Python
Handling matrices with NumPy

Ricco Rakotomalala

http://data-mining-tutorials.blogspot.fr/

Numpy ?

- NumPy (numerical python) is a package for scientific computing. It provides tools for handling n-dimensional arrays (especially vectors and matrices).
- The objects are all the same type into a NumPy arrays structure
- The package offers a large number of routines for fast access to data (e.g. search, extraction), for various manipulations (e.g. sorting), for calculations (e.g. statistical computing)
- Numpy arrays are more efficient (speed, volume management) than the usual Python collections (list, tuple).
- Numpy arrays are underlying to many packages dedicated to scientific computing in Python.
- Note that a matrix is actually a 2 dimensional array

To go further, see the reference manual (used to prepare this slideshow).

http://docs.scipy.org/doc/numpy/reference/index.html

Creation on the fly, generation of a sequence, loading from a file

CREATING A NUMPY MATRIX

First, we must import the « NumPy » module

```
import numpy as np
```

np is the alias used for accessing to the routines of "NumPy".

Converting Python array_like objects (e.g. list)

$$\begin{pmatrix} 1.2 & 2.5 \\ 3.2 & 1.8 \\ 1.1 & 4.3 \end{pmatrix}$$

```
a = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
```

Note the role of [ ] and [ ] to define the parts of the matrix

Information about the structure

```
#object type
print(type(a))  #<class 'numpy.ndarray'>
#data type
print(a.dtype)  #float64
#number of dimensions
print(a.ndim)  #2 (because it is a matrix)
#number of rows and columns (tuple)
print(a.shape)  #(3,2) → 3 rows and 2 columns
#total number of values
print(a.size)  #6, nb.rows x nb.columns
```

Setting the data type

Visualizing the matrix

```
#print the whole object
print(a)
```

→

```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

Setting the data type may be implicit or explicit

```
#creating a matrix – implicit typing
a = np.array([[1,2],[4,7]])
print(a.dtype)  #int32

#explicit typing – preferable !
a = np.array([[1,2],[4,7]],dtype=float)
print(a.dtype)  #float64
```

As with vectors, creating a matrix of complex objects (other than the basic types) is possible

Creating matrix from a sequence of numbers

```
#evenly spaced values
#the dimensions must be compatible
a = np.arange(0,10).reshape(2,5)
print(a)
```

arange() generate values, 0 to 9.
reshape() reorganize the values in a matrix with 2 rows and 5 columns.

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

```
#a vector can be converted into matrix
a = np.array([2.1,3.4,6.7,8.1,3.5,7.2])
print(a.shape) # (6,)
#reshape in 3 rows x 2 columns
b = a.reshape(3,2)
print(b.shape) # (3, 2)
print(b)
```

```
[[ 2.1  3.4]
 [ 6.7  8.1]
 [ 3.5  7.2]]
```

```
#repeating 8 times the value 0
#e.g. for an initialization process
a = np.zeros(shape=(2,4))
print(a)
```

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

```
#more generally, repeating 8 times the value 0.1
a = np.full(shape=(2,4),fill_value=0.1)
print(a)
```

```
[[ 0.1  0.1  0.1  0.1]
 [ 0.1  0.1  0.1  0.1]]
```

The values can be stored in a text file (loadtxt for reading, savetxt for writting)

Note : if necessary, we can modify the default directory with the command **chdir()** of the module **os** (that must be imported)

matrice.txt

```
1  #age     poids    taille
2  45   62   168
3  34   68   187
4  53   85   159
```

The first row must be ignored. We use the # symbol.

```
#explicit typing
#column separator = tabulation « \t »
a = np.loadtxt("matrice.txt",delimiter="\t",dtype=float)
print(a)
```

```
[[  45.   62.  168.]
 [  34.   68.  187.]
 [  53.   85.  159.]]
```

We can convert a Python sequence type in a « numpy » array

```
#list of values
lst = [1.2,3.1,4.5,6.3]
print(type(lst)) # <class 'list'>
#conversion, 2 steps : asarray() and reshape()
a = np.asarray(lst,dtype=float).reshape(2,2)
print(a)
```

```
[[ 1.2  3.1]
 [ 4.5  6.3]]
```

Modifying the size of a matrix

a =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#matrix: 3 rows and 2 columns
a = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
```

Row binding (axis = 0)

```
#adding a new row
b = np.array([[4.1,2.6]])
c = np.append(a,b,axis=0)
print(c)
```

```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]
 [ 4.1  2.6]]
```

Column binding (axis = 1)

```
#adding a new column
d = np.array([[7.8],[6.1],[5.4]])
print(np.append(a,d,axis=1))
```

```
[[ 1.2  2.5  7.8]
 [ 3.2  1.8  6.1]
 [ 1.1  4.3  5.4]]
```

Insert a new row (axis = 0) at the row n°1

```
#insertion
print(np.insert(a,1,b,axis=0))
```

```
[[ 1.2  2.5]
 [ 4.1  2.6]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

Delete a row (axis = 0) from the position n°1

```
#removing
print(np.delete(a,1,axis=0))
```

```
[[ 1.2  2.5]
 [ 1.1  4.3]]
```

Modify the size of an existing matrix

```
#modify the size of a matrix
#reading the data the row by row
h = np.resize(a,new_shape=(2,3))
print(h)
```

```
[[ 1.2  2.5  3.2]
 [ 1.8  1.1  4.3]]
```

Indexing with indices of boolean array

EXTRACTING VALUES

```
v = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])

#printing all the values
print(v)
```

v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#indexed access – first valus
print(v[0,0]) # 1.2

#last value – note the use of "shape" which is a tuple
print(v[v.shape[0]-1,v.shape[1]-1]) # 4.3

#printing all the values, note the use of :
print(v[:,:])

#continguous indices
print(v[0:2,:])
```

```
[[ 1.2  2.5]
 [ 3.2  1.8]]
```

```
#extreme values, rows: start to 2 (not included), all columns
print(v[:2,:])
```

```
[[ 1.2  2.5]
 [ 3.2  1.8]]
```

```
#extreme values, rows:  1 to last, all columns
print(v[1:,:])
```

```
[[ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#negative index – last row and all the columns
print(v[-1,:])
```

```
[ 1.1  4.3]
```

```
#negative indices – last two rows and all columns
print(v[-2:,:])
```

```
[[ 3.2  1.8]
 [ 1.1  4.3]]
```

Note:

(1) Apart from singletons, the generated matrix is of type numpy.ndarray

(2) As with vectors, we can use non-contiguous indices

v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#indexing with a vector of booleans
#if b is too short, the remainder is considered False
b = np.array([True,False,True],dtype=bool)
print(v[b,:])
```

```
[[ 1.2  2.5]
 [ 1.1  4.3]]
```

#example: extract the row for which the sum is the lowest (among all the rows)

```
#calculate the sum of columns for each row
s = np.sum(v,axis=1)
print(s) # [ 3.7  5.  5.4 ]

#detect the rows for which the sum corresponds to the minimum
# maybe several rows are detected
b = (s == np.min(s))
print(b) # [ True  False  False]

#apply the boolean filter
print(v[b,:])
```

```
[[ 1.2  2.5]]
```

Note the square brackets [ ] : we obtain a **matrix** with 1 row and 2 columns.

Sorting and searching

v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#get the max of rows (axis = 0) for each column
print(np.max(v,axis=0)) # [ 3.2   4.3 ] -- 3.2 is the highest value of rows into the column 0, 4.3 is the highest for column 1

#get the max of columns (axis = 1) for each row
print(np.max(v,axis=1)) # [ 2.5  3.2  4.3]

#get the index of max within rows (axis = 0) for each column
print(np.argmax(v,axis=0)) # [ 1   2 ]

#sort the rows (axis = 0) for each column
# the relationship between the values of the same row is lost
print(np.sort(v,axis=0))
```

```
[[ 1.1  1.8]
 [ 1.2  2.5]
 [ 3.2  4.3]]
```

```
#get the sorted indices of rows for each column
print(np.argsort(v,axis=0))
```

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

Strategy to visit all the elements of a matrix

ITERATING OVER MATRIX

With indices, we can access to all the elements of a matrix: row by row, or column by column.

v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#indexed nested loop
s = 0.0
for i in range(0,v.shape[0]):
    for j in range(0,v.shape[1]):
        print(v[i,j])
        s = s + v[i,j]
print("Somme = ",s)
```

```
1.2
2.5
3.2
1.8
1.1
4.3
Somme =  14.1
```

“nditer” allows to visit every element of the matrix without using indices

v =

```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#iterator – accessing row by row
s = 0.0
for x in np.nditer(v):
    print(x)
    s = s + x
print("Somme = ",s)
```

```
1.2
2.5
3.2
1.8
1.1
4.3
Somme =  14.1
```

```
#iterator – accessing column by column
#"F" for" Fortran order "
s = 0.0
for x in np.nditer(v,order="F"):
    print(x)
    s = s +x
print("Somme = ",s)
```

```
1.2
3.2
1.1
2.5
1.8
4.3
Somme =  14.1
```

The iterator object is sophisticated and efficient. See http://docs.scipy.org/doc/numpy/reference/arrays.nditer.html

STATISTICAL ROUTINES

Principle: the calculations are performed over an axis (0: treating the values in rows for each column; 1: vice versa)

v =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

```
#mean of rows for each column
print(np.mean(v,axis=0)) # [1.833   2.867]

#mean of columns for each row
print(np.mean(v,axis=1)) # [1.85   2.5   2.7]

#cumulative sum of values in rows for each column
print(np.cumsum(v,axis=0))
```

```
[[ 1.2  2.5]
 [ 4.4  4.3]
 [ 5.5  8.6]]
```

```
#correlation matrix
#rowvar = 0 means the variables are organized in columns
m = np.corrcoef(v,rowvar=0)
print(m)
```

```
[[ 1.         -0.74507396]
 [-0.74507396  1.        ]]
```

The statistical functions are not numerous, we will need SciPy (and other)

NumPy shows its potential in the matrix calculations

MATRIX CALCULATIONS

x =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

y =
```
[[ 2.1  0.8]
 [ 1.3  2.5]]
```

```
#transposition
print(np.transpose(x))
```
→
```
[[ 1.2  3.2  1.1]
 [ 2.5  1.8  4.3]]
```

```
#multiplication
print(np.dot(x,y))
```
→
```
[[  5.77   7.21]
 [  9.06   7.06]
 [  7.9   11.63]]
```

```
#determinant
print(np.linalg.det(y))
```
4.21

```
#inverse
print(np.linalg.inv(y))
```
→
```
[[ 0.59382423 -0.19002375]
 [-0.3087886   0.49881235]]
```

x =
```
[[ 1.2  2.5]
 [ 3.2  1.8]
 [ 1.1  4.3]]
```

y =
```
[[ 2.1  0.8]
 [ 1.3  2.5]]
```

Solving

Y.**a** = z

```
#solve a linear matrix equation
z = np.array([1.7,1.0])
print(np.linalg.solve(y,z)) # [0.8195   -0.0261]
```

We can do

a = Y^{-1}.z

```
#checking
print(np.dot(np.linalg.inv(y),z)) # [0.8195   -0.0261]
```

#symmetric matric with X^TX

```
s = np.dot(np.transpose(x),x)
print(s)
```

```
[[ 12.89  13.49]
 [ 13.49  27.98]]
```

```
#eigenvalues and eigenvectors of a symmetric matrix
print(np.linalg.eigh(s))
```

```
(array([  4.97837925,  35.89162075]),
array([[-0.86259502,  0.50589508],
       [ 0.50589508,  0.86259502]]))
```

References

Course materials (in French)
http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_python.html

Python website
Welcome to Python - https://www.python.org/
Python **3.4.3** documentation - https://docs.python.org/3/index.html

NumPy Manual
Numpy User Guide and Numpy Reference

POLLS (KDnuggets)
Data Mining / Analytics Tools Used
Python, 4th in 2015
Primary programming language for Analytics, Data Mining, Data Science tasks
Python, 2nd in 2015 (next R)