**Lecture 9 (week 5.2)**

Thursday October 6th 2016

Numpy 2

This lecture will give you a more advanced description of numpy arrays and arrays operations. It is far from covering everything but it gives you the basic toolkit you need in order to start using numpy arrays for many scientific applications.

Numpy array views and memory usage

We have already seen the basic array creation with **numpy.array()** and slicing. If you have a list, you can also create a numpy array from it:

**import numpy as np**

**mylist=[[1,2,3],[4,5,6],[7,8,9]]**

**a=np.array(mylist)**

**type(a)**

remember that numpy arrays have *attributes* (**shape**, **size**, **dtype**, etc…). Remember also *slicing*, e.g. using the array **a** defined above:

**In [5]: a**

**Out[5]:**

**array([[1, 2, 3],**

**[4, 5, 6],**

**[7, 8, 9]])**

**In [6]: a[:,0]**

**Out[6]: array([1, 4, 7])**

**In [7]: a[0,:]**

**Out[7]: array([1, 2, 3])**

**In [9]: a[:,0]=[9,9,9]**

**In [10]: a**

**Out[10]:**

**array([[9, 2, 3],**

**[9, 5, 6],**

**[9, 8, 9]])**

Arrays keep their types even if elements are changed:

**In [11]: a=np.array([1,2,3,4])**

**In [12]: a[0]=3.2**

**In [13]: a**

**Out[13]: array([3, 2, 3, 4])**

Note that it is possible to change an array type with the **astype()** method:

**c=a.astype(np.float32)**

Numpy prefers to use *views* of an original arrays used under a different name rather than create a new array in memory. The example below is a bit counter intuitive, however, very important!

**In [29]: a=np.array([1.,2.,3.,4.,5.])**

**In [30]: a**

**Out[30]: array([ 1., 2., 3., 4., 5.])**

**In [31]: b=a**

**In [32]: np.may\_share\_memory(a,b)** *# this is asking if b and a*

*are sharing some memory space*

**Out[32]: True**

**In [33]: b**

**Out[33]: array([ 1., 2., 3., 4., 5.])**

**In [34]: b[0]=99.**

**In [35]: b**

**Out[35]: array([ 99., 2., 3., 4., 5.])**

**In [36]: a**

**Out[36]: array([ 99., 2., 3., 4., 5.])**

In this example, changing b is also changing a! Note the use of the **may\_share\_memory()** method to check if two arrays share memory or not. To create a separate copy of the initial array, you can use the method **np.copy():**

**b=np.copy(a)**

Array filtering and sorting

Array filtering is the action of accessing a subarray from an existing array. You can access a subarray directly (i.e. knowing the indices pointing to specific location in an array) or using some conditional statement.

An example of direct access is when you use lists (or arrays) of indices to access arrays elements:

**In [15]: arr = np.arange(100, 200)**

**In [17]: select = [5, 25, 50, 75, -5]** *# this is a list*

**In [18]: type(select)**

**Out[18]: list**

**In [19]: print(arr[select])**

**[105 125 150 175 195]**

**In [20]: print(arr[np.array(select)])**

**[105 125 150 175 195]**

It is possible to “filter” an array by applying a condition on the array elements. Imagine you want to select all positive elements from a numpy array of 10 random numbers drawn from a normal distribution:

I**n [12]: import numpy as np**

**In [13]: a=np.random.randn(10)**

**In [14]: select=a>0**

**In [15]: select**

**Out[15]: array([False, False, False, False, False, False, True, True, True, False], dtype=bool)**

**In [16]: a[select]**

**Out[16]: array([ 0.86346597, 1.46032268, 0.91075946])**

**In [17]: a**

**Out[17]:**

**array([-0.06171418, -2.39908035, -1.01373213, -0.93808913, -1.28325634,**

**-1.08430785, 0.86346597, 1.46032268, 0.91075946, -1.28469151])**

Note that you can use combined several filters in one, for instance the following command creates a filter that selects arrays elements that are between 0 and one:

**select = (a>0) & (a<1)**

You can also extract the indices where a condition is true using the numpy **where()** method:

**In [17]: a**

**Out[17]:**

**array([-0.06171418, -2.39908035, -1.01373213, -0.93808913, -1.28325634,**

**-1.08430785, 0.86346597, 1.46032268, 0.91075946, -1.28469151])**

**In [18]: np.where(a > 0)**

**Out[18]: (array([6, 7, 8]),)**

**In [19]: ind=np.where(a > 0)**

**In [21]: a[ind]**

**Out[21]: array([ 0.86346597, 1.46032268, 0.91075946])**

The last statement allows you to access the indices the array where the condition is true and create a subarray with the selected values.

**IMPORTANT**: Accessing subarrays with masks or lists as shown above ALWAYS returns new copies of the original array never views.

Along with filtering, another useful array manipulation tool is sorting. The method **sort()** sorts the elements of a numpy array. Sorting can be very time consuming with big arrays, the numpy sorting routine allows you to select different sorting methods as function of the result you want to achieve:

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html>

Note that sorting multidimensional arrays is possible and the **sort()** method can be called with different options depending on how you want to sort the different axis of the array.

Numpay arrays and functions

Another numpy feature important to know is how functions behave when called with numpy arrays. If you call a numpy function with a numpy array, it returns a numpy array. It is also the case for user defined functions ONLY if the returning value is a numpy (or numpy compliant) operation. For instance the function **squarecos()**:

**def squarecos(b):**

**return b\*b+np.cos(b)**

will return a numpy array if called with a numpay array, but it will not work (you get a runtime error) if you use the **math.cos(a)** function instead of the **np.cos(a)**, experiment with it.

In reality there is no clear cut between numpy-compliant and numpy-non compliant functions. For instance the function below uses numpy-compliant operations:

**def my\_sign(a):**

**if a > 0:**

**return 1**

**if a < 0:**

**return -1**

**if a == 0:**

**return 0**

However, you get an error if you run it with an array, e.g.

**print(my\_sign(np.array([-1,0,1]))**

The reason is because the array elements are treated differently by the function. There will be more on this with today’s lab.

Numpy meshgrid and 2D plots

An extremely useful feature of numpy is the method **meshgrid()**. It allows you to expand a 1D vector into 2 or more dimensions, which is very useful when you want to represent functions on grid. It is formally explained there:

<http://docs.scipy.org/doc/numpy/reference/generated/numpy.meshgrid.html>

Consider the following example: imagine you want to plot an image of the function **f(x,y)=exp(-(x^2+y^2))** in the region **x=[-3,3]** and **y=[0,2]**. It would be very convenient to be able to do this with one call to the function **exp()** and one line of code, i.e. you don’t want to do this with a **for** loop over all possible values of **x** and **y**. **meshgrid()** allows you create a **x** and **y** 2-dimensional template that covers the region you want.

First create 1-dimensional vectors x1d and y1d which cover the desired range:

**In [71]: x1d=np.array([-3,-2,-1,0,1,2,3])**

**In [72]: y1d=np.array([0,1,2])**

Now use **meshgrid()** to create the 2-dimensional templates x2d and y2d:

**In [74]: x2d,y2d=np.meshgrid(x1d,y1d)**

**In [75]: x2d**

**Out[75]:**

**array([[-3, -2, -1, 0, 1, 2, 3],**

**[-3, -2, -1, 0, 1, 2, 3],**

**[-3, -2, -1, 0, 1, 2, 3]])**

**In [76]: y2d**

**Out[76]:**

**array([[0, 0, 0, 0, 0, 0, 0],**

**[1, 1, 1, 1, 1, 1, 1],**

**[2, 2, 2, 2, 2, 2, 2]])**

Now you can create the array r which contains the 2-dimensional array of the squared distance from the origin (0,0) for the desired region:

**In [77]: r2d=x2d\*\*2+y2d\*\*2**

**In [78]: r2d**

**Out[78]:**

**array([[ 9, 4, 1, 0, 1, 4, 9],**

**[10, 5, 2, 1, 2, 5, 10],**

**[13, 8, 5, 4, 5, 8, 13]])**

Now, the following command calculates **f(x,y)=exp(-(x^2+y^2))** in one line, no loops, computing time is optimized (this is especially true for big arrays!):

**In [80]: z=np.exp(-r2d)**

The sampling in the above example was is too crude to make nice images. Now redo the same thing with a much finer sampling using this script:

**import numpy as np**

**import matplotlib.pyplot as plt**

**x1d=np.arange(-3,3,0.01)**

**y1d=np.arange(-1,3,0.01)**

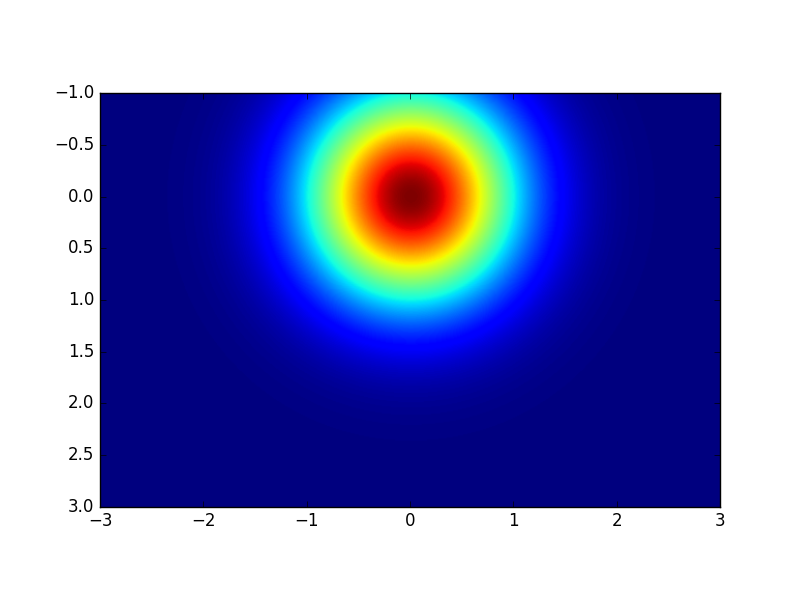
**x2d,y2d=np.meshgrid(x1d,y1d)**

**r2d=x2d\*\*2+y2d\*\*2**

**z=np.exp(-r2d)**

**plt.imshow(z,extent=(-3,3,3,-1))**

**plt.show(block=False)**

you get the nice image:

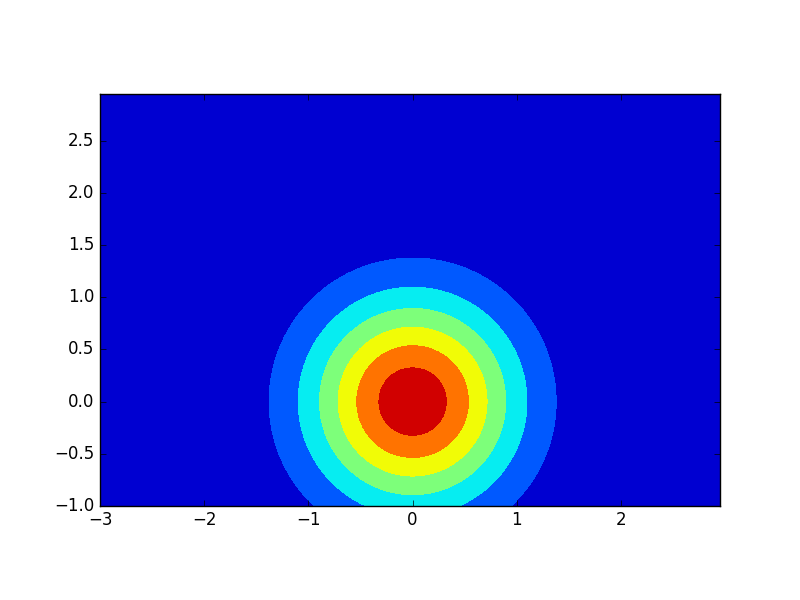
You can also get a contour plot using the **contourf()** method of pyplot (I leave it to you to explore the arguments of **contourf()**):

**In [15]: plt.axes().set\_aspect('equal')**

**In [16]: plt.contourf(x2d,y2d,z)**

**Out[16]: <matplotlib.contour.QuadContourSet at 0x113b6b5c0>**

**In [17]: plt.show()**

As a home practice/exercise, think how you would filter a 2-dimensional array.