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** : **multidimentioanl** array

  - Fast **Mathematical functions** and **operations** with **ndarray** including **reading** and **writing** array data from/to **disk**

  - **Linear algebra**, **random** **number** generation, and **Fourrier transform** capabilities

  - A **C** **API** for connecting **NumPy** with libraries written in **C**, C++, or **FORTRAN**.

- We will focus int this course on the **3** first points.

[By Amina Delali]

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
```

```
[1, 2, 3]
```

a is a list

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

     #creating an ndarray by transforming a list using array function
     b = np.array(a)

     # b is an ndarray
     b
```

```
array([1, 2, 3])
```

Import Numpy to use "**array**" function

array function used to transform a list to an **ndarray**

b is an **ndarray**

[By Amina Delali]

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)

     #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)
```

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

2 elements of dimension 2
(a 2 dimensionl element has2 external brackets)

3 elements of dimension 1

3 elements of dimension 0 =scalars

Number of external Brackets = dimension (=3)

The ndarray d is a 3 dimensional array, composed of: **2** elements of dimesion 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

[By Amina Delali]

1-Numpy: ndarray

Creating ndarray

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

[30]

```python
# asarray function : create an ndarray from the input.
# if the input is an ndarray, it will not be copied:
# the output and the input will refer to the same element.
g = np.asarray(b)
g[0]=155
print ("g=",g)
print("b=",b)

'''
array function : create an ndarray from the input.
even if the input is an ndarray, it will by default be copied:
the output and the input will refer to different elements.
To behave like asarray, it must be called with the optional argument 'copy'
set to false: copy(b,copy=False)
'''

h = np.array(b)
h[1]= 156
print("h=",h)
print("b=",b)
```

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

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**

[By Amina Delali]

Creating ndarray

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

Axis 1 == columns

id1

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1. & 0. & 0. \\ 1 & 0. & 1. & 0. \\ 2 & 0. & 0. & 1. \end{array}$$

1 in diagonal, **0** elsewhere

Axis 0 == lines

[By Amina Delali]

1-Numpy: ndarray

Creating ndarray

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

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

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

[By Amina Delali]

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 stirng 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”**

```python
a = np.full (5,3.2,dtype="i8")
print(a)
```

```
[3 3 3 3 3]
```

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

[By Amina Delali]

2- Indexing

indexes

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

c

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

c[0]==[1,2,3]

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

```
array([1, 2, 3])
```

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

Ind == [1, 1]
c[ind]==[[4,5,6],[4,5,6]]

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

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

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

C[: ,0:2]==[[1,2],[4,5]]

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

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

“ : “

0:2

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**)

```
ind2 = np.full_like(c,False, dtype="?")
print("ind2=",ind2)
ind2[0]=True
print("ind2=",ind2)
ind2[1,0]=True
print("ind2=",ind2)
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

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

[By Amina Delali]

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**

c

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

part

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1 & 2 \\ 1 & 4 & 5 \end{array}$$

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1 & 1000 \\ 1 & 4 & 5 \end{array}$$

[39]

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

c

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 1000 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

2- Indexing

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**

c

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

part2

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1 & 2 \\ 1 & 4 & 5 \end{array}$$

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1 & 1000 \\ 1 & 4 & 5 \end{array}$$

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

```
part2=
 [[1 2]
 [4 5]]
now part2 =
 [[   1 1000]
 [   4    5]]
and c always =
 [[1 2 3]
 [4 5 6]]
```

c

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

Arithmetic operations & Linear Algebra

3- Operations with Ndarrays

[52]

```python
# element wise multiplication: cell by corresponding cell,
# matrices with the same shape
res = c * np.eye(3)[0:2,0:3]

# the result is a matrix with the same shape
print ("res=",res)

# matrix multiplication:  line by columuns
# different shapes but:
# number of colums of the first matrix == number of lines of the second matrix
res = c @ np.eye(3)  # same as np.dot(c,np.eye(3)) or c.dot(np.eye(3))

# the result is a matrix wiht:
# number of lines == number of lines of the first matrix
# number of columns == number of columnus of the first matrix
print("res=",res)
```

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

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array} * \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{array} \rightarrow \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 0 & 0 \\ 1 & 0 & 5 & 0 \end{array}$$

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array} @ \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 0 & 0 \\ & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{array} \rightarrow \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

c @ indentity== c

[By Amina Delali]

3- Operations with Ndarrays

Element wise multiplication vs Matrix multiplication

Element wise multiplication

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array} \;*\; \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & -1 & 0 \\ 1 & -5 & 1 & 2 \end{array} \;\rightarrow\; \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & -2 & 0 \\ 1 & -20 & 5 & 12 \end{array}$$

1 * 1 == 1

6 * 2 == 12

Matrix multiplication

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array} \;@\; \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1 & -1 \\ 1 & 0 & -5 \\ 2 & 1 & 2 \end{array} \;\rightarrow\; \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 4 & -5 \\ 1 & 10 & -17 \end{array}$$

Line **0** * Column **0** == Cell (**0** , **0**):
1 * 1 + 2 * 0 + 3 * 1 == **4**

Line **1** * Column **1** == Cell (**1** , **1**)
4 * -1 + 5 * -5 + 6 * 2 = **-17**

[By]

Arithmetic and Logical operations

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

3- Operations with Ndarrays

Arithmetic and Logical operations

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

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

```
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

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

```
array([3, 4, 5, 6])
```


[By]

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:

$$x_1 + x_2 = 4$$
$$x_1 - x_2 = 0$$

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

And the solution will be : $$x_{(2,1)} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

Linear Algebra

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

```
# diag returns the diagonal of a square matrix
mat = np.random.randn(3,3)
print ("mat==",mat)
print("mat diagonal ==", np.diag(mat))
# trace retruns the sum of the diagonal elements
print(np.trace(np.eye(4)))
# inv return the inverse of a square matrix : mat * inv(mat)== identity matrix
print("mat inverse==",np.linalg.inv(mat))
# solve return the solution of the equation Ax=B (the values of x)
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

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

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

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

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

```
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']]
```

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

[By]

3- Operations with Ndarrays

Some functions and methods

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

```
print("c==",c)
# creating new array by appending a new values as a column (axis=1)
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”

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

c didn’t change

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

[By]

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]
print("c==",c)
print ("maximum element of c==",c.max())
print ("the sum of elements of c ==",c.sum())
print ("the cumulative sum of elements of c ==", c.cumsum())
print("the mean of values of c ==",c.mean())
```

```
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
```

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

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

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

[By]

Save and Load

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

```
]
# save c to "file_c.npy"
np.save("file_c",c)
# loading c from "file_c.npy" into
c2= np.load("file_c.npy")
print("c2==",c2)
# saving multiple ndarrays: c and d into "files.npz"
np.savez("files",c=c,d=d)
#loading c and d from "files.npz"
res = np.load("files.npz")
print("c==",res["c"])
print("d==",res["d"])
```

If the extensions “**npy**” or “**npz**” are not specified they will be **added**.

The extension has to be specified in loading data

Access to the arrays with the names used in the saving

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

4- Array saving and loading

5- Structures with dtype

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]
myType = np.dtype([("code","U5"),("Value","i4")])
myAr = np.array([("A",10),("B",2),("C",15)],myType)
print("myAr==",myAr)
print("first element==",myAr[0])
print ("Codes in myAr==",myAr["code"])
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