Numpy - arreglos de datos multidimensionales

Versión original en inglés de J.R. Johansson (http://jrjohansson.github.io/) (robert@riken.jp).

Traducido/Adaptado por G.F. Rubilar (http://google.com/+GuillermoRubilar).

La última versión de estos Notebooks (http://ipython.org/notebook.html) está disponible en http://github.com/gfrubi/clases-python-cientifico (http://github.com/gfrubi/clases-python-cientifico).

La última versión del original (en inglés) está disponible en http://github.com/jrjohansson/scientific-python-lectures (http://github.com/jrjohansson/scientific-python-lectures). Los otros notebooks de esta serie están listados en http://jrjohansson.github.com (http://jrjohansson.github.com).

Introducción

El paquete (módulo) Numpy es usado en casi todos los cálculos numéricos usando Python. Es un paquete que provee a Python de estructuras de datos vectoriales, matriciales, y de rango mayor, de alto rendimiento. Está implementado en C y Fortran, de modo que cuando los cálculos son vectorizados (formulados con vectores y matrices), el rendimiento es muy bueno.

Para usar Numpy necesitamos importar el módulo usando, por ejemplo:

In [1]:
```
from numpy import *
```

En el paquete `numpy` la terminología usada para vectores, matrices y conjuntos de datos de dimensión mayor es la de un *arreglo*.

Creando arreglos de `numpy`

Existen varias formas para inicializar nuevos arreglos de `numpy`, por ejemplo desde

- Listas o tuplas Python
- Usando funciones dedicadas a generar arreglos `numpy`, como `arange`, `linspace`, etc.
- Leyendo datos desde archivos

Desde listas

Por ejemplo, para crear nuevos arreglos de matrices y vectores desde listas Python podemos usar la función `numpy.array`.

In [2]:
```
# un vector: el argumento de la función array es una lista de Python
v = array([1,2,3,4])

v
```

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

In [3]:
```
# una matriz: el argumento de la función array es una lista anidada de Python
M = array([[1, 2], [3, 4]])

M
```

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

Los objetos `v` y `M` son ambos del tipo `ndarray` que provee el módulo `numpy`.

In [4]:
```
type(v), type(M)
```

Out[4]:
```
(numpy.ndarray, numpy.ndarray)
```

La diferencia entre los arreglos `v` y `M` es sólo su forma. Podemos obtener información de la forma de un arreglo usando la propiedad `ndarray.shape`

In [5]:
```
v.shape
```

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

In [6]:
```
M.shape
```

Out[6]:
```
(2, 2)
```

El número de elementos de un arreglo puede obtenerse usando la propiedad `ndarray.size`:

In [7]:
```
M.size
```

Out[7]:
```
4
```

Equivalentemente, podemos usar las funciones `numpy.shape` y `numpy.size`

In [8]:
```
shape(M)
```

Out[8]:
```
(2, 2)
```

In [9]:
```
size(M)
```

Out[9]:
```
4
```

Hasta el momento el arreglo `numpy.ndarray` luce como una lista Python (anidada). Entonces, ¿por qué simplemente no usar listas para hacer cálculos en lugar de crear un tipo nuevo de arreglo?

Existen varias razones:

- Las listas Python son muy generales. Ellas pueden contener cualquier tipo de objeto. Sus tipos son asignados dinámicamente. Ellas no permiten usar funciones matemáticas tales como la multiplicación de matrices, el producto escalar, etc. El implementar tales funciones para las listas Python no sería muy eficiente debido a la asignación dinámica de su tipo.
- Los arreglos Numpy tienen tipo **estático** y **homogéneo**. El tipo de elementos es determinado cuando se crea el arreglo.
- Los arreglos Numpy son eficientes en el uso de memoria.
- Debido a su tipo estático, se pueden desarrollar implementaciones rápidas de funciones matemáticas tales como la multiplicación y la suma de arreglos `numpy` usando lenguajes compilados (se usan C y Fortran).

Usando la propiedad `dtype` (tipo de dato) de un `ndarray`, podemos ver qué tipo de dato contiene un arreglo:

In [10]:
```
M.dtype
```

Out[10]:
```
dtype('int64')
```

Se obtiene un error si intentamos asignar un valor de un tipo equivocado a un elemento de un arreglo numpy:

In [11]:
```
M[0,0] = "hola"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-11-fc4abc9a80ec> in <module>()
----> 1 M[0,0] = "hola"

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

Si lo deseamos, podemos definir explícitamente el tipo de datos de un arreglo cuando lo creamos, usando el argumento `dtype`:

In [12]:
```
M = array([[1, 2], [3, 4]], dtype=complex)

M
```

Out[12]:
```
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Algunos tipos comunes que pueden ser usados con `dtype` son: `int`, `float`, `complex`, `bool`, `object`, etc.

Podemos también definir explícitamente el número de bit de los tipos de datos, por ejemplo: `int64`, `int16`, `float64`, `complex64`.

Usando funciones que generan arreglos

En el caso de arreglos más grandes no es práctico inicializar los datos manualmente, usando listas Python explícitas. En su lugar, podemos usar una de las muchas funciones en `numpy` que generan arreglos de diferentes formas. Algunas de los más comunes son:

arange

In [13]:
```
# crea un arreglo con valores en un rango

x = arange(0,10,1) # argumentos: desde, hasta (no se incluye!), paso

x
```

Out[13]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [14]:
```
x = arange(-1,1,0.1)

x
```

Out[14]:
```
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 y logspace

In [15]:
```
# usando linspace, ambos puntos finales SON incluidos. Formato: (desde, hasta, número de elementos)
linspace(0,10,25)
```

Out[15]:
```
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 [16]:
```
# logspace también incluye el punto final. Por defecto base=10
logspace(0,10,11, base=e) # produce e elevado a cada valor en linspace(0, 10,
 11), e.d.[e**0, e**1,...,e**10]
```

Out[16]:
```
array([  1.00000000e+00,   2.71828183e+00,   7.38905610e+00,
         2.00855369e+01,   5.45981500e+01,   1.48413159e+02,
         4.03428793e+02,   1.09663316e+03,   2.98095799e+03,
         8.10308393e+03,   2.20264658e+04])
```

mgrid

In [17]:
```
x, y = mgrid[0:5, 0:5] # similar a meshgrid en MATLAB
```

In [18]:
```
x
```

Out[18]:
```
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 [19]:
```
y
```

Out[19]:
```
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]])
```

Datos aleatorios

In [20]:
```
from numpy import random
```

In [21]:
```
# números aleatorios con distribución de probabilidad uniforme en [0,1]
random.rand(10)
```

Out[21]:
```
array([ 0.24505667,  0.73920374,  0.43203417,  0.20683999,  0.6392161 ,
        0.04194899,  0.13378878,  0.52801467,  0.69030967,  0.52480182])
```

In [22]:
```
# números aleatorios con distribución normal (gaussiana de media 0 y varianza
 1).
random.randn(3,3)
```

Out[22]:
```
array([[-0.98227309,  0.461191  ,  0.72575093],
       [-0.60268008, -2.27446301, -0.12867387],
       [ 0.19399359,  1.51916762,  1.15105401]])
```

diag

In [23]:
```
# una matriz diagonal
diag([1,2,3])
```

Out[23]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

In [24]:
```
# diagonal desplazada desde la diagonal principal
diag([1,2,3], k=1)
```

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

ceros y unos

In [25]:
```
zeros((3,3))
```

Out[25]:
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

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

Out[26]:
```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Entrada/Salida desde/hasta archivos

Valores separados por coma (Comma-separated values, CSV)

Un formato muy común para archivos de datos es el de valores separados por comas, o formatos relacionados, como por ejemplo TSV (tab-separated values, valores separados por tabs). Para leer datos desde tales archivos a un arreglo Numpy podemos usar la función `numpy.genfromtxt` . Por ejemplo,

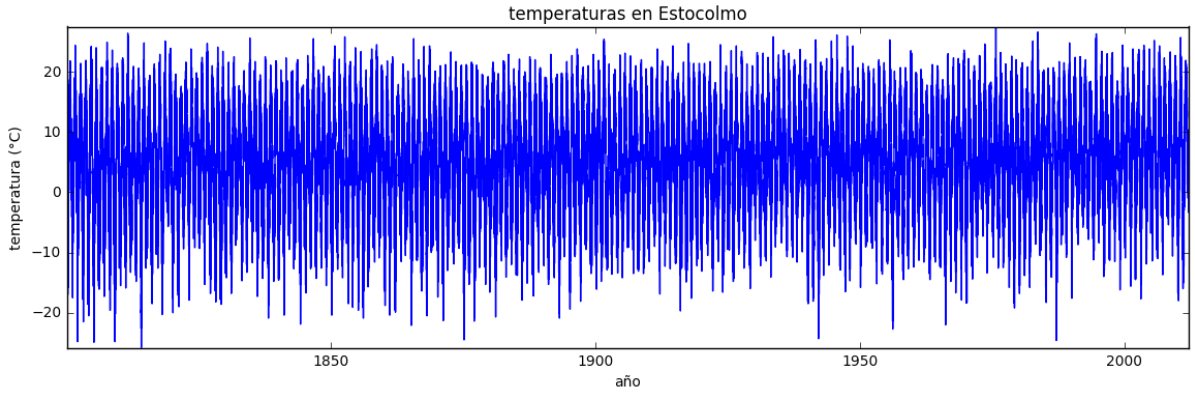
In [27]:
```
data = genfromtxt('stockholm_td_adj.dat')  # asigna los datos desde el archivo
'stockholm_td_adj.dat' al arreglo data
```

In [28]:
```
data.shape
```

Out[28]:
```
(77431, 7)
```

In [29]:
```
# ¿qué hace esta línea?. La respuesta en la clase 04
%matplotlib inline
from matplotlib.pyplot import *
```

In [30]:
```
figure(figsize=(14,4))
plot(data[:,0]+data[:,1]/12.0+data[:,2]/365, data[:,5])
axis('tight')
title('temperaturas en Estocolmo')
xlabel(u'año')
ylabel(u'temperatura (°C)');
```



Usando `numpy.savetxt` podemos almacenar un arreglo Numpy a un archivo en formato CSV:

In [31]:
```
M = random.rand(3,3)

M
```

Out[31]:
```
array([[ 0.44373475,  0.09421772,  0.22699008],
       [ 0.56588745,  0.33473943,  0.57484992],
       [ 0.36831004,  0.77844943,  0.47212619]])
```

In [32]:
```
savetxt("matriz-aleatoria.csv", M)
```

In [33]:
```
savetxt("matriz-aleatoria.csv", M, fmt='%.5f') # fmt especifica el formato (5
 decimales)
```

El formato de archivo nativo de Numpy

Este formato es útil cuando se almacenan arreglos de datos y luego se leen nuevamente con `numpy` . Use las funciones `numpy.save` y `numpy.load` :

In [34]:
```
save("matriz-aleatoria.npy", M)
```

In [35]:
```
load("matriz-aleatoria.npy")
```

Out[35]:
```
array([[ 0.44373475,  0.09421772,  0.22699008],
       [ 0.56588745,  0.33473943,  0.57484992],
       [ 0.36831004,  0.77844943,  0.47212619]])
```

Más propiedades de los arreglos Numpy

In [36]:
```
M.itemsize # los bits de cada elemento
```

Out[36]: 8

In [37]:
```
M.nbytes # número de bytes
```

Out[37]: 72

In [38]:
```
M.ndim # número de dimensiones
```

Out[38]: 2

Manipulando arreglos

Indexando

Podemos indexar elementos en un arreglo usando paréntesis cuadrados e índices, tal como con las listas:

In [39]:
```
# v es un vector, tiene por lo tanto sólo una dimensión, y requiere un índice
v[0]
```

Out[39]: 1

In [40]:
```
# M es una matriz, es decir un arreglo bidimensional, requiere dos índices
M[1,1]
```

Out[40]: 0.33473943047565435

Si omitimos un índice de un arreglo multidimensional `Numpy` entrega la fila completa (o, en general, al arreglo de dimensión N-1 correspondiente)

In [41]:
```
M
```

Out[41]:
```
array([[ 0.44373475,  0.09421772,  0.22699008],
       [ 0.56588745,  0.33473943,  0.57484992],
       [ 0.36831004,  0.77844943,  0.47212619]])
```

In [42]:
```
M[1]
```

Out[42]:
```
array([ 0.56588745,  0.33473943,  0.57484992])
```

Puede obtenerse lo mismo usando `:` en el lugar de un índice:

```
In [43]: M[1,:] # fila 1
```

```
Out[43]: array([ 0.56588745,  0.33473943,  0.57484992])
```

```
In [44]: M[:,1] # columna 1
```

```
Out[44]: array([ 0.09421772,  0.33473943,  0.77844943])
```

Podemos asignar nuevos valores a los elementos de un arreglo usando el indexado:

```
In [45]: M[0,0] = 1
```

```
In [46]: M
```

```
Out[46]: array([[ 1.        ,  0.09421772,  0.22699008],
                [ 0.56588745,  0.33473943,  0.57484992],
                [ 0.36831004,  0.77844943,  0.47212619]])
```

```
In [47]: # también funciona para filas y columnas completas
         M[1,:] = 0
         M[:,2] = -1
```

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

```
Out[48]: array([[ 1.        ,  0.09421772, -1.        ],
                [ 0.        ,  0.        , -1.        ],
                [ 0.36831004,  0.77844943, -1.        ]])
```

Corte de índices

Corte (slicing) de índices es el nombre para la sintaxis `M[desde:hasta:paso]` que extrae una parte de un arreglo:

```
In [49]: A = array([1,2,3,4,5])
         A
```

```
Out[49]: array([1, 2, 3, 4, 5])
```

```
In [50]: A[1:3]
```

```
Out[50]: array([2, 3])
```

Los cortes de índices son *mutables*: si se les asigna un nuevo valor el arreglo original es modificado:

In [51]:
```
A[1:3] = [-2,-3]

A
```

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

Podemos omitir cualquiera de los tres parámetros en `M[desde:hasta:paso]` :

In [52]:
```
A[::] # desde, hasta y paso asumen los valores por defecto
```

Out[52]:
```
array([ 1, -2, -3,  4,  5])
```

In [53]:
```
A[::2] # el paso es 2, desde y hasta se asumen desde el comienzo hasta el fin
 del arreglo
```

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

In [54]:
```
A[:3] # primeros tres elementos
```

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

In [55]:
```
A[3:] # elementos desde el índice 3
```

Out[55]:
```
array([4, 5])
```

Los índices negativos se cuentan desde el fin del arreglo (los índices positivos desde el comienzo):

In [56]:
```
A = array([1,2,3,4,5])
```

In [57]:
```
A[-1] # el último elemento del arreglo
```

Out[57]:
```
5
```

In [58]:
```
A[-3:] # los últimos 3 elementos
```

Out[58]:
```
array([3, 4, 5])
```

El corte de índices funciona exactamente del mismo modo para arreglos multidimensionales:

In [59]:
```
A = array([[n+m*10 for n in range(5)] for m in range(5)])

A
```

Out[59]:
```
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 [60]:
```
# un bloque parte del arreglo original
A[1:4, 1:4]
```

Out[60]:
```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

In [61]:
```
# elemento por medio
A[::2, ::2]
```

Out[61]:
```
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])
```

Indexado Fancy

Se llama *indexado fancy* cuando un arreglo o una lista es usado en lugar de un índice:

In [62]:
```
indices_fila = [1, 2, 3]
A[indices_fila]
```

Out[62]:
```
array([[10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [63]:
```
indices_col = [1, 2, -1] # recuerde que el índice -1 corresponde al último ele
mento
A[indices_fila, indices_col]
```

Out[63]:
```
array([11, 22, 34])
```

Podemos también usar **máscaras de índices**: Si la máscara de índice es un arreglo `Numpy` con tipo de dato booleano (`bool`), entonces un elemento es seleccionado (True) o no (False) dependiendo del valor de la máscara de índice en la posición de cada elemento:

In [64]:
```
B = arange(5)
B
```

Out[64]:
```
array([0, 1, 2, 3, 4])
```

In [65]:
```
masc_fila = array([True, False, True, False, False])
B[masc_fila]
```

Out[65]:
```
array([0, 2])
```

In [66]:
```
# lo mismo
masc_fila = array([1,0,1,0,0], dtype=bool)
B[masc_fila]
```

Out[66]:
```
array([0, 2])
```

Esta característica es muy útil para seleccionar en forma condicional elementos de un arreglo, usando por ejemplo los operadores de comparación:

In [67]:
```
x = arange(0, 10, 0.5)
x
```

Out[67]:
```
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 [68]:
```
masc = (5 < x) * (x < 7.5) # * Entre 2 booleanos actúa como AND, entregando 0
 (False) y 1 (True)

masc
```

Out[68]:
```
array([False, False, False, False, False, False, False, False, False,
       False, False,  True,  True,  True,  True, False, False, False,
       False, False], dtype=bool)
```

In [69]:
```
x[masc]
```

Out[69]:
```
array([ 5.5,  6. ,  6.5,  7. ])
```

Funciones para extraer información desde arreglos y para crear nuevos arreglos

where

Las máscaras de índices pueden ser convertidas en posiciones de índices usando la función `where` ("dónde"):

In [70]:
```
indices = where(masc)

indices
```

Out[70]:
```
(array([11, 12, 13, 14]),)
```

In [71]:
```
x[indices] # este indexado es equivalente al indexado fancy x[masc]
```

Out[71]:
```
array([ 5.5,  6. ,  6.5,  7. ])
```

diag

Con la función diag podemos extraer la diagonal y las subdiagonales de un arreglo:

In [72]:
```
diag(A)
```

Out[72]:
```
array([ 0, 11, 22, 33, 44])
```

In [73]:
```
diag(A, -1)
```

Out[73]:
```
array([10, 21, 32, 43])
```

take

La función `take` es similar al indexado fancy descrito anteriormente:

In [74]:
```
v2 = arange(-3,3)
v2
```

Out[74]:
```
array([-3, -2, -1,  0,  1,  2])
```

In [75]:
```
indices_fila = [1, 3, 5]
v2[indices_fila] # indexado fancy
```

Out[75]:
```
array([-2,  0,  2])
```

In [76]:
```
v2.take(indices_fila)
```

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

Pero la función `take` también funciona sobre listas y otros objetos:

In [77]:
```
take([-3, -2, -1,  0,  1,  2], indices_fila)
```

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

choose

Construye un arreglo tomando elementos desde varios arreglos:

In [78]:
```
cuales = [1, 0, 1, 0]
posibilidades = [[1,2,3,4], [5,6,7,8]]

choose(cuales, posibilidades)
```

Out[78]:
```
array([5, 2, 7, 4])
```

Álgebra lineal

El vectorizar el código es la clave para realizar cálculos numéricos eficientes usando Python/Numpy. Esto significa que la mayor parte de un programa debería ser formulado en términos de operaciones con matrices y vectores, como por ejemplo la multiplicación de matrices.

Operaciones escalar-arreglo

Podemos usar los operadores aritméticos usuales para multiplicar, sumar, restar, y dividir arreglos por números (escalares):

In [79]:
```
v1 = arange(0,5)
```

In [80]:
```
2*v1
```

Out[80]:
```
array([0, 2, 4, 6, 8])
```

In [81]:
```
v1 + 2
```

Out[81]:
```
array([2, 3, 4, 5, 6])
```

In [82]:
```
3*A
```

Out[82]:
```
array([[  0,   3,   6,   9,  12],
       [ 30,  33,  36,  39,  42],
       [ 60,  63,  66,  69,  72],
       [ 90,  93,  96,  99, 102],
       [120, 123, 126, 129, 132]])
```

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

Out[83]:
```
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]])
```

Operaciones elemento a elemento entre arreglos

Cuando sumamos, sustraemos, multiplicamos y dividimos dos arreglos, el comportamiento por defecto es operar *elemento a elemento*:

In [84]:
```
A * A # multiplicación elemento a elemento
```

Out[84]:
```
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 [85]:
```
A**2 # mismo resultado que A*A
```

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])
```

Si multiplicamos arreglos con formas compatibles, obtenemos una multiplicación elemento a elemento de cada fila:

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]])
```

Álgebra matricial

¿Y la multiplicación de matrices? Podemos realizarla de dos formas. Podemos usar la función `dot`, que aplica una multiplicación matriz-matriz, matriz-vector o un producto interno entre vectores a sus dos argumentos:

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
```

Alternativamente, podemos transformar el arreglo al tipo `matrix` . Esto cambia el comportamiento de los operadores aritméticos estándar `+, -, *` al de álgebra de matrices.

In [92]:
```
M = matrix(A)
v = matrix(v1).T # aplica la traspuesta, convirtiéndolo en vector columna
```

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]:
```
# producto interior
v.T * v
```

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

In [97]:
```
# con objetos matriciales, el álgebra matricial estándar es usada
v + M*v
```

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

Si intentamos sumar, restar, o multiplicar objetos con formas incompatibles, obtendremos un 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

/opt/anaconda/lib/python2.7/site-packages/numpy/matrixlib/defmatrix.pyc in __
mul__(self, other)
    341         if isinstance(other, (N.ndarray, list, tuple)) :
    342             # This promotes 1-D vectors to row vectors
--> 343             return N.dot(self, asmatrix(other))
    344         if isscalar(other) or not hasattr(other, '__rmul__') :
    345             return N.dot(self, other)

ValueError: shapes (5,5) and (6,1) not aligned: 5 (dim 1) != 6 (dim 0)
```

Vea también las funciones relacionadas: `inner` , `outer` , `cross` , `kron` , `tensordot` . Por ejemplo, introduzca `help(kron)` .

Transformaciones de arreglos/matrices

Antes hemos usado `.T` para transponer un vector `v` . Podemos también usar la función `transpose` para conseguir el mismo resultado.

Otras funciones matemáticas que transforman objetos matriciales son:

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]])
```

Hermítico conjugado: transpuesta + conjugado

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

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

Podemos extraer las partes reales e imaginarias de un arreglo con elementos complejos usando `real` y `imag` :

In [104]:
```
real(C) # lo mismo que: C.real
```

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

In [105]:
```
imag(C) # lo mismo que: C.imag
```

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

Podemos también extraer el módulo y el argumento complejo

In [106]:
```
angle(C+1) # Atención usuarios de MATLAB, se usa angle en lugar de arg
```

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

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

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

Cálculos con matrices

Inversa

In [108]:
```
linalg.inv(C) # equivalente a 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,   0.00000000e+00+0.j],
        [  1.11022302e-16+0.j,   1.00000000e+00+0.j]])
```

Determinante

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

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

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

Out[111]:
```
(0.49999999999999967+0j)
```

Cálculos con datos

A menudo es útil almacenar datos en arreglos Numpy. Numpy provee funciones para realizar cálculos estadísticos de los datos en un arreglo.

Por ejemplo, calculemos algunas propiedades de los datos de la temperatura de Estocolmo que discutimos anteriormente.

In [112]:
```
# recuerde, los datos de la temperatura están almacenados en la variable data
shape(data)
```

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

mean

In [113]:
```
# la temperatura está almacenada en la columna 3
mean(data[:,3])
```

Out[113]:
```
6.1971096847515854
```

La temperatura diaria promedio en Estocolmo en los últimos 200 años ha sido aproximadamente 6.2 C.

Desviación estándar y varianza

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

Out[114]:
```
(8.2822716213405734, 68.596023209663414)
```

min y max

In [115]:
```
# valor mínimo del promedio diario de temperatura. Lo mismo que data[:,3].min()
min(data[:,3])
```

Out[115]:
```
-25.800000000000001
```

In [116]:
```
# valor máximo del promedio diario de temperatura. Lo mismo que data[:,3].max()
max(data[:,3])
```

Out[116]:
```
28.300000000000001
```

sum, prod y trace

In [117]:
```
d = arange(1,11)**2
d
```

Out[117]:
```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

In [118]:
```
# suma todos los elementos
sum(d)
```

Out[118]:
```
385
```

In [119]:
```
# multiplica todos los elementos
prod(d)
```

Out[119]:
```
13168189440000
```

In [120]:
```
# suma acumulativa
cumsum(d)
```

Out[120]:
```
array([  1,   5,  14,  30,  55,  91, 140, 204, 285, 385])
```

In [121]:
```
# producto acumulativo
cumprod(d)
```

Out[121]:
```
array([             1,              4,             36,            576,
                14400,         518400,       25401600,     1625702400,
            131681894400, 13168189440000])
```

In [122]:
```
# lo mismo que: diag(A).sum()
trace(A)
```

Out[122]:
```
110
```

Cálculos con subconjuntos de un arreglo

Podemos calcular usando subconjuntos de los datos de un arreglo usando el indexado, indexado fancy, y los otros métodos para extraer datos desde un arreglo (descrito más arriba).

Por ejemplo, consideremos nuevamente los datos de temperatura de Estocolmo:

El formato de los datos es: año, mes, día, temperatura promedio diaria, mínima, máxima, lugar.

Si estamos interesados sólo en la temperatura promedio de un mes particular, Febrero por ejemplo, podemos crear una máscara de índice y seleccionar sólo los datos de ese mes usando:

In [123]:
```
unique(data[:,1]) # la columna mes asume valores entre 1 y 12, `unique` lista
 los valores distintos
```

Out[123]:
```
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.,
        12.])
```

In [124]:
```
masc_feb = (data[:,1] == 2) # los paréntesis () son opcionales
```

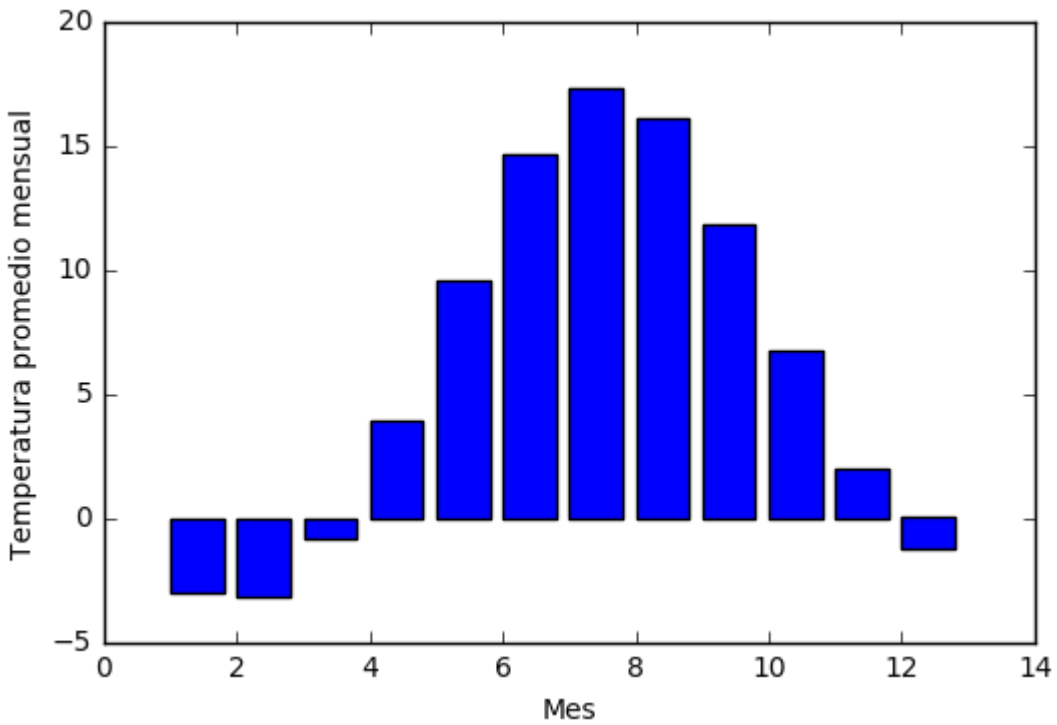
In [125]:
```
# los datos de temperatura están en la columna 3
mean(data[masc_feb,3])
```

Out[125]:
```
-3.2121095707365961
```

Estas funciones ponen a nuestra disposición herramientas muy poderosas para procesar datos. Por ejemplo, para extraer las temperaturas promedio mensuales para cada mes del año sólo necesitamos unas pocas líneas de código:

In [126]:
```
meses = arange(1,13)
media_mensual = [mean(data[data[:,1] == mes, 3]) for mes in meses]

bar(meses, media_mensual)
xlabel("Mes")
ylabel("Temperatura promedio mensual");
```



Cálculos con datos multidimensionales

Cuando se aplican funciones como `min`, `max`, etc., a arreglos multidimensionales, a veces se desea aplicarlas al arreglo completo, y en otras ocasiones sólo por filas o columnas. Podemos especificar cómo se comportan estas funciones usando el argumento `axis` (eje):

In [127]:
```
m = random.rand(3,3)
m
```

Out[127]:
```
array([[ 0.69848249,  0.95112965,  0.95984403],
       [ 0.34837068,  0.5614132 ,  0.69146514],
       [ 0.11750508,  0.45918034,  0.97143975]])
```

In [128]:
```
# máximo global
m.max()
```

Out[128]:
```
0.97143975235812652
```

In [129]:
```
# máximo en cada columna
m.max(axis=0)
```

Out[129]:
```
array([ 0.69848249,  0.95112965,  0.97143975])
```

In [130]:
```
# máximo en cada fila
m.max(axis=1)
```

Out[130]:
```
array([ 0.95984403,  0.69146514,  0.97143975])
```

Muchas otras funciones y métodos de las clases `array` y `matrix` aceptan el argumento (opcional) `axis`.

Cambiando la forma, redimensionando y apilando arreglos

La forma de un arreglo Numpy puede ser modificada sin copiar las datos involucrados, lo que hace que esta operación sea rápida, incluso con arreglos grandes.

In [131]:
```
A
```

Out[131]:
```
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 [132]:
```
n, m = A.shape
```

In [133]:
```
B = A.reshape((1,n*m)) # convierte el arreglo a uno de dimensiones (1,n*m)
B
```

Out[133]:
```
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 [134]:
```
B[0,0:5] = 5 # modifica el arreglo

B
```

Out[134]:
```
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 [135]:
```
A # la variable original es también cambiada. B sólo constituye una forma dist
inta de ver los mismos datos
```

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]])
```

Podemos también usar la función `flatten` para transformar un arreglo multidimensional a un vector. A diferencia de `reshape` esta función crea una copia de los datos.

In [136]:
```
B = A.flatten()

B
```

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])
```

In [137]:
```
B[0:5] = 10

B
```

Out[137]:
```
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 [138]:
```
A # ahora A no ha cambiado, ya que los datos de B han sido duplicados desde A
```

Out[138]:
```
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]])
```

Agregando una dimensión adicional: `newaxis`

Con `newaxis`, podemos insertar una nueva dimension en un arreglo, por ejemplo, para convertir un vector en la fila o columna de una matriz:

In [139]:
```
v = array([1,2,3])
```

In [140]:
```
shape(v)
```

Out[140]:
```
(3,)
```

In [141]:
```
# crea una matriz con el vector v como su columna
v[:, newaxis]
```

Out[141]:
```
array([[1],
       [2],
       [3]])
```

In [142]:
```
# matriz columna
v[:,newaxis].shape
```

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

In [143]:
```
# matriz fila
v[newaxis,:]
```

Out[143]:
```
array([[1, 2, 3]])
```

In [144]:
```
v[newaxis,:].shape
```

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

Apilando y repitiendo arreglos

Podemos crear vectores y matrices más grandes a partir de otras más pequeñas usando las funciones `repeat` (repetir), `tile` (teselar, "embaldosar"), `vstack` (apilar verticalmente), `hstack` (apilar horizontalmente), y `concatenate` (concatenar):

`tile` y `repeat`

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

Out[145]:
```
array([[1, 2],
       [3, 4]])
```

In [146]:
```
# repite cada elemento 3 veces
repeat(a, 3)
```

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

In [147]:
```
# repite la matriz 3 veces
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 y 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 y "deep copy"

Para alcanzar un alto desempeño, las asignaciones en Python usualmente no copian los objetos involucrados. Esto es importante cuando se pasan objetos a funciones, para así evitar uso excesivo de memoria copiando cuando no es necesario (término técnico: paso por referencia)

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

A
```

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

In [154]:
```
# ahora B apunta al mismo arreglo que A
B = A
```

In [155]:
```
# cambiar B afecta a A
B[0,0] = 10

B
```

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

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

Out[156]:
```
array([[10,  2],
       [ 3,  4]])
```

Si queremos evitar este comportamiento, para así obtener un nuevo objecto `B` copiado desde `A`, pero totalmente independiente de `A`, necesitamos realizar una "copia profunda" ("deep copy") usando la función `copy`:

In [157]:
```
B = copy(A)
```

In [158]:
```
# ahora A no cambia si modificamos B
B[0,0] = -5

B
```

Out[158]:
```
array([[-5,  2],
       [ 3,  4]])
```

In [159]:
```
A
```

Out[159]:
```
array([[10,  2],
       [ 3,  4]])
```

Iterando sobre elementos de un arreglo

Generalmente, deseamos evitar iterar sobre los elementos de un arreglo donde sea posible (a cualquier precio!). La razón es que en un lenguaje interpretado como Python (o MATLAB), las iteraciones son realmente lentas comparadas con las operaciones vectorizadas.

Sin embargo, algunas veces es ineludible. En tales casos el bucle Python `for` es la forma más conveniente para iterar sobre un arreglo:

In [160]:
```
v = array([1,2,3,4])

for elemento in v:
    print elemento
```

```
1
2
3
4
```

In [161]:
```
M = array([[1,2], [3,4]])

for fila in M:
    print "fila", fila

    for elemento in fila:
        print elemento
```

```
fila [1 2]
1
2
fila [3 4]
3
4
```

Cuando necesitamos iterar sobre cada elemento de un arreglo y modificar sus valores, es conveniendo usar la función `enumerate` para obtener tanto el elemento como su índice en el bucle `for` :

In [162]:
```
for ind_fila, fila in enumerate(M):
    print "ind_fila", ind_fila, "fila", fila

    for ind_colu, elemento in enumerate(fila):
        print "ind_colu", ind_colu, "elemento", elemento

        # actualiza la matriz: eleva al cuadrado cada elemento
        M[ind_fila, ind_colu] = elemento**2
```

```
ind_fila 0 fila [1 2]
ind_colu 0 elemento 1
ind_colu 1 elemento 2
ind_fila 1 fila [3 4]
ind_colu 0 elemento 3
ind_colu 1 elemento 4
```

In [163]:
```
# cada elemento en M está ahora al cuadrado
M
```

Out[163]:
```
array([[ 1,  4],
       [ 9, 16]])
```

Vectorizando funciones

Como se ha mencionado en varias ocasiones, para obtener un buen rendimiento deberíamos tratar de evitar realizar bucles sobre los elementos de nuestros vectores y matrices, y en su lugar usar algoritmos vectorizados. El primer paso para convertir un algoritmo escalar a uno vectorizado es asegurarnos de que las funciones que escribamos funcionen con argumentos vectoriales.

In [164]:
```
def Theta(x):
    """
    implementación escalar de la función escalón de Heaviside.
    """
    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-d6359217b50a> in Theta(x)
      3     implementación escalar de la función escalón de Heaviside.
      4     """
----> 5     if x >= 0:
      6         return 1
      7     else:

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

Ok, eso no funcionó porque no definimos la función `Theta` de modo que pueda manejar argumentos vectoriales. Para obtener una *versión vectorizada* de Theta podemos usar la función `vectorize` de Numpy. En muchos casos, puede vectorizar automáticamente una función:

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])
```

Podemos también implementar la función de modo que desde el comienzo acepte un argumento vectorial (esto requiere más esfuerzo, para mejorar el rendimiento):

In [168]:
```
def Theta(x):
    """
    Implementación preparada para vectores de la función escalón de Heaviside.
    """
    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]:
```
# y también funciona para escalares!
Theta(-1.2), Theta(2.6)
```

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

Usando arreglos en sentencias condicionales

Cuando se usan arreglos en sentencias condicionales, por ejemplo en sentencias `if` y otras expresiones booleanas, necesitamos usar `any` o bien `all`, que requiere que todos los elementos de un arreglo se evalúen con `True`:

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

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

In [172]:
```
if any(M > 5): # equivalente a (M > 5).any():
    print "al menos un elemento de M es mayor que 5"
else:
    print "ningún elemento de M es mayor que 5"
```

```
al menos un elemento de M es mayor que 5
```

In [173]:
```
if all(M > 5): # equivalente a (M > 5).all():
    print "todos los elementos de M son mayores que 5"
else:
    print "no todos los elementos de M son mayores que 5"
```

```
no todos los elementos de M son mayores que 5
```

Conversión de tipo

Como los arreglos de Numpy son *de tipo estático*, el tipo de un arreglo no puede ser cambiado luego de que es creado. Sin embargo, podemos convertir explícitamente un arreglo de un tipo a otro usando las funciones `astype` (ver también las funciones similares `asarray`). Esto crea un nuevo arreglo de un nuevo tipo:

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(complex)

M3
```

Out[177]:
```
array([[  1.+0.j,   4.+0.j],
       [  9.+0.j,  16.+0.j]])
```

Lectura adicional

- Numpy (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) - Una guía de Numpy para usuario de MATLAB.