0  0  0]
 [-3 -3 -3]]

matriz - matriz =
 [[0 0 0]
 [0 0 0]]
```

Multiplicação de arrays/matrizes por números

Exatamente o mesmo que soma e subtração, só que agora vamos multiplicar.

Nota

Eu tenho falado: "Soma/Subtração/Multiplicação de arrays/matrizes por números" - mas uma forma de o dizer mais "corretamente", é substituir *números* por *escalares*.

In [35]:

```
print( "lista * 5 = ", lista * 5 )
print( "matriz * 5 =\n", matriz * 5 )
```

```
lista * 5 =  [ 5 10 15]
matriz * 5 =
 [[ 5 10 15]
 [20 25 30]]
```

Multiplicação de arrays/matrizes por arrays/matrizes

Este é outro caso que é igual a todos os outros. **Não se habituem que a seguir vem um tipo de multplicação diferente!**

In [36]:

```
print( "lista * lista = ", lista * lista )
print( "\nlista * matriz =\n", lista * matriz )
print( "\nmatriz * matriz =\n", matriz * matriz )
```

```
lista * lista =  [1 4 9]

lista * matriz =
 [[ 1  4  9]
 [ 4 10 18]]

matriz * matriz =
 [[ 1  4  9]
 [16 25 36]]
```

Produto Escalar entre matrizes

Aqui é um pouco diferente, mas vamos primeiro ao exemplo.

In [37]:

```
# Criar uma matriz de 3 linhas e 4 colunas
a = np.array([
    [1,2,3,4],
    [5,6,7,8],
    [9,10,11,12]
])

# Criar outra matriz de 3 linhas e 4 colunas
b = np.array([
    [13,14,15,16],
    [17,18,19,20],
    [21,22,23,24]
])

print( "Matriz a =\n", a)
print( "\nMatriz b =\n", b)
```

```
Matriz a =
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

Matriz b =
 [[13 14 15 16]
 [17 18 19 20]
 [21 22 23 24]]
```

In [38]:

```
print( "Produto escalar entre a e b =\n", a.dot(b.T))
```

```
Produto escalar entre a e b =
 [[150 190 230]
 [382 486 590]
 [614 782 950]]
```

HORA DA MATEMÁTICA!

Não se assustem, com calma vamos todos entender isto.

Primeiramente, um array é um vetor. E uma matriz é um vetor multidimensional.

Aqui o que se aplica é a regra **RC - Row by Column**, que em português ficaria **LC - Linha por Coluna**. Nós pegamos a primeira linha da matriz *a* = [1,2,3,4] e pegamos a primeira coluna da matriz *b* =

[13,
14,
15,
16]

-*Espera lá! A primeira coluna do vetor b não tem os valores: 13, 17 e 21 ?!?!*

É verdade, mas se repararem nós aplicamos um `.T` (uma transposição). Observem abaixo:

In [39]:

```
print( "\nMatriz b transposta =\n", b.T)
```

```
Matriz b transposta =
 [[13 17 21]
 [14 18 22]
 [15 19 23]
 [16 20 24]]
```

Temos de fazer isto porque a regra **LC ou RC** obriga-nos a que **a matriz *a* tenha tantas linhas quanto a matriz *b* tenha de colunas**. Isto é, se *a* tem 3 linhas, a matriz *b* tem de ter 3 colunas.

Vamos confirmar abaixo:

In [40]:

```
print("Shape da matriz a = ", a.shape)
print("Shape da matriz b transposta = ", b.T.shape)
```

```
Shape da matriz a =  (3, 4)
Shape da matriz b transposta =  (4, 3)
```

Confirma-se, *a* tem 3 linhas e *b transposto* tem 3 colunas.

Continuando a explicação do produto escalar ... Após termos 2 matrizes: uma com N linhas e outra com N colunas - pegamos na 1ª linha e 1º valor da matriz *a* e multplicamos pelo 1º valor da 1ª coluna da matriz *b*.

Depois pegamos na 1ª linha e 2º valor da matriz *a* e multiplicamos pelo 2º valor da 1ª coluna da matriz *b*... e assim vamos.

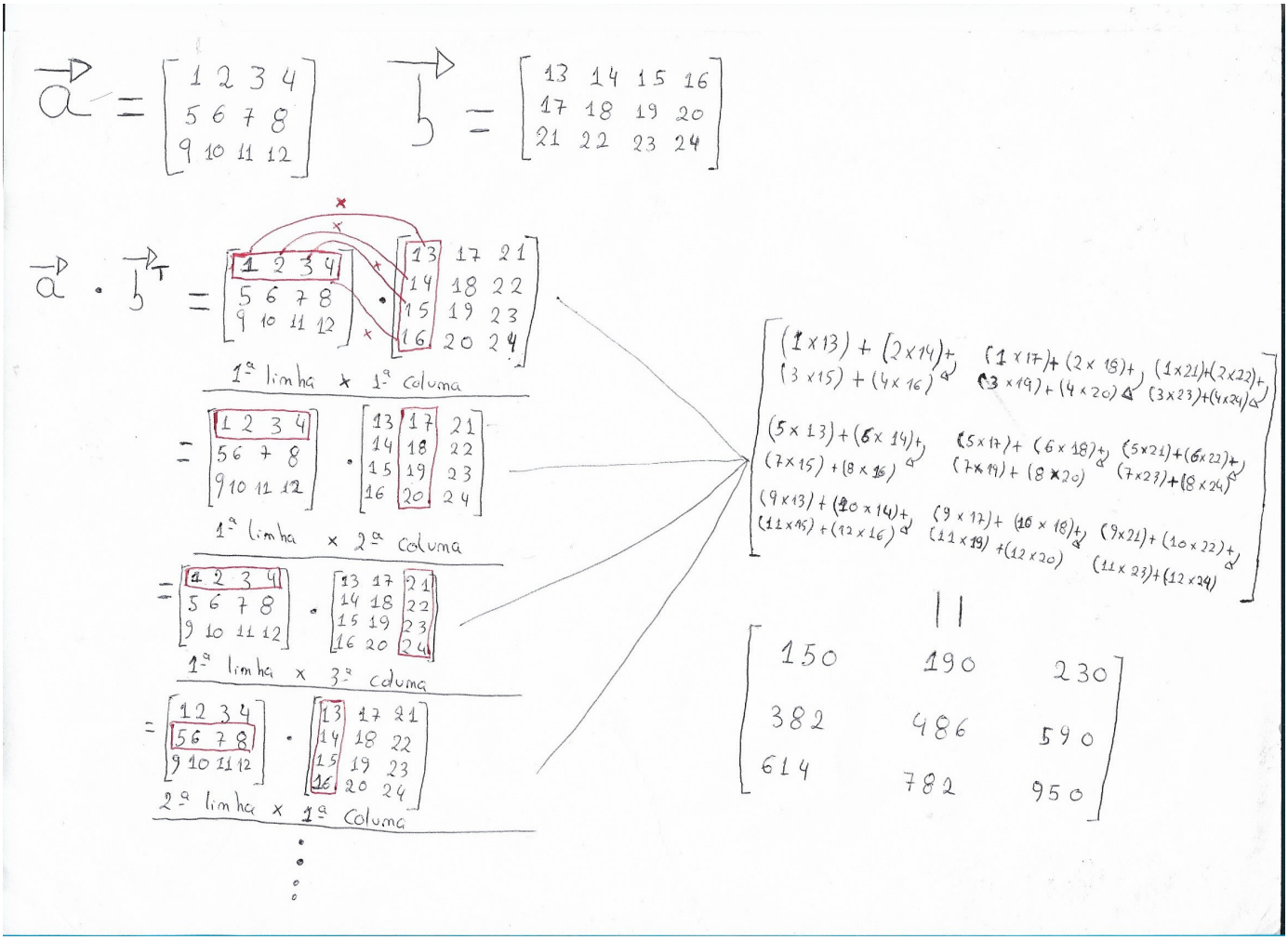
No fim, somamos todos os produtos gerados entre a 1ª linha de uma matriz e 1ª coluna da outra matriz. Esse valor é o produto escalar da linha 1, da matriz *a*, pela coluna 1 da matriz *b*.

De seguida pegamos novamente na 1ª linha e no 1º valor da matriz *a* e multplicamos pela 2ª coluna e 1ª valor da matriz *b*. E vamos multplicar valor por valor, para no fim fazermos novamente a soma de produtos.

Após a 1ª linha da matriz *a* multplicar por todas as colunas da matriz *b*, partimos para a 2ª linha da matriz *a* e repetimos todo o processo...

No fim obtemos uma **matriz quadrada**, isto é, uma matriz com o mesmo número de linhas e colunas. Isto devido à tal regra de **Linha por Coluna**, pois o número de linhas de *a* tem de ser o mesmo que o número das colunas de *b*.

Por ser algo difícil de explicar por palavras, vou mostrar uma imagem que poderá ajudar



Peço desculpa pelas minhas habilidades de esboço serem quase nulas, mas espero que assim dê para esclarecer alguma confusão.

Podemos confirmar os resultados que o numpy obteve e os resultados que calculei manualmente ... e deu exatamente o mesmo!

Nota

O vetor *b* tem expoente *T*, porque é a forma de assinalar transposição em matemática. No numpy limitamo-nos a: `b.T`

Multiplicação de arrays/matrizes por matriz identidade

Eu disse que explicaria o que é matriz identidade mais para a frente, e aqui está.

In [62]:

```
# Criar uma matriz de 3 linhas e 2 colunas
matriz = np.arange(1,7).reshape((3,2))

# Criar uma matriz identidade com 2 linhas e 2 colunas
matriz_identidade = np.identity(2)

print("Matriz 3x2 =\n", matriz)
print("\nMatriz Identidade 2x2 =\n", matriz_identidade)
```

```
Matriz 3x2 =
 [[1 2]
 [3 4]
 [5 6]]

Matriz Identidade 2x2 =
 [[1. 0.]
 [0. 1.]]
```

In [156]:

```
print("Produto escalar entre matriz e matriz identidade =\n", matriz.dot(matriz_identidade))
```

```
Produto escalar entre matriz e matriz identidade =
 [[1. 2.]
 [3. 4.]
 [5. 6.]]
```

Parece que multiplicar a nossa *matriz* pela *matriz identidade*, resulta na própria *matriz*!
O que podemos concluir é que a matriz identidade é como o 1 na multiplicação!

MAS ATENÇÃO: A matriz identidade é uma matriz quadrada (número de linhas = número de colunas), e esse número de linhas e colunas tem que ser **igual** ao número de colunas da matriz em que estão a calcular o produto escalar! Caso contrário obterão erros.

Eu não dei nenhum exemplo com arrays nas últimas explicações, mas é exatamente o mesmo, a diferença que em vez de linhas e colunas, temos apenas elementos (1 linha com N elementos/colunas). O produto escalar no caso não irá ser uma matriz quadrada e sim um número/escalar.

Divisão de arrays/matrizes por escalares

Novamente, é tudo igual às restantes operações. O escalar divide cada elemento do array/matriz.

In [70]:

```
# Crirar uma matriz 3x2
matriz = np.arange(1,7).reshape((3,2))

print("matriz / 2 =\n", matriz/2)
```

```
matriz / 2 =
 [[0.5 1. ]
 [1.5 2. ]
 [2.5 3. ]]
```

Divisão de arrays/matrizes por arrays/matrizes

Bem aqui o caso já muda. Não existe divisão de arrays/matrizes, mas é possível dividí-los.
-O quê!?

Já vamos entender isso, primeiro vamos dividir 2 matrizes e ver o resultado.

In [81]:

```
# Criar uma matriz 3x3
matriz1 = np.arange(1,10).reshape((3,3))

# Criar outra matriz 3x3
matriz2 = np.arange(10, 19).reshape((3,3))
```

In [85]:

```
print("matriz1 / matriz2 =\n", matriz1 / matriz2)
```

```
matriz1 / matriz2 =
 [[0.1        0.18181818 0.25      ]
 [0.30769231 0.35714286 0.4       ]
 [0.4375     0.47058824 0.5       ]]
```

Parece que dividiu (mesmo não "existindo" divisão)!!
Na verdade o que se passou aqui foi uma divisão elemento por elemento (como na multiplicação sem ser com produto escalar). Nós queremos realmente dividir a matriz por outra matriz e não elemento por elemento. E é isso que vou explicar daqui para a frente.

Nota

Os arrays/matrizes têm de ter o mesmo shape (número de linhas, colunas etc).

HORA DA MATEMÁTICA
Agora eu vou explicar isso de "Não haver divisão".

Primeiro quero que pensem em um número, vou usar o 14. E quero que se lembrem que o 14 parece que está sozinho, mas na realidade não está.

$$\frac{14^1}{1}$$

Podemos ver que o número 14 tem expoente 1 e está a ser dividido por 1. Nós omitimos isso, porque 14 elevado a expoente 1 = 14 e 14 dividido por 1 = 14.

Muito bem, agora imaginem que estão num mundo onde não há divisão direta, ou seja, não podem fazer isto: *14 / 7*

No entanto, vocês querem dividir 14 tarefas por 7 pessoas, por exemplo. Como podem fazer isto ?!

A resposta é ... **Multiplicação de um número pelo inverso de outro número**!
Se 7 é o mesmo que 7 dividido por 1, o inverso seria 1 / 7 !

$$\frac{1}{7}$$

Vamos testar então!

$$\frac{14}{1} \times \frac{1}{7} = \frac{14 \times 1}{1 \times 7} = \frac{14}{7} = 7$$

Bateu certo ! 14 / 7 = 7

Tudo isto para explicar que a divisão de 2 matrizes: $a / b = a \,.\, b^{-1}$

$$\vec{a} \div \vec{b} = \vec{a}.\vec{b}^{-1}$$

In [132]:

```
# Criar matriz 2x2
a = np.array([
    [1,2],
    [3,4]
])

# Criar outra matriz
b = np.array([
    [5,6],
    [7,8]
])

# Vamos inverter a matriz b usando o método linalg.inv()
b_inv = np.linalg.inv(b)

print("Matriz a =\n", a)
print("\nMatriz b =\n", b)
print("\nMatriz b inversa =\n", b_inv)
```

```
Matriz a =
 [[1 2]
 [3 4]]

Matriz b =
 [[5 6]
 [7 8]]

Matriz b inversa =
 [[-4.   3. ]
 [ 3.5 -2.5]]
```

Para comprovarmos se a matriz inversa foi corretamente calculada, basta fazer o produto escalar da matriz original pela sua inversa, e o resultado é a matriz identidade!

$$\vec{b}.\vec{b^{-1}}$$

In [134]:

```
print("Produto escalar entre b e b inversa =\n", np.round( b.dot(b_inv) ))
```

```
Produto escalar entre b e b inversa =
 [[1. 0.]
 [0. 1.]]
```

Parece que deu certo!! Eu apliquei um arredondamento, porque nem sempre os valores dão 1 ou 0!
Devido aos arrendondamentos e aos formatos dos números (inteiros, decimais etc), há certas imprecisões (0.00000000000000000000000000234 ou 1,000000000000000004), ao aplicar arredondamento a 0 casas decimais eu escondi essas imprecisões.

In [148]:

```
print("Produto escalar entre a e b inversa =\n", a.dot(b_inv) )
```

```
Produto escalar entre a e b inversa =
 [[ 3. -2.]
 [ 2. -1.]]
```

Bem aplicá-mos a fórmula para calcular a divisão entre 2 matrizes. Têm de ter atenção que há matrizes que não são **inversíveis**, portanto não a podemos dividir por outra matriz. Chamamos a essas matrizes, **matrizes singulares**

-Como podemos calcular a matriz inversa??

Há uma fórmula para isso, mas eu só vou mostrar utilizando uma matriz 2x2, porque à medida que a matriz cresce mais difícil fica aplicar a fórmula (é necessário fazer muitos mais cálculos).

Tendo uma matriz como a seguinte:

$$\begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix}$$

A inversa é calculada através da seguinte fórmula:

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix}^{-1} = \frac{1}{a.d - b.c} . \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Se aplicarmos a fórmula à nossa matriz, resulta em:

$$\begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix}^{-1} = \frac{1}{6 \times 2 - 2 \times 1} . \begin{bmatrix} 2 & -2 \\ -1 & 6 \end{bmatrix} \Leftrightarrow \begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix}^{-1} = \frac{1}{10} . \begin{bmatrix} 2 & -2 \\ -1 & 6 \end{bmatrix} \Leftrightarrow \begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix}^{-1} = \begin{bmatrix} 0.2 & -0.2 \\ -0.1 & 0.6 \end{bmatrix}$$

E assim obtemos a matriz inversa da nossa matriz original.

Outras operações em arrays/matrizes

Podemos também calcular raízes quadradas e aplicar expoentes nos elementos do array/matriz.

In [150]:

```
print("Matriz a =\n", a)
```

```
Matriz a =
 [[1 2]
 [3 4]]
```

In [153]:

```
print("Raíz quadrada da Matriz a =\n", np.sqrt(a))
print("\nMatriz a expoente 2 =\n", a ** 2)
```

```
Raíz quadrada da Matriz a =
 [[1.         1.41421356]
 [1.73205081 2.        ]]

Matriz a expoente 2 =
 [[ 1  4]
 [ 9 16]]
```

Conclusão

Aqui pudemos ver um pouco do potencial do numpy. Lembro que não há mal algum não quererem já focar na matemática por detrás dos métodos. Primeiro de tudo brinquem com tudo isto, façam os vossos próprios testes, releiam etc.

Isto não é um curso, mas sim um conjunto de apontamentos, portanto levem isto como um dicionário com alguns métodos e explicações de Numpy.

Talvez ainda adicione e explique outros métodos que me recorde e que ache importante vocês terem em mente.
Finalizo pedindo que caso haja dúvidas ou queiram inserir algo, é só contactar-me (todos os meus contactos estão aqui abaixo).

Deixo aqui alguns links para vídeo aulas da khan academy para ajudar-vos com a matemática:

Soma e subtração de matrizes (https://pt.khanacademy.org/math/algebra-home/alg-matrices/alg-adding-and-subtracting-matrices/v/matrix-addition-and-subtraction-1)
Multiplicação de matrizes por escalares (https://pt.khanacademy.org/math/algebra-home/alg-matrices/alg-multiplying-matrices-by-scalars/v/scalar-multiplication)
Multiplicação de matrizes por matrizes (https://pt.khanacademy.org/math/algebra-home/alg-matrices/alg-multiplying-matrices-by-matrices/v/matrix-multiplication-intro)
Introdução às matrizes inversas (https://pt.khanacademy.org/math/algebra-home/alg-matrices/alg-intro-to-matrix-inverses/v/inverse-matrix-part-1)

Contactos

Twitter (https://twitter.com/iN127pkt)
Instagram (https://www.instagram.com/t_1g4_x/)
Email (tiagodeha@protonmail.com)