NumPy - multidimensional data arrays

Introduction

The NumPy package (module) is used in almost all numerical computation using Python. It is a package that provides 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. Here we will import all of NumPy into the global namespace. We do this for convenience in this notebook, since we will be using NumPy functions extensively. However, you will often see the convention used to import NumPy into a namespace called np (`'import numpy as np'`). This is often seen in iPython notebooks or interactive sessions, and we will follow this convention in subsequent lecture notebooks. But for now, be aware that most of the functions you will see demonstrated below are actually from the NumPy package.

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 tuple
- using functions that are dedicated to generating NumPy arrays, such as `arange` , `linspace` , etc.
- reading data from files

From lists

One of the most basic ways to create a NumPy array is to initialize it from an existing Python list. For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function:

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

v
```

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

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

M
```

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

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

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

Out[4]:
```
(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 [5]:
```
v.shape
```

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

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

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

v is a 1 dimensional vector, with 4 elements in it. M is a 2 dimensional matrix, with 2 rows and 2 columns (for a total of 4 elements).

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

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

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

Out[9]: 4

So far the `numpy.ndarray` looks a lot 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, and they cannot be changed once 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 are used).

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

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

Out[10]:
```
dtype('int32')
```

In this case, the M array contains integer elements (int64 indicates that we use 64 bits to represent each integer element, also known as a long integer, where a 32 bit integer is usually considered a regular sized integer).

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

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[11], line 1
----> 1 M[0,0] = "hello"

ValueError: invalid literal for int() 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 [12]:
```
M = array([[1, 2], [3, 4]], dtype=float)

print(M)
print(M.dtype)
```

```
[[1. 2.]
 [3. 4.]]
float64
```

Common 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 impractical to initialize the data manually, using explicit Python lists. Instead we can use one of the many functions in NumPy that generates arrays of different forms (or reads in the data from some other source, e.g. files, see next section). Some of the more common are:

arange

In [13]:
```
# create a range

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

x
```

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

In [14]:
```
set_printoptions(4, suppress=True) # show only four decimals
x = arange(-1, 1, 0.1)

x
```

Out[14]:
```
array([-1. , -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, -0. ,
        0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

random data

We will have a whole section in this class on random models and random number generation. Here are some examples of creating arrays with randomly generated data using `NumPy` functions.

In [15]:
```
from numpy import random
import numpy as np
```

In [16]:
```
# uniform random numbers in range [0,1]
# generate a 2 dimensional array of random numbers, with 5 elements along each dimension.
random.rand(5,5)
```

Out[16]:
```
array([[0.2628, 0.143 , 0.6731, 0.4783, 0.8872],
       [0.2121, 0.1503, 0.084 , 0.5165, 0.7287],
       [0.4011, 0.9229, 0.082 , 0.2902, 0.6218],
       [0.9435, 0.2906, 0.9348, 0.1064, 0.1519],
       [0.4729, 0.746 , 0.8816, 0.163 , 0.4153]])
```

In [17]:
```
# standard normally distributed random numbers (mean or mu = 0.0, standard deviation
# or sigma = 1.0
# create a 3 dimensional array with 3 elements in each dimension
x = random.randn(3,3,3)
print(x)
print(x.mean())
print(x.std())
```

```
[[[ 0.7893 -1.4189 -0.5957]
  [-0.9714 -0.0473  0.1049]
  [ 0.0582  0.5605 -0.6347]]

 [[-0.5994 -0.3274 -0.4326]
  [ 0.492  -0.169  -0.8582]
  [ 1.3877 -1.3215 -0.7176]]

 [[ 0.5975  0.9197 -1.3692]
  [ 0.4592 -0.6368  0.2133]
  [ 1.3632  0.4346 -0.6868]]]
-0.12615682738484515
0.7780460653500236
```

An often seen idiom allocates a two-dimensional array, and then fills in one-dimensional arrays from some function:

In [18]:
```
twod = np.zeros((5, 2))
twod
```

Out[18]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])
```

In [19]:
```
for i in range(twod.shape[0]):
    twod[i, :] = np.random.random(2)
twod
```

Out[19]:
```
array([[0.0321, 0.6094],
       [0.0469, 0.1921],
       [0.2818, 0.5321],
       [0.8946, 0.6138],
       [0.553 , 0.4776]])
```

Benchmark the difference in performance from the preceeding code to the following code that does the same thing.

In [20]:
```
twod = np.random.random(size=(5,2))
twod
```

Out[20]:
```
array([[0.2094, 0.1038],
       [0.5627, 0.5451],
       [0.7036, 0.4717],
       [0.8783, 0.7325],
       [0.8409, 0.4699]])
```

diag

In [21]:
```
# a diagonal matrix
diag([1,2,3])
```

Out[21]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

In [22]:
```
# diagonal with offset from the main diagonal
diag([1,2,3], k=1)
```

Out[22]:
```
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```

zeros and ones

In [23]:
```
# a 3x3 2 dimensional array, filled with zeros
zeros((3,3))
```

Out[23]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [24]:
```
# a vector (1 dimensional array) of 10 ones
ones((10,))
```

Out[24]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

List Performance vs. NumPy Performance

In the cell below, time the numpy sum vs the list sum using `%time` to see the difference

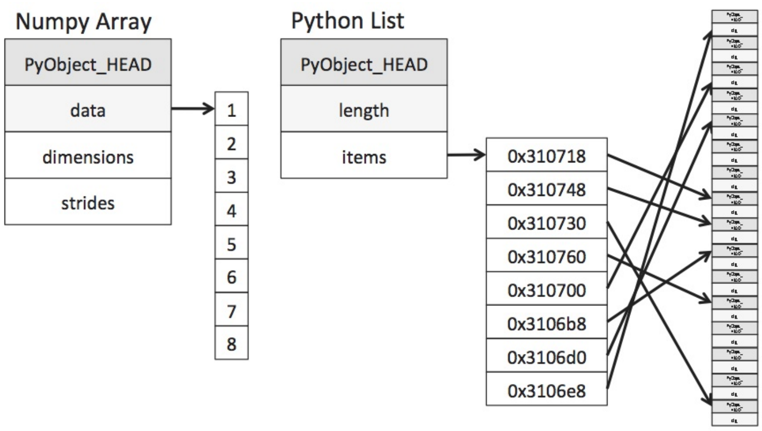
In [25]:
```
Nelements = 10000
Ntimeits = 10000

x = arange(Nelements)
y = range(Nelements)
```

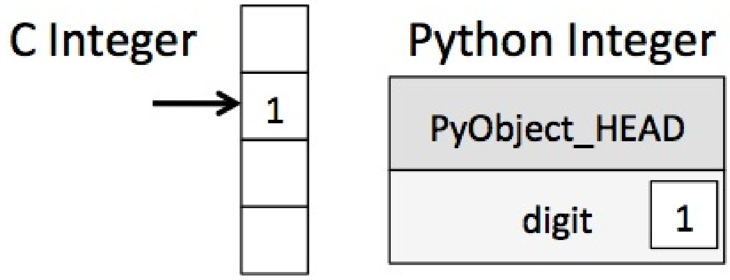
Numpy Arrays vs. Python Lists?

1. Why the need for numpy arrays? Can't we just use Python lists?
2. Iterating over numpy arrays is slow. Slicing is faster

Python lists may contain items of different types. This flexibility comes at a price: Python lists store *pointers* to memory locations. On the other hand, numpy arrays are typed, where the default type is floating point. Because of this, the system knows how much memory to allocate, and if you ask for an array of size 100, it will allocate one hundred contiguous spots in memory, where the size of each spot is based on the type. This makes access extremely fast.



BUT, iteration slows things down again. In general you should not access numpy array elements by iteration. This is because of type conversion. Numpy stores integers and floating points in C-language format. When you operate on array elements through iteration, Python needs to convert that element to a Python int or float, which is a more complex beast (a `struct` in C jargon). This has a cost.



If you want to know more, read this from Jake Vanderplas's Data Science Handbook. You will find that book an incredible resource.

Why is slicing faster? The reason is technical: slicing provides a view onto the memory occupied by a numpy array, instead of creating a new array. That is the reason the code above this cell works nicely as well. However, if you iterate over a slice, then you have gone back to the slow access.

By contrast, functions such as `np.dot` are implemented at C-level, do not do this type conversion, and access contiguous memory. If you want this kind of access in Python, use the struct module or Cython. Indeed many fast algorithms in numpy, pandas, and C are either implemented at the C-level, or employ Cython.

File I/O

For small examples and tests we often simply randomly or systematically generate the data we need into arrays, as we have just done. But for real computational problems and simulations, we usually need to get or load data that was generated from some external source or experiment in order to analyse it. `NumPy` supports reading in data from regular files in several formats, including a space efficient 'NumPy' native format.

BTW, we won't get into it in this course, but Python and `NumPy` support more complex and currently popular formats for doing huge big data analysis projects. These include relational database queries, JSON and things such as HDF5 (Hierarchical Data Format) and many others.

Comma-separated values (CSV)

A very common but basic file format for data files are the comma-separated values (CSV), or related format such as TSV (tab-separated values) or space-separated values. To read data from such file into NumPy arrays we can use the `numpy.genfromtxt` function. For example, the stockholm_td_adj file contains temperature data recored from Stockholm, SW. A recording was made each day of the year (presumably at the same time of day). The first 3 columns are the year, month and day. The next 3 columns hold the temperature in degress Celcius:

Note that you may need to change the path to the data below

In [31]:
```
data = genfromtxt('stockholm_td_adj.dat')
```

In [32]:
```
data.shape
```

Out[32]:
```
(77431, 7)
```

This indicates that the data consists of 77431 rows or records. Each row has 7 columns, or elements. As we mentioned, the first 3 columns are the year, month and day, and the next 3 columns are the temperature recordings.

We can look at the first 10 values from columns 0, 1 and 2 (the year, month and day).

In [33]:
```
data[0:10,0:3]
```

Out[33]:
```
array([[1800.,    1.,    1.],
       [1800.,    1.,    2.],
       [1800.,    1.,    3.],
       [1800.,    1.,    4.],
       [1800.,    1.,    5.],
       [1800.,    1.,    6.],
       [1800.,    1.,    7.],
       [1800.,    1.,    8.],
       [1800.,    1.,    9.],
       [1800.,    1.,   10.]])
```

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

In [34]:
```
M = random.rand(3,3)

M
```

Out[34]:
```
array([[0.5829, 0.8486, 0.5251],
       [0.8062, 0.2064, 0.2644],
       [0.6745, 0.5751, 0.4411]])
```

In [35]:
```
savetxt("random-matrix.csv", M)
```

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)
```

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

Out[37]:
```
array([[0.5829, 0.8486, 0.5251],
       [0.8062, 0.2064, 0.2644],
       [0.6745, 0.5751, 0.4411]])
```

More properties of 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, as we have done before with regular Python lists:

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

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

If we want to access `NumPy` arrays with 2 or more dimensions, we can specify each element of each dimension, sepearating the dimension indexes with a ,

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

```
[[0.5829 0.8486 0.5251]
 [0.8062 0.2064 0.2644]
 [0.6745 0.5751 0.4411]]
0.20636378875212835
```

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([[0.5829, 0.8486, 0.5251],
       [0.8062, 0.2064, 0.2644],
       [0.6745, 0.5751, 0.4411]])
```

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

Out[44]:
```
array([0.8062, 0.2064, 0.2644])
```

The same thing can be achieved with using `:` instead of an index (this is actually a slice):

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

Out[45]:
```
array([0.8062, 0.2064, 0.2644])
```

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

Out[46]:
```
array([0.8486, 0.2064, 0.5751])
```

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.    , 0.8486, 0.5251],
       [0.8062, 0.2064, 0.2644],
       [0.6745, 0.5751, 0.4411]])
```

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

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

Out[50]:
```
array([[ 1.    ,  0.8486, -1.    ],
       [ 0.    ,  0.    , -1.    ],
       [ 0.6745,  0.5751, -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])
```

WARNING: Array slices are *mutable*: if they are assigned a new value the original array from which the slice was extracted is modified (so they are really a view into the original data/memory of the array):

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])
```

Functions for extracting data from arrays and creating arrays

diag

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

In [66]:
```
diag(A)
```

Out[66]:
```
array([ 0, 11, 22, 33, 44])
```

In [67]:
```
diag(A, -1)
```

Out[67]:
```
array([10, 21, 32, 43])
```

choose

Constructs an array by picking elements form several arrays:

In [68]:
```
which = [1, 0, 2, 3]
choices = [[-1, -2, -3, -4], [1,2,3,4]]

choose(which, choices, mode='wrap')
```

Out[68]:
```
array([ 1, -2, -3,  4])
```

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 [69]:
```
v1 = arange(0, 5)
```

In [70]:
```
v1 * 2
```

Out[70]:
```
array([0, 2, 4, 6, 8])
```

In [71]:
```
v1 + 2
```

Out[71]:
```
array([2, 3, 4, 5, 6])
```

In [72]:
```
A * 2, A + 2
```

Out[72]:
```
(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 [73]:
```
A * A # element-wise multiplication
```

Out[73]:
```
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 [74]:
```
v1 * v1
```

Out[74]:
```
array([ 0,  1,  4,  9, 16])
```

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

In [75]:
```
A.shape, v1.shape
```

Out[75]:
```
((5, 5), (5,))
```

In [76]:
```
A * v1
```

Out[76]:
```
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` function, which applies a matrix-matrix, matrix-vector, or inner vector multiplication to its two arguments:

In [77]:
```
dot(A, A)
```

Out[77]:
```
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 [78]:
```
dot(A, v1)
```

Out[78]:
```
array([ 30, 130, 230, 330, 430])
```

In [79]:
```
dot(v1, v1)
```

Out[79]:
```
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 [80]:
```
M = matrix(A)
v = matrix(v1).T # make it a column vector
```

In [81]:
```
v
```

Out[81]:
```
matrix([[0],
        [1],
        [2],
        [3],
        [4]])
```

In [82]:
```
M*M
```

Out[82]:
```
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 [83]:
```
M*v
```

Out[83]:
```
matrix([[ 30],
        [130],
        [230],
        [330],
        [430]])
```

In [84]:
```
# inner product
v.T * v
```

Out[84]:
```
matrix([[30]])
```

In [85]:
```
# with matrix objects, standard matrix algebra applies
v + M*v
```

Out[85]:
```
matrix([[ 30],
        [131],
        [232],
        [333],
        [434]])
```

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

In [86]:
```
v = matrix([1,2,3,4,5,6]).T
```

In [87]:
```
shape(M), shape(v)
```

Out[87]:
```
((5, 5), (6, 1))
```

In [88]:
```
M * v
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[88], line 1
----> 1 M * v

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\numpy\matrixlib\defmatrix.py:218, in matrix.__mul__(self, other)
    215 def __mul__(self, other):
    216     if isinstance(other, (N.ndarray, list, tuple)) :
    217         # This promotes 1-D vectors to row vectors
--> 218         return N.dot(self, asmatrix(other))
    219     if isscalar(other) or not hasattr(other, '__rmul__') :
    220         return N.dot(self, other)

File <__array_function__ internals>:200, in dot(*args, **kwargs)

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

Data processing

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 [89]:
```
# reminder, the tempeature dataset is stored in the data variable:
shape(data)
```

Out[89]:
```
(77431, 7)
```

mean

In [90]:
```
# the temperature data is in column 3
mean(data[:,3])
```

Out[90]:
```
6.197109684751585
```

The daily mean temperature in Stockholm over the last 200 year so has been about 6.2 C.

standard deviations and variance

In [91]:
```
std(data[:,3]), var(data[:,3])
```

Out[91]:
```
(8.282271621340573, 68.59602320966341)
```

min and max

In [92]:
```
# lowest daily average temperature
data[:,3].min()
```

Out[92]:
```
-25.8
```

In [93]:
```
# highest daily average temperature
data[:,3].max()
```

Out[93]:
```
28.3
```

sum, prod, and trace

In [94]:
```
d = arange(0, 10)
d
```

Out[94]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [95]:
```
# sum up all elements
sum(d)
```

Out[95]:
```
45
```

In [96]:
```
# product of all elements, do you understand why we passed d+1 to the prod() function?
prod(d+1)
```

Out[96]:
```
3628800
```

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 [97]:
```
A
```

Out[97]:
```
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 [98]:
```
n, m = A.shape
```

In [99]:
```
B = A.reshape((1,n*m))
B
```

Out[99]:
```
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 [100…
```
B[0,0:5] = 5 # modify the array

B
```

Out[100…
```
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 [101…
```
A # and the original variable is also changed. B is only a different view of the same data
```

Out[101…
```
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 [102…
```
B = A.flatten()

B
```

Out[102…
```
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 [103…
```
B[0:5] = 10

B
```

Out[103…
```
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 [104…
```
A # now A has not changed, because B's data is a copy of A's, not refering to the same data
```

Out[104…
```
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]])
```

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 [105…
```
a = array([[1, 2], [3, 4]])
```

In [106…
```
# repeat each element 3 times
repeat(a, 3)
```

Out[106…
```
array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
```

In [107…
```
# tile the matrix 3 times
tile(a, 3)
```

Out[107…
```
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4]])
```

concatenate

In [108…
```
b = array([[5, 6]])
```

In [109…
```
concatenate((a, b), axis=0)
```

Out[109…
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [110…
```
concatenate((a, b.T), axis=1)
```

Out[110…
```
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 [111…
```
A = array([[1, 2], [3, 4]])

A
```

Out[111… array([[1, 2],
       [3, 4]])

In [112…
```
# now B is referring to the same array data as A
B = A
```

In [113…
```
# changing B affects A
B[0,0] = 10

B
```

Out[113…
```
array([[10,  2],
       [ 3,  4]])
```

In [114…
```
A
```

Out[114…
```
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 [115…
```
B = copy(A)
```

In [116…
```
# now, if we modify B, A is not affected
B[0,0] = -5

B
```

Out[116…
```
array([[-5,  2],
       [ 3,  4]])
```

In [117…
```
A
```

Out[117…
```
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 [118…
```
v = array([1,2,3,4])

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

```
1
2
3
4
```

In [119…
```
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 [120…
```
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 [121…

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

Out[121…

```
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 [122…

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

In [123…

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[123], line 1
----> 1 Theta(array([-3,-2,-1,0,1,2,3]))

Cell In[122], line 5, in Theta(x)
      1 def Theta(x):
      2     """
      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 a.all()
```

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 [124…

```
Theta_vec = vectorize(Theta)
```

In [125…

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

Out[125…

```
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 [126…

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

In [127…

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

Out[127…

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

In [128…

```
# still works for scalars as well
Theta(-1.2), Theta(2.6)
```

Out[128…

```
(0, 1)
```

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 [129…
```
M.dtype
```

Out[129…
```
dtype('int32')
```

In [130…
```
M2 = M.astype(float)

M2
```

Out[130…
```
array([[ 1.,  4.],
       [ 9., 16.]])
```

In [131…
```
M2.dtype
```

Out[131…
```
dtype('float64')
```

In [132…
```
M3 = M.astype(bool)

M3
```

Out[132…
```
array([[ True,  True],
       [ True,  True]])
```

Benchmarking

You can calculate the size of the numpy array (in bytes) by using the following code. Try adding a few more lines to this to compare the size (storage requirements) of different arrays.

In [133…
```
M.data.nbytes
```

Out[133…
```
16
```

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.

Acknowledgements

Original versions of these notebooks created by J.R. Johansson (robert@riken.jp) http://dml.riken.jp/~rob/. Adjustments have been made by Dr. Derek Riley

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js