



Data manipulation: Numpy

AAA-Python Edition



Plan

- 1- Numpy: ndarray
- 2- indexing
- 3- Operations with ndarray
- 4- File saving and loading
- 5- Structures with dtype



1- Numpy: ndarray

Numpy

- **Numpy** for **Numerical Python**, a library for **numerical computing** in Python.
- It defines:
 - **ndarray** : **multidimensional** array
 - Fast **Mathematical functions** and **operations** with **ndarray** including **reading** and **writing** array data from/to **disk**
 - **Linear algebra**, **random number** generation, and **Fourier transform** capabilities
 - A **C API** for connecting **NumPy** with libraries written in **C**, **C++**, or **FORTRAN**.
- We will focus in this course on the **3** first points.



1-Numpy: ndarray

ndarray

- **ndarray** is a **multidimensional array** object = a generic multidimensional **container** for **data** of the **same type**.

```
[ ] a= [1,2,3]
    # a is a list
    a
```

a is a list

```
↳ [1, 2, 3]
```

```
[ ] # import numpy to use ndarray
    import numpy as np
```

Import Numpy to use “array” function

```
#creating an ndarray by transforming a list using array function
```

```
b = np.array(a)
```

array function used to transform
a list to an ndarray

```
# b is an ndarray
b
```

b is an ndarray

```
↳ array([1, 2, 3])
```



1-Numpy: ndarray

ndarray

- ndarray** is characterized by its **shape** and **dimension**

```
[19] #create an ndarray from a list of two same sized list
```

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

2 elements of dimension 2
(a 2 dimensional element has 2 external brackets)

```
#create a 3 dimension ndarray
```

```
d= np.array([[[5,0,1],[9,7,-1],[2,3,5]],[[11,21,33],[22,5,16],[7,8,9]]])
```

```
#the ndim (dimension) and shape attributes
```

```
print("d.dimension=",d.ndim)  
print("d.shape =",d.shape)
```

3 elements of
dimension 1

3 elements
of dimension 0 =scalars



```
[[1 2 3]  
 [4 5 6]]  
d.dimension= 3  
d.shape = (2, 3, 3)
```

Number of external
Brackets = dimension
(=3)

The ndarray d is a 3 dimensional array, composed of: 2 elements of dimension 2.
Each dimension 2 element is composed of: 3 elements of dimension 1
Each dimension 1 element is composed of: 3 elements of dimension 0
==> the shape of d = 2 x 3 x 3



1-Numpy: ndarray

Creating ndarray

- like **array** function, other functions exist to **create** an ndarray

```
[30] 1 # asarray function : create an ndarray from the input.  
      2 # if the input is an ndarray, it will not be copied:  
      3 # the output and the input will refer to the same element.  
      4 g = np.asarray(b)  
      5 g[0]=155  
      6 print ("g=",g)  
      7 print("b=",b)  
      8  
      9 ...  
     10 array function : create an ndarray from the input.  
     11 even if the input is an ndarray, it will by default be copied:  
     12 the output and the input will refer to different elements.  
     13 To behave like asarray, it must be called with the optional argument 'copy'  
     14 set to false: copy(b,copy=False)  
     15 ...  
     16  
     17 h = np.array(b)  
     18 h[1]= 156  
     19 print("h=",h)  
     20 print("b=",b)  
     21
```

Since **b** is an ndarray, it will not be copied.
g and **b** will refer to the same element

Using array function, the array **b** will be copied in a new element **h**

Modifying the second element of **h** will **not modify** the second element of **b**

Modifying the first element of **g**, will modify also the first element of **b**

```
g= [155  2  3]  
b= [155  2  3]  
h= [155 156  3]  
b= [155  2  3]
```



1-Numpy: ndarray

Creating ndarray

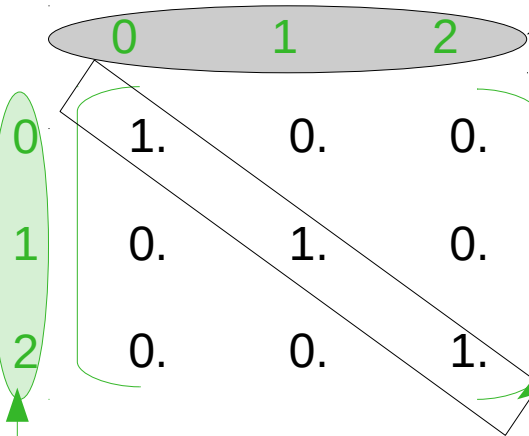
```
1 # function eye (or identity), returns the identity matrix
2 id1 = np.eye(3)
3 print (id1)
4 # arrange return a range in an ndarray format
5 ar = np.arange(1,9,3)
6 print (ar)
```

```
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]
 [1  4  7]]
```

Identity matrix (2 dimensions:
rows and columns)

Range from 1 to (9-1)
with step 3

id1



Axis 1 == columns

1 in diagonal,
0 elsewhere

Axis 0 == lines



Creating ndarray

- Each of the following functions has two versions: **function-name** and **function-name_like**

```
[48] 1 # 2 x 3 ndarray of ones
      2 on = np.ones((2,2))
      3 print ("on=",on)
      4 # ndarray with the same shape and type as "c"
      5 on_l = np.ones_like(c)
      6 print ("on_l=",on_l)
```

```
on= [[1. 1.]
      [1. 1.]]
on_l= [[1 1 1]
       [1 1 1]]
```

We can specify in these functions the **dtype** argument (the values type)

```
1 # ndarray full with zeros
2 f = np.zeros(2)
3 print("f=",f)
4 # ndarray "empty"= no default values (random)
5 k = np.empty(6)
6 print("k=",k)
7 # ndarray full with the given value
8 y = np.full((2,4),0.5)
9 print("y=",y)
```

```
f= [0. 0.]
k= [5.e-324 5.e-324 5.e-324 5.e-324 5.e-324 5.e-324]
y= [[0.5 0.5 0.5 0.5]
     [0.5 0.5 0.5 0.5]]
```




1-Numpy: ndarray

dtype

- The ndarray can be created specifying a type "**dtype**"
- The types can be:
 - int : signed (i1, i2, i4 or i8) and unsigned (u1, u2, u4 or u8)
 - float: f2, f4 or f, f8 or d, f16 or g
 - complex: c8, c16, c32
 - boolean: ?
 - object: O
 - String: S . Fixed length ASCII String type, (S"number" for a string of "number" byte size)
 - Unicode: U . Fixed length Unicode type, (U"number" for unicode of "number" of certain_byte size)

These codes can be used as arguments: **dtype="i8"**

The **float** fill value(**3.2**) is converted to **int**

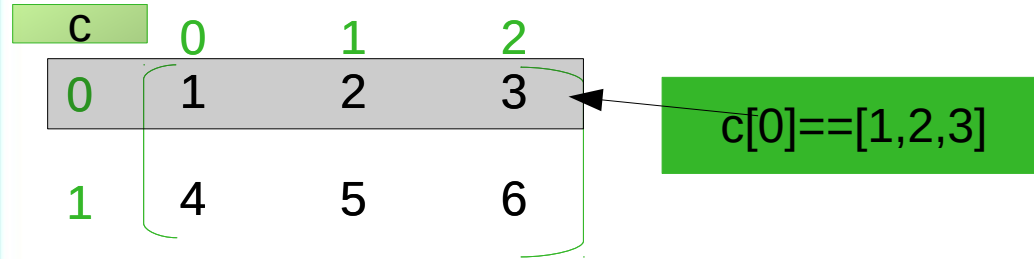
```
1 a = np.full (5,3.2,dtype="i8")  
2 print(a)  
[3 3 3 3 3]
```



2- Indexing

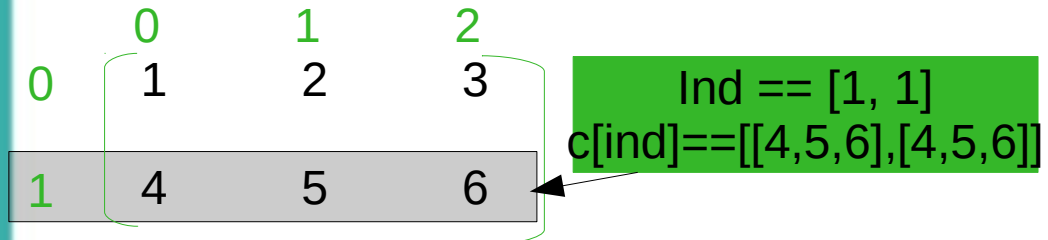
indexes

- ndarray** can be **indexed** by: **integers, arrays, slices**, and Boolean



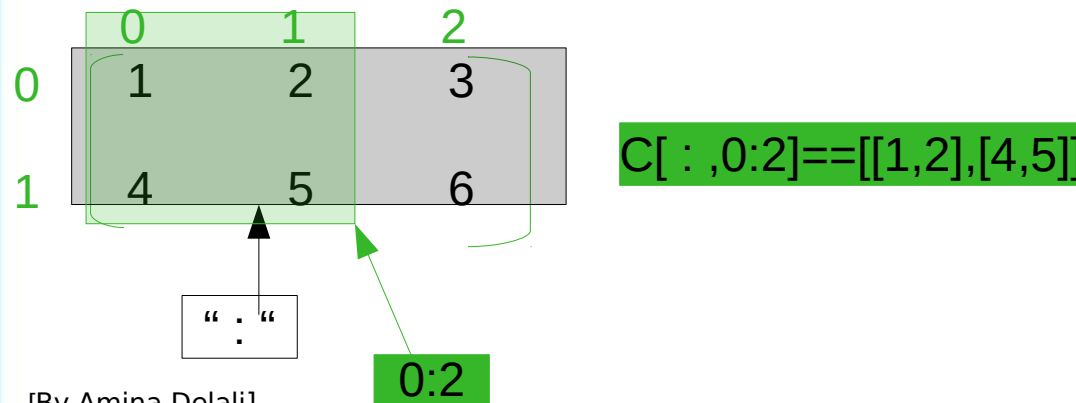
```
[97] 1 | c[0]
```

```
↳ array([1, 2, 3])
```



```
[101] 1 | ind = np.ones(2,dtype="i4")
      2 | c[ind]
```

```
↳ array([[4, 5, 6],
        [4, 5, 6]])
```



```
[102] 1 | c[:, 0:2]
```

```
↳ array([[1, 2],
        [4, 5]])
```



2- Indexing

indexes

- ndarray** can be **indexed** by: integers, arrays, slices, and **Boolean**

c	0	1	2
0	1	2	3
1	4	5	6

ind2	0	1	2
0	False	False	False
1	False	False	False

ind2	0	1	2
0	True	True	True
1	False	False	False

Assigning one value(**True**)
to the entire **line (0)**

[By Amina Delali]

Selecting values from **c**
corresponding to **True** in
ind2

```
1 ind2 = np.full_like(c, False, dtype="?")
2 print("ind2=", ind2)
3 ind2[0]=True
4 print("ind2=", ind2)
5 ind2[1,0]=True
6 print("ind2=", ind2)
7 c[ind2]
```

Assigning one value
(**True**) to one cell (1,0)

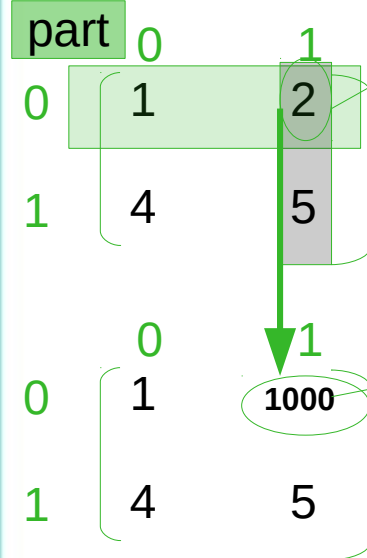
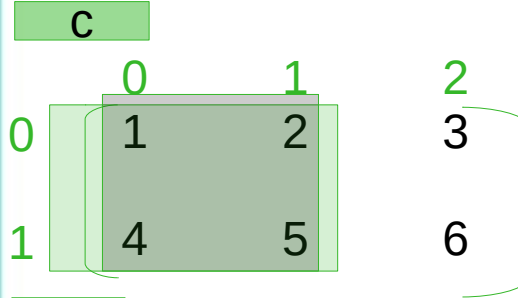
```
ind2= [[False False False]
       [False False False]]
ind2= [[ True  True  True]
       [False False False]]
ind2= [[ True  True  True]
       [ True False False]]
array([1, 2, 3, 4])
```

	0	1	2
0	True	True	True
1	True	False	False



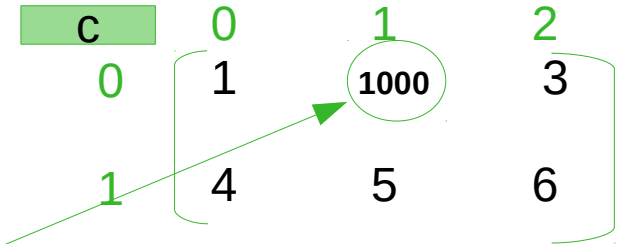
Slices and copies

- Using **slices** to create arrays from **other** ndarrays **doesn't** create **copies**. To have **distinct** arrays, we have to use the **method copy**



```
[39] 1 # creating part ndarray from a slice of c ndarray
      2 part = c[:2,0:2]
      3 print("part=\n",part)
      4 part[0,1]=1000
      5 # modification of cell(0,1) of part will
      6 # alter c values too.
      7 print("now part = \n",part)
      8 print("and c now =\n",c)
```

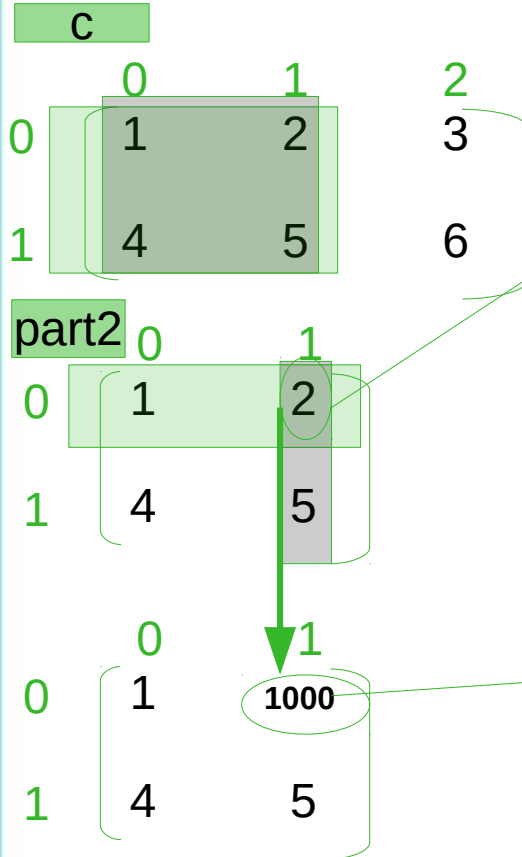
```
↳ part=
[[1 2]
 [4 5]]
now part =
[[ 1 1000]
 [ 4   5]]
and c now =
[[ 1 1000  3]
 [ 4   5  6]]
```



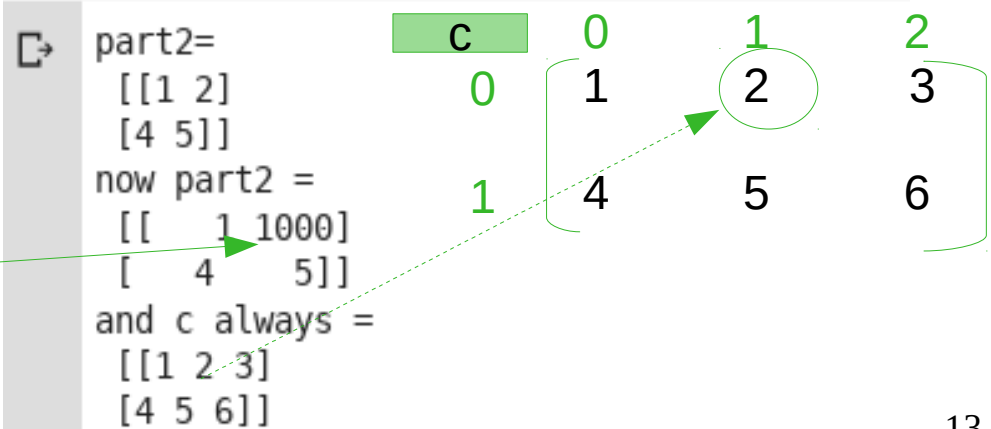


Slices and copies

- Using **slices** to create arrays from **other** ndarrays **doesn't** create **copies**. To have **distinct** arrays, we have to use the **method copy**



```
1 c= np.array([[1,2,3],[4,5,6]])
2 # part2 is a slice copy of c
3 part2 = c[:2,0:2].copy()
4 print("part2=\n",part2)
5 part2[0,1]=1000
6 # modification of cell(0,1) of part2
7 # will not alter c values.
8 print("now part2 = \n",part2)
9 print("and c always =\n",c)
10
```





3- Operations with Ndarrays

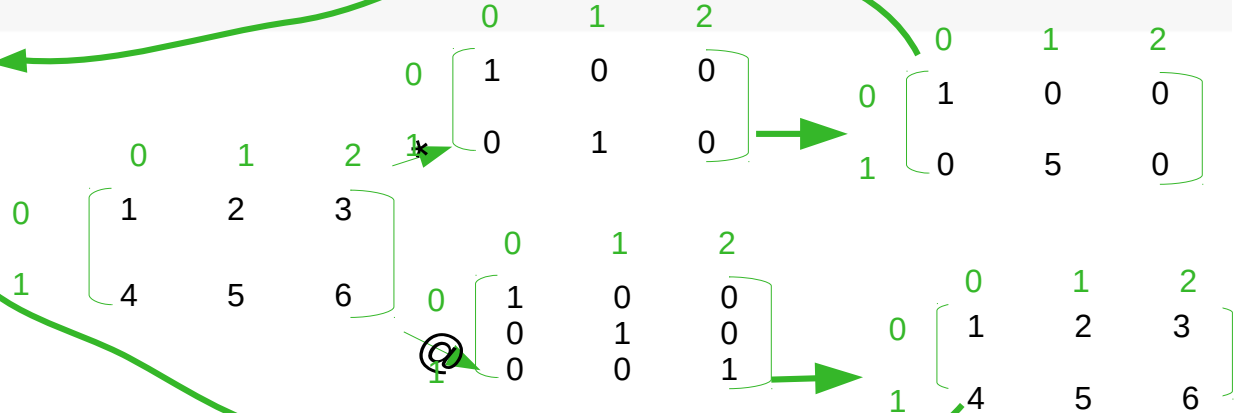
Arithmetic operations & Linear Algebra

```
[52] 1 # element wise multiplication: cell by corresponding cell,  
2 # matrices with the same shape  
3 res = c * np.eye(3)[0:2,0:3] ←  
4  
5 # the result is a matrix with the same shape  
6 print ("res=",res)  
7  
8 # matrix multiplication: line by columns  
9 # different shapes but:  
10 # number of columns of the first matrix == number of lines of the second matrix  
11 res = c @ np.eye(3) # same as np.dot(c,np.eye(3)) or c.dot(np.eye(3))  
12  
13 # the result is a matrix wiht:  
14 # number of lines == number of lines of the first matrix  
15 # number of columns == number of columns of the first matrix  
16 print("res=",res)  
17
```

```
res= [[1. 0. 0.]  
      [0. 5. 0.]]  
res= [[1. 2. 3.]  
      [4. 5. 6.]]
```

c @ identity == c

[By Amina Delali]

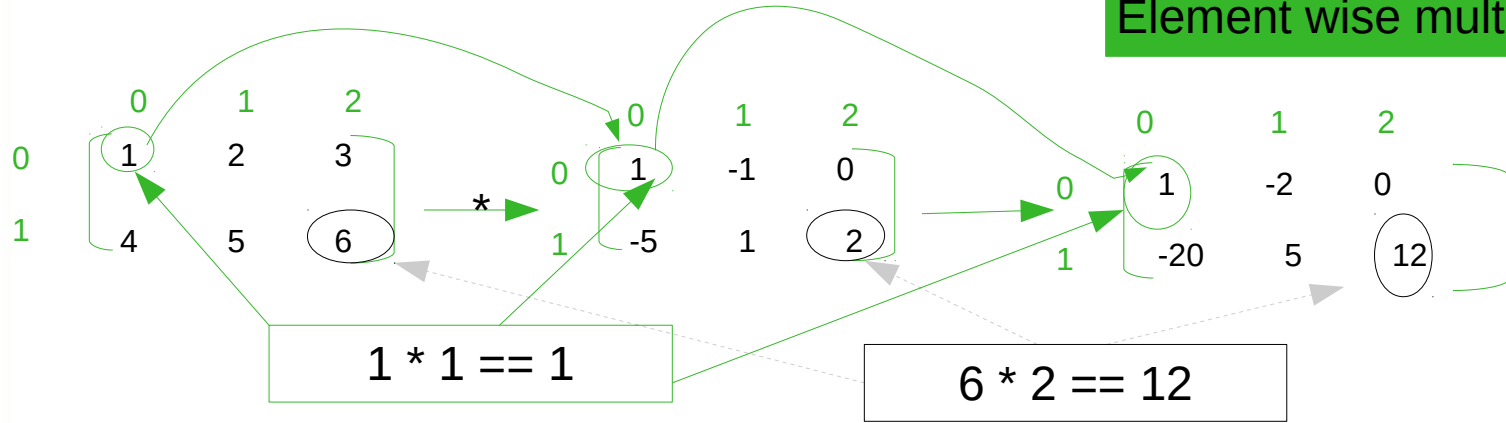




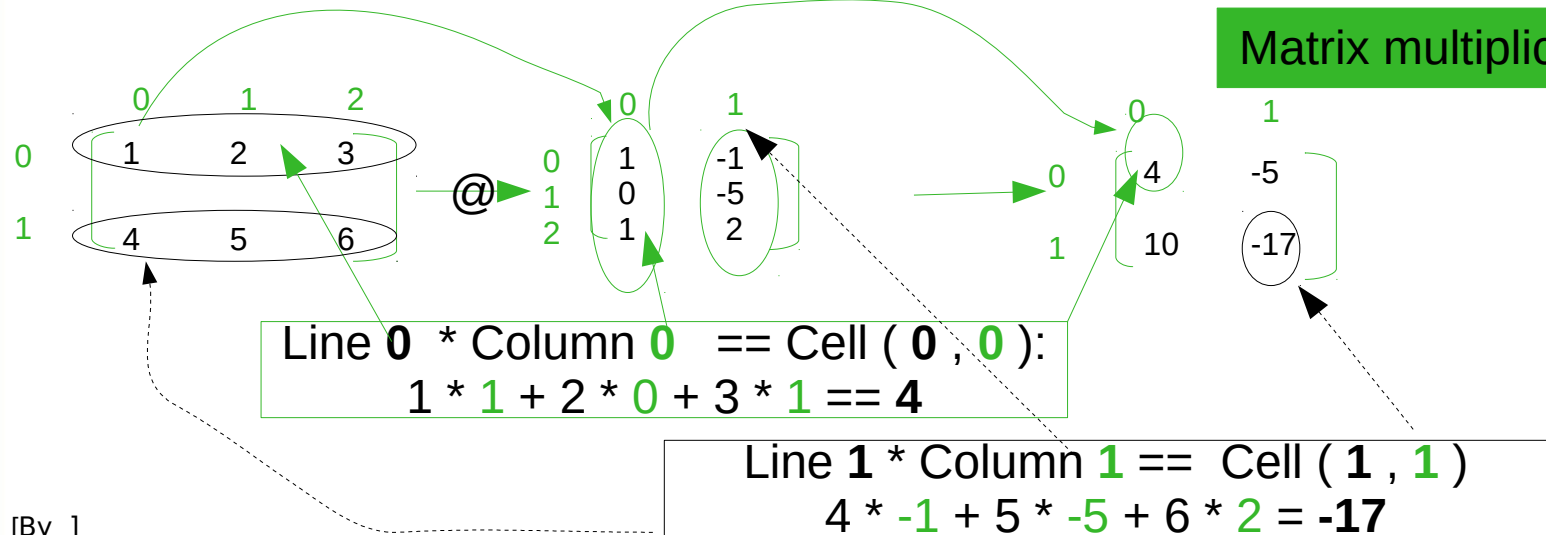
3- Operations with Ndarrays

Element wise multiplication vs Matrix multiplication

Element wise multiplication



Matrix multiplication





3- Operations with Ndarrays

```
1 # element wise addition same as matrix addition
2 res = c + d.reshape(c.shape)
3 print("res=",res)
4 # a different kind of addition
5 # 5 will be added to all values of c
6 res2 = c + 5
7 print("res2=",res2)
8 # inverse of values of an ndarray
9 res3 = 1 / c
10 print("res3=",res3)
```

Same operation can be done
With: -, /, *

```
res= [[ 2  1  3]
      [-1  6  8]]
res2= [[ 6  7  8]
       [ 9 10 11]]
res3= [[1.         0.5         0.33333333]
       [0.25       0.2        0.16666667]]
```




3- Operations with Ndarrays

Arithmetic and Logical operations

```
[56] 1 #comparison
      2 res = c > 3
      3 print("res=",res)
      4 # logical and: any value different from 0 is a True
      5 res2 = np.logical_and(c, np.array([[1,0,1],[0,0,1]]))
      6 print("res2=",res2)
      7
```

The result is an ndarray (each value Greater than 3 will produce a True value)

```
➞ res= [[False False False]
        [ True  True  True]]
    res2= [[ True False  True]
          [False False  True]]
```

- We can use logical operations to select certain elements of an array

```
➞ 1 # selecting elements greater than 2
   2 c [c>2]
```

```
➞ array([3, 4, 5, 6])
```



3- Operations with Ndarrays

Linear Algebra

- We already seen the matrix multiplication using **dot** method or **np.dot** function or the operator **@**
- There are other functions related to linear Algebra as: **diag, trace, inv, solve**, ... etc.
- as for sacalrs, matrices have inverse regarding the matrix multiplication operation:

$$mat * mat^{-1} = I$$

I is the Identity matrix

- A system of linear equations can be represented by matrices:

$$mat * x = y$$

for example:

$$x_1 + x_2 = 4 \quad mat_{(2,2)} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, x_{(2,1)} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad y_{(2,1)} = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$$

$$x_1 - x_2 = 0$$

And the solution will be :

$$x_{(2,1)} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$



3- Operations with Ndarrays

Linear Algebra

A square matrix means:
number of rows ==
number of columns

```
1 # diag returns the diagonal of a square matrix
2 mat = np.random.randn(3,3)
3 print ("mat==",mat)
4 print("mat diagonal ==", np.diag(mat))
5 # trace retruns the sum of the diagonal elements|
6 print(np.trace(np.eye(4)))
7 # inv return the inverse of a square matrix : mat * inv(mat)== identity matrix
8 print("mat inverse==",np.linalg.inv(mat))
9 # solve return the solution of the equation Ax=B (the values of x)
10 print("solution of mat * x = I is: ",np.linalg.solve(mat,np.eye(3,3)))
```

```
mat== [[-0.22704671 -0.91749631  1.94312276]
 [ 0.72634263  0.53660225  0.07718055]
 [-1.27634468 -1.51152533 -1.19382702]]
mat diagonal == [-0.22704671  0.53660225 -1.19382702]
4.0
mat inverse== [[ 0.37728297  2.90363646  0.80180073]
 [-0.55346257 -1.98103216 -1.02891193]
 [ 0.29738779 -0.59611705 -0.39214027]]
solution of mat * x = I is: [[ 0.37728297  2.90363646  0.80180073]
 [-0.55346257 -1.98103216 -1.02891193]
 [ 0.29738779 -0.59611705 -0.39214027]]
```

The solution must be equal to
the inverse matrix of **mat**



3- Operations with Ndarrays

Some functions and methods

- Numpy defines a list of **element wise functions** applicable to:
 - One ndarray, as: **sqrt**, **exp**, **modf**, **log**, **sign**, **ceil** and **floor**, **cos**, **logical_not**, ... etc
 - Two ndarray as: **add**, **mod**, **maximum**... etc

```
1 # the fractional and integer parts of values of an ndarray
2 # it returns 2 ndarray
3 print("fractional part of c/2=", np.modf(c/2)[0])
4 print("integer part of c/2=", np.modf(c/2)[1])
5 # the sign function returns the signs of the ndarray elements: 1 , 0 or -1
6 print("signs of d=", np.sign(d))
7 # maximum between the elements of two ndarrays
8 print("maximum values are:", np.maximum(-c, d.reshape(c.shape)))
```

Access to the first ndarray

```
fractional part of c/2= [[0.5 0.  0.5]
 [0.  0.5 0.  ]]
integer part of c/2= [[0.  1.  1.]
 [2.  2.  3.]]
signs of d= [[ 1 -1]
 [ 0 -1]
 [ 1  1]]
maximum values are: [[ 1 -1  0]
 [-4  1  2]]
```

1 for positive elements, -1 for negative elements and 0 for 0 values



3- Operations with Ndarrays

Some functions and methods

- There is a list of functions that permit the generation of ndarrays with certain values. For example: **randn**, **meshgrid**, and **where**

```
1 val = np.arange (0, 5, 1)
2 # the two arrays can be used to generate functions values
3 x, y= np.meshgrid(val, val)
4 print("x=",x)
5 print ("y=",y)
6 # function randn(2,3) will generate a (2x3) ndarray with random values
7 val = np.random.randn(2,3)
8 print("generated random values=",val)
9 # with "where" function we can generate ndarray values using conditional
10 # the folwo
11 res= np.where (c>3, "G","L")
12 print("res=",res)
13
```

```
x= [[0 1 2 3 4]
     [0 1 2 3 4]
     [0 1 2 3 4]
     [0 1 2 3 4]
     [0 1 2 3 4]]
y= [[0 0 0 0 0]
     [1 1 1 1 1]
     [2 2 2 2 2]
     [3 3 3 3 3]
     [4 4 4 4 4]]
```

```
generated random values= [[ 0.28066364 -0.53650679  2.40150812]
 [ 1.91066572  0.85300811 -1.19599321]]
[['L' 'L' 'L']
 ['G' 'G' 'G']]
```

If a value from **c** is greater than **3** it will return "**G**" else it will return "**L**"

Each value from the generated range Can be associated with all values



3- Operations with Ndarrays

Some functions and methods

- With the function **append** we can create a **new** ndarray by appending **new** values

```
1 print("c==",c)
2 # creating new array by appending a new values as a column (axis=1)
3 cn=np.append(c,[[7],[8]],axis= 1)
```

```
c== [[1 2 3]
     [4 5 6]]
c still == [[1 2 3]
            [4 5 6]]
first new ndarray= [[1 2 3 7]
                    [4 5 6 8]]
```

The given values must have the same dimension as the first argument"

```
4 # creating new array by appending a new values as a row (axis=0)
5 cn2=np.append(c,[[7,8,9]],axis= 0)
6 print("c still ==",c)
7 print("first new ndarray=",cn)
8 print("second new ndarray=",cn2)
```

c didn't change

```
second new ndarray= [[1 2 3]
                     [4 5 6]
                     [7 8 9]]
```



3- Operations with Ndarrays

Some functions and methods

- ndarray objects define a list of useful **methods** like: **mean, sum, cumsum, max, sort, T, ...etc**

```
[98] 1 print("c==",c)
      2 print ("maximum element of c==",c.max())
      3 print ("the sum of elements of c ==",c.sum())
      4 print ("the cumulative sum of elements of c ==", c.cumsum())
      5 print("the mean of values of c ==",c.mean())
      6
```

```
↳ c= [[1 2 3]
      [4 5 6]]
maximum element of c== 6
the sum of elements of c == 21
the cumulative sum of elements of c == [ 1  3  6 10 15 21]
the mean of values of c == 3.5
```

Lines **0,1** become
columns **0,1**.
And columns **0,1,2**
become lines **0,1,2**

```
▶ 1 # the T method: retruns the transpose of a matrix
   2 # the lines become columns and vise versa
   3 print("c==",c)
   4 print ("c.T==",c.T)
```

```
↳ c= [[1 2 3]
      [4 5 6]]
c.T= [[1 4]
      [2 5]
      [3 6]]
```



4- Array saving and loading

Save and Load

- It is possible to **save** and **load** ndarrays into binray **format**

```
] 1 # save c to "file_c.npy"
2 np.save("file_c", c)
3 # loading c from "file_c.npy" into
4 c2= np.load("file_c.npy")
5 print("c2==", c2)
6 # saving multiple ndarrays: c and d into "files.npy"
7 np.savez("files", c=c, d=d)
8 #loading c and d from "files.npy"
9 res = np.load("files.npy")
10 print("c==", res["c"])
11 print("d==", res["d"])
```

If the extensions "**npz**" or "**npz**" are not specified they will be **added**.

```
➤ c2== [[1 2 3]
       [4 5 6]]
c== [[1 2 3]
     [4 5 6]]
d== [[ 1 -1]
     [ 0 -5]
     [ 1  2]]
```

Access to the arrays with the names used in the saving

The extension has to be specified in loading data



Some functions and methods

- **dtype constructor** can be used to create **structured** type.

Each myType element is defined by two values: "code" and "Value"

```
[122] 1 myType = np.dtype([("code", "U5"), ("Value", "i4")])
      2 myAr = np.array([("A", 10), ("B", 2), ("C", 15)], myType)
      3 print("myAr==", myAr)
      4 print("first element==", myAr[0])
      5 print("Codes in myAr==", myAr["code"])
      6 print("second element value==", myAr[1]["Value"])
```

name and type

```
↳ myAr== [('A', 10) ('B', 2) ('C', 15)]
first element== ('A', 10)
Codes in myAr== ['A' 'B' 'C']
second element value== 2
```

Initialized by tuples of two values corresponding to myType definition



References

- Wes McKinney. Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. O'Reilly Media, Inc, 2018.
- SciPy.org. Data type objects. On-line at <https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.dtypes.html>. Accessed on 05-10-2018.



Thank you!

FOR ALL YOUR TIME