PACKAGES FOR MACHINE LEARNING

Extending Python's capabilities in Data Science

Here are some important Python libraries, already present in the Anaconda environment, that will make our ML project development task much easier and more productive

- NumPy (Documentation: https://numpy.org/doc/)
  - *for multidimensional arrays, linear algebra, and other high-level mathematical computing operations*
- Pandas (Documentation: https://pandas.pydata.org/docs/)
  - *representation and processing data in the form of tables (dataframes)*
- Matplotlib (Documentation: https://matplotlib.org/contents.html)
  - *for graphical visualization of the data*
- Scikit-learn (https://scikit-learn.org/stable/user_guide.html)
  - *for ML (provides us with the main ML algorithms)*

Brief introduction to NumPy[1]

- In Python, we use a list to easily collect elements
  - *a list is a concept somewhat equivalent to the arrays of other languages, although it gives us other flexibility*
    - for example, the list allows us to collect a heterogeneous set of elements (elements of different types)

```
list=[‘wednesday’, 28, ‘October’, 2020]
```

  - *But this interesting feature of lists can also prove to be a disadvantage when large amounts of data are concerned*
    - which is typically what goes on with the datasets handled in ML
- To make lists flexible structures, Python ensures access to and manipulation of your elements with computationally inefficient operations
  - *NumPy helps overcome this difficulty,*
    - by supporting multidimensional arrays
    - and a whole set of very useful mathematical operations and functions, which allow us to efficiently operate the large volumes of data represented in these structures.
    - However, unlike lists, in NumPy arrays the elements have to be all of the same type (homogeneous collections)

[1] *With content adapted from “A Whirlwind Tour of Python”, of Jake Vanderplas, O’Reilly, 2016 (with CC0 licensing)*

Arrays NumPy[1]

The NumPy package then provides us with an efficient way to store and manipulate multidimensional arrays in Python

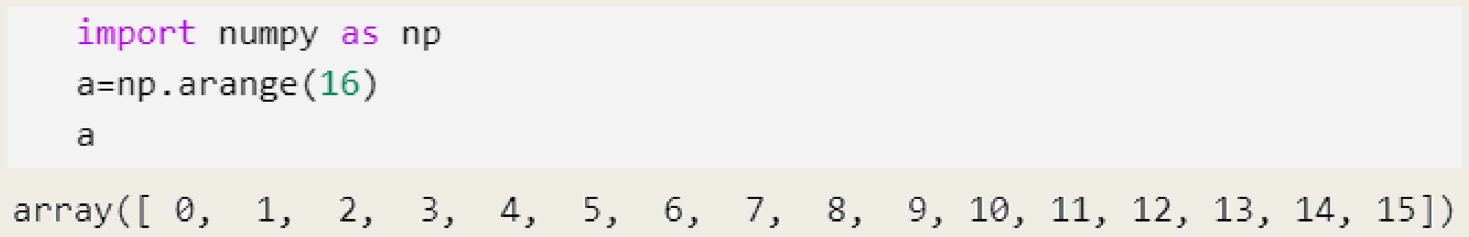
As we study later, Python's main library for ML – scikit-learn – uses two-dimensional NumPy arrays to represent the datasets processed by its algorithms

Key Features of NumPy:

- *Provides a structure ndarray, (n-dimensional array) that enables efficient storage and manipulation of vectors, matrices, and other higher-order structures.*
- *Provides readable and efficient syntax for operating with these data structures, from simple elemental aritmetics to more complex linear algebraic operations.*

In the simplest cases, NumPy arrays look a lot like Python lists

- *For example, the following is an NumPy array containing the number range from 0 to 15 (equivalent to python's range()) function:*



```
import numpy as np
a=np.arange(16)
a
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

Multidimensional NumPy Arrays

- NumPy arrays make both data storage and element-by-element operations efficient.
  - *For example, to square each array element, simply apply the operator directly to the array**:*

```
a**2
```

```
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144, 169, 196, 225])
```

- Operation that would result much more tangled if it was performed python-style, using lists comprehension:

```
[x**2 for x in range(16)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
```

- Unlike Python lists (which are limited to one dimension), NumPy arrays can be multidimensional
  - *For example, we'll change the shape of array x, from one-dimensional to two-dimensional 4x4:*

```
m=a.reshape(4,4)
m
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Two-dimensional NumPy Arrays

- A two-dimensional array represents an array, and NumPy knows how to efficiently perform typical array operations
  - *For example, the transposition of an array can be easily obtained by invoking the T operator (the same as invoking the transpose() method):*

```
m.T
```

```
array([[ 0,  4,  8, 12],
       [ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15]])
```

  - *and the product of an array by a vector can be performed with np.dot():*

```
np.dot(m, [3,2,1,0])
```

```
array([ 4, 28, 52, 76])
```

  - *and even more sophisticated operations such as the decomposition of a matrix into own values:*

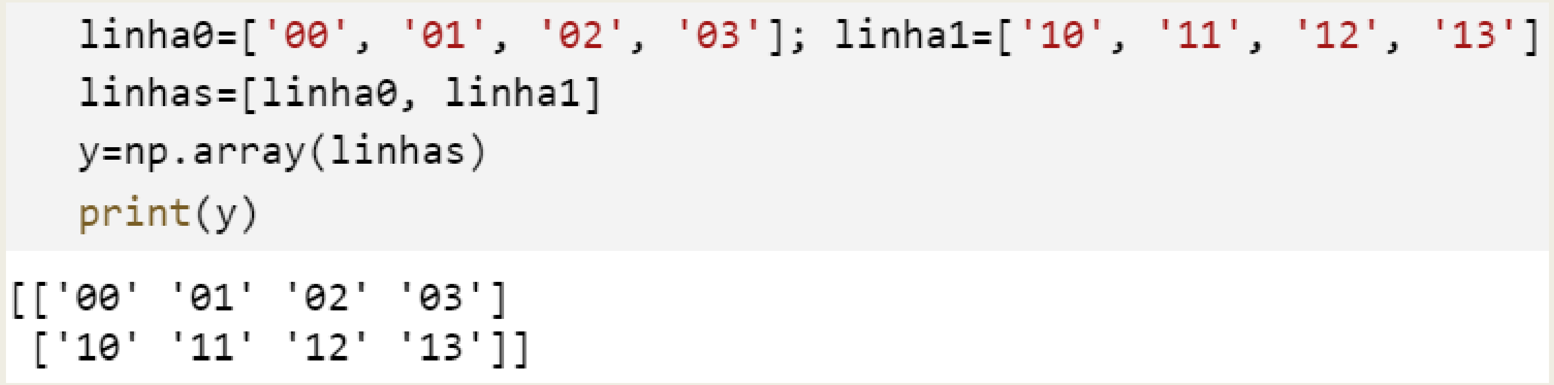
```
np.linalg.eigvals(m)
```

```
array([ 3.24642492e+01, -2.46424920e+00,  1.92979794e-15, -4.09576009e-16])
```

- Linear algebra operations support much of the data analysis currently in ML and data mining.

Creating NumPy arrays

- With the numpay.array() function, we were able to create any multidimensional array, using a list structure to define its shape and content



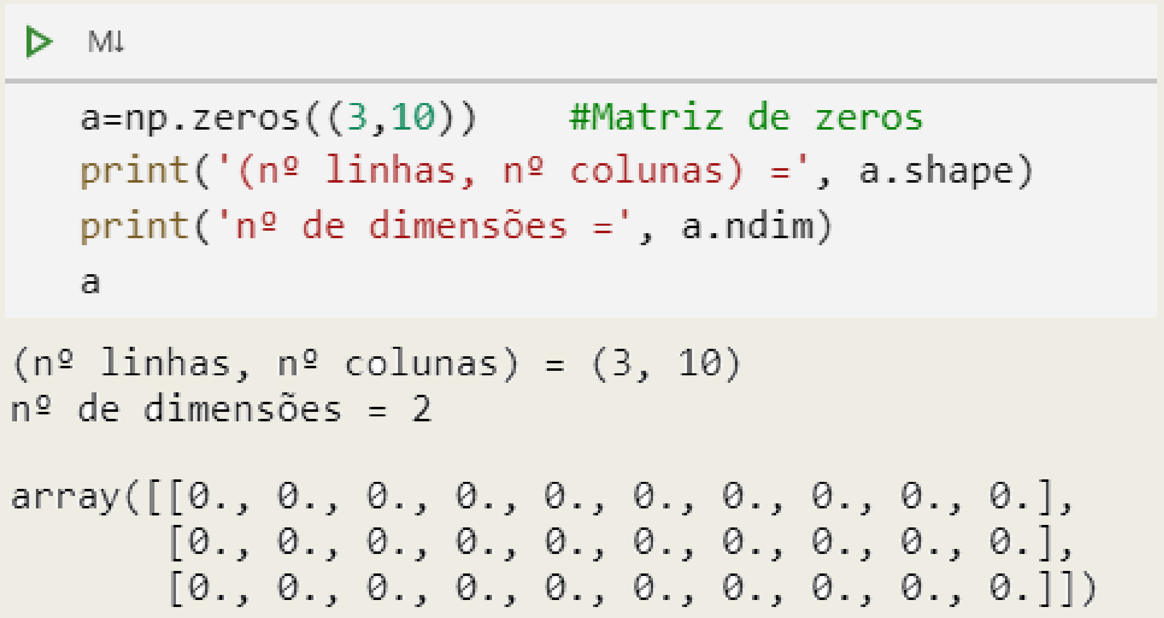
```
linha0=['00', '01', '02', '03']; linha1=['10', '11', '12', '13']
linhas=[linha0, linha1]
y=np.array(linhas)
print(y)
```

```
[['00' '01' '02' '03']
 ['10' '11' '12' '13']]
```

- In addition to the numpay.array() function and the numpay.arange() function, which allows us to only create NumPy arrays of a single dimension, there are other functions to create arrays already filled with values:

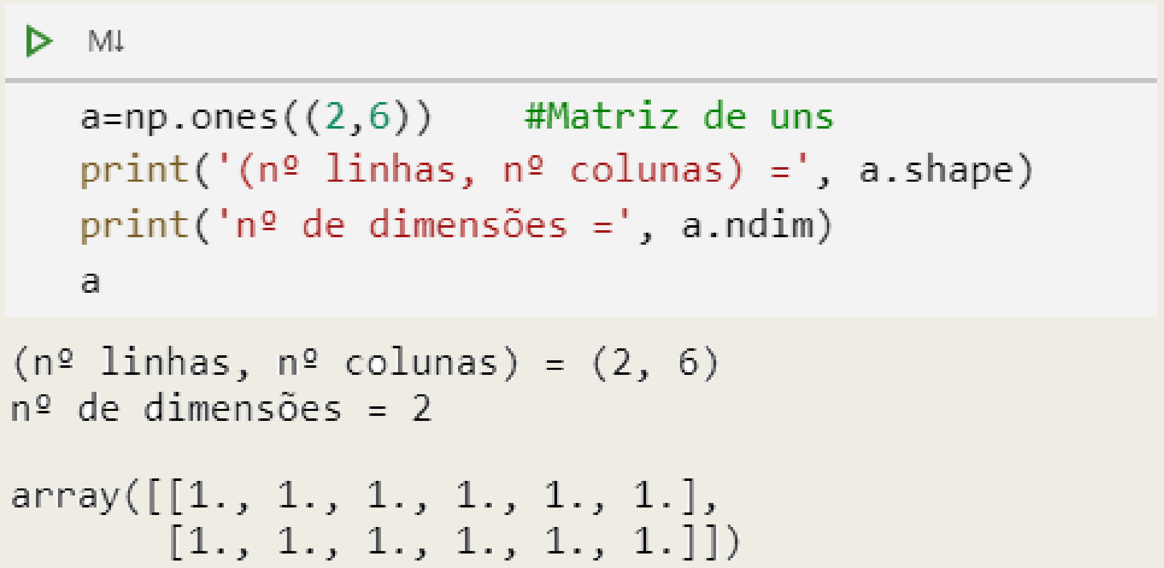
Function	Array created
numpy.zeros()	array of 0's
numpy.ones()	Array of 1’s
numpy.full()	array filled with a specific value
numpy.random.random()	array of random values evenly distributed between 0 and 1
numpy.eye()	identity matrix
numpy.diag()	diagonal matrix

numpy.zeros() / numpy.ones()



```
a=np.zeros((3,10))    #Matriz de zeros
print('(nº linhas, nº colunas) =', a.shape)
print('nº de dimensões =', a.ndim)
a
```

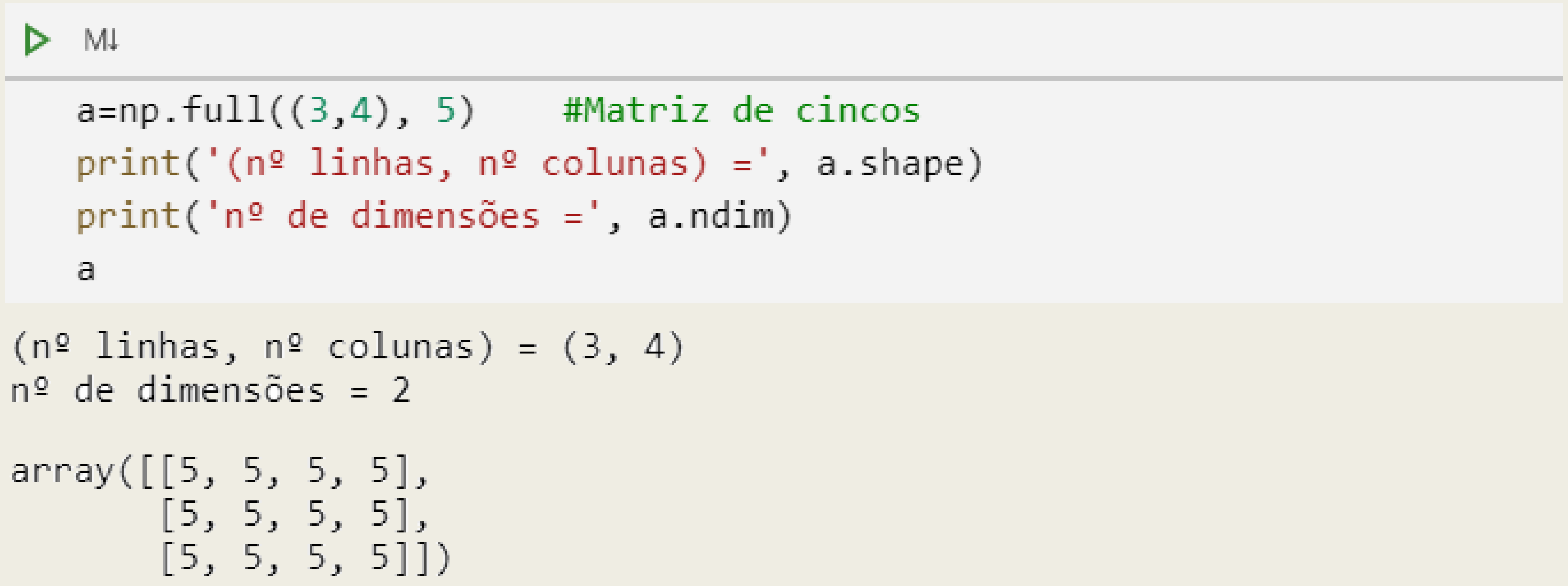
```
(nº linhas, nº colunas) = (3, 10)
nº de dimensões = 2

array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```



```
a=np.ones((2,6))    #Matriz de uns
print('(nº linhas, nº colunas) =', a.shape)
print('nº de dimensões =', a.ndim)
a
```

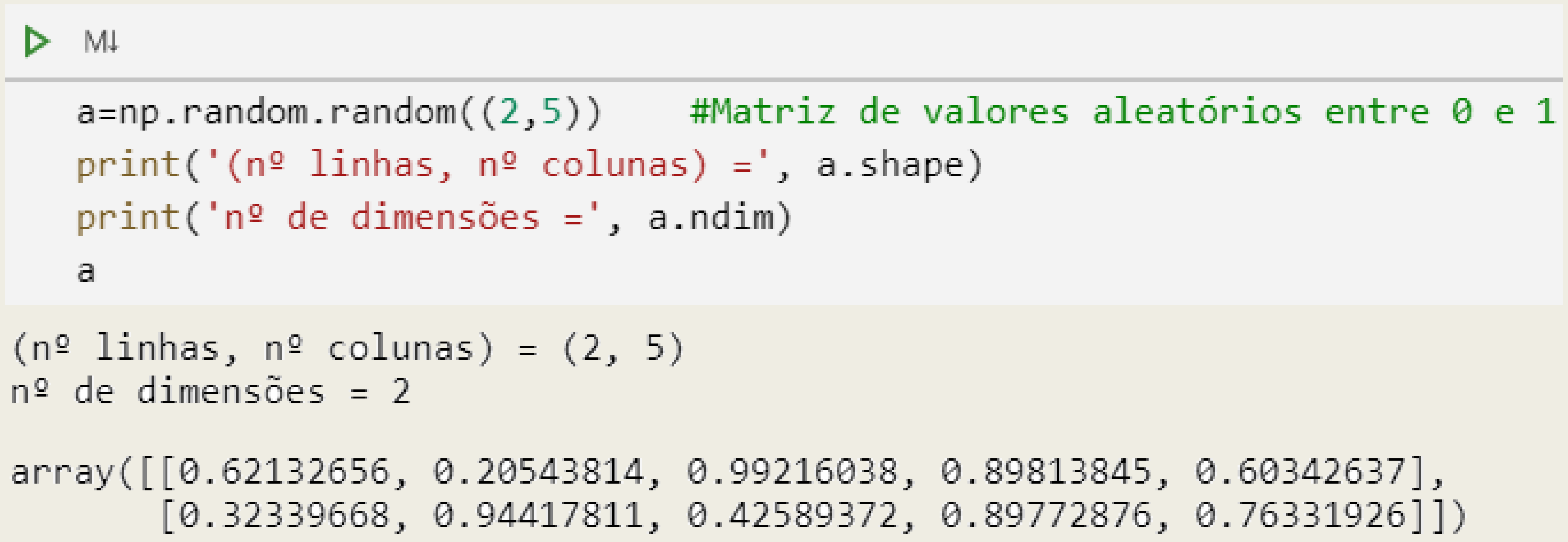
```
(nº linhas, nº colunas) = (2, 6)
nº de dimensões = 2

array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

numpy.full() / numpy.random.random()



```python
a=np.full((3,4), 5)    #Matriz de cincos
print('(nº linhas, nº colunas) =', a.shape)
print('nº de dimensões =', a.ndim)
a
```

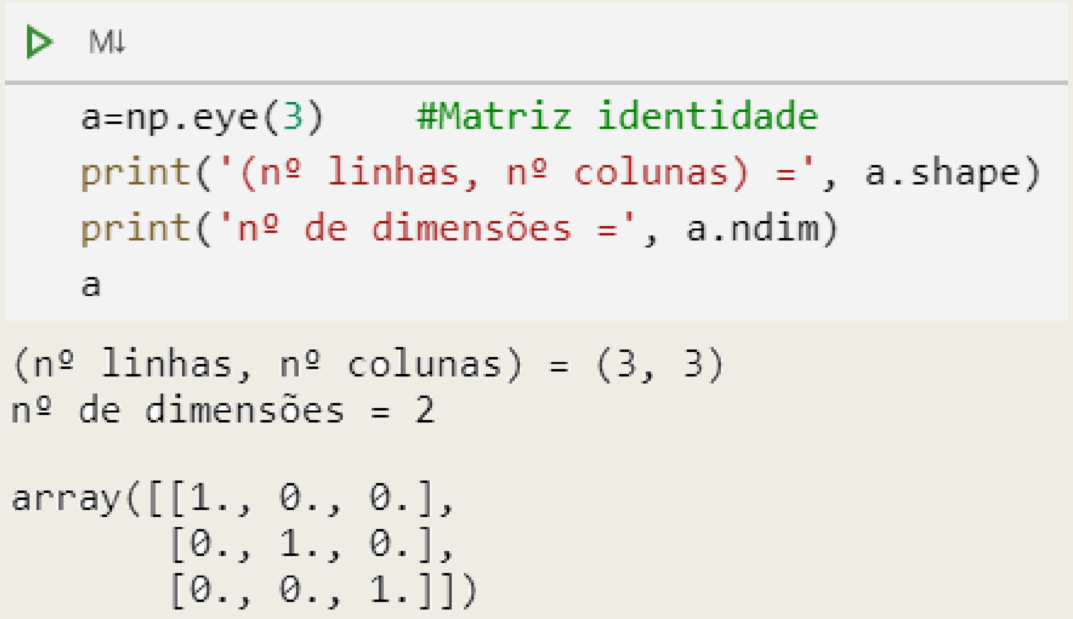
```
(nº linhas, nº colunas) = (3, 4)
nº de dimensões = 2

array([[5, 5, 5, 5],
       [5, 5, 5, 5],
       [5, 5, 5, 5]])
```



```python
a=np.random.random((2,5))    #Matriz de valores aleatórios entre 0 e 1
print('(nº linhas, nº colunas) =', a.shape)
print('nº de dimensões =', a.ndim)
a
```

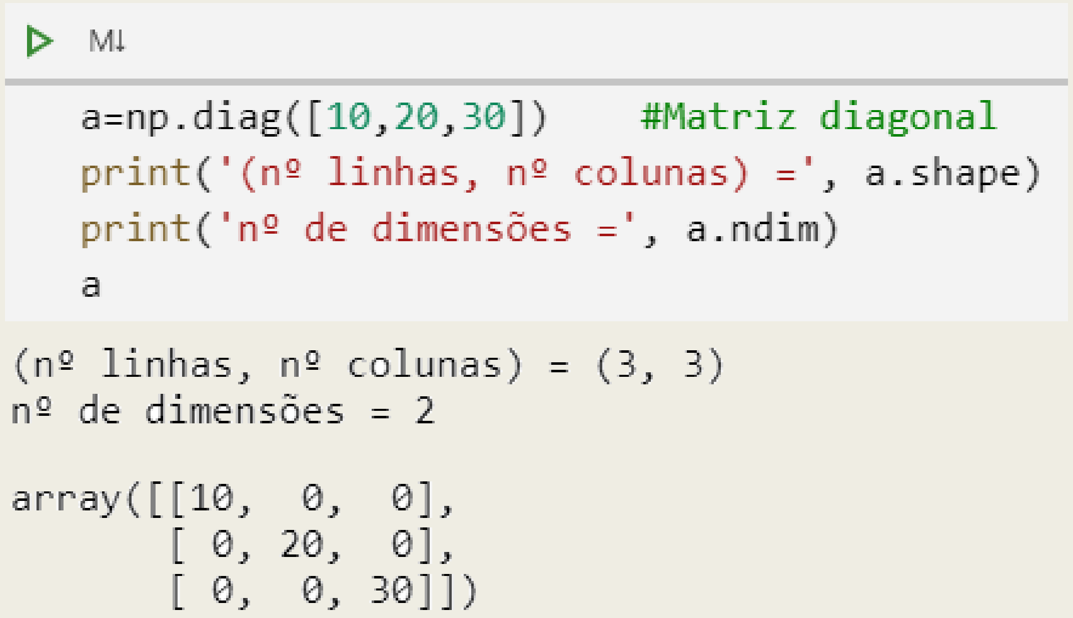
```
(nº linhas, nº colunas) = (2, 5)
nº de dimensões = 2

array([[0.62132656, 0.20543814, 0.99216038, 0.89813845, 0.60342637],
       [0.32339668, 0.94417811, 0.42589372, 0.89772876, 0.76331926]])
```

numpy.eye() / numpy.diag()



```python
a=np.eye(3)    #Matriz identidade
print('(nº linhas, nº colunas) =', a.shape)
print('nº de dimensões =', a.ndim)
a
```

```
(nº linhas, nº colunas) = (3, 3)
nº de dimensões = 2

array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```



```python
a=np.diag([10,20,30])    #Matriz diagonal
print('(nº linhas, nº colunas) =', a.shape)
print('nº de dimensões =', a.ndim)
a
```

```
(nº linhas, nº colunas) = (3, 3)
nº de dimensões = 2

array([[10,  0,  0],
       [ 0, 20,  0],
       [ 0,  0, 30]])
```

NumPy array indexing

- Access to elements of an NumPy array is done using indexing, just as we do in other languages

```
a0=np.arange(0,100,10)
a0
```

```
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

```
a1=a0.reshape((2,5))
a1
```

```
array([[ 0, 10, 20, 30, 40],
       [50, 60, 70, 80, 90]])
```

```
a0[5]
```

```
50
```

```
a1[1,0]
```

```
50
```

- Boolean indexing
  - *Additionally, NumPy allows indexes to be used in True or False logical values at all array positions to signal elements that, respectively, will be selected*

```
a0[[False,True,False,False,False,True,False,False,False,False]]
```

```
array([10, 50])
```

  - *And because using an array in a condition converts positions into logical values*

```
a0>40
```

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

the selection of the elements according to this condition is simplified

```
a0[a0>40]
```

```
array([50, 60, 70, 80, 90])
```

  - *This is a very useful form of indexing. See the following example*

```
a2=np.arange(15)
```

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

```
a2[a2 % 2 != 0] #escolher apenas os impares
```

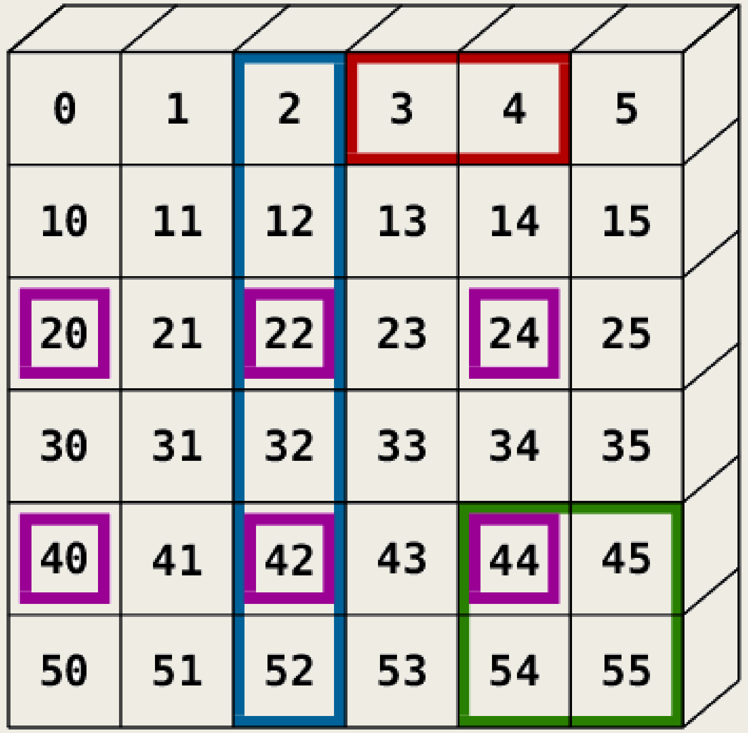
```
array([ 1,  3,  5,  7,  9, 11, 13])
```

Slicing arrays (slicing) [2]

- Slicing an NumPy array is similar to slicing Python lists, but applied to the various dimensions

```
a=np.arange(60)  # [0,  1,  2,  3,  4,  5,  6, .., 58, 59]

a=a[a%10<=5] # [0,  1,  2,  3,  4,  5, 10, 11, .., 54, 55]

a=a.reshape((6,6))
```

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 45],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

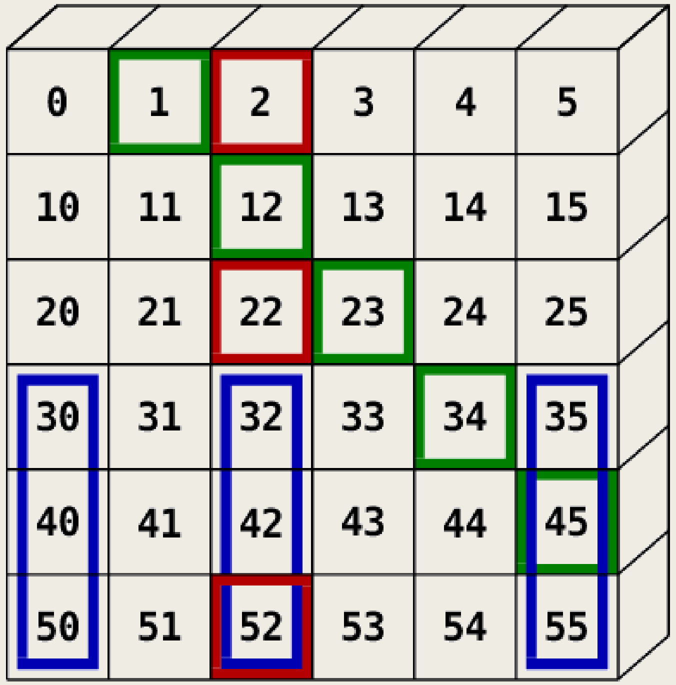


[2] With content adapted from "Scipy Lecture Notes", DOI 10.5281/zenodo.594102, 2020 (with licensing CC BY 4.0)

Different ways of indexing[2]

Three distinct ways to index a NumPay array

```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```



Mathematical operations

- It is possible to perform, in a simple way, mathematical operations on NumPy arrays
  - *including basic aritmtic operations, element by element*

```
x=np.array([[0,1,2],[3,4,5]])
x
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
y=x+6   #broadcasting
y       #(operação de difusão)
```

```
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

```
soma=x+y
```

```
array([[ 6,  8, 10],
       [12, 14, 16]])
```

```
prod=x*y
```

```
array([[ 0,  7, 16],
       [27, 40, 55]])
```

```
dif=x-y
```

```
array([[-6, -6, -6],
       [-6, -6, -6]])
```

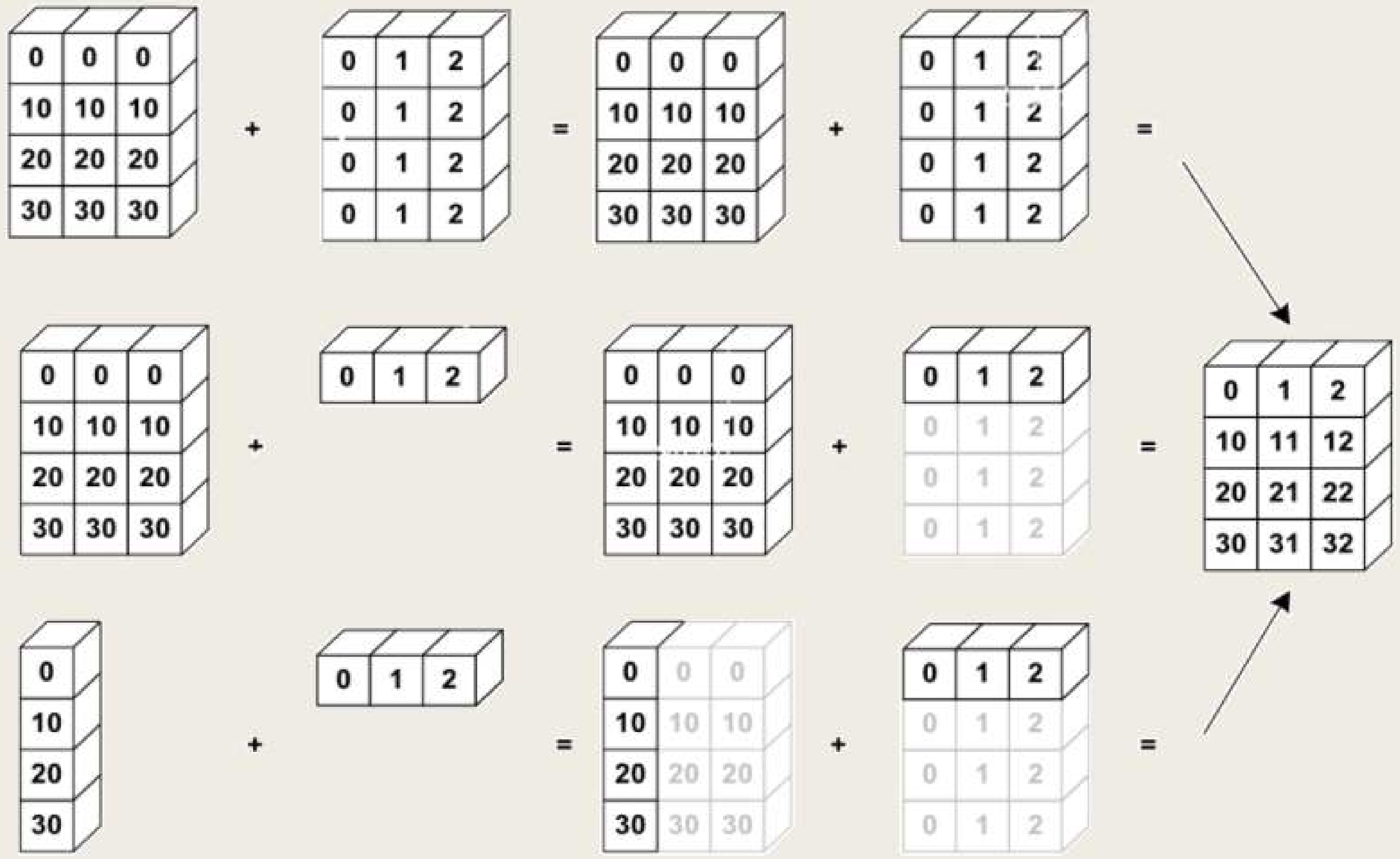
```
div=x/y
```

```
array([[0.   , 0.143, 0.25 ],
       [0.333, 0.4  , 0.455]])
```

  - *The same operations can be carried out through:*
    - np.add(x,y), np.subtract(x,y), np.multiply(x,y), np.divide(x,y)
- To perform the internal product between two vectors, represented by 2 one-dimensional NumPy arrays, the function np.dot() is used, resulting in a scalar value
  - *Example:* `v1=np.array([1,2,3]); v2=np.array([3,2,1]); np.dot(v1,v2)` `10`
  - *When used in two-dimensional arrays, it is equivalent to multiplying arrays.*

Broadcasting

- Faced with two arrays of different dimensions involved in an arithmetic operation, NumPy attempts to convert these arrays to the same dimension, replicating the pattern of existing values in the new dimensions created
    - *This conversion is called '**broadcasting**'*



NumPay Matrices

- NumPay provides us with a special type of array: the matrix type
  - *This is a subclass of ndarray*
  - *It is therefore an array but necessarily two-dimensional*
  - *The creation of an array is similar to the creation of arrays*

```
m1=np.matrix([[0,1,2],[3,4,5]])
m1
```

```
matrix([[0, 1, 2],
        [3, 4, 5]])
```

  - *You can also convert an array to an array, with the function asmatrix()*

```
a2=np.array([[5,4,3],[2,1,0]])
m2=np.asmatrix(a2)
m2
```

```
matrix([[5, 4, 3],
        [2, 1, 0]])
```

and an array in an array, with the inverse function asarray().

- Another important difference between arrays and matrix is when multiplying these types of objects (with the *operator)
  - *The multiplication of arrays is carried out element by element, as seen before*
  - *The multiplication of matrices is equivalent to the np.dot() function, that is, it implements the behavior of the multiplication of linear algebra matrices*

```
print(m1)        print(m2.T)       m1*m2.T                np.asarray(m1)*np.asarray(m2)

[[0 1 2]         [[5 2]            matrix([[10,  1],      array([[0, 4, 6],
 [3 4 5]]         [4 1]                    [46, 10]])            [6, 4, 0]])
                  [3 0]]
```

Aggregation functions

- Multiple aggregation functions can be applied to NumPay arrays (and arrays)
  - *to all values ( ), column by column (axis=0), or line by row (axis=1)*

```
print(a2)          [[5 4 3]
                    [2 1 0]]
```

```
print(a2.sum(), a2.sum(axis=0), a2.sum(axis=1), sep=' | ')
```

```
15 | [7 5 3] | [12  3]
```

```
print(a2.mean(), a2.mean(axis=0), a2.mean(axis=1), sep=' | ')
```

```
2.5 | [3.5 2.5 1.5] | [4. 1.]
```

```
print(a2.max(), a2.max(axis=0), a2.max(axis=1), sep=' | ')
```

```
5 | [5 4 3] | [5 2]
```

```
print(a2.argmax(), a2.argmax(axis=0), a2.argmax(axis=1), sep=' | ')
```

```
0 | [0 0 0] | [0 0]
```

```
print(a2.min(), a2.min(axis=0), a2.min(axis=1), sep=' | ')
```

```
0 | [2 1 0] | [3 0]
```

```
print(a2.argmin(), a2.argmin(axis=0), a2.argmin(axis=1), sep=' | ')
```

```
5 | [1 1 1] | [2 2]
```

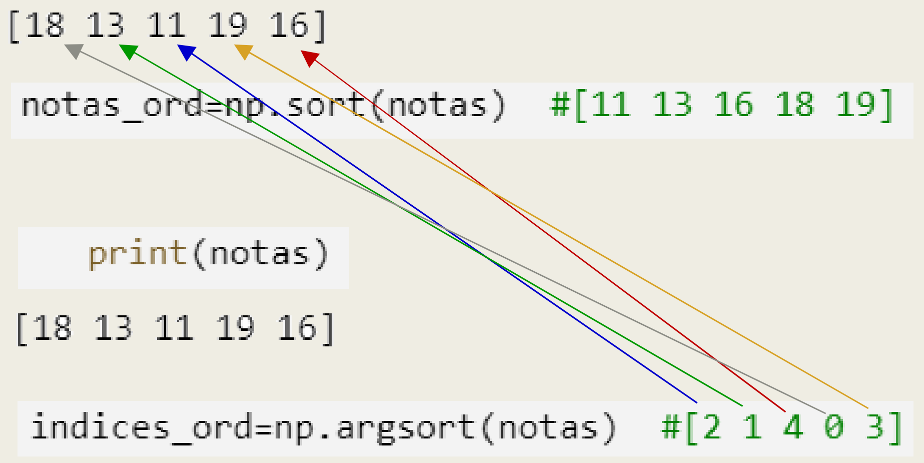
- Other NumPy aggregation functions:
  - *average(), product(), median(), std(), var(), cumprod(), cumsum(), corrcoef().*

Ordering NumPy Arrays

NumPy provides us with functions for efficient ordering of arrays

Let's start by creating an array with the names of 5 students and another with their grades (randomly generated but positive...)

```
nomes=np.array(['Ana', 'Rui', 'Ze', 'To', 'Rita'])
notas=np.random.random_integers(10,20,5)
print(notas)
```



```
[18 13 11 19 16]
```

- We can easily sort notes with **numpy.sort()**

```
notas_ord=np.sort(notas)  #[11 13 16 18 19]
```

- The sort() function does not modify the original array, as can be seen

```
print(notas)
```

```
[18 13 11 19 16]
```

- Another very interesting function, is **NumPy.argsort()**, since it returns the indexes (positions) that allow us to sort the array

```
indices_ord=np.argsort(notas)  #[2 1 4 0 3]
```

- If we reverse the order of these indexes,

```
indices_ord_inv=indices_ord[::-1] #[3, 0, 4, 1, 2]
```

we can easily show students by the value of their grade, starting with the highest.

```
print('Alunos ordenados pela nota: ', nomes[indices_ord_inv])
print('Respetivas notas: ', notas[indices_ord_inv])
```

```
Alunos ordenados pela nota:  ['To' 'Ana' 'Rita' 'Rui' 'Ze']
Respetivas notas:  [19 18 16 13 11]
```

Resizing

- We already know a method that returns an Array NumPy resizing

```
x=np.array([10,11,12,13,14,15]); y1=x.reshape(2,3); print(y1)
```

```
[[10 11 12]
 [13 14 15]]
```

  - *and with parameter -1 we can let it be the method to define one of the dimensions, in order to beat right with the amount of elements*

```
y2=x.reshape(2,-1);  print(y2)
```

```
[[10 11 12]
 [13 14 15]]
```

```
y3=x.reshape(-1,3);  print(y3)
```

```
[[10 11 12]
 [13 14 15]]
```

- But if the goal is to change the array itself, then we can resize it by assigning a tuple to its 'shape' property

```
x.shape=2,-1; print(x)
```

```
[[10 11 12]
 [13 14 15]]
```

  - *and put it back in its one-dimensional form by changing the same property*

```
x.shape=-1; print("x={} ndim={} shape={}".format(x, x.ndim, x.shape))
```

```
x=[10 11 12 13 14 15] ndim=1 shape=(6,)
```

Copying NumPy Arrays

- Copy by reference (reference copy)
  - *As with other Python objects, when we assign one NumPy Array to another, only the reference is copied*
    - thus, any change in one, affects the other

```
x=np.array([1,2,3,4]); y=x; y[1]=0; y.shape=(2,2); print(x,y)
```

```
[[1 0]   [[1 0]
 [3 4]]   [3 4]]
```

- Surface copying (*shallow copy*)
  - *NumPy arrays have, however, the view() method, which creates a new array, but shares the same data with the original*
    - *passes is to have its own form independent of that of the original array*

```
x=np.array([1,2,3,4]); y=x.view(); y.shape=(2,2); y[1,1]=0; print(x,y)
```

```
[1 2 3 0] [[1 2]
           [3 0]]
```

  - *It is this type of copy that also happens with slicing and resizing*

```
x=np.array([1,2,3,4]); y=x[2:]; y[1]=0; print(x,y)
```

```
[1 2 3 0] [3 0]
```

```
x=np.array([1,2,3,4]); y=x.reshape(2,2); y[1,1]=0; print(x,y)
```

```
[1 2 3 0] [[1 2]
           [3 0]]
```

- In-depth copying (*deep copy*)
  - *The copy() method is one that allows a duplication of both the array and the elements contained in it, completely delinking the created array from what gave rise to it*

```
x=np.array([1,2,3,4]); y=x.copy(); y.shape=(2,2); y[1,1]=0; print(x,y)
```

```
[1 2 3 4] [[1 2]
           [3 0]]
```

NumPy, a support package

- In conclusion, it should be seen that NumPy in addition to being Python's main numerical computing package, supports many other Data Science packages, as is precisely the case for those we're also going to work with: Pandas, Matplotlib, and Scikit-learn.

solve exercise #9, #10 and #11

from the book of exercises