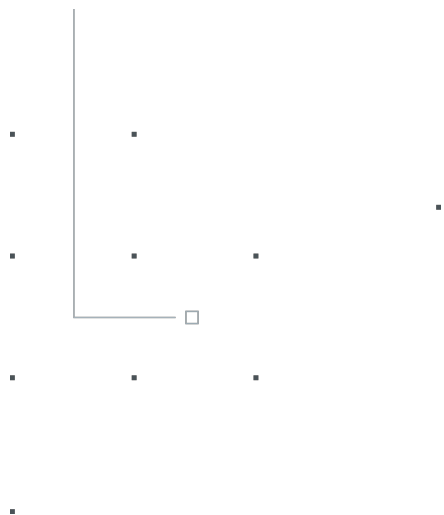


FIAP

NBA



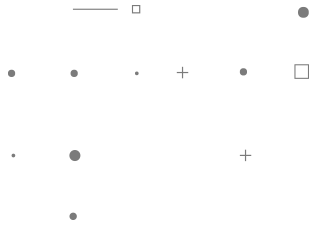
Numpy

Dheny R. Fernandes

1. Numpy

1. Introdução
2. Nddarray
3. Operações entre Arrays e Escalares
4. Indexação e Fatiamento
5. Indexação Booleana
6. Funções Universais
7. Localização Baseada em Condição e Métodos Estatísticos
8. Álgebra Linear
9. Geração de Números Aleatórios
10. Funções Avançadas
11. Outros Métodos

Numpy



Numpy, abreviação para *Numerical Python*, é o pacote fundamental requerido para **computação científica de alta performance e análise de dados**.

Alguns recursos:

- ndarray***: um rápido e multidimensional array que provê operações aritméticas vetorizadas;

- Funções matemáticas padrões para rápidas operações em todo array de dados sem ter de escrever loops

- Álgebra linear, geração de números aleatórios e transformada de Fourier

O principal motivo para entender bem *numpy* é tornar o uso da biblioteca *pandas* mais eficiente.

Pandas e numpy serão a base para praticamente tudo que estudaremos à frente:

- rápidas operações vetorizadas de arrays para *data munging (wrangling)*, limpeza de dados, filtragem e subconjuntos e transformação
- Ordenação, operação com conjuntos e unicidade
- Estatística descritiva e sumarização de dados
- Junção de conjuntos de dados heterogêneos
- Manipulação de dados (agregação, transformação, aplicação de funções)

Uma das razões do NumPy ser tão importante para cálculos computacionais é pelo motivo de ter sido projetado para eficiência em grandes arrays de dados. Há algumas razões para isso:

- Numpy armazena internamente os dados num bloco de memória contíguo, independentemente de outros objetos python. A biblioteca de algoritmos do NumPy escritos na linguagem C pode operar nesta memória sem qualquer verificação de tipo ou outra sobrecarga. Os arrays NumPy também usam muito menos memória do que as sequências(listas, tuplas) internas do Python.
- As operações NumPy executam cálculos complexos em arrays inteiros sem a necessidade de loops for do Python, que podem ser lentos para grandes sequências. O NumPy é mais rápido que o código Python regular porque seus algoritmos baseados em C evitam a sobrecarga presente com o código Python interpretado.
- Veja o código para exemplificação:

Ok, mas por qual motivo usar numpy arrays e não listas?

- A principal diferença entre um array e uma lista é que **arrays foram projetados para manipular operações vetorizadas**, enquanto as listas não.
- Exemplo: somar 1 a cada um dos elementos de uma lista:

```
import numpy as np
list1 = [0,1,2,3,4]
arr1d = np.array(list1)
# principal diferença entre list e array
print(arr1d + 2)
print(list1 + 2)  # erro
```


Outra importante diferença é que, uma vez criado o numpy array, **seu tamanho não pode ser aumentado**.

- Já para list, isso é facilmente conseguido através do método *append()*.

Todos os itens de um array devem ser do **mesmo tipo**, diferentemente os itens de uma lista.

Veja um exemplo:

```
x = [1,2,3,4.5,5.7]
print('----- lists ----- \n')
print(type(x))
print(x)
print(type(x[1]))
print(type(x[4]))
print('\n')
print('----- array ----- \n')
y = np.array(x)
print(type(y))
print(y)
print(type(y[1]))
print(type(y[4]))
```

O ndarray, ou *N-dimensional array*, é um rápido e flexível container para grandes conjuntos de dados em python.

Considere o exemplo:

```
import numpy as np
data1 = [6,7.5,8,0,1]
arr1 = np.array(data1)
arr1
```

O ndarray é um container genérico multidimensional para **dados homogêneos**, isto é, todos os elementos precisam ser do mesmo tipo.

Todo array possui o atributo **shape**, uma tupla indicando o tamanho de cada dimensão, e o atributo **dtype**, que descreve o tipo de dado presente no array:

```
print(arr1.shape)
print(arr1.dtype)
```

Numpy provê outras maneiras de criar um array:

- **np.zeros**: cria arrays de 0's
- **np.ones**: cria arrays de 1's
- **np.empty**: cria arrays com valores aleatórios

```
data3 = np.zeros(10)
data4 = np.ones(10)
data5 = np.empty([2,2])
print(data3)
print(data4)
print(data5)
```

É possível criar números sequenciais com numpy usando a função *arange()*.

```
data6 = np.arange(15)  
print(data6)
```

A matriz identidade pode ser criada usando a função *eye()*.

```
data7 = np.eye(5)  
print(data7)
```

A tabela abaixo resume os principais métodos para criação de um array:

| Function | Description |
|---|---|
| <code>array</code> | Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a data type or explicitly specifying a data type; copies the input data by default |
| <code>asarray</code> | Convert input to ndarray, but do not copy if the input is already an ndarray |
| <code>arange</code> | Like the built-in <code>range</code> but returns an ndarray instead of a list |
| <code>ones</code> , <code>ones_like</code> | Produce an array of all 1s with the given shape and data type; <code>ones_like</code> takes another array and produces a ones array of the same shape and data type |
| <code>zeros</code> , <code>zeros_like</code> | Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0s instead |
| <code>empty</code> , <code>empty_like</code> | Create new arrays by allocating new memory, but do not populate with any values like <code>ones</code> and <code>zeros</code> |
| <code>full</code> , <code>full_like</code> | Produce an array of the given shape and data type with all values set to the indicated “fill value”; <code>full_like</code> takes another array and produces a filled array of the same shape and data type |
| <code>eye</code> , <code>identity</code> | Create a square $N \times N$ identity matrix (1s on the diagonal and 0s elsewhere) |

O tipo de dado, ou *dtype*, é um objeto especial que contém a informação que o Python precisa para interpretar um pedaço de memória como um particular tipo de dado.

Existem diversos tipos de dados que podem ser usados, como mostra a tabela a seguir:

| Tipo | Código | Descrição |
|-------------------|----------|--|
| int8, uint8 | i1, u1 | Signed and unsigned 8-bit (1 byte) integer types |
| int16, uint16 | i2, u2 | Signed and unsigned 16-bit integer types |
| int32, uint32 | i4, u4 | Signed and unsigned 32-bit integer types |
| int64, uint64 | i8, u8 | Signed and unsigned 32-bit integer types |
| float16 | f2 | Half-precision floating point |
| float32 | f4 or f | Standard single-precision floating point. Compatible with C float |
| float64, float128 | f8 or d | Standard double-precision floating point. Compatible with C double and Python float object |
| float128 | f16 or g | Extended-precision floating point |

| Tipo | Código | Descrição |
|-----------------------------------|--------------|--|
| complex64, complex128, complex256 | c8, c16, c32 | Complex numbers represented by two 32, 64, or 128 floats, respectively |
| bool | ? | Boolean type storing True and False values |
| object | O | Python object type |
| string_ | S | Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'. |
| unicode_ | U | Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_ (e.g. 'U10'). |

Declarando um tipo num array:

```
arr1 = np.array([1, 2, 3], dtype=np.float64)
print(arr1.dtype)
arr2 = np.array([1, 2, 3], dtype=np.int32)
print(arr2.dtype)
```

É possível converter (*cast*) de um *dtype* para outro usando *astype()*.

```
arr = np.array([1, 2, 3, 4, 5])
print(arr.dtype)
float_arr = arr.astype(np.float64)
print(float_arr.dtype)
```

Se converter um float num int, o Python irá truncar a parte decimal.

```
arr2 = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])  
print(arr2)  
arr2.astype(np.int32)
```

Convertendo array de string representando número em float.

```
numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)  
numeric_strings.astype(float)
```

Arrays são importantes pois eles **permitem realizar operações em bloco de dados sem a necessidade de um *loop*.**

À isto, denominamos **vetorização**.

Em qualquer operação aritmética entre arrays de mesmo tamanho é aplicada uma operação elemento-a-elemento

Operação entre arrays

```
arr3 = np.array([[1., 2., 3.], [4., 5., 6.]])  
print(arr3)  
print('Multiplicação: \n', arr3 * arr3)  
print('Subtração: \n', arr3 - arr3)
```

Operação entre array e escalar

```
arr3 = np.array([[1., 2., 3.], [4., 5., 6.]])  
print(arr3)  
print('Escalar dividindo array: \n', 1/arr3)  
print('Raiz quadrada: \n', arr3 ** 0.5)
```

De maneira simples, selecionar um subconjunto de dados ou um elemento individual é semelhante às listas:

```
arr = np.arange(10)
print(arr)
print(arr[5])
print(arr[5:8])
arr[5:8] = 12 #atribuição
print(arr)
```



Esse processo é denominado *broadcast*

É importante destacar que arrays são *views* do array original. Isto significa que os dados não são copiados e qualquer modificação na *view* será refletida no array original.

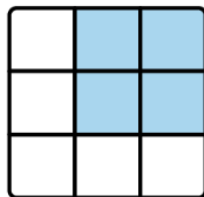
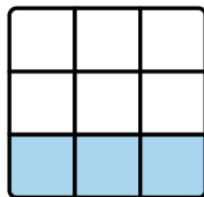
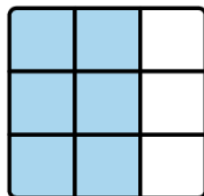
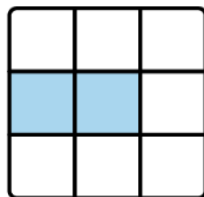
```
arr_slice = arr[5:8]
print(arr_slice)
arr_slice[1] = 12345
print(arr_slice)
print(arr)
```


Indexação – array 2D

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
print(arr2d[2])  
print(arr2d[0][2]) #igual a print(arr2d[0, 2])
```

Fatiamento – array 2D

```
print(arr2d[:2, 1:])  
print(arr2d[1, :2])  
print(arr2d[:, :1])
```

**Expression**`arr[:2, 1:]`**Shape**`(2, 2)``arr[2]``(3,)``arr[2, :]``(3,)``arr[2:, :]``(1, 3)``arr[:, :2]``(3, 2)``arr[1, :2]``(2,)``arr[1:2, :2]``(1, 2)`

Vamos considerar um exemplo em que temos alguns dados num array e um array de nomes com valores duplicados.

```
names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2], [-12, -4], [3, 4]])
```

Suponha que cada nome corresponda a uma linha no array data e queremos selecionar todas as linhas que correspondam ao nome “Bob”. Da mesma maneira que operações aritméticas, comparações (==) são também vetorizadas. Observe:

```
names == "Bob"
```

Podemos passar esse array boolean para indexar o array:

```
data[names == "Bob"]
```

Para fazer a operação contrária, ou seja, selecionar todos os nomes menos “Bob”, podemos fazer o seguinte:

```
~(names == "Bob")
```

```
data[~(names == "Bob")]
```

Para combinar múltiplas condições booleanas, use operadores booleanos como & (and) e | (or):

```
mask = (names == "Bob") | (names == "Will")  
data[mask]
```

Importante notar que as palavras “and” e “or” não funcionam com arrays booleanos

Resolver o exercício no notebook

Transposta: A transposta da matriz $M = [a]_{ij}$ é a matriz $M^T = [a]_{ji}$, para todo elemento de M

De maneira simplificada, toda linha de M vira coluna em M^T e toda coluna de M vira linha em M^T .

```
arr = np.arange(15).reshape((3, 5))  
print(arr)  
print(arr.T)
```

Flattening é o processo de **transformar um array multidimensional num array 1D**.

Existem 2 maneiras de implementar esse processo: usando o método *flatten()* ou usando o método *ravel()*:

- A diferença entre eles é que o novo array criado pelo método *ravel()* é, na verdade, referência ao array pai.
- Assim, qualquer mudança no novo array irá refletir no array pai
- No entanto, é mais eficiente em questão de memória

Exemplo:

```
print('----- flatten() -----\\n')
print(arr2d.flatten())
b1 = arr2d.flatten()
b1[0] = 100
print(arr2d, '\\n')
print('----- ravel() -----\\n')
b2 = arr2d.ravel()
b2[0] = 101
print(arr2d)
```


Uma função universal é uma função que **realiza operações elemento-a-elemento nos dados de um *ndarray***. Você pode considerá-los como wrappers vetorizados rápidos para funções simples que usam um ou mais valores escalares e produzem um ou mais resultados escalares.

```
arr = np.arange(10)
print(arr)
np.sqrt(arr) #raiz quadrada
np.exp(arr) #e^x
```

| Function | Description |
|--|---|
| <code>abs, fabs</code> | Compute the absolute value element-wise for integer, floating point, or complex values. Use <code>fabs</code> as a faster alternative for non-complex-valued data |
| <code>sqrt</code> | Compute the square root of each element. Equivalent to <code>arr ** 0.5</code> |
| <code>square</code> | Compute the square of each element. Equivalent to <code>arr ** 2</code> |
| <code>exp</code> | Compute the exponent e^x of each element |
| <code>log, log10, log2, log1p</code> | Natural logarithm (base e), log base 10, log base 2, and $\log(1 + x)$, respectively |
| <code>sign</code> | Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative) |
| <code>ceil</code> | Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element |
| <code>floor</code> | Compute the floor of each element, i.e. the largest integer less than or equal to each element |
| <code>rint</code> | Round elements to the nearest integer, preserving the dtype |
| <code>modf</code> | Return fractional and integral parts of array as separate array |
| <code>isnan</code> | Return boolean array indicating whether each value is NaN (Not a Number) |
| <code>isfinite, isinf</code> | Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively |
| <code>cos, cosh, sin, sinh, tan, tanh</code> | Regular and hyperbolic trigonometric functions |
| <code>arccos, arccosh, arcsin, arcsinh, arctan, arctanh</code> | Inverse trigonometric functions |
| <code>logical_not</code> | Compute truth value of <code>not x</code> element-wise. Equivalent to <code>-arr</code> . |

| Function | Description |
|---|--|
| <code>add</code> | Add corresponding elements in arrays |
| <code>subtract</code> | Subtract elements in second array from first array |
| <code>multiply</code> | Multiply array elements |
| <code>divide</code> , <code>floor_divide</code> | Divide or floor divide (truncating the remainder) |
| <code>power</code> | Raise elements in first array to powers indicated in second array |
| <code>maximum</code> , <code>fmax</code> | Element-wise maximum. <code>fmax</code> ignores NaN |
| <code>minimum</code> , <code>fmin</code> | Element-wise minimum. <code>fmin</code> ignores NaN |
| <code>mod</code> | Element-wise modulus (remainder of division) |
| <code>copysign</code> | Copy sign of values in second argument to values in first argument |
| <code>greater</code> , <code>greater_equal</code> , <code>less</code> , <code>less_equal</code> , <code>equal</code> , <code>not_equal</code> | Perform element-wise comparison, yielding boolean array. Equivalent to infix operators <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>==</code> , <code>!=</code> |
| <code>logical_and</code> , <code>logical_or</code> , <code>logical_xor</code> | Compute element-wise truth value of logical operation. Equivalent to infix operators <code>&</code> , <code> </code> , <code>^</code> |

Suponha que eu queira pegar um valor de *xarr* se o valor correspondente em *cond* for True ou então eu pego o valor de *yarr*.

```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])  
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])  
cond = np.array([True, False, True, True, False])
```

```
result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]  
result
```

Entretando, além da existir uma maneira mais pythonica, essa solução é lenta para arrays muito grandes.

Podemos usar `numpy.where`:

- `numpy.where`: é a versão **vetorizada** da expressão ternária *x if condição else y*:

```
result = np.where(cond, xarr, yarr)
result
```

Numpy.where funciona muito bem com escalares também:

```
arr = np.random.randn(4, 4)
print(arr)
print(np.where(arr > 0, 2, -2)) # atribui 2 p/ x>0, -2 caso contrário
print(np.where(arr > 0, 2, arr)) # atribui 2 p/ x>0, mantém original caso contrário
```

Existe um conjunto de funções matemáticas que calcula estatísticas num determinado array ou sobre apenas um dos eixos.

```
arr = np.random.randn(5, 4)
print(arr)
print(arr.mean()) #ou np.mean(arr)
print(arr.sum())
print(arr.mean(axis=1))
print(arr.sum(0))
```

Outros métodos estatísticos:

| Method | Description |
|----------------|---|
| sum | Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0. |
| mean | Arithmetic mean. Zero-length arrays have NaN mean. |
| std, var | Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n). |
| min, max | Minimum and maximum. |
| argmin, argmax | Indices of minimum and maximum elements, respectively. |
| cumsum | Cumulative sum of elements starting from 0 |
| cumprod | Cumulative product of elements starting from 1 |

Álgebra Linear, como multiplicação de matriz, decomposição, determinantes e outros cálculos matemáticos é uma importante parte dos arrays e base para entender modelos de ML.

Multiplicação de matrizes:

```
x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])
print(x)
print(y)
z = np.matmul(x,y) #igual aa np.dot(x,y)
print(z)
```

numpy.linalg possui um conjunto de decomposição de matrizes e outras coisas como inversa e determinante.

Inversa:

```
import numpy as np
from numpy.linalg import inv
X = np.random.randn(5, 5)
print(X)
mat = np.matmul(X.T,X) #transposta * X
print(mat)
print(inv(mat)) #inversa
```

| Function | Description |
|----------|--|
| diag | Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal |
| dot | Matrix multiplication |
| trace | Compute the sum of the diagonal elements |
| det | Compute the matrix determinant |
| eig | Compute the eigenvalues and eigenvectors of a square matrix |
| inv | Compute the inverse of a square matrix |
| pinv | Compute the Moore-Penrose pseudo-inverse inverse of a square matrix |
| qr | Compute the QR decomposition |
| svd | Compute the singular value decomposition (SVD) |
| solve | Solve the linear system $Ax = b$ for x , where A is a square matrix |
| lstsq | Compute the least-squares solution to $y = Xb$ |

Geração de número aleatório é uma constante em ciência de dados e o numpy apresenta diversas funções para fazermos isso de modo eficiente:

Simple random data

| | |
|--|--|
| <code>rand</code> (d0, d1, ..., dn) | Random values in a given shape. |
| <code>randn</code> (d0, d1, ..., dn) | Return a sample (or samples) from the "standard normal" distribution. |
| <code>randint</code> (low[, high, size, dtype]) | Return random integers from <i>low</i> (inclusive) to <i>high</i> (exclusive). |
| <code>random_integers</code> (low[, high, size]) | Random integers of type np.int between <i>low</i> and <i>high</i> , inclusive. |
| <code>random_sample</code> ([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| <code>random</code> ([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| <code>ranf</code> ([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| <code>sample</code> ([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| <code>choice</code> (a[, size, replace, p]) | Generates a random sample from a given 1-D array |
| <code>bytes</code> (length) | Return random bytes. |

Permutations

`shuffle` (x) Modify a sequence in-place by shuffling its contents.

`permutation` (x) Randomly permute a sequence, or return a permuted range.

Distributions

`beta` (a, b[, size])

Draw samples from a Beta distribution.

`binomial` (n, p[, size])

Draw samples from a binomial distribution.

`chisquare` (df[, size])

Draw samples from a chi-square distribution.

`dirichlet` (alpha[, size])

Draw samples from the Dirichlet distribution.

`exponential` ([scale, size])

Draw samples from an exponential distribution.

`f` (dfnum, dfden[, size])

Draw samples from an F distribution.

`gamma` (shape[, scale, size])

Draw samples from a Gamma distribution.

`geometric` (p[, size])

Draw samples from the geometric distribution.

`gumbel` ([loc, scale, size])

Draw samples from a Gumbel distribution.

`hypergeometric` (ngood, nbad, nsample[, size])

Draw samples from a Hypergeometric distribution.

Random generator

| | |
|--------------------------------|--|
| <code>RandomState</code> | Container for the Mersenne Twister pseudo-random number generator. |
| <code>seed</code> ([seed]) | Seed the generator. |
| <code>get_state</code> () | Return a tuple representing the internal state of the generator. |
| <code>set_state</code> (state) | Set the internal state of the generator from a tuple. |

Alguns exemplos:

```
# gera 4 números aleatórios
# entre [0.0,1.0)
samples = np.random.random(4)
samples
```

Alguns exemplos:

```
# A geração de números é determinística  
# Ou seja, é baseada num estado interno  
# que chamamos de seed  
np.random.seed(42) # set seed  
print(np.random.random())  
np.random.seed(42) # reset seed  
print(np.random.random())
```

```
# shuffle reordena um array  
# é muito usado antes do processo  
# de treino de algoritmo de ML  
x = np.arange(10)  
np.random.shuffle(x)  
x
```

vectorize() proporciona a possibilidade de uma função escrita para trabalhar em números individuais operar em vetores.

Considere o exemplo:

```
def modulo(x):  
    if x % 2 == 1:  
        return x**2  
    else:  
        return x/2  
  
print('x = 10 retorna ', modulo(10))  
print('x = 11 retorna ', modulo(11))  
print('x = [10, 11, 12] retorna ', modulo([10, 11, 12])) # Erro
```


Assim, podemos usar *numpy.vectorize()* para aplicar essa função a todos os elementos de um vetor:

```
modulo_vect = np.vectorize(modulo, otypes=[float])

print('x = [10, 11, 12] retorna ', modulo_vect([10, 11, 12]))
print('x = [[10, 11, 12], [1, 2, 3]] retorna ', modulo_vect([[10, 11, 12], [1, 2, 3]]))
```

Como encontrar a diferença entre o valor máximo e mínimo em cada linha?

- A abordagem normal seria escrever um loop que itera ao longo da linha e computar *max-min*.

Essa não é a maneira mais pythônica de resolver o problema

Uma maneira mais **elegante** de resolver o problema é utilizar *numpy.apply_along_axis*.

Esse método recebe 3 argumentos:

- Função que opera num vetor 1D;
- Eixo no qual aplicar a função:
 - 1 para linha, 0 para coluna
- Vetor que será aplicado a função

Criando a matriz:

```
#criando matriz  
np.random.seed(100)  
arr_x = np.random.randint(1,10,size=[4,10])  
arr_x
```

Definindo a função e aplicando:

```
#função que calcula diferença  
def max_minus_min(x):  
    return np.max(x) - np.min(x)  
  
# aplicando nas linhas  
print('Por linha: ', np.apply_along_axis(max_minus_min, 1, arr=arr_x))  
  
# aplicando nas colunas  
print('Por coluna: ', np.apply_along_axis(max_minus_min, 0, arr=arr_x))
```

Any e All:

```
bools = np.array([False, False, True, False])  
print(bools.any()) # se houver pelo menos um True  
print(bools.all()) # se todos forem True
```

Ordenação

```
arr = np.random.randn(8)  
print(arr)  
arr.sort() #in-place  
print(arr)
```

Operações com conjuntos (válidos para array 1D):

- *Unique*: retorna um array de valores ordenados

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
print(np.unique(names))
ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
print(np.unique(ints))
```

Relação de Pertencimento: retorna um array booleano indicando True se um elemento de y pertence a x e False, caso contrário

```
values = np.array([6, 0, 0, 3, 2, 5, 6])
np.in1d(values, [2, 3, 6])
```

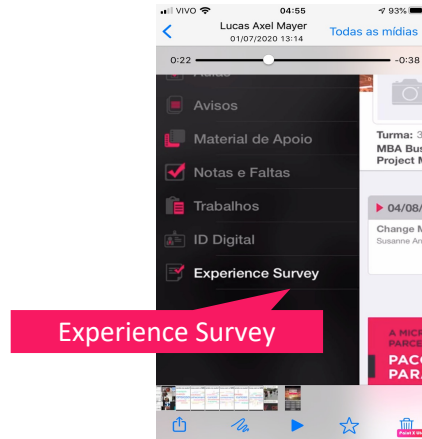
Demais operações com conjunto:

| Method | Description |
|--------------------------------|--|
| <code>unique(x)</code> | Compute the sorted, unique elements in x |
| <code>intersect1d(x, y)</code> | Compute the sorted, common elements in x and y |
| <code>union1d(x, y)</code> | Compute the sorted union of elements |
| <code>in1d(x, y)</code> | Compute a boolean array indicating whether each element of x is contained in y |
| <code>setdiff1d(x, y)</code> | Set difference, elements in x that are not in y |
| <code>setxor1d(x, y)</code> | Set symmetric differences; elements that are in either of the arrays, but not both |

Resolver o exercício no notebook

O que você achou da aula de hoje?


Entrar no aplicativo FIAPP, e no menu clicar em Experience Survey



Ou pelo link: <https://fiap.me/Pesquisa-MBA>

Obrigado!

profdheny.fernandes@fiap.com.br

 /dhenyfernandes

FIAP MBA⁺

Copyright © 2018 | Professor (a) Nome do Professor
Todos os direitos reservados. Reprodução ou divulgação total ou parcial deste documento, é expressamente proibido sem consentimento formal, por escrito, do professor/autor.

FIAP