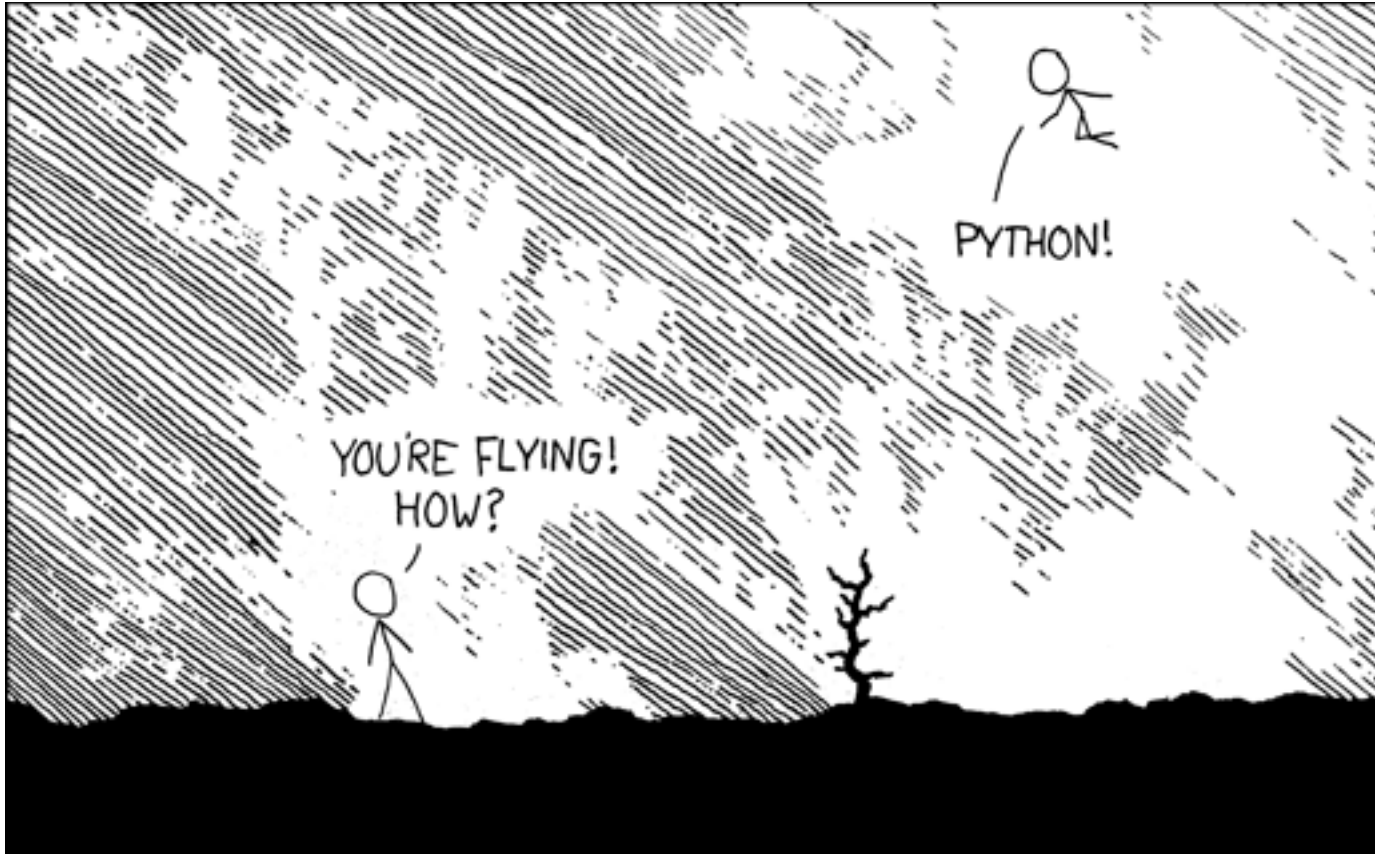


# NUMERICAL PYTHON



*Python for computational science*

19 - 21 Nov 2012

CINECA

[m.cestari@cineca.it](mailto:m.cestari@cineca.it)



# Introduction (1)

- Numerical Python:
  - first implemented as 2 different libraries  
`Numeric`, `Numarray`
  - NumPy comprehends all the features of the two libraries
    - It is the *de facto* **standard**
- NumPy offers efficient array computation; feat highly required by scientific users
- most of the scientific and mathematical Python-based packages use internally NumPy arrays

# Introduction (2)



<b>Sub-Packages</b>	<b>Purpose</b>	<b>Comments</b>
<b>core</b>	basic objects	all names exported to numpy
<b>lib</b>	Additional utilities	all names exported to numpy
<b>linalg</b>	Basic linear algebra	LinearAlgebra derived from Numeric
fft	Discrete Fourier transforms	FFT derived from Numeric
<b>random</b>	Random number generators	RandomArray derived from Numeric
distutils	Enhanced build and distribution	improvements built on standard distutils
testing	unit-testing	utility functions useful for testing
<b>f2py</b>	Automatic wrapping of Fortran code	a useful utility needed by SciPy

## Introduction (3)

- As usual, to import the NumPy module type:

```
>>> import numpy
```

```
>>> from numpy import *
```

```
>>> import numpy as np      # default for this presentation  
                             # and for standard documentation
```

# Introduction (3)

- **Ipython:**

- **enhance** the prompt capabilities

- tab completion for functions, modules, variables, files

- works neatly with **matplotlib**

- filesystem navigation (cd, ls, pwd)

- has access to the standard Python help and ?/?? information

- Search commands (Ctrl-n, Ctrl-p, Ctrl-r)

- the output of the nth command is in `_n`

- magic commands: type % → (tab) to list them all

- %whos

- %run script.py

- %timeit

- %logstart name

- improves the **interactive mode** usage

# NumPy (main) objects



- NumPy provides two fundamental objects:
  - **ndarray**; described in the following slides. From now on when we say array we always mean ndarray
  - **ufunc**; functions that operate on one or more ndarrays **element-by-element**

**ndarray**



# ndarray (1)



- Like Lists, tuples and sets, ndarrays are collection of items
- ndarray is a **homogeneous** collection (lists, tuples, sets can be not)
  - this means that items are occupying a fixed number of bytes :
  - typically representing a numerical type ...
  - but also an arbitrary record made by a collection of numerical types and/or string

- ndarray differs from the standard Python sequences:
  - NumPy arrays have a **fixed size** at creation unlike Python lists which can grow dynamically (changing ndarray size produces a new ndarray and deletes the original)
  - the elements in the ndarray must be the **same data type** (and thus they will be the same size in memory)
  - same data type does not mean single item
    - `dtype([('f1', '<u4'), ('f2', '|S3')])`

# Ndarray (1) cont'ed



- ndarray examples:

```
>>> a = np.array([1, 2, 3])
```

```
>>> b = np.array([[1.5, 2.2, 34.4], [2.3, 4, 6.5]])
```

```
>>> c = np.array(['lof', 'l', 'ga'])
```

# Ndarray (1) cont'ed



- ndarray examples:

```
>>> list = [2, 3, 'lof'] # list → [2, 3, 'lof']
```

```
>>> a = np.array(list) # doesn't work as expected
```

# ndarray (2)



(numpy/core/include/numpy/ndarrayobject.h)

```
typedef struct PyArrayObject {
    char *data;          /* pointer to raw data buffer */
    PyArray_Descr *descr; /* Pointer to type structure */
    int nd;              /* number of dimensions, also
                          called ndim */
    npy_intp *dimensions; /* size in each dimension */
    npy_intp *strides;    /* bytes to jump to get to the next
                          element in each dimension */
    PyObject *base;      /* This object should be decref'd
                          upon deletion of array */
                          /* For views it points to the
                          original array */
    int flags;           /* Flags describing array */
} PyArrayObject;
```

# ndarray: data



```
char *data;          /* pointer to raw data buffer */  
PyArray_Descr *descr; /* pointer to type structure */
```

- data is just a pointer to bytes in memory
- data needs some sort of descriptor (dtype) to be interpreted

```
In [1]: a = np.array([1, 2, 3]) # there's a buffer of  
In [2]: a.dtype                # 12 bytes, which are  
Out[2]: dtype('int32')        # interpreted as 3 ints
```

- int32: 32 are the number of bits used to represent the integer
- There are 21 built-in **data type** objects that can be used to form ndarray.

→ Numpy supports more numerical types than Python

# ndarray: dtype



Data Type	np.dtype	char. code	Description
bool	np.bool	'b'	Boolean (True or False) stored as a byte
string	'S'	'S', 'a'	String datatype
int	np.int	'i4' or 'i8'	Platform integer (normally either int32 or int64)
int8	np.int8	'i1'	Byte (-128 to 127)
int16	np.int16	'i2'	Integer (-32768 to 32767)
int32	np.int32	'i4'	Integer (-2147483648 to 2147483647)
int64	np.int64	'i8'	Integer (-9223372036854775808 to 9223372036854775807)
uint32	np.uint32	'u4'	Unsigned integer (0 to 4294967295)
uint64	np.uint64	'u8'	Unsigned integer (0 to 18446744073709551615)
float	np.float	'f8'	Shorthand for float64
float32	np.float32	'f4'	Single precision float, sign bit, 8 bits exponent, 23 bits mantissa
float64	np.float64	'f8'	Double precision float, sign bit, 11 bits exponent, 52 bits mantissa
complex64	np.complex64	'c8'	Complex number represented by two 32-bit floats
complex128	np.complex128	'c16'	Complex number represented by two 64-bit floats

# ndarray: dtype cont'ed



- the elements of a ndarray can be specified via a dtype object
- Examples:

## numerical

```
>>> np.dtype(np.int32) # 32 are the bits used to
dtype('np.int32') # represent the integer
# (4 bytes int)
```

## string

```
>>> np.dtype('S3') # mind the case here
```

## both

```
>>> np.dtype([('f1', np.uint), ('f2', 'S3')])
dtype([('f1', '<u4'), ('f2', '|S3')]) # list of tuples
```



# Exercise 1: dtype



- Try to construct a dtype object to represent an atom in a 3d space. It should contain:
  - a string ('S3') to represent the symbol (i.e. Na, Cl, H, O)
  - an integer (np.int) for the atomic weight (i.e. 11, 17, 1)
  - 3 floats 64 bits (np.float64) for the cartesian coordinates
- Help needed? Try (after having imported numpy):

```
>>> help(np.dtype)           (python)
```

```
>>> np.dtype?                (ipython)
```

# Exercise 1: dtype



- Solution

```
>>> dt = np.dtype([('symb', 'S3'), ('PA', np.int),  
('x', np.float64), ('y', np.float64), ('z', np.float64)])  
  
dtype([('symb', '|S3'), ('Z', '<i4'), ('x', '<f8'), ('y',  
'<f8'), ('z', '<f8')])
```

# ndarray: dimensions



```
int nd;          /* number of dimensions,  
                 also called ndim */
```

```
numpy_intp *dimensions;    /* size in each dimension */
```

```
In [1]: b = np.array([[1,2], [3,4]])
```

```
In [2]: b.ndim
```

```
Out[2]: 2
```

```
In [3]: b.shape
```

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

# ndarray: strides



```
numpy_intp *strides;      /* bytes to jump to get to the
                          next element of each dimensions */
```

```
In [36]: b
```

```
Out[36]:
```

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

```
In [37]: b.dtype.itemsize
```

```
Out[37]: 4           # bytes to represent item
```

```
In [38]: b.strides
```

```
Out[38]: (8, 4)      # (skip_bytes_row, skip_bytes_col)
           # (2*itemsize, itemsize)
```

# ndarray: flags



```
int flags;          /* Flags describing array
                    see below */
```

```
In [36]: b.flags
```

```
Out[39]:
```

```
C_CONTIGUOUS : True
F_CONTIGUOUS : False
```

```
OWNDATA : True      # are we responsible for memory
                    # handling?
```

```
WRITEABLE : True    # may we change the data?
```

```
ALIGNED : True      # appropriate hardware alignment
```

```
UPDATEIFCOPY : False # update .base w/ its data on
                    # deallocation
```

# ndarrays memory layout

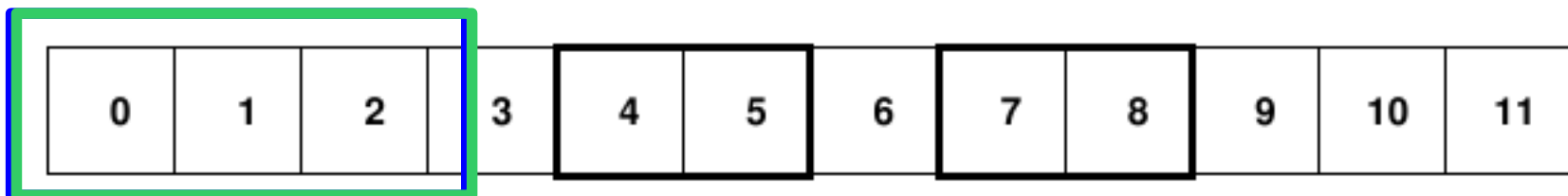


0	1	2
(0,0)	(0,1)	(0,2)
3	4	5
(1,0)	(1,1)	(1,2)
6	7	8
(2,0)	(2,1)	(2,2)
9	10	11
(3,0)	(3,1)	(3,2)

C

0	3	6	9
(0,0)	(0,1)	(0,2)	(0,3)
1	4	7	10
(1,0)	(1,1)	(1,2)	(1,3)
2	5	8	11
(2,0)	(2,1)	(2,2)	(2,3)

Fortran



- In C-style the last index varies first, than the second-to-last varies second, and so on (**default behavior for numpy**).
- Algorithms that work on N-dimensional arrays that are written in Fortran typically expect Fortran-style arrays

# ndarray: base pointer



```
PyObject *base; /* This object should be decref'd
                  upon deletion of array. For views it
                  points to the original array */
```

```
In [60]: a = b.view()      # a = array([[1, 2],
                               [3, 4]])
```

```
In [61]: a.flags
```

```
C_CONTIGUOUS : True
```

```
F_CONTIGUOUS : False
```

```
OWNDATA : False
```

```
In [62]: a[0] = 1, 5      # now also b = array([[1, 5],
                               [3, 4]])
```

# Creating arrays (1)



- **5 general mechanism** for creating arrays:

**(1)** Conversion from other Python structures (e.g. lists, tuples)

```
>>> list = [0,2,3,7];
>>> a = np.array(list)      # or np.array([0,2,3,7])
      Array([0, 2, 3, 7])
>>> b = np.array([l,m])    # l, m python lists
>>> b = np.asarray(b)      # convert b to an array if b is,
                          # i.e, a list; otherwise do
                          # nothing
```



# Creating arrays (2)



**(2)** Intrinsic NumPy array creation functions (e.g. `arange`, `ones`, `zeros`, etc.)

```
>>> a = np.zeros(10)
```

```
>>> a = np.ones((3,2))
```

```
>>> a = np.linspace(0,10,11) # start, stop, npoints
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  
       7.,  8.,  9., 10.]
```

```
>>> a = np.arange(10)
```

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

```
>>> a = np.arange(1,10,2) # start, stop, step
```

```
array([1, 3, 5, 7, 9])
```

# Creating arrays (3)



Try the shortcut: **np.r\_**

```
>>> a = np.r_[0,1,2]      # ~ np.array
>>> a = np.r_[0:10:0.1]   # np.arange
>>> a = np.r_[0:10:100j]  # np.linspace
```

## (3) Creating arrays from strings or buffers

```
>>> np.fromstring('1, 2', dtype=np.int, sep=',')
array([1, 2])
```

# Creating arrays (4)



(4) Reading arrays from disk, either from standard or custom formats (mind the **byte order!!**)

- `loadtxt, genfromtxt` # custom ASCII formats
- `fromfile(), <ndarray>.tofile()` # custom binary  
# formats
- HDF5: `h5py` # standard formats
- netcdf: `netcdf-python`

# Creating arrays (5)



- Reading from ASCII text files

<pre><b>loadtxt</b>(fname[, dtype=, comments=, delimiter=, skiprows=, converters=, usecols=, ...])</pre>	Load data from a text file.
<pre><b>savetxt</b>(fname, X[, fmt=, delimiter=])</pre>	Save an array to a text file (auto recognizes if .gz) x is the array
<pre><b>genfromtxt</b>(fname[, dtype=, comments=, missing= ...])</pre>	Load data from a text file, with missing values handled as specified.

- `loadtxt` is faster but less flexible
- `genfromtxt` runs two loops, the first reads all the lines (→ strings), the second parses the line and converts each string to a data type
  - It has the advantage of taking into account missing data

## Exercise 2: creating array



1. Create a ndarray from the lists [1, 12, 123] and 2-dimensional array from [1,2,3,4] and [10,10,11,11]

```
>>> help(np.array)
```

2. Create a ndarray of 6x5 elements, all set to 1

```
>>> help(np.ones)
```

3. Download 'mol.xyz' from [here](#) and build a new array of atom dtype

```
>>> help(np.loadtxt)
```

# Exercise 2: creating array



## Solutions

1.

```
>>> np.array([1,12,123])
```

```
>>> np.array([[1,2,3,4],[10,10,11,11]])
```

2.

```
>>> np.ones((6,5))
```

3.

```
>>> dt = np.dtype([('symb','|S3'),('PA',np.int),  
('x',np.float64),('y',np.float64),('z',np.float64)])
```

```
>>> arr = np.loadtxt('mol.xyz',dtype=dt)
```

# Creating arrays (6)



- From the ndarray just built we can save a binary file with the ndarray data

```
>>> arr.tofile('mol_bin')
```

- It can be read with `np.fromfile`

```
>>> arr2 = np.fromfile('mol_bin', dtype=dt)
```

- Be careful when using `np.fromfile` w/ fortran binary files
  - You may need to deal with field separators...

# Creating arrays (7)



**(5)** Special library functions (e.g. random, ...) or *ad hoc* defined function

```
>>> def func(i,j):  
    return i*j+1  
  
>>> a = np.fromfunction(func, (3,4)) # meshgrid  
array([[ 1.,  1.,  1.,  1.],  
       [ 1.,  2.,  3.,  4.],  
       [ 1.,  3.,  5.,  7.]])
```

A blue arrow points from the text "i = 1, j = 3" to the value "4." in the second row, fourth column of the array.

```
In [1]: np.random.rand(4)
```

```
Out[1]:
```

```
array([ 0.14046702,  0.31755881,  0.44131823,  
        0.1219652])
```



# (some) ndarray attributes



- Some useful ndarray attributes are (let `arr` be the array)
  - `arr.shape` returns a tuple with the size of the array. Changing the tuple re-shapes the array
  - `arr.dtype` data-type object for this array

```
a = np.ones(10);    a.dtype
dtype('float64')
```
  - `arr.ndim` number of dimension in array
  - `arr.size` total number of elements
  - `arr.base` object the array is using for its data buffer if any

# (some) ndarray functions



- Some useful ndarray functions are:
  - `arr.copy()` returns a copy of the array
  - `arr.view()` returns a new view of the array using the same data of `arr` (i.e. dtype can be varied)
  - `arr.reshape()` returns an array with a new shape
  - `arr.transpose()` returns an array view with the shape transposed
  - `arr.ravel()` return a flattered array

## (some) ndarray functions (2)



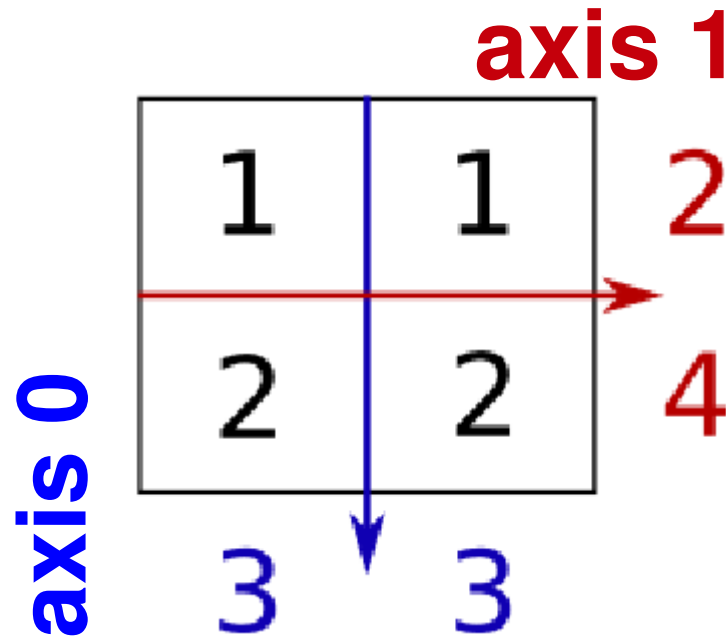
- other ndarray functions are:
  - `arr.min|max()` return min (or max) value
  - `arr.argmax|min()` return the index of min (max) value
  - `arr.sum()`

```
>>> a = np.array([[1,4,2],[3,1,8]])  
>>> a.sum(axis=0)  
>>> array([ 4,  5, 10])
```
  - `arr.var|mean|std|prod()` take the usual meaning

# (some) ndarray functions (2) cont'd



```
>>> a = np.array([[1,1],[2,2]])  
>>> a.sum(axis=0)  
>>> array([ 3,  3])
```



# (some) ndarray functions (3)



→ **sort** array sorting

```
>>> a = np.array([[1,4],[3,1]])
```

```
>>> np.sort(a)      # sort along the last axis
```

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

```
>>> np.sort(a, axis=0)  # sort along x axis
```

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

```
>>> np.sort(a, axis=None) # sort the flattened
                          # array
```

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

```
>>> np.sort(a, order=field) # 'field' of a
                          # structured array
```

# Basic routines overview



- Creating arrays routines (mentioned before in this presentation)
- Operation on two or more arrays (concatenate, dot, ...)
- Shape functions (split, resize, ...)
- Basic functions (average, cov, vectorize, ...)

Vectorize example:

```
def myfunc(a,b):  
    if (a>b): return a  
    else: return b-1  
  
vecfunc = np.vectorize(myfunc)  
  
vecfunc([[1,2,3],[5,6,9]], [[7,4,5],[7,4,5]])  
array([[6, 3, 4],  
       [6, 6, 9]])
```

- Two-dimensional array functions (eye, diag, ...)

- Indexing refers to the action of selecting array values by their position (indexes)
- `ndarrays` can be indexed using the following syntax:

**X[obj]**

- Depending on the nature of **obj**, a different indexing is triggered:
  - **Basic slicing** when **obj** is a an integer, or a tuple (or list) of integers, and slice objects.
    - ✓ always returns a **view** of the array
    - ✓ in this way is **faster** and **save memory**
    - ✓ a copy can always be obtained with the **.copy()** function
  - **Advanced selection** when **obj** contains a `ndarray` of data type integer or bool.
    - ✓ always returns a **copy** of the array

# Basic slicing (1)



- A slice object is represented by `X[I:J:K]`
  - `I` = **first element** (default is 0)
  - `J` = **last element** (excluded, default is length of the array)
  - `K` = **step or stride** (default is 1)
- Slicing for `ndarray` works similarly to standard Python sequences but can be done over **multiple dimensions**



# Basic slicing (2)



- Examples: given a two dimensional 5x10 ndarray "A"
  - $A[3] = A[3, :] = A[3, ::]$   
represents the **4th row** of the array (10 elements)
  - $A[:, 1] = A[:, :, 1]$   
represents the **2nd column** of the array (5 elements)
  - $A[0, ::2]$   
represents the **1st element** of every **even column**

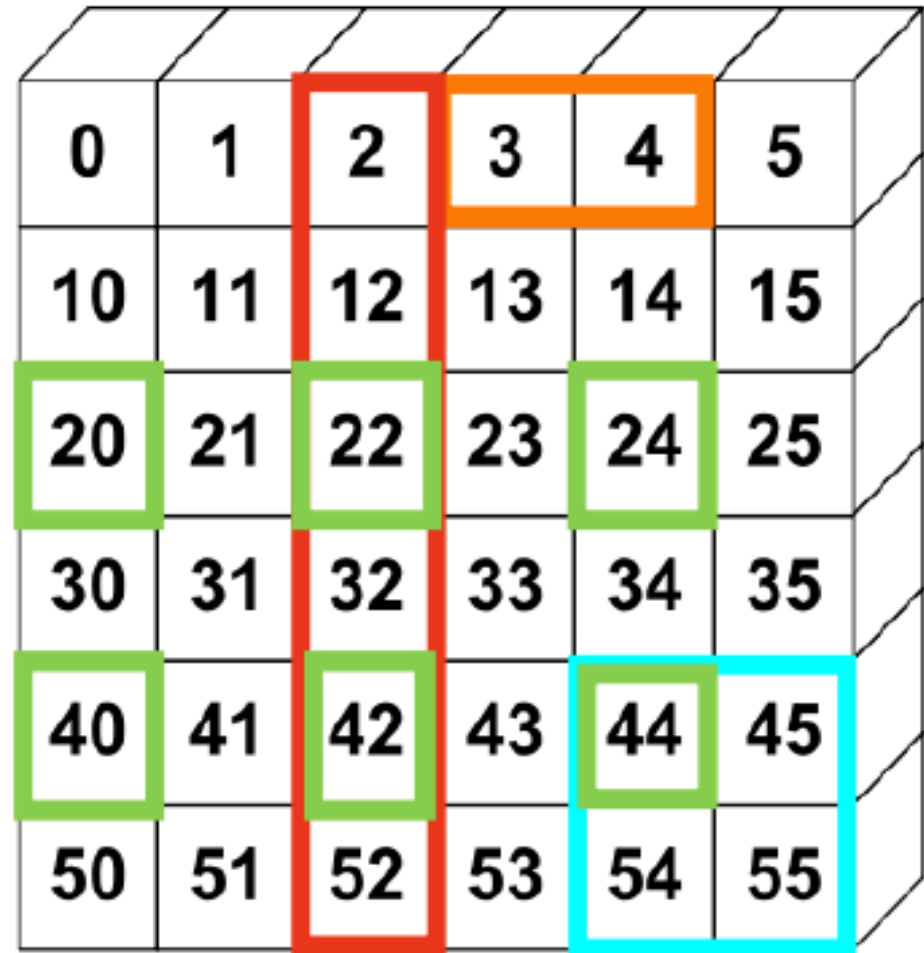
# Basic slicing (3)

```
>>> a[0, 3:5]
```

```
>>> a[4:, 4:]
```

```
>>> a[:, 2]
```

```
>>> a[2::2, ::2]
```



0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

# Basic slicing (4)



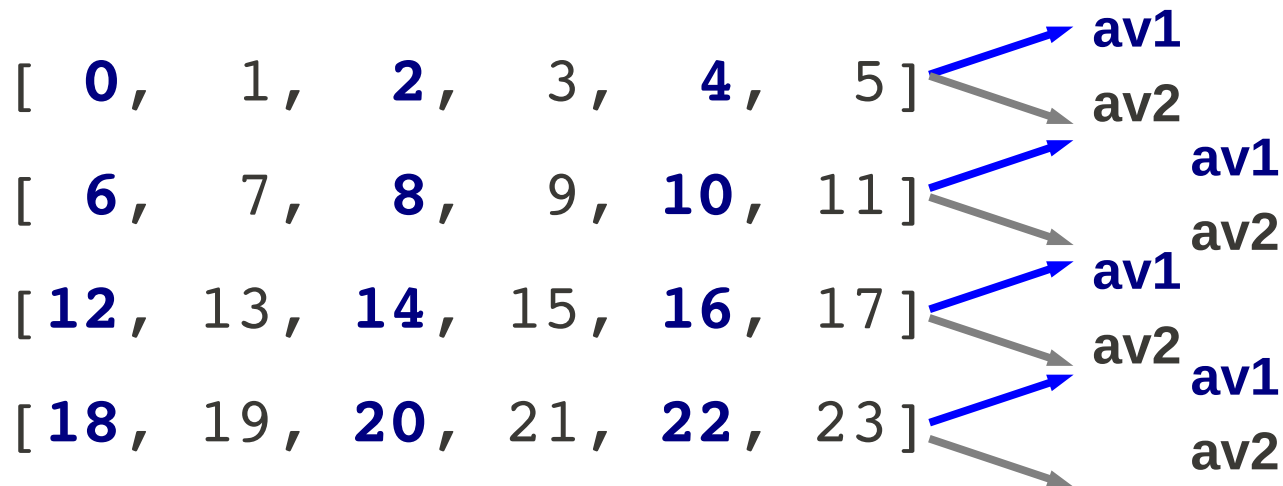
- **Ellipses** (...) can be used to replace zero or more ":" terms so that the total number of dimensions in the slicing tuple **matches the shape** of the ndarray

```
a = np.arange(60); a.shape=(3,4,5)
print a[... ,3]      # print a[:, :, 3] → (3,4)
print a[1, ... ,3]  # print a[1, :, 3] → (4)
```

# Exercise 3: Basic slicing



- Build a 2-dim array, then calculate all the average values along the rows for its odd and even columns.



hint: `np.mean()`

## Exercise 3: Basic slicing



```
>>> a = np.arange(24).reshape(4,6)
```

```
>>> a[:,0::2].mean(axis=1)
```

```
>>> a[:,1::2].mean(axis=1)
```

or alternatively

```
>>> np.mean(a[:,0::2],axis=1) # [2., 8., 14., 20.]
```

```
>>> np.mean(a[:,1::2],axis=1) # [3., 9., 15., 21.]
```

# Advanced selection



- Indexing arrays with other arrays
- This means  $X[\text{obj}] \rightarrow \text{obj}$  is a `ndarray`
- There are two different ways:
  - **Integer index arrays.** Use one or more arrays of index values.
  - **Boolean (mask) index arrays.** Involve giving a boolean array of the proper shape to indicate the values to be selected

# Adv. selection: integer index arrays (1) NumPy

- Simple case: indexing 1d arrays
- The indexing arrays must be of the type integer

```
>>> x = np.arange(10,1,-1)
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
x[np.array([3,3,-2,8])]      # np.array([3,3,-2,8])
                             # → index array
array([7,  7,  3,  2])
>>> x[np.array([[3,3],[-2,8]])] # 2-d index array
array([[7,  7],
       [3,  2]])
```

- A new array is returned:
  - with the **same shape** as the **index array**
  - with the type and values of the array being indexed

# Adv. selection: integer index arrays (2) NumPy

- Indexing multi-dimensional arrays with more index arrays is allowed
- Simplest case scenario: index arrays have a **matching shape**; and there's one for each dimension of the array being indexed

```
>>> y = np.arange(18).reshape(3,6)
```

```
array([[ 0,  1,  2,  3,  4,  5],  
       [ 6,  7,  8,  9, 10, 11],  
       [12, 13, 14, 15, 16, 17]])
```

**matching  
shape**



```
>>> y[np.array([0,2,2]), np.array([0,1,2])]
```

```
array([ 0, 13, 14])      # y[0,0] y[2,1] y[2,2]
```



## Adv. selection: integer index arrays (3) NumPy

- If the index arrays do not have a matching shape they will be broadcasted (if possible)

```
>>> y[np.array([0,1,2]), 4] # scalar broadcasted
                                # to (3,)
array([ 4, 10, 16]) # y[0,4] y[1,4] y[2,4]
```

- It is possible to partially index the array with an index array

```
>>> y[np.array([0,2])]
array([[ 0,  1,  2,  3,  4,  5],
       [12, 13, 14, 15, 16, 17]])
```

# Adv. selection: boolean index arrays NumPy

- `a = np.array([True, False, True]) # dtype('bool')`
- Boolean arrays must be of the (or broadcastable to the) same shape of the array being indexed

```
>>> y
```

```
array([[ 0,  1,  2,  3,  4,  5],  
       [ 6,  7,  8,  9, 10, 11],  
       [12, 13, 14, 15, 16, 17]])
```

```
>>> b = y>10
```

```
>>> y[b] # y[y>10]
```

```
array([11, 12, 13, 14, 15, 16, 17]) # 1-d array
```

# ndarray selection



To summarize:

- NumPy provides different and very flexible ways to perform a selection on a ndarray.
- Selection can save you **time** and **code lines**...
- and enhances the code **performances**
  - by avoiding traversing array through python loops (internal (C) loops are still performed... more on that later)

## Exercise 4 ndarray selection



- From the atoms array calculate the contribution to the molecular weight for each of the different kind of atoms (C, N, H, O) making use of the Numpy selection

ex: `Sum_c PA(c)`

- Hint: use the advanced boolean selection

# Exercise 4 ndarray selection



- Solution

```
>>> arr[arr['symb']=='C']['PA'].sum() # 372
```

```
>>> arr[arr['symb']=='H']['PA'].sum() # 26
```

```
>>> arr[arr['symb']=='N']['PA'].sum() # 28
```

```
>>> arr[arr['symb']=='O']['PA'].sum() # 32
```

(does it break some of The Zen of Python's rules??)

(anyway don't tell Simone about it ...)

# NumPy (main) objects



- NumPy provides two fundamental objects:
  - **ndarray**; described in the following slides. From now on when we say array we always mean ndarray
  - **ufunc**; functions that operate on one or more ndarrays **element-by-element**

**ufunc**

# Universal functions (1)



- Universal functions, `ufuncs` are mathematical functions that operate on the `ndarray` objects
- all `ufuncs` are instances of a general class, thus they behave the same. All `ufuncs` perform **element-by-element** operations on the entire `ndarray(s)`
- These functions are **wrapper** of some core functions, typically implemented in compiled (C or Fortran) code



# Universal functions (2)



array in1(n1,n2)

array out1(n1,n2)

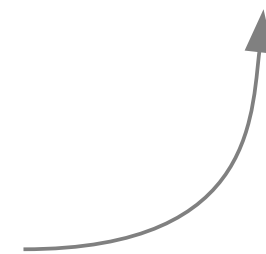
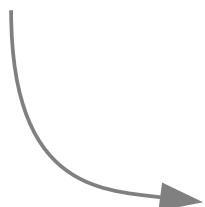
**UFUNC**

- BROADCASTING
- FINDS the optimized loop based on input types

- CREATES output array

ufunc\_loop

- LOOP iteration
- APPLY elementwise function



# Universal functions (2)



- Ufunc loop may look something like

```
void ufunc_loop(void **args, int *dimensions, int *steps,
                void *data)
{
    /* int8 output = elementwise_function(int8 input_1, int8
    * input_2)
    *
    * This function must compute the ufunc for many values at
    * once, in the way shown below.
    */

    char *input_1 = (char*)args[0];
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];
    int i;
    for (i = 0; i < dimensions[0]; ++i) {
        *output = elementwise_function(*input_1, *input_2);
        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

# Universal functions (2)



- Examples:

```
>>> a = np.array([a1, a2, ..., an]) # a -> ndarray
```

```
np.cos(a) = ([cos(a1), cos(a2), ..., cos(an)])
```

**ufunc!!**

**elementwise  
function**

```
>>> b = 1.5*a + 2 # * + / - all ufuncs
```

- all the elements of "a" are multiplied by 1.5; result stored in a **temp array**
- All the elements of the temp array are increased by 2; result stored in a **2<sup>nd</sup> temp array**
- b becomes a reference to the 2<sup>nd</sup> temp array

# Universal functions (3)



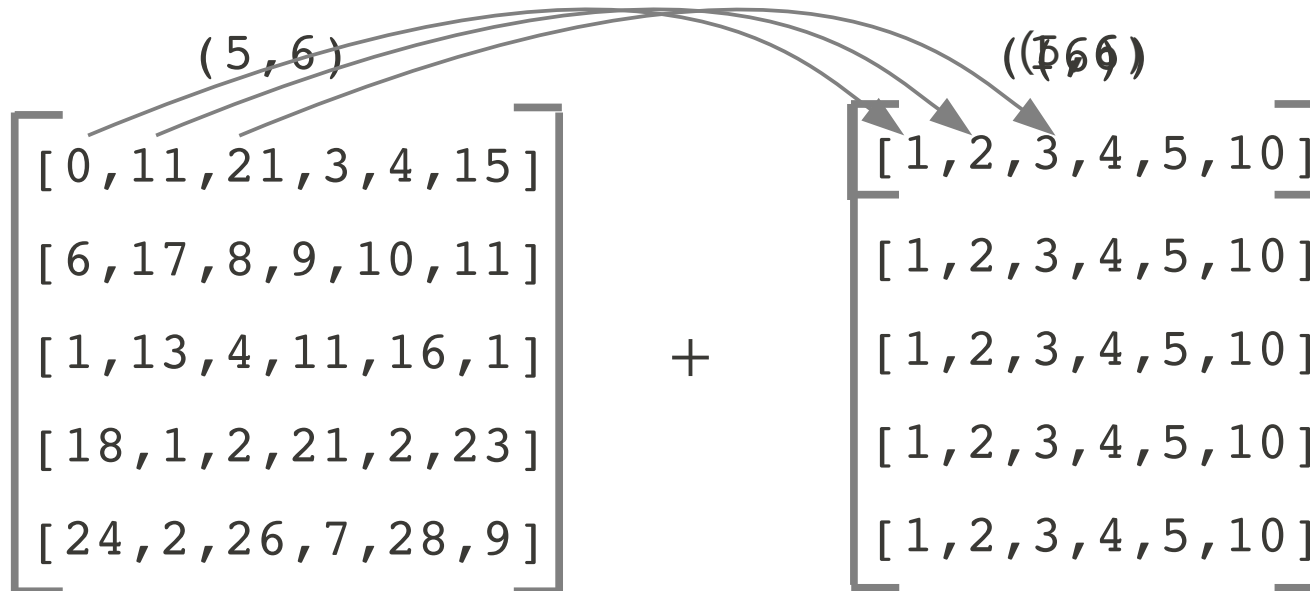
- What happens when ufunc has to operate on **two arrays** of **different dimensions**?

$$\begin{bmatrix} [0, 11, 21, 3, 4, 15] \\ [6, 17, 8, 9, 10, 11] \\ [1, 13, 4, 11, 16, 1] \\ [18, 1, 2, 21, 2, 23] \\ [24, 2, 26, 7, 28, 9] \end{bmatrix} + [1, 2, 3, 4, 5, 10]$$

- To be efficient, element-wise operations are required
- To operate element-by-element **broadcasting** is needed.

# Broadcasting (1)

- After having applied the broadcasting rules the shape of all arrays must match



# Broadcasting (2)



- Broadcasting allows to deal in a meaningful way with inputs that do not have the same shape.
- broadcasting follows these rules:
  - (1) If all input arrays do not have the same number of dimensions, a “1” will be repeatedly **prepended** to the shapes of the smaller arrays until all the arrays have the same number of dimensions
  - (2) Arrays with a size of 1 along a particular dimension act like the array with the **largest size** along that dimension. The **values** of the array are assumed to be **the same along the broadcasted dimension**

# Broadcasting (3)



- 1d-ndarrays are always broadcastable

```
a.shape → (3)          b.shape → (4,3);
```

```
a.shape → (1,3)       # first rule, n. of dim matches
```

```
a.shape → (4,3)       # second rule
```

```
c = a*b      c.shape → (4,3)  # as expected
```

- Another example

```
a.shape → (2,1,3)      b.shape → (2,4,1);
```

```
c = a*b      c.shape → (2,4,3)  # second rule
```

- Arrays not broadcastable

```
a.shape → (2,5,3)
```

```
b.shape → (2,4,1);      c = a*b      # shape mismatch
```

# Broadcasting (4)



- If needed it is possible to add a **trailing dimension** to the shape of a `ndarray`. This is a convenient way of taking the outer product (or any other outer operation) of two arrays

```
a.shape → (3)      b.shape → (4);
```

```
c = a[:,np.newaxis]      # c.shape → (3,1)
```

```
c * b      c.shape → (3,4)
```



# Universal functions: performance



- Iterations over array should be avoided, when possible, because they are inefficient.
- Performance using array arithmetic is better than using python loops

## Using ufunc

```
In [1]: x = np.arange(10**7)
```

```
In [2]: %timeit y = x*3
```

```
10 loops, best of 3: 76 ms per loop
```

## Using for loop

```
In [3]: %timeit y = [x*3 for x in xrange (10**7)]
```

```
1 loops, best of 3: 1.33 s per loop
```

**~ 18x speedup** (~ standard across NumPy)

# Efficiency note (1)



- NumPy functions are efficient only on arrays (but can work on scalars as well)
- i.e. `math` function `asin(x)` is 12 times faster than **numpy** `np.arcsin(x)` on a scalar `x`
- Therefore use **math** for **scalar arguments**

## Efficiency note (2)



- The operation

```
>>> b = 2*a + 2
```

can be done with in-place operations

```
>>> b = a      # b is a reference to a
```

```
>>> b *= 2
```

```
>>> b += 2
```

More efficient and doesn't require storing temp array

- These operations affect array "a" as well; we can avoid this with

```
>>> b = a.copy()
```

- Other in-place operations

```
>>> b += a      >>> b *= a      >>> b **= a
```

```
>>> b -= a      >>> b /= a
```

## Efficiency note (3):



- Let's consider the function

```
>>> def myfunc(x):           # if x is an array
>>>     if x<0:              # x<0 becomes a bool array
>>>         return 0
>>>     else:
>>>         return sin(x)
```

- If  $x<0$  results in a boolean array. Such array **cannot be evaluated by the if statement**

# Efficiency note (3): Vectorization



- We could use an xrange loop but it would result in a **SLOW** code

- We can vectorize our function using the function where

```
>>> def vec_myfunc(x):  
>>>     y = np.where(x<0, 0, np.sin(x))  
>>>     return y      # now returns an array
```

- `np.where(condition, x, y)` # condition → bool array  
# condition is true → x  
# condition is false → y

# NumPy universal functions (1)



- There are more than 60 ufuncs
- Some ufuncs are called automatically: i.e. `np.multiply(x,y)` is called internally when `a*b` is typed
- NumPy offers trigonometric functions, their inverse counterparts, and hyperbolic versions as well as the exponential and logarithmic functions. Here, a few examples:

```
>>> b = np.sin(a)
```

```
>>> b = np.arcsin(a)
```

```
>>> b = np.sinh(a)
```

```
>>> b = a**2.5 # exponential
```

```
>>> b = np.log(a)
```

```
>>> b = np.exp(a)
```

```
>>> b = np.sqrt(a)
```

# NumPy universal functions (2)



- Comparison functions:

**greater, less, equal, logical\_and/\_or/\_xor/\_nor, maximum, minimum, ...**

```
>>> a = np.array([2,0,3,5])
```

```
>>> b = np.array([1,1,1,6])
```

```
>>> np.maximum(a,b)
```

```
array([2, 1, 3, 6])
```

- Floating point functions:

**floor, ceil, isreal, iscomplex, ...**

# Exercise 5: universal functions



- Calculate the molecular center of mass

$$x_{cm} = \frac{1}{MW} \sum_i x_i m_i \quad y_{cm} = \frac{1}{MW} \sum_i y_i m_i \quad z_{cm} = \frac{1}{MW} \sum_i z_i m_i$$

- Calculate the moment of inertia tensor

$$I = \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix}$$

$$I_{11} = I_{xx} = \sum_i m_i (y_i^2 + z_i^2)$$

$$I_{22} = I_{yy} = \sum_i m_i (x_i^2 + z_i^2)$$

$$I_{33} = I_{zz} = \sum_i m_i (x_i^2 + y_i^2)$$

$$I_{12} = I_{xy} = -\sum_i m_i x_i y_i$$

$$I_{13} = I_{xz} = -\sum_i m_i x_i z_i$$

$$I_{23} = I_{yz} = -\sum_i m_i y_i z_i$$



## Exercise 5: universal functions



- Calculate the molecular center of mass

```
>>> xcm = 1./b['PA'].sum() * (b['PA']*b['x']).sum()
```

```
>>> ycm = 1./b['PA'].sum() * (b['PA']*b['y']).sum()
```

```
>>> zcm = 1./b['PA'].sum() * (b['PA']*b['z']).sum()
```

- Calculate the moment of inertia tensor

```
>>> Ixx = (b['PA']*(b['y']**2 + b['z']**2)).sum()
```

```
>>> Iyy = (b['PA']*(b['x']**2 + b['z']**2)).sum()
```

```
>>> Izz = (b['PA']*(b['x']**2 + b['y']**2)).sum()
```

```
>>> Ixy = -(b['PA']*b['x']*b['y']).sum()
```

```
>>> Ixz = -(b['PA']*b['x']*b['z']).sum()
```

```
>>> Iyz = -(b['PA']*b['y']*b['z']).sum()
```

```
>>> TI = np.array([[Ixx, Ixy, Ixz],[Ixy, Iyy, Iyz],  
[Ixz, Iyz, Izz]])
```

# **Standard classes inheriting from ndarray**

# Standard classes



- NumPy provides standard classes which **inherit** from the `ndarray` class or use `ndarray` as their internal structure
- Some of these classes are:
  - **Matrix objects**
  - **Record arrays**

# Matrix objects



There are some differences between ndarray and Matrix objects

- Matrix can be created using a Matlab-style notation

```
>>> a = np.matrix('1 2; 3 4')
```

```
>>> print a
```

```
[[1 2]
```

```
 [3 4]]
```

- Matrix are always **two-dimensional** objects
- Matrix objects over-ride multiplication to be **matrix-multiplication**, and over-ride power to be **matrix raised to a power**
- Matrix objects have higher **priority** than ndarrays. Mixed operations result therefore in a matrix object

# Matrix objects (2)



- `Matrix` objects have special attributes:
  - `.T` returns the transpose of the matrix
  - `.H` returns the conjugate transpose of the matrix
  - `.I` returns the inverse of the matrix
  - `.A` returns a view of the data of the matrix as a 2-d array

# Record arrays (1)



- Structured array example

```
>>> arr = np.array([(1.0, 2), (3.0, 4)], dtype=[('x',  
np.float), ('y', np.int)])  
  
array([(1.0, 2), (3.0, 4)],  
      dtype=[('x', '<f8'), ('y', '<i4')])
```

- To access array columns by name:

```
>>> arr['x']  
  
array([ 1.,  3.]
```

# Record arrays (2)



- Record array is a subclass of `ndarray` that just adds **attribute access** to the columns of a structured arrays

```
>>> rarr = arr.view(np.recarray)
```

```
>>> rarr.x
```

```
array([ 1.,  3.])
```

```
>>> rarr.y
```

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

# Basic modules



# Basic modules: linear algebra



- The `linalg` module is automatically imported typing `import numpy`
- It contains functions to solve **linear systems**, finding the **inverse** and the **determinant** of a matrix, and computing **eigenvalues** and **eigenvectors**

```
A = np.zeros((10,10))    # arrays initialization
x = np.arange(10)/2.0
for i in range(10):
...     for j in range(10):
...         A[i,j] = 2.0 + float(i+1)/float(j+i+1)
b = np.dot(A, x)
y = np.linalg.solve(A, b) # A*y=b → y=x
np.allclose(y,x) # True, mind the tolerance!
```

# Basic modules: linear algebra



- Eigenvalues can also be computed:

```
# eigenvalues only:
```

```
>>> A_eigenvalues = np.linalg.eigvals(A)
```

```
# eigenvalues and eigenvectors:
```

```
>>> A_eigenvalues, A_eigenvectors = np.linalg.eig(A)
```

## Exercise 6: matrices and linalg



(1) From the array  $TI$  of the last exercise build a matrix ( $mTI$ ), calculate eigenvalues ( $e$ ) and eigenvectors ( $\mathbf{E}\mathbf{v}$ ) and prove that

$$\mathbf{E}\mathbf{v}^T mTI \mathbf{E}\mathbf{v} = \begin{bmatrix} e_1 & 0 & 0 \\ 0 & e_2 & 0 \\ 0 & 0 & e_3 \end{bmatrix}$$

(2) Now, for example, you can rotate all the atoms coordinates by multiplying them by the  $\mathbf{E}\mathbf{v}$  matrix

# Basic modules: random numbers



- Calling the standard random module of Python in a loop to generate a sequence of random number is inefficient. Instead:

```
>>> np.random.seed(100)
```

```
>>> x = np.random.random(4)
```

```
array([ 0.89132195,  0.20920212,  0.18532822,  
        0.10837689])
```

```
>>> y = np.random.uniform(-1, 1, n) # n uniform  
numbers in interval (-1,1)
```

- This module provides more general distributions like the normal distribution

```
>>> mean = 0.0; stdev = 1.0
```

```
>>> u = np.random.normal(mean, stdev, n)
```

# Exercise 7: random numbers



- Given a game in which you win back 10 times your bet if the sum of 4 dice is less than 10, determine (brute force, like 100000 trows) wheter it is convenient to play the game or not

hint: use `np.random.randint` to create a bi-dimensional array of random integers with shape `(4, n)`



**Numpy: good to know**

# Masked arrays (1)



```
>>> x = np.r_[0:20].reshape(4,5) # 2d array
>>> x[2:,2:] = 2 # assignment
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11,  2,  2,  2],
       [15, 16,  2,  2,  2]])
>>> x[x<5] # x<5 bool array
array([0, 1, 2, 3, 2, 2, 2, 2, 2, 2]) # 1d array
```

- What if we want to maintain the original shape of the array
- We can use masked arrays: the **shape of the array is kept**
- Invalid data can be flagged and ignored in math computations



# Masked arrays (2)



```
>>> import numpy.ma as ma
>>> y = ma.masked_less(x, 4)
>>> masked_array(data =
  [[-- -- -- -- 4]
  [5 6 7 8 9]
  [10 11 -- -- --]
  [15 16 -- -- --]],
  mask =
  [[ True  True  True  True False]
  [False False False False False]
  [False False  True  True  True]
  [False False  True  True  True]],
  fill_value = 999999)
>>> y.sum(axis=0)          # → [30 33 7 8 13]
```

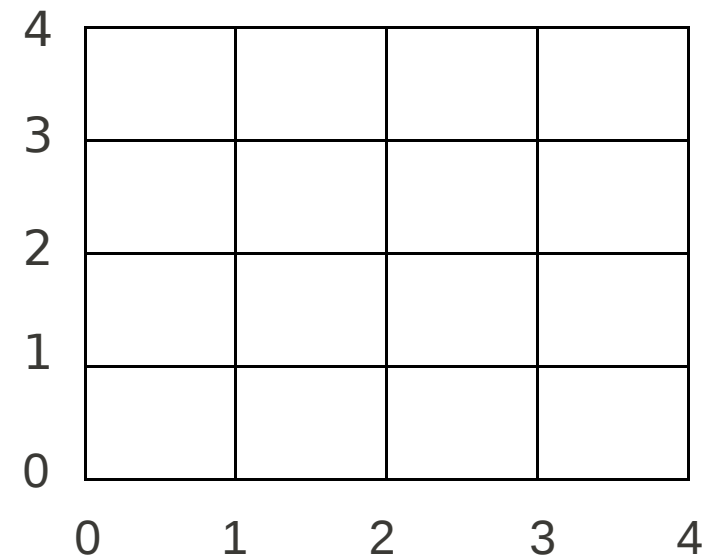
# Meshgrids (1)



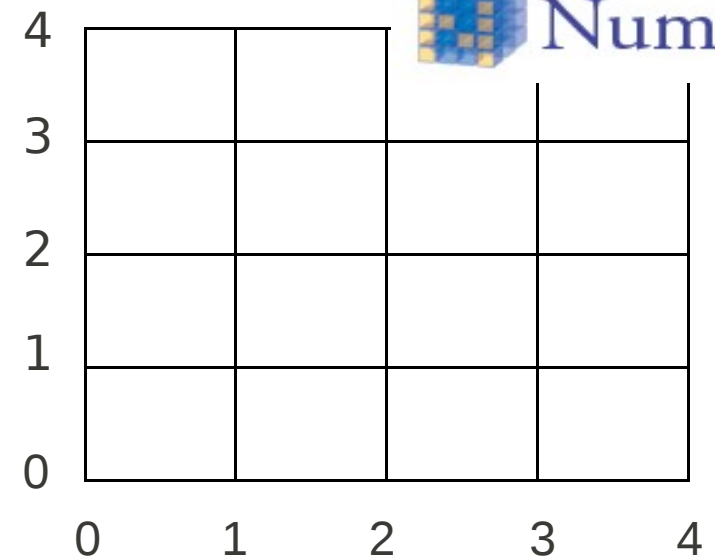
- Given a two-dimensional grid of points, namely  $a_{i,j} = (x_i, y_j)$  we want to calculate for each point of the grid the values of a function  $f(a_{i,j}) = f(x_i, y_j)$
- Suppose  $f = x/2 + y$
- $f$  is a ufunc (implicitly uses ufuncs: add, divide) so it operates element-by-element

```
>>> x = np.linspace(0,4,5)
array([ 0.,  1.,  2.,  3.,  4.])
>>> y = x.copy()
>>> def f(x,y): return x/2 + y
>>> values = f(x,y)
array([ 0. ,  1.5,  3. ,  4.5,  6. ])
```

**WRONG!**



# Meshgrids (2)



```
>>> x = np.linspace(0,4,5)
>>> y = x.copy()
>>> xv,yv = np.meshgrid(x,y)
```

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

```
>>> def f(x,y): return x/2 + y # still operates elementwise
>>> values = f(xv,yv)
```

# Bibliography



- <http://docs.scipy.org/doc/>
- Guide to NumPy (Travis E. Oliphant)
- NumPy User Guide
- Numpy Reference Guide
  
- Python Scripting for Computational Science (Hans Petter Langtangen)