Numpy - multidimensional data arrays

J.R. Johansson (jrjohansson at gmail.com)

The latest version of this IPython notebook (http://ipython.org/notebook.html) lecture is available at http://github.com/jrjohansson/scientific-python-lectures (http://github.com/jrjohansson/scientific-python-lectures).

The other notebooks in this lecture series are indexed at http://jrjohansson.github.io (http://jrjohansson.github.io).

In [13]:

```
# what is this line all about?!? Answer in lecture 4
%matplotlib inline
import matplotlib.pyplot as plt
```

Introduction

The `numpy` package (module) is used in almost all numerical computation using Python. It is a package that provide high-performance vector, matrix and higher-dimensional data structures for Python. It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), performance is very good.

To use `numpy` you need to import the module, using for example:

In [1]:

```
from numpy import *
```

In the `numpy` package the terminology used for vectors, matrices and higher-dimensional data sets is *array*.

Creating `numpy` arrays

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples
- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files

From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function.

In [2]:

```
import numpy as np #alias
# a vector: the argument to the array function is a Python list
v = np.array([1,2,3,4])

v
```

Out[2]:

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

In [4]:

```
# a matrix: the argument to the array function is a nested Python list
M = np.array([[1, 2], [3, 4]])
M
#type(M)
```

Out[4]:

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

The `v` and `M` objects are both of the type `ndarray` that the `numpy` module provides.

In [5]:

```
type(v), type(M)
```

Out[5]:

```
(numpy.ndarray, numpy.ndarray)
```

The difference between the `v` and `M` arrays is only their shapes. We can get information about the shape of an array by using the `ndarray.shape` property.

In [6]:

```
v.shape
```

Out[6]:

```
(4,)
```

In [7]:

```
M.shape
```

Out[7]:

```
(2, 2)
```

The number of elements in the array is available through the `ndarray.size` property:

In [7]:

```
M.size
```

Out[7]:

```
4
```

Equivalently, we could use the function `numpy.shape` and `numpy.size`

In [9]:

```
shape(M)
```

Out[9]:

```
(2, 2)
```

In [10]:

```
size(M)
```

Out[10]:

```
4
```

So far the `numpy.ndarray` looks awefully much like a Python list (or nested list). Why not simply use Python lists for computations instead of creating a new array type?

There are several reasons:

- Python lists are very general. They can contain any kind of object. They are dynamically typed. They do not support mathematical functions such as matrix and dot multiplications, etc. Implementing such functions for Python lists would not be very efficient because of the dynamic typing.
- Numpy arrays are **statically typed** and **homogeneous**. The type of the elements is determined when the array is created.
- Numpy arrays are memory efficient.
- Because of the static typing, fast implementation of mathematical functions such as multiplication and addition of `numpy` arrays can be implemented in a compiled language (C and Fortran is used).

Using the `dtype` (data type) property of an `ndarray`, we can see what type the data of an array has:

In [11]:

```
M.dtype
```

Out[11]:

```
dtype('int64')
```

We get an error if we try to assign a value of the wrong type to an element in a numpy array:

In [5]:

```
M[1,0] = 5
M
```

Out[5]:

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

If we want, we can explicitly define the type of the array data when we create it, using the `dtype` keyword argument:

In [11]:

```
M =array([[1, 2], [3, 4]], dtype=complex)

M
```

Out[11]:

```
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Common data types that can be used with `dtype` are: `int`, `float`, `complex`, `bool`, `object`, etc.

We can also explicitly define the bit size of the data types, for example: `int64`, `int16`, `float128`, `complex128`.

Using array-generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in `numpy` that generate arrays of different forms. Some of the more common are:

arange

In [13]:

```
# create a range

x =np.arange(0, 10, 0.4) # arguments: start, stop, step

x
```

Out[13]:

```
array([0. , 0.4, 0.8, 1.2, 1.6, 2. , 2.4, 2.8, 3.2, 3.6, 4. , 4.4, 4.8,
       5.2, 5.6, 6. , 6.4, 6.8, 7.2, 7.6, 8. , 8.4, 8.8, 9.2, 9.6])
```

In [15]:

```
x = np.arange(-1, 1, 0.1)

x
```

Out[15]:

```
array([ -1.00000000e+00,  -9.00000000e-01,  -8.00000000e-01,
        -7.00000000e-01,  -6.00000000e-01,  -5.00000000e-01,
        -4.00000000e-01,  -3.00000000e-01,  -2.00000000e-01,
        -1.00000000e-01,  -2.22044605e-16,   1.00000000e-01,
         2.00000000e-01,   3.00000000e-01,   4.00000000e-01,
         5.00000000e-01,   6.00000000e-01,   7.00000000e-01,
         8.00000000e-01,   9.00000000e-01])
```

linspace and logspace

In [6]:

```
# using linspace, both end points ARE included
np.linspace(0, 10, 9)
```

Out[6]:

```
array([ 0.  ,  1.25,  2.5 ,  3.75,  5.  ,  6.25,  7.5 ,  8.75, 10.  ])
```

In [17]:

```
np.logspace(0, 10, 10, base=e)
```

Out[17]:

```
array([  1.00000000e+00,   3.03773178e+00,   9.22781435e+00,
         2.80316249e+01,   8.51525577e+01,   2.58670631e+02,
         7.85771994e+02,   2.38696456e+03,   7.25095809e+03,
         2.20264658e+04])
```

mgrid

In [7]:

```
x, y = np.mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
```

In [8]:

```
x
```

Out[8]:

```
array([[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]])
```

In [20]:

```
y
```

Out[20]:

```
array([[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]])
```

random data

In [10]:

```
from numpy import random
```

In [7]:

```
# uniform random numbers in [0,1]
np.random.rand(3,3)
```

Out[7]:

```
array([[0.73334395, 0.96200168, 0.25332436],
       [0.5218926 , 0.9207092 , 0.08943997],
       [0.22059853, 0.72792416, 0.87695863]])
```

In [23]:

```
# standard normal distributed random numbers
np.random.randn(5,5)
```

Out[23]:

```
array([[ 0.117907  , -1.57016164,  0.78256246,  1.45386709,  0.54744436],
       [ 2.30356897, -0.28352021, -0.9087325 ,  1.2285279 , -1.00760167],
       [ 0.72216801,  0.77507299, -0.37793178, -0.31852241,  0.84493629],
       [-0.10682252,  1.15930142, -0.47291444, -0.69496967, -0.58912034],
       [ 0.34513487, -0.92389516, -0.216978  ,  0.42153272,  0.86650101]])
```

diag

In [24]:

```
# a diagonal matrix
np.diag([1,2,3])
```

Out[24]:

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

In [11]:

```
# diagonal with offset from the main diagonal
diag([1,2,3], k=2)
```

Out[11]:

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

zeros and ones

In [32]:

```
np.zeros((5,5))
```

Out[32]:

```
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.]])
```

In [27]:

```
np.ones((3,3))
```

Out[27]:

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

File I/O

Comma-separated values (CSV)

A very common file format for data files is comma-separated values (CSV), or related formats such as TSV (tab-separated values). To read data from such files into Numpy arrays we can use the `numpy.genfromtxt` function. For example,

In [14]:

```
!head data.csv
```

```
'head' is not recognized as an internal or external command,
operable program or batch file.
```

In [14]:

```
data = np.genfromtxt('data.csv', delimiter=',')
data
```

Out[14]:

```
array([[ 218475.  ,  218475.  ,  218475.  ,  218475.  ,  199030.29],
       [1400904.  , 1400904.  , 1400904.  , 1400904.  , 1772984.1 ],
       [   4365.  ,    4365.  ,    4365.  ,    4365.  ,    4438.05],
       [   4365.  ,    4365.  ,    4365.  ,    4365.  ,    6095.72],
       [  39789.  ,   39789.  ,   39789.  ,   39789.  ,   58106.58],
       [  24867.  ,   24867.  ,   24867.  ,   24867.  ,   18969.79],
       [ 213876.  ,  213876.  ,  213876.  ,  213876.  ,  261435.18],
       [  69435.  ,   69435.  ,   69435.  ,   69435.  ,   93674.34],
       [  14922.  ,   14922.  ,   14922.  ,   14922.  ,   12333.03],
       [ 165546.  ,  165546.  ,  165546.  ,  165546.  ,  239134.51],
       [  72837.  ,   72837.  ,   72837.  ,   72837.  ,   86637.86],
       [  72837.  ,   72837.  ,   72837.  ,   72837.  ,   98147.86],
       [  19440.  ,   19440.  ,   19440.  ,   19440.  ,   30658.59],
       [   9945.  ,    9945.  ,    9945.  ,    9945.  ,   11551.12]])
```

In [27]:

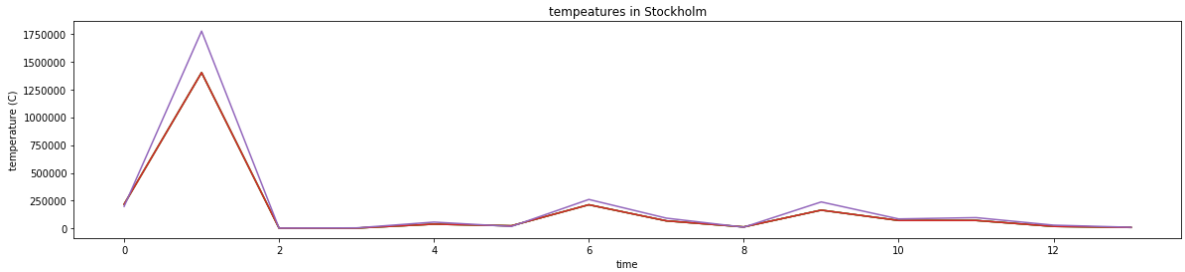
```
data.shape
```

Out[27]:

```
(14, 5)
```

In [17]:

```
fig, ax = plt.subplots(figsize=(20,4))
ax.plot(data[:,:])
ax.axis('tight')
ax.set_title('tempeatures in Stockholm')
ax.set_xlabel('time')
ax.set_ylabel('temperature (C)');
```



Using `numpy.savetxt` we can store a Numpy array to a file in CSV format:

In [49]:

```
M = random.rand(3,3)

M
```

Out[49]:

```
array([[0.79832236, 0.45569923, 0.02023379],
       [0.73959414, 0.78307576, 0.43381118],
       [0.419933  , 0.44028841, 0.17959896]])
```

In [50]:

```
np.savetxt("random-matrix.csv", M)
```

In [ ]:

```

```

In [53]:

```
np.savetxt("random-matrix.csv", M, fmt='%.5f') # fmt specifies the format
```

In [61]:

```
A = array([[n+m*10 for n in range(5)] for m in range(5)])

A
```

Out[61]:

```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

In [62]:

```
# a block from the original array
A[1:4, 1:4] #A[r,c]
```

Out[62]:

```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

In [63]:

```
# strides
A[::2, ::2]
```

Out[63]:

```
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])
```

In [66]:

```
B = np.array([n for n in range(5)])
B
```

Out[66]:

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

In [67]:

```
row_mask = array([True, False, True, False, False])
B[row_mask]
```

Out[67]:

```
array([0, 2])
```

In [68]:

```
# same thing
row_mask = array([1,0,1,0,0], dtype=bool)
B[row_mask]
```

Out[68]:

```
array([0, 2])
```

This feature is very useful to conditionally select elements from an array, using for example comparison operators:

In [69]:

```
x = arange(0, 10, 0.5)
x
```

Out[69]:

```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
        5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
```

In [70]:

```
mask = (5 < x) * (x < 7.5)

mask
```

Out[70]:

```
array([False, False, False, False, False, False, False, False, False,
       False, False,  True,  True,  True,  True, False, False, False,
       False, False], dtype=bool)
```

In [71]:

```
x[mask]
```

Out[71]:

```
array([ 5.5,  6. ,  6.5,  7. ])
```

Calculations with higher-dimensional data

When functions such as `min`, `max`, etc. are applied to a multidimensional arrays, it is sometimes useful to apply the calculation to the entire array, and sometimes only on a row or column basis. Using the `axis` argument we can specify how these functions should behave:

In [35]:

```
m =np.random.rand(3,3)
m
```

Out[35]:

```
array([[0.39757376, 0.57489109, 0.32198865],
       [0.17202624, 0.66385436, 0.56517576],
       [0.82489285, 0.22452581, 0.01511898]])
```

In [37]:

```
# global max
m.min()
```

Out[37]:

```
0.015118975672845703
```

In [40]:

```
# max in each column
m.max(axis=0)
```

Out[40]:

```
array([0.82489285, 0.66385436, 0.56517576])
```

In [131]:

```
# max in each row
m.max(axis=1)
```

Out[131]:

```
array([ 0.2850926 ,  0.80070487,  0.87010206])
```

Copy and "deep copy"

To achieve high performance, assignments in Python usually do not copy the underlaying objects. This is important for example when objects are passed between functions, to avoid an excessive amount of memory copying when it is not necessary (technical term: pass by reference).

In [42]:

```
A = np.array([[1, 2], [3, 4]])

A
```

Out[42]:

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

In [43]:

```
# now B is referring to the same array data as A
B = A
B
```

Out[43]:

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

In [45]:

```
# changing B affects A
B[0,0] = 10

B
```

Out[45]:

```
array([[10,  2],
       [ 3,  4]])
```

In [46]:

```
A
```

Out[46]:

```
array([[10,  2],
       [ 3,  4]])
```

If we want to avoid this behavior, so that when we get a new completely independent object `B` copied from `A`, then we need to do a so-called "deep copy" using the function `copy`:

In [157]:

```
B = copy(A)
```

In [158]:

```
# now, if we modify B, A is not affected
B[0,0] = -5

B
```

Out[158]:

```
array([[-5,  2],
       [ 3,  4]])
```

In [159]:

```
A
```

Out[159]:

```
array([[10,  2],
       [ 3,  4]])
```

Iterating over array elements

Generally, we want to avoid iterating over the elements of arrays whenever we can (at all costs). The reason is that in a interpreted language like Python (or MATLAB), iterations are really slow compared to vectorized operations.

However, sometimes iterations are unavoidable. For such cases, the Python `for` loop is the most convenient way to iterate over an array:

In [160]:

```
v = array([1,2,3,4])

for element in v:
    print(element)
```

```
1
2
3
4
```

In [161]:

```
M = array([[1,2], [3,4]])

for row in M:
    print("row", row)

    for element in row:
        print(element)
```

```
('row', array([1, 2]))
1
2
('row', array([3, 4]))
3
4
```

When we need to iterate over each element of an array and modify its elements, it is convenient to use the `enumerate` function to obtain both the element and its index in the `for` loop:

In [163]:

```
# each element in M is now squared
M
```

Out[163]:

```
array([[ 1,  4],
       [ 9, 16]])
```

Vectorizing functions

As mentioned several times by now, to get good performance we should try to avoid looping over elements in our vectors and matrices, and instead use vectorized algorithms. The first step in converting a scalar algorithm to a vectorized algorithm is to make sure that the functions we write work with vector inputs.

Using arrays in conditions

When using arrays in conditions,for example `if` statements and other boolean expressions, one needs to use `any` or `all`, which requires that any or all elements in the array evalutes to `True`:

In [61]:

```
M=array([[ 1,  4],
       [ 9, 16]])
M
```

Out[61]:

```
array([[ 1,  4],
       [ 9, 16]])
```

In [62]:

```
if (M > 5).any():
    print("at least one element in M is larger than 5")
else:
    print("no element in M is larger than 5")
```

```
at least one element in M is larger than 5
```

In [63]:

```
if (M > 5).all():
    print("all elements in M are larger than 5")
else:
    print("all elements in M are not larger than 5")
```

```
all elements in M are not larger than 5
```

Type casting

Since Numpy arrays are *statically typed*, the type of an array does not change once created. But we can explicitly cast an array of some type to another using the `astype` functions (see also the similar `asarray` function). This always create a new array of new type:

In [174]:

```
M.dtype
```

Out[174]:

```
dtype('int64')
```

In [175]:

```
M2 = M.astype(float)

M2
```

Out[175]:

```
array([[  1.,   4.],
       [  9.,  16.]])
```

In [176]:

```
M2.dtype
```

Out[176]:

```
dtype('float64')
```

In [177]:

```
M3 = M.astype(bool)

M3
```

Out[177]:

```
array([[ True,  True],
       [ True,  True]], dtype=bool)
```

Further reading

- http://numpy.scipy.org (http://numpy.scipy.org)
- http://scipy.org/Tentative_NumPy_Tutorial (http://scipy.org/Tentative_NumPy_Tutorial)
- http://scipy.org/NumPy_for_Matlab_Users (http://scipy.org/NumPy_for_Matlab_Users) - A Numpy guide for MATLAB users.

Versions

In [178]:

```
%reload_ext version_information

%version_information numpy
```

Out[178]:

Software	Version
Python	2.7.10 64bit [GCC 4.2.1 (Apple Inc. build 5577)]
IPython	3.2.1
OS	Darwin 14.1.0 x86_64 i386 64bit
numpy	1.9.2
	Sat Aug 15 11:02:09 2015 JST