Numpy - multidimensional data arrays

J.R. Johansson (robert@riken.jp) http://dml.riken.jp/~rob/

The latest version of this IPython notebook (http://ipython.org/ipython-doc/dev/interactive/htmlnotebook.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.com (http://jrjohansson.github.com).

In [1]:

```
# what is this line all about?!? Answer in lecture 4
%pylab inline
```

```
Welcome to pylab, a matplotlib-based Python environment [backend: module://IPython.zmq.pylab.bac
For more information, type 'help(pylab)'.
```

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` need to import the module it using of example:

In [2]:

```
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 [3]:

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

v
```

Out[3]:

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

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

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 [8]:
```
M.size
```

Out[8]:
```
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. Implementating 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 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 [12]:
```
M[0,0] = "hello"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-a09d72434238> in <module>()
----> 1 M[0,0] = "hello"

ValueError: invalid literal for long() with base 10: 'hello'
```

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

In [13]:
```
M = array([[1, 2], [3, 4]], dtype=complex)

M
```

Out[13]:
```
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Common type 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 pythons lists. Instead we can use one of the many functions in `numpy` that generates arrays of different forms. Some of the more common are:

arange

In [14]:
```
# create a range

x = arange(0, 10, 1) # arguments: start, stop, step

x
```

Out[14]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [15]:
```
x = 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 [16]:
```
# using linspace, both end points ARE included
linspace(0, 10, 25)
```

Out[16]:
```
array([  0.        ,   0.41666667,   0.83333333,   1.25      ,
         1.66666667,   2.08333333,   2.5       ,   2.91666667,
         3.33333333,   3.75      ,   4.16666667,   4.58333333,
         5.        ,   5.41666667,   5.83333333,   6.25      ,
         6.66666667,   7.08333333,   7.5       ,   7.91666667,
         8.33333333,   8.75      ,   9.16666667,   9.58333333,  10.        ])
```

In [17]:
```
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 [18]:
```
x, y = mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
```

In [19]:
```
x
```

Out[19]:
```
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 [21]:
```
from numpy import random
```

In [22]:
```
# uniform random numbers ini [0,1]
random.rand(5,5)
```

Out[22]:
```
array([[ 0.38514869,  0.65611855,  0.30951719,  0.90606323,  0.45323021],
       [ 0.4829053 ,  0.71078648,  0.27249177,  0.06156748,  0.49899315],
       [ 0.81852145,  0.65724548,  0.77194554,  0.29973648,  0.87633625],
       [ 0.37501764,  0.10998782,  0.5567457 ,  0.26298218,  0.97630491],
       [ 0.81460477,  0.8886327 ,  0.46886708,  0.29431937,  0.16157934]])
```

In [23]:
```
# standard normal distributed random numbers
random.randn(5,5)
```

Out[23]:
```
array([[ 1.17984323,  0.12248472,  0.16712688, -0.63193807, -1.0372697 ],
       [-0.28335305, -0.92302383,  1.41181247,  0.46338623, -1.53910004],
       [ 0.08862918,  1.12887421,  1.07811757, -0.27373696, -1.25380144],
       [ 2.80918157, -0.79861234,  0.27846162, -1.21928768, -0.0844151 ],
       [-0.29196407, -0.5398782 , -0.18096382, -1.12382364, -0.92178747]])
```

diag

In [24]:
```
# a diagonal matrix
diag([1,2,3])
```

Out[24]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

In [25]:
```
# diagonal with offset from the main diagonal
diag([1,2,3], k=1)
```

Out[25]:
```
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```

zeros and ones

In [26]:
```
zeros((3,3))
```

Out[26]:
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

In [27]:
```
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 are the comma-separated values (CSV), or related format such as TSV (tab-separated values). To read data from such file into Numpy arrays we can use the `numpy.genfromtxt` function. For example,

In [28]:
```
!head stockholm_td_adj.dat
```

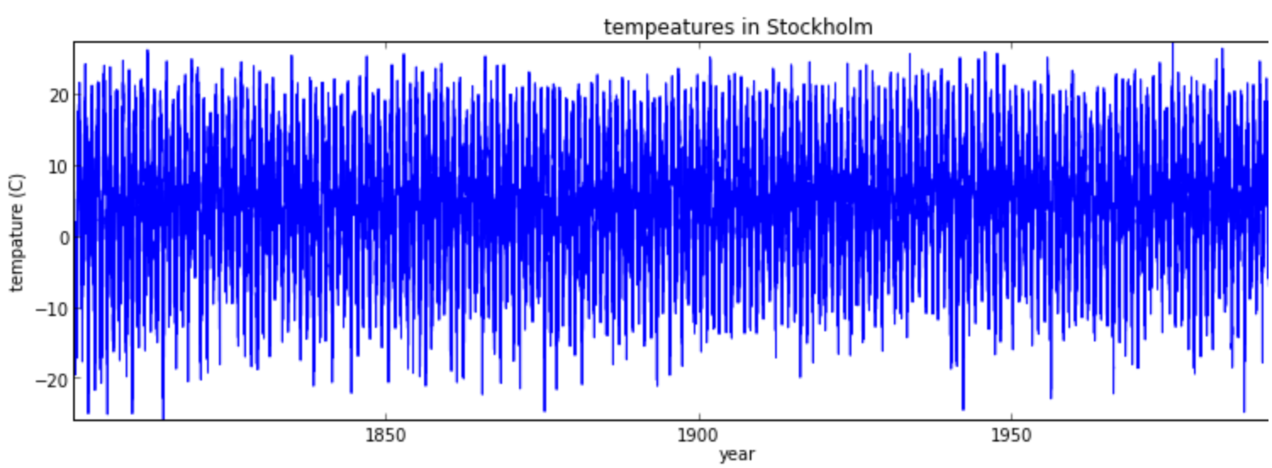
```
1800  1  1    -6.1    -6.1    -6.1 1
1800  1  2   -15.4   -15.4   -15.4 1
1800  1  3   -15.0   -15.0   -15.0 1
1800  1  4   -19.3   -19.3   -19.3 1
1800  1  5   -16.8   -16.8   -16.8 1
1800  1  6   -11.4   -11.4   -11.4 1
1800  1  7    -7.6    -7.6    -7.6 1
1800  1  8    -7.1    -7.1    -7.1 1
1800  1  9   -10.1   -10.1   -10.1 1
1800  1 10    -9.5    -9.5    -9.5 1
```

In [29]:
```
data = genfromtxt('stockholm_td_adj.dat')
```

In [30]:
```
data.shape
```

Out[30]:
```
(77431, 7)
```

In [31]:
```
fig, ax = subplots(figsize=(14,4))
ax.plot(data[:,0]+data[:,1]/12.0+data[:,2]/365, data[:,5])
ax.axis('tight')
ax.set_title('tempeatures in Stockholm')
ax.set_xlabel('year')
ax.set_ylabel('tempature (C)');
```



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

In [32]:
```
M = rand(3,3)

M
```

Out[32]:
```
array([[ 0.43893135,  0.46635226,  0.4070475 ],
       [ 0.53910705,  0.64968622,  0.85079048],
       [ 0.45170465,  0.97032227,  0.31628198]])
```

In [33]:
```
savetxt("random-matrix.csv", M)
```

In [34]:
```
!cat random-matrix.csv
```

```
4.389313531846058547e-01 4.663522629835443745e-01 4.070474979109756086e-01
5.391070529944648193e-01 6.496862221899694090e-01 8.507904796404367476e-01
4.517046464380731763e-01 9.703222696663832414e-01 3.162819794660202133e-01
```

In [35]:
```
savetxt("random-matrix.csv", M, fmt='%.5f') # fmt specifies the format

!cat random-matrix.csv
```

```
0.43893 0.46635 0.40705
0.53911 0.64969 0.85079
0.45170 0.97032 0.31628
```

Numpy's native file format

Useful when storing and reading back numpy array data. Use the functions `numpy.save` and `numpy.load`:

In [36]:
```
save("random-matrix.npy", M)

!file random-matrix.npy
```

```
random-matrix.npy: data
```

In [37]:
```
load("random-matrix.npy")
```

Out[37]:
```
array([[ 0.43893135,  0.46635226,  0.4070475 ],
       [ 0.53910705,  0.64968622,  0.85079048],
       [ 0.45170465,  0.97032227,  0.31628198]])
```

More properties of the numpy arrays

In [38]:
```
M.itemsize # bytes per element
```

Out[38]: 8

In [39]:
```
M.nbytes # number of bytes
```

Out[39]: 72

In [40]:
```
M.ndim # number of dimensions
```

Out[40]: 2

Manipulating arrays

Indexing

We can index elements in an array using the square bracket and indices:

In [41]:
```
# v is a vector, and has only one dimension, taking one index
v[0]
```

Out[41]: 1

In [42]:
```
# M is a matrix, or a 2 dimensional array, taking two indices
M[1,1]
```

Out[42]:
```
0.64968622218996941
```

If we omit an index of a multidimensional array it returns the whole row (or, in general, a N-1 dimensional array)

In [43]:
```
M
```

Out[43]:
```
array([[ 0.43893135,  0.46635226,  0.4070475 ],
       [ 0.53910705,  0.64968622,  0.85079048],
       [ 0.45170465,  0.97032227,  0.31628198]])
```

In [44]:
```
M[1]
```

Out[44]:
```
array([ 0.53910705,  0.64968622,  0.85079048])
```

The same thing can be achieved with using `:` instead of an index:

In [45]:
```
M[1,:] # row 1
```

Out[45]:
```
array([ 0.53910705,  0.64968622,  0.85079048])
```

In [46]:
```
M[:,1] # column 1
```

Out[46]:
```
array([ 0.46635226,  0.64968622,  0.97032227])
```

We can assign new values to elements in an array using indexing:

In [47]:
```
M[0,0] = 1
```

In [48]:
```
M
```

Out[48]:
```
array([[ 1.        ,  0.46635226,  0.4070475 ],
       [ 0.53910705,  0.64968622,  0.85079048],
       [ 0.45170465,  0.97032227,  0.31628198]])
```

In [49]:
```
# also works for rows and columns
M[1,:] = 0
M[:,2] = -1
```

In [50]:
```
M
```

Out[50]:
```
array([[ 1.        ,  0.46635226, -1.        ],
       [ 0.        ,  0.        , -1.        ],
       [ 0.45170465,  0.97032227, -1.        ]])
```

Index slicing

Index slicing is the technical name for the syntax `M[lower:upper:step]` to extract part of an array:

In [51]:
```
A = array([1,2,3,4,5])
A
```

Out[51]:
```
array([1, 2, 3, 4, 5])
```

In [52]:
```
A[1:3]
```

Out[52]:
```
array([2, 3])
```

Array slices are *mutable*: if they are assigned a new value the original array from which the slice was extracted is modified:

In [53]:
```
A[1:3] = [-2,-3]

A
```

Out[53]:
```
array([ 1, -2, -3,  4,  5])
```

We can omit any of the three parameters in `M[lower:upper:step]`:

In [54]:
```
A[::] # lower, upper, step all take the default values
```

Out[54]:
```
array([ 1, -2, -3,  4,  5])
```

In [55]:
```
A[::2] # step is 2, lower and upper defaults to the beginning and end of the
array
```

Out[55]:
```
array([ 1, -3,  5])
```

In [56]:
```
A[:3] # first three elements
```

Out[56]:
```
array([ 1, -2, -3])
```

In [57]:
```
A[3:] # elements from index 3
```

Out[57]:
```
array([4, 5])
```

Negative indices counts from the end of the array (positive index from the begining):

In [58]:
```
A = array([1,2,3,4,5])
```

In [59]:
```
A[-1] # the last element in the array
```

Out[59]:
```
5
```

In [60]:
```
A[-3:] # the last three elements
```

Out[60]:
```
array([3, 4, 5])
```

Index slicing works exactly the same way for multidimensional arrays:

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]
```

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]])
```

Fancy indexing

Fancy indexing is the name for when an array or list is used in-place of an index:

In [64]:
```
row_indices = [1, 2, 3]
A[row_indices]
```

Out[64]:
```
array([[10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [65]:
```
col_indices = [1, 2, -1] # remember, index -1 means the last element
A[row_indices, col_indices]
```

Out[65]:
```
array([11, 22, 34])
```

We can also index masks: If the index mask is an Numpy array of with data type `bool`, then an element is selected (True) or not (False) depending on the value of the index mask at the position each element:

In [66]:
```
B = 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. ])
```

Functions for extracting data from arrays and creating arrays

where

The index mask can be converted to position index using the `where` function

In [72]:
```
indices = where(mask)

indices
```

Out[72]:
```
(array([11, 12, 13, 14]),)
```

In [73]:
```
x[indices] # this indexing is equivalent to the fancy indexing x[mask]
```

Out[73]:
```
array([ 5.5,  6. ,  6.5,  7. ])
```

diag

With the diag function we can also extract the diagonal and subdiagonals of an array:

In [74]:
```
diag(A)
```

Out[74]:
```
array([ 0, 11, 22, 33, 44])
```

In [75]:
```
diag(A, -1)
```

Out[75]:
```
array([10, 21, 32, 43])
```

take

The `take` function is similar to fancy indexing described above:

In [76]:
```
v2 = arange(-3,3)
v2
```

Out[76]:
```
array([-3, -2, -1,  0,  1,  2])
```

In [77]:
```
row_indices = [1, 3, 5]
v2[row_indices] # fancy indexing
```

Out[77]:
```
array([-2,  0,  2])
```

In [78]:
```
v2.take(row_indices)
```

Out[78]:
```
array([-2,  0,  2])
```

But `take` also works on lists and other objects:

In [79]:
```
take([-3, -2, -1,  0,  1,  2], row_indices)
```

Out[79]:
```
array([-2,  0,  2])
```

choose

Constructs and array by picking elements form several arrays:

In [80]:
```
which = [1, 0, 1, 0]
choices = [[-2,-2,-2,-2], [5,5,5,5]]

choose(which, choices)
```

Out[80]:
```
array([ 5, -2,  5, -2])
```

Linear algebra

Vectorizing code is the key to writing efficient numerical calculation with Python/Numpy. That means that as much as possible of a program should be formulated in terms of matrix and vector operations, like matrix-matrix multiplication.

Scalar-array operations

We can use the usual arithmetic operators to multiply, add, subtract, and divide arrays with scalar numbers.

In [81]:
```
v1 = arange(0, 5)
```

In [82]:
```
v1 * 2
```

Out[82]:
```
array([0, 2, 4, 6, 8])
```

In [83]:
```
v1 + 2
```

Out[83]:
```
array([2, 3, 4, 5, 6])
```

In [84]:
```
A * 2, A + 2
```

Out[84]:
```
(array([[ 0,  2,  4,  6,  8],
       [20, 22, 24, 26, 28],
       [40, 42, 44, 46, 48],
       [60, 62, 64, 66, 68],
       [80, 82, 84, 86, 88]]),
 array([[ 2,  3,  4,  5,  6],
       [12, 13, 14, 15, 16],
       [22, 23, 24, 25, 26],
       [32, 33, 34, 35, 36],
       [42, 43, 44, 45, 46]]))
```


Element-wise array-array operations

When we add, subtract, multiply and divide arrays with each other, the default behaviour is **element-wise** operations:

In [85]:
```
A * A # element-wise multiplication
```

Out[85]:
```
array([[   0,    1,    4,    9,   16],
       [ 100,  121,  144,  169,  196],
       [ 400,  441,  484,  529,  576],
       [ 900,  961, 1024, 1089, 1156],
       [1600, 1681, 1764, 1849, 1936]])
```

In [86]:
```
v1 * v1
```

Out[86]:
```
array([ 0,  1,  4,  9, 16])
```

If we multiply arrays with compatible shapes, we get an element-wise multiplication of each row:

In [87]:
```
A.shape, v1.shape
```

Out[87]:
```
((5, 5), (5,))
```

In [88]:
```
A * v1
```

Out[88]:
```
array([[  0,   1,   4,   9,  16],
       [  0,  11,  24,  39,  56],
       [  0,  21,  44,  69,  96],
       [  0,  31,  64,  99, 136],
       [  0,  41,  84, 129, 176]])
```

Matrix algebra

What about matrix mutiplication? There are two ways. We can either use the `dot` functions, which applies a matrix-matrix, matrix-vector, or inner vector multiplication to its two arguments:

In [89]:
```
dot(A, A)
```

Out[89]:
```
array([[ 300,  310,  320,  330,  340],
       [1300, 1360, 1420, 1480, 1540],
       [2300, 2410, 2520, 2630, 2740],
       [3300, 3460, 3620, 3780, 3940],
       [4300, 4510, 4720, 4930, 5140]])
```

In [90]:
```
dot(A, v1)
```

Out[90]:
```
array([ 30, 130, 230, 330, 430])
```

In [91]:
```
dot(v1, v1)
```

Out[91]:
```
30
```

Alternatively, we can cast the array objects to the type `matrix`. This changes the behavior of the standard arithmetic operators `+, -, *` to use matrix algebra.

In [92]:
```
M = matrix(A)
v = matrix(v1).T # make it a column vectorm
```

In [93]:
```
v
```

Out[93]:
```
matrix([[0],
        [1],
        [2],
        [3],
        [4]])
```

In [94]:
```
M*M
```

Out[94]:
```
matrix([[ 300,  310,  320,  330,  340],
        [1300, 1360, 1420, 1480, 1540],
        [2300, 2410, 2520, 2630, 2740],
        [3300, 3460, 3620, 3780, 3940],
        [4300, 4510, 4720, 4930, 5140]])
```

In [95]:
```
M*v
```

Out[95]:
```
matrix([[ 30],
        [130],
        [230],
        [330],
        [430]])
```

In [96]:
```
# inner product
v.T * v
```

Out[96]:
```
matrix([[30]])
```

In [97]:
```
# with matrix objects, standard matrix algebra applies
v + M*v
```

Out[97]:
```
matrix([[ 30],
        [131],
        [232],
        [333],
        [434]])
```

If we try to add, subtract or multiply objects with incomplatible shapes we get an error:

In [98]:
```
v = matrix([1,2,3,4,5,6]).T
```

In [99]:
```
shape(M), shape(v)
```

Out[99]:
```
((5, 5), (6, 1))
```

In [100]:
```
M * v
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-100-995fb48ad0cc> in <module>()
----> 1 M * v

/usr/lib/python2.7/dist-packages/numpy/matrixlib/defmatrix.pyc in __mul__(self, other)
    328         if isinstance(other,(N.ndarray, list, tuple)) :
    329             # This promotes 1-D vectors to row vectors
--> 330             return N.dot(self, asmatrix(other))
    331         if isscalar(other) or not hasattr(other, '__rmul__') :
    332             return N.dot(self, other)

ValueError: objects are not aligned
```

See also the related functions: `inner`, `outer`, `cross`, `kron`, `tensordot`. Try for example `help(kron)`.

Array/Matrix transformations

Above we have used the `.T` to transpose the matrix object `v`. We could also have used the `transpose` function to accomplish the same thing.

Other mathematical functions that transforms matrix objects are:

In [101]:
```
C = matrix([[1j, 2j], [3j, 4j]])
C
```

Out[101]:
```
matrix([[ 0.+1.j,  0.+2.j],
        [ 0.+3.j,  0.+4.j]])
```

In [102]:
```
conjugate(C)
```

Out[102]:
```
matrix([[ 0.-1.j,  0.-2.j],
        [ 0.-3.j,  0.-4.j]])
```

Hermitian conjugate: transpose + conjugate

In [103]:
```
C.H
```

Out[103]:
```
matrix([[ 0.-1.j,  0.-3.j],
        [ 0.-2.j,  0.-4.j]])
```

We can extract the real and imaginary parts of complex-valued arrays using `real` and `imag`:

In [104]:
```
real(C) # same as: C.real
```

Out[104]:
```
matrix([[ 0.,  0.],
        [ 0.,  0.]])
```

In [105]:
```
imag(C) # same as: C.imag
```

Out[105]:
```
matrix([[ 1.,  2.],
        [ 3.,  4.]])
```

Or the complex argument and absolute value

In [106]:
```
angle(C+1) # heads up MATLAB Users, angle is used instead of arg
```

Out[106]:
```
array([[ 0.78539816,  1.10714872],
       [ 1.24904577,  1.32581766]])
```

In [107]:
```
abs(C)
```

Out[107]:
```
matrix([[ 1.,  2.],
        [ 3.,  4.]])
```

Matrix computations

Inverse

In [108]:
```
inv(C) # equivalent to C.I
```

Out[108]:
```
matrix([[ 0.+2.j ,  0.-1.j ],
        [ 0.-1.5j,  0.+0.5j]])
```

In [109]:
```
C.I * C
```

Out[109]:
```
matrix([[  1.00000000e+00+0.j,   4.44089210e-16+0.j],
        [  0.00000000e+00+0.j,   1.00000000e+00+0.j]])
```

Determinant

In [110]:
```
det(C)
```

Out[110]:
```
(2.0000000000000004+0j)
```

In [111]:
```
det(C.I)
```

Out[111]:
```
(0.50000000000000011+0j)
```

Data computations

Often it is useful to store datasets in Numpy arrays. Numpy provides a number of functions to calculate statistics of datasets in arrays.

For example, let's calculate some properties data from the Stockholm temperature dataset used above.

In [112]:
```
# reminder, the tempeature dataset is stored in the data variable:
shape(data)
```

Out[112]:
```
(77431, 7)
```

mean

In [113]:
```
# the temperature data is in column 3
mean(data[:,3])
```

Out[113]:
```
6.1971096847515925
```

standard deviations and variance

In [114]:
```
std(data[:,3]), var(data[:,3])
```

Out[114]: `(8.2822716213405663, 68.596023209663286)`

min and max

In [115]:
```
# lowest daily average temperature
data[:,3].min()
```

Out[115]: `-25.800000000000001`

In [116]:
```
# highest daily average temperature
data[:,3].max()
```

Out[116]: `28.300000000000001`

sum, prod, and trace

In [117]:
```
d = arange(0, 10)
d
```

Out[117]: `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`

In [118]:
```
# sum up all elements
sum(d)
```

Out[118]: `45`

In [119]:
```
# product of all elements
prod(d+1)
```

Out[119]: `3628800`

In [120]:
```
# cummulative sum
cumsum(d)
```

Out[120]: `array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])`

In [121]:
```
# commulative product
cumprod(d+1)
```

Out[121]:
```
array([      1,       2,       6,      24,     120,     720,    5040,
         40320,  362880, 3628800])
```

In [122]:
```
# same as: diag(A).sum()
trace(A)
```

Out[122]: `110`

Computations on subsets of arrays

We can compute with subsets of the data in an array using indexing, fancy indexing, and the other methods of extracting data from an array (described above).

For example, let's go back to the temperature dataset:

In [123]:
```
!head -n 3 stockholm_td_adj.dat
```

```
1800  1  1    -6.1    -6.1    -6.1 1
1800  1  2   -15.4   -15.4   -15.4 1
1800  1  3   -15.0   -15.0   -15.0 1
```

The dataformat is: year, month, day, daily average temperature, low, high, location.

If we are interested in the average temperature only in a particular month, say February, then we can create a index mask and use the select out only the data for that month using:

In [124]:
```
unique(data[:,1]) # the month column takes values from 1 to 12
```

Out[124]:
```
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.,
        12.])
```

In [125]:
```
mask_feb = data[:,1] == 2
```

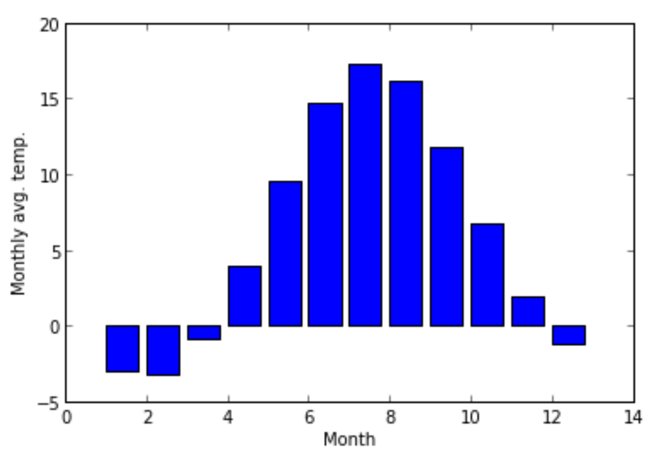
In [126]:
```
# the temperature data is in column 3
mean(data[mask_feb,3])
```

Out[126]:
```
-3.2121095707366085
```

With these tools we have very powerful data processing capabilities at our disposal. For example, to extract the average monthly average temperatures for each month of the year only takes a few lines of code:

In [127]:
```
months = arange(1,13)
monthly_mean = [mean(data[data[:,1] == month, 3]) for month in months]

fig, ax = subplots()
ax.bar(months, monthly_mean)
ax.set_xlabel("Month")
ax.set_ylabel("Monthly avg. temp.");
```



Calculations with higher-dimensional data

When functions such as `min`, `max`, etc., is 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 [128]:
```
m = rand(3,3)
m
```

Out[128]:
```
array([[ 0.69804365,  0.76194991,  0.46126281],
       [ 0.23147764,  0.20841975,  0.01492889],
       [ 0.61958415,  0.85989264,  0.85552498]])
```

In [129]:
```
# global max
m.max()
```

Out[129]:
```
0.85989264081824257
```

In [130]:
```
# max in each column
m.max(axis=0)
```

Out[130]:
```
array([ 0.69804365,  0.85989264,  0.85552498])
```

In [131]:
```
# max in each row
m.max(axis=1)
```

Out[131]:
```
array([ 0.76194991,  0.23147764,  0.85989264])
```

Many other functions and methods in the `array` and `matrix` classes accept the same (optional) `axis` keyword argument.

Reshaping, resizing and stacking arrays

The shape of an Numpy array can be modified without copying the underlaying data, which makes it a fast operation even for large arrays.

In [132]:
```
A
```

Out[132]:
```
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 [133]:
```
n, m = A.shape
```

In [134]:
```
B = A.reshape((1,n*m))
B
```

Out[134]:
```
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 [135]:
```
B[0,0:5] = 5 # modify the array


B
```

Out[135]:
```
array([[ 5,  5,  5,  5,  5, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
         32, 33, 34, 40, 41, 42, 43, 44]])
```

In [136]:
```
A # and the original variable is also changed. B is only a different view of
the same data
```

Out[136]:
```
array([[ 5,  5,  5,  5,  5],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

We can also use the function `flatten` to make a higher-dimensional array into a vector. But this function create a copy of the data.

In [137]:
```
B = A.flatten()

B
```

Out[137]:
```
array([ 5,  5,  5,  5,  5, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
       32, 33, 34, 40, 41, 42, 43, 44])
```

In [138]:
```
B[0:5] = 10

B
```

Out[138]:
```
array([10, 10, 10, 10, 10, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
       32, 33, 34, 40, 41, 42, 43, 44])
```

In [139]:
```
A # now A has not changed, because B's data is a copy of A's, not refering to
the same data
```

Out[139]:
```
array([[ 5,  5,  5,  5,  5],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

Adding a new dimension: newaxis

With `newaxis`, we can insert new dimensions in an array, for example converting a vector to a column or row matrix:

In [140]:
```
v = array([1,2,3])
```

In [141]:
```
shape(v)
```

Out[141]:
```
(3,)
```

In [142]:
```
# make a column matrix of the vector v
v[:, newaxis]
```

Out[142]:
```
array([[1],
       [2],
       [3]])
```

In [143]:
```
# column matrix
v[:,newaxis].shape
```


Out[143]:
```
(3, 1)
```

In [144]:
```
# row matrix
v[newaxis,:].shape
```

Out[144]:
```
(1, 3)
```

Stacking and repeating arrays

Using function `repeat`, `tile`, `vstack`, `hstack`, and `concatenate` we can create larger vectors and matrices from smaller ones:

tile and repeat

In [145]:
```
a = array([[1, 2], [3, 4]])
```

In [146]:
```
# repeat each element 3 times
repeat(a, 3)
```

Out[146]:
```
array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
```

In [147]:
```
# tile the matrix 3 times
tile(a, 3)
```

Out[147]:
```
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4]])
```

concatenate

In [148]:
```
b = array([[5, 6]])
```

In [149]:
```
concatenate((a, b), axis=0)
```

Out[149]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [150]:
```
concatenate((a, b.T), axis=1)
```

Out[150]:
```
array([[1, 2, 5],
       [3, 4, 6]])
```

hstack and vstack

In [151]:
```
vstack((a,b))
```

Out[151]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [152]:
```
hstack((a,b.T))
```

Out[152]:
```
array([[1, 2, 5],
       [3, 4, 6]])
```

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 (techincal term: pass by reference).

In [153]:
```
A = array([[1, 2], [3, 4]])

A
```

Out[153]:
```
array([[1, 2],
       [3, 4]])
```

In [154]:
```
# now B is referring to the same array data as A
B = A
```

In [155]:
```
# changing B affects A
B[0,0] = 10

B
```

Out[155]:
```
array([[10,  2],
       [ 3,  4]])
```

In [156]:
```
A
```

Out[156]:
```
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 [1 2]
1
2
row [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 [162]:
```
for row_idx, row in enumerate(M):
    print "row_idx", row_idx, "row", row

    for col_idx, element in enumerate(row):
        print "col_idx", col_idx, "element", element

        # update the matrix M: square each element
        M[row_idx, col_idx] = element ** 2
```

```
row_idx 0 row [1 2]
col_idx 0 element 1
col_idx 1 element 2
row_idx 1 row [3 4]
col_idx 0 element 3
col_idx 1 element 4
```

In [163]:
```
# each element in M are 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.

In [164]:
```
def Theta(x):
    """
    Scalar implemenation of the Heaviside step function.
    """
    if x >= 0:
        return 1
    else:
        return 0
```

In [165]:
```
Theta(array([-3,-2,-1,0,1,2,3]))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-165-6658efdd2f22> in <module>()
----> 1 Theta(array([-3,-2,-1,0,1,2,3]))

<ipython-input-164-9a0cb13d93d4> in Theta(x)
      3     Scalar implemenation of the Heaviside step function.
      4     """
----> 5     if x >= 0:
      6         return 1
      7     else:

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or
```

OK, that didn't work because we didn't write the `Theta` function so that it can handle with vector input...

To get a vectorized version of Theta we can use the Numpy function `vectorize`. In many cases it can automatically vectorize a function:

In [166]:
```
Theta_vec = vectorize(Theta)
```

In [167]:
```
Theta_vec(array([-3,-2,-1,0,1,2,3]))
```

Out[167]:
```
array([0, 0, 0, 1, 1, 1, 1])
```

We can also implement the function to accept vector input from the beginning (requires more effort but might give better performance):

In [168]:
```
def Theta(x):
    """
    Vector-aware implemenation of the Heaviside step function.
    """
    return 1 * (x >= 0)
```

In [169]:
```
Theta(array([-3,-2,-1,0,1,2,3]))
```

Out[169]:
```
array([0, 0, 0, 1, 1, 1, 1])
```

In [170]:
```
# still works for scalars as well
Theta(-1.2), Theta(2.6)
```

Out[170]:
```
(0, 1)
```

Using arrays in conditions

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

In [171]:
```
M
```

Out[171]:
```
array([[ 1,  4],
       [ 9, 16]])
```

In [172]:
```
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 [173]:
```
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://scipy.org/Tentative_NumPy_Tutorial
- http://scipy.org/NumPy_for_Matlab_Users - A Numpy guide for MATLAB users.