030_Numpy

March 9, 2022

[1]:
```
from IPython.display import Image
```

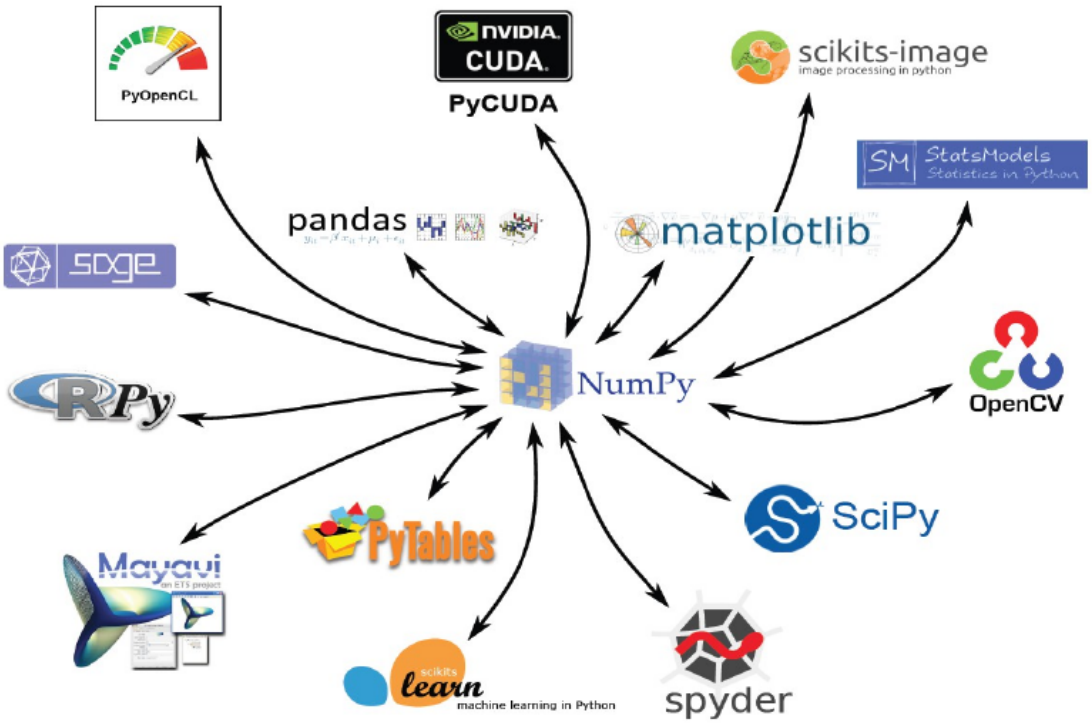
1 Numpy Basics

Section ??
Section ??
Section ??
Section ?? Section ??
Section ??
Section ??
Section ??
Section ??
Section ??
Section ??

2 Scientific Python Ecosystem

[2]:
```
Image(filename="eco.png")
```

[2]:



Numpy contains core routines for doing fast vector, matrix, and linear algebra-type operations in Python. Scipy contains additional routines for optimization, special functions, and other advanced tasks. Both contain modules written in C and Fortran and optionally linked to BLAS/LAPACK/MKL …; hence, that they're as fast as possible. Using them together, you can do with Python (roughly) most of the things you did with Matlab (Disclaimer: I am not doing a comparison; just, fields of application are the same).

In fact, if you're an experienced Matlab user, there's a guide to Numpy for Matlab users just for you.

Numpy and Scipy are the core of a wider scientific python ecosystem extending in many fields. In addition to those cited in the picture there are many other framework for specific fields such as:

- bioinformatics: Biopython
- computational chemistry and biophysics: MDAnalysis
- graphs and networks: NetworkX
- astrophysics: [Astropy][https://en.wikipedia.org/wiki/Astropy]

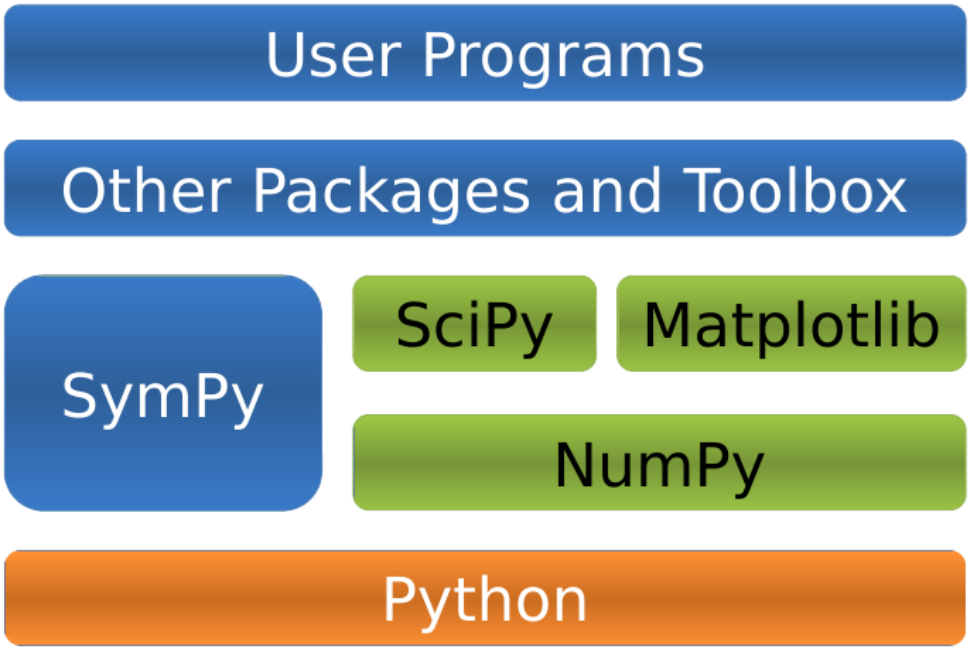
and many more. Not all of them support Python3.x and some may be based on specific distributions (e.g. Anaconda).

The success of Numpy as a numerical library is due to three main features (in addition of being Python):

- OOS
- Flexible definition of arrays and other objects, inspired from MatLab
- Upon building, may be compiled using user provided math libraries such as BLAS, LAPACK, ATLAS, FFTW, MKL. This yields **significant** speed improvement

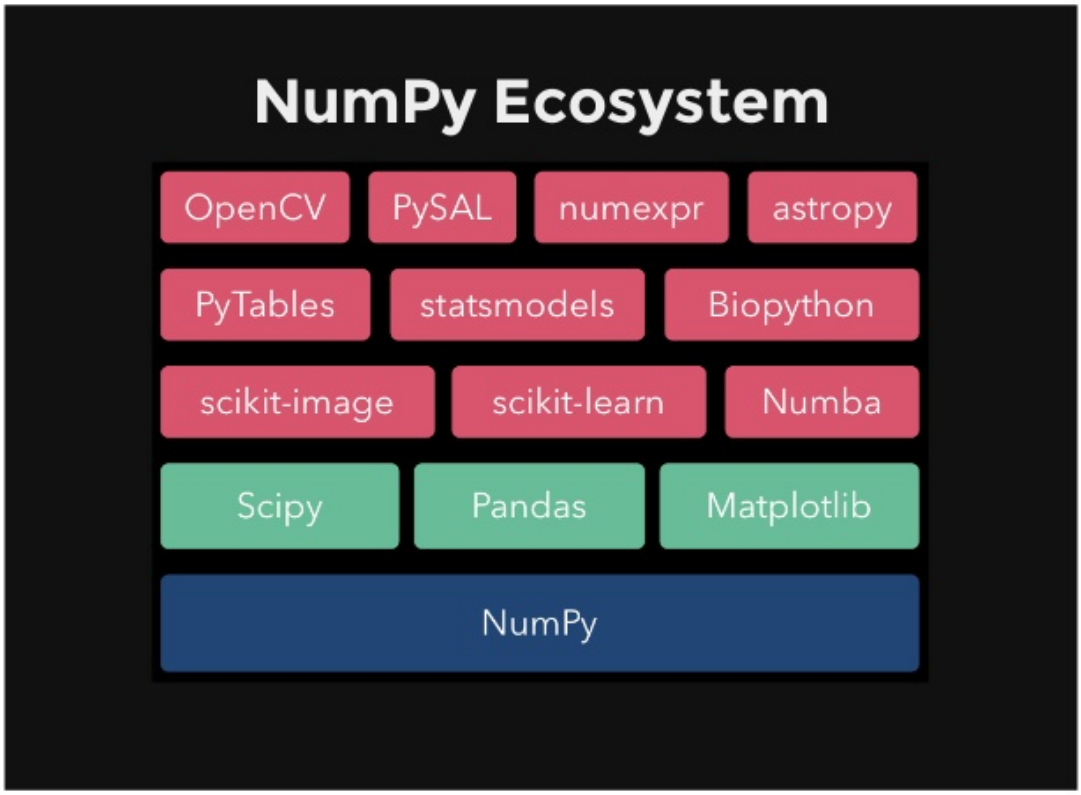
[3]:
```
Image(filename="stack.png")
```

[3]:



[4]:
```
Image(filename="pydata_stack.jpg")
```

[4]:



In addition to basic definitions, **Numpy** itself is divided in many submodules:

- **.core** and **.lib**: basic objects and utilities

- **.linalg**: Linear Algebra utilities
- **.fft**: (discrete) Fast Fourier Transform
- **.random**: Random number generator
- **.f2py**: Automated wrapping of fortran code. *Needed by Scipy*

3 Numpy Basics

3.0.1 Numpy arrays: the workhorse datatype of scientific Python

NumPy is a Python C extension library for array-oriented computing. The **numpy array** is the fundamental data type. It has some super awesome features:

1. Efficient.
2. In-memory Contiguous (or Strided)

A numpy array has some basic attributes, such as:

Size: number of elements in the array

Shape: integer tuple with number of elements for each dimension

3.0.2 Glossary

We have heard that ``everything in Python is an object''. For the time being, we consider an object as predefined or user defined data type with different *properties*.
E. g. an integer has the property of *value*. We call such properties **attributes**. The **size** and **shape** properties are thus **attributes**. An attribute of an object can be accessed with the . syntax:

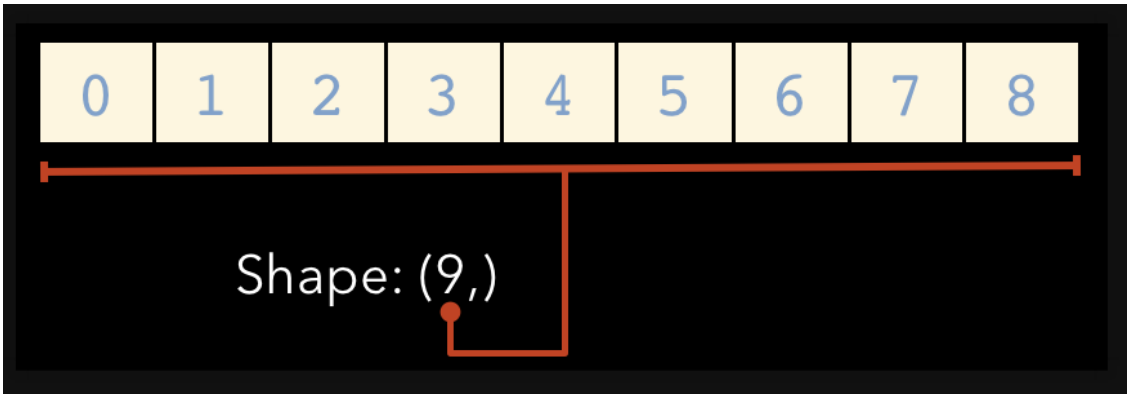
[5]:
```
a = dict()
a.keys()
```

[5]:
```
dict_keys([])
```

One dimensional arrays have a shape formed by a 1 tuple:

[6]:
```
Image(filename="size1.png")
```

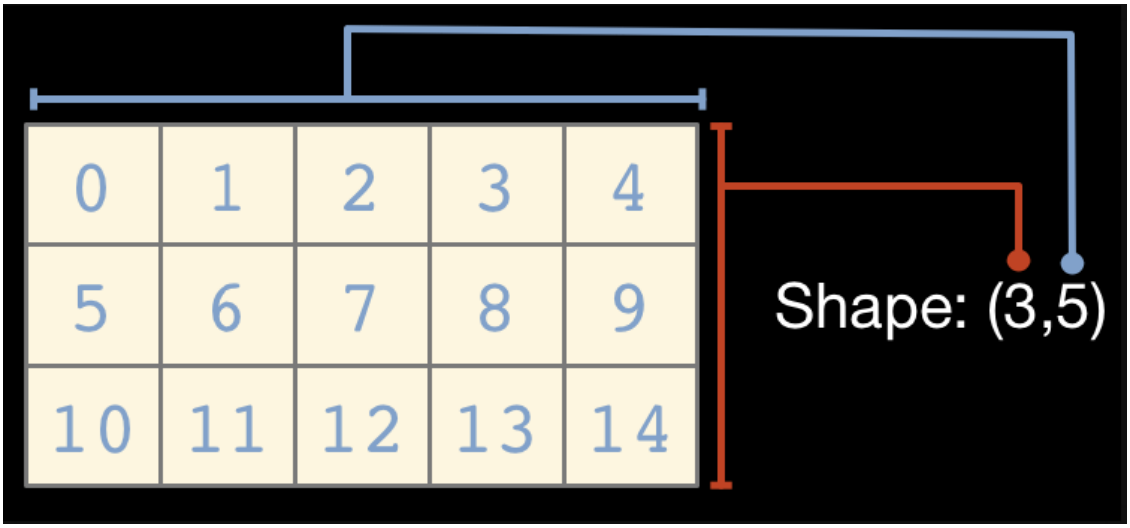
[6]:



Two dimensional …

[7]:
```
Image(filename="size2.png")
```

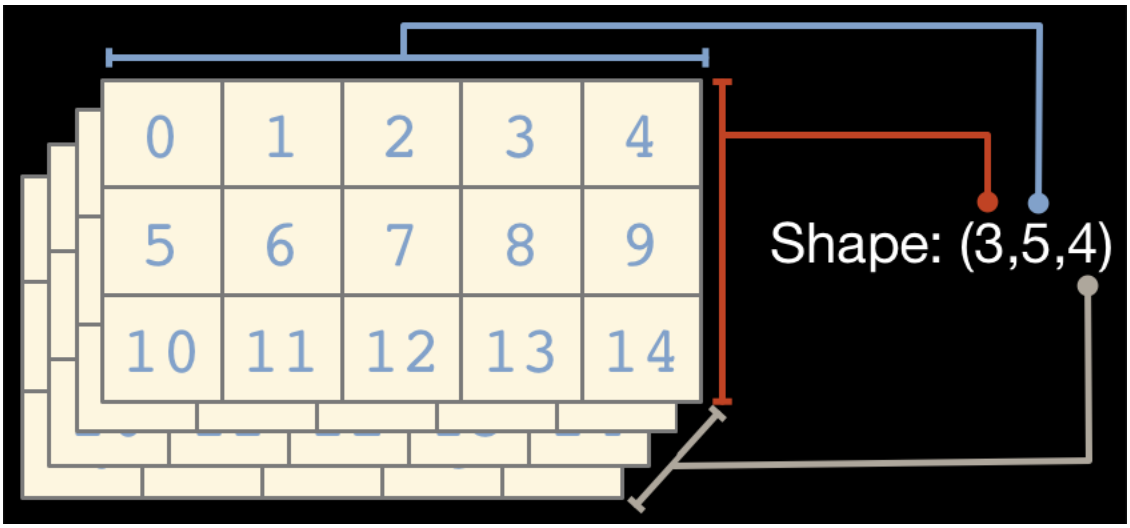
[7]:



Three dimensional …

[8]:
```
Image(filename="size3.png")
```

[8]:



Array types are:

[9]:
```
Image(filename="nptypes.png")
```

[9]:

Type	Description
bool	Boolean (True or False) stored as byte
int	Platform integer (normally either int32 or int64)
int8	Byte (-128, 127)
int16	Integer (-32768, 32767)
int32	Integer (-2147483648, 2147483647)
int64	Integer (-9223372036854775808, 9223372036854775807)
uint8	unsigned integer (0, 255)
uint16	unsigned integer (0, 65535)

[10]:
```
Image(filename="nptypes2.png")
```

[10]:

Type	Description
uint32	unsigned integer (0, 4294967295)
uint64	unsigned integer (0, 18446744073709551615)
float	float64
float16	Half precision float
float32	Single precision float
float64	Double precision float
complex	complex128
complex64	complex represented by 2 32-bits float
complex128	complex represented by 2 64-bits float

3.0.3 Array creation and manipulation

Let's create and manipulate some Numpy arrays:

[11]:
```
import numpy as np #a very common abbreviation
```

[12]:
```
a=np.array((1,2,3,4)); a
```

[12]:
```
array([1, 2, 3, 4])
```

[13]:
```
a=np.array([(1,2,3,4),(5,6,7,8)]); a
```

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

[14]:
```
a=a-10; a
```

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

[15]:
```
a=a*2; a
```

[15]:
```
array([[-18, -16, -14, -12],
       [-10,  -8,  -6,  -4]])
```

[16]:
```
a.size
```

[16]:
```
8
```

the **shape** attribute tells you the number of elements in each axis

[17]:
```
a.shape
```

[17]:
```
(2, 4)
```

[18]:
```
a.T-a.transpose(), a
```

[18]:
```
(array([[0, 0],
        [0, 0],
        [0, 0],
        [0, 0]]),
 array([[-18, -16, -14, -12],
        [-10,  -8,  -6,  -4]]))
```

a is not changed by viewing its transpose

[19]:
```
b=a.T
b
```

[19]:
```
array([[-18, -10],
       [-16,  -8],
       [-14,  -6],
       [-12,  -4]])
```

[20]:
```
a=-a
a
```

[20]:
```
array([[18, 16, 14, 12],
       [10,  8,  6,  4]])
```

[21]:
```
b
```

[21]:
```
array([[-18, -10],
       [-16,  -8],
       [-14,  -6],
       [-12,  -4]])
```

b is a new array

[22]:
```
b=a
b
```

[22]:
```
array([[18, 16, 14, 12],
       [10,  8,  6,  4]])
```

[23]:
```
a*b
```

[23]:
```
array([[324, 256, 196, 144],
       [100,  64,  36,  16]])
```

[24]:
```
try:
    a*b.T
except:
    print("this is not a scalar product")
```

```
this is not a scalar product
```

[25]:
```
np.dot(a,a.T)
```

[25]:
```
array([[920, 440],
       [440, 216]])
```

You can assign single elements or general attributes such as the array **shape**:

[26]:
```
a[1,1]=1111
a
```

[26]:
```
array([[  18,   16,   14,   12],
       [  10, 1111,    6,    4]])
```

[27]:
```
a.shape
```

[27]:
```
(2, 4)
```

[28]:
```
a.shape = (4,2)
```

[29]:
```
a
```

[29]:
```
array([[  18,   16],
       [  14,   12],
       [  10, 1111],
       [   6,    4]])
```

Easy creation of useful arrays

[30]:
```
np.zeros((3,3),'d')
```

[30]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[31]:
```
np.ones((3,3))
```

[31]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

[32]:
```
np.ones((3,3),dtype="int32")
```

[32]:
```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

[33]:
```
np.eye(3)
```

[33]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

if we want to specify boundaries or intervals:

[34]:
```
np.arange(1,10)
```

[34]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[35]:
```
np.arange(2.3,3.3,.1)
```

[35]:
```
array([2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. , 3.1, 3.2])
```

[36]:
```
np.linspace(2.3,3.3,10)
```

[36]:
```
array([2.3       , 2.41111111, 2.52222222, 2.63333333, 2.74444444,
       2.85555556, 2.96666667, 3.07777778, 3.18888889, 3.3       ])
```

[37]:
```
np.linspace(2.3,3.3,10,False)
```

[37]:
```
array([2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. , 3.1, 3.2])
```

[38]:
```
a=np.array(list(map(bool,(0,1,0,1,1,0,1))))
a
```

[38]:
```
array([False,  True, False,  True,  True, False,  True])
```

Often, you will not create an array element by element; **zeros**, **ones**, **eye**, **arange** and **linspace** provide simple tools for creating arrays.

What if you want to load a table from a text file?

[39]:
```
%%bash
awk 'BEGIN{print 0,1,2;print 3,4,5;print 6,7,8}' > table.txt
head table.txt
```

```
0 1 2
3 4 5
6 7 8
```

[40]:
```
np.loadtxt("table.txt")
```

[40]:
```
array([[0., 1., 2.],
       [3., 4., 5.],
       [6., 7., 8.]])
```

3.1 Exercises

1. Become familiar with array creation. Read the documentation of **np.savetxt()** and try it. If some of you uses spreadsheets and want to import some data, have a look at the **csv** module.

2. Be sure to understand when to use **arange** or **linspace**.

3. Read the documentation and experiment with **hstack** and **vstack** in conjunction with **savetxt**. Disclaimer: I never remember which is which.

4. Create a null vector of size 10 but the fifth value which is 1.

5. Find what requisite a tuple must have to be used to redefine the shape of an existing array.

[41]:
```
# Solution 4
a = np.zeros(10,dtype='int')
a[4] = 1
a
```

[41]:
```
array([0, 0, 0, 0, 1, 0, 0, 0, 0, 0])
```

4 Array indexing and slicing

Having created arrays we may need to apply operations to some specific elements, i.e. we have to refer to specific elements in the array

[42]:
```
a=np.arange(5)
a
```

[42]:
```
array([0, 1, 2, 3, 4])
```

[43]:
```
a[1]
```

[43]:
```
1
```

[44]:
```
a[-1]
```

[44]:
```
4
```

[45]:
```
a[1:]
```

[45]:
```
array([1, 2, 3, 4])
```

[46]:
```
a[-1::-1]
```

[46]:
```
array([4, 3, 2, 1, 0])
```

Iteration is on the first axis:

[47]:
```
a[0]
```

[47]:
```
0
```

[48]:
```
for i in range(3): print(a[3-i-1])
```

```
2
1
0
```

Indexing is very similar to lists; the order of iteration corresponds to the order of dimensions in the array specification. *But* multiple elements in the selection are used for different dimensions.

[49]:
```
print(a)
try:
    a[0,1]
except IndexError as e:
    print(e)
```

```
[0 1 2 3 4]
too many indices for array: array is 1-dimensional, but 2 were indexed
```

[50]:
```
try:
    a[0][1]
except Exception as e:
    print(e)
```

```
invalid index to scalar variable.
```

[51]:
```
a = np.arange(6); a.shape = (2,3)
print(a[0,1],a[0][1])
```

```
1 1
```

[52]:
```
a[0,1:], len(a)
```

[52]:
```
(array([1, 2]), 2)
```

The general syntax is:

i:j:k

with:

- i=start (defaults to zero, i.e. first element on axis)
- j=end (defaults to a.shape[n] for axis n)
- k=stride (defaults to 1, i.e. all elements).

Ellipsis, or … is expanded as :,:, up to the number of axes not explicitly set.

[53]:
```
a[...,2]
```

[53]:
```
array([2, 5])
```

Other basic utilities save you a lot of time:

[54]:
```
np.max(a)
```

[54]:
```
5
```

[55]:
```
np.mean(a)
```

[55]: 2.5

you can specify only a part of an array:

[56]:
```
np.mean(a[0])
```

[56]: 1.0

Array dimensions are listed in **axes**. The **axis** keyword allows to specify rows or columns. Many numpy methods do reduction operations (e. g. a sum) over one or more axes. The default is **axis=None** hence the reduction is over all elements (the array becomes a scalar).

[57]:
```
a = np.arange(10)
a.shape = (2,5)
a
```

[57]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

[58]:
```
a.sum()
```

[58]: 45

Reduction on first (0) dimension is done with **axis=0**:

[59]:
```
a.sum(axis=0)
```

[59]: array([5, 7, 9, 11, 13])

Reduction on second (1) dimension is done with **axis=1**:

[60]:
```
print(a)
a.mean(axis=1), a.std(axis=1)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

[60]: (array([2., 7.]), array([1.41421356, 1.41421356]))

4.1 Exercises

1. Create a 2x2x2 array with random values. Use introspection.

2. Create an array whose first row goes from 0 to 100 (closed) with a stride of 4 and the second from 200 to 100 with stride 4.

3. Create a 4x4 random matrix with values in [0,100]. Then transform it so that values have mean 0 and std. dev. 1

4.1.1 Hints

Section ??

[61]:
```
# Solution 1
A = np.random.random((2,2,2))
A
```

[61]:
```
array([[[0.44259195, 0.66238183],
        [0.13984184, 0.78947311]],

       [[0.99437303, 0.93465712],
        [0.68629323, 0.67812431]]])
```

[62]:
```
# Solution 2
a = np.arange(1,201,4)
a[25:] = a[-1:24:-1]
a.shape = (2,25)
a
```

[62]:
```
array([[  1,   5,   9,  13,  17,  21,  25,  29,  33,  37,  41,  45,  49,
         53,  57,  61,  65,  69,  73,  77,  81,  85,  89,  93,  97],
       [197, 193, 189, 185, 181, 177, 173, 169, 165, 161, 157, 153, 149,
        145, 141, 137, 133, 129, 125, 121, 117, 113, 109, 105, 101]])
```

[63]:
```
# solution 3
A = np.random.random((4,4))
mu,sigma = A.mean(), A.std()
A = (A-mu)/sigma
print(A.mean(), A.std())
A
```

```
-1.8041124150158794e-16 0.9999999999999999
```

[63]:
```
array([[-0.61945769, -1.7119858 , -0.14910592,  1.43372074],
       [-0.22031101,  1.38224339, -1.21520111, -0.3762895 ],
       [-1.04118657,  0.62816905,  0.37214917,  0.23016827],
       [ 1.95615416, -0.18001175,  0.58657956, -1.07563497]])
```

5 View and copy

[64]:
```
a=np.arange(25)
a.shape = (5,5)
a
```

[64]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
```

```
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

[65]:
```
b=a[1]; b
```

[65]:
```
array([5, 6, 7, 8, 9])
```

[66]:
```
b=b*10; b
```

[66]:
```
array([50, 60, 70, 80, 90])
```

[67]:
```
a
```

[67]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

[68]:
```
b=a[1]; b[:] = 100; b
```

[68]:
```
array([100, 100, 100, 100, 100])
```

[69]:
```
a
```

[69]:
```
array([[  0,   1,   2,   3,   4],
       [100, 100, 100, 100, 100],
       [ 10,  11,  12,  13,  14],
       [ 15,  16,  17,  18,  19],
       [ 20,  21,  22,  23,  24]])
```

What happened? when you have created **b** in the first place, it was a **view** or **reference** to some elements in a; assigning it to a new variable caused the **unbinding** of b from a.

But just reassiging all the elements was done on the view of a.

Each operation basic slicing operation yields a view of the target array or (in C terms) creates a reference to a section of the target array even if they have different ids.

A view is an array that does not own its data, but refers to another array's data instead.

[70]:
```
id(a)
```

[70]:
```
140182148306480
```

[71]:
```
b=a[:,2]; print(id(b)); b
```

```
140182148308400
```

[71]:
```
array([  2, 100,  12,  17,  22])
```

[72]:
```
b=a[1].copy()
b
```

[72]:
```
array([100, 100, 100, 100, 100])
```

[73]:
```
b=b/10
b
```

[73]:
```
array([10., 10., 10., 10., 10.])
```

[74]:
```
a
```

[74]:
```
array([[  0,   1,   2,   3,   4],
       [100, 100, 100, 100, 100],
       [ 10,  11,  12,  13,  14],
       [ 15,  16,  17,  18,  19],
       [ 20,  21,  22,  23,  24]])
```

Selection works on both axes at the same time; *a is not an array of arrays.*

[75]:
```
a=np.array(range(25))
a.shape=(5,5)
a
```

[75]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

[76]:
```
a[1:4:2,1:4:2]
```

[76]:
```
array([[ 6,  8],
       [16, 18]])
```

Summary of slicing operations:

[77]:
```
Image(filename="slice.png")
```

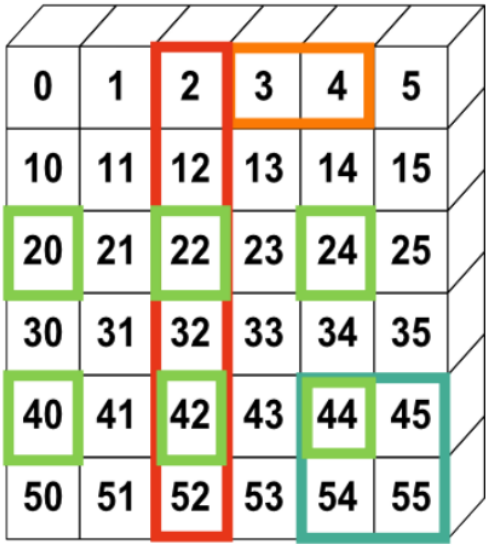
[77]:

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



5.1 Exercises

1. Create the array below (not element-wise), then extract the 2 nd and 4th row:

   ```
   [10, 65, 110],
   [21, 76, 121],
   [32, 87, 132],
   [43, 98, 143],
   [54, 109, 154]]
   ```

2. Calculate the mean on the 1st and 3rd column.

3. Declare a 8x8 matrix and fill it with a checkerboard patern, like this:

   ```
   [[0,1],
   [1,0]]
   ```

5.1.1 Hints

1,2 -- Use the reshape and mean methods. On one line you can build more than one array on the fly.

3 -- Start with a white board a use slicing to colour it.

[78]:
```
#Solution 1,2:
a = np.array([i for i in range(1,16)])*10 + np.array([i%10 for i in range(15)])
a
```

[78]:
```
array([ 10,  21,  32,  43,  54,  65,  76,  87,  98, 109, 110, 121, 132,
       143, 154])
```

[79]:
```
a.shape = (5,3)
a.shape
```

[79]:
```
(5, 3)
```

[80]:
```
a
```

[80]:
```
array([[ 10,  21,  32],
       [ 43,  54,  65],
       [ 76,  87,  98],
       [109, 110, 121],
       [132, 143, 154]])
```

[81]:
```
np.mean(a[:,0])
```

[81]:
```
74.0
```

[82]:
```
#Solution 3
Z = np.zeros((8,8))
Z
```

[82]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])
```

[83]:
```
#start with odd rows
Z[::2,1::2] = 1
Z
```

[83]:
```
array([[0., 1., 0., 1., 0., 1., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0., 1., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0., 1., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0., 1., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])
```

[84]:
```
Z[1::2,::2] = 1
Z
```

[84]:
```
array([[0., 1., 0., 1., 0., 1., 0., 1.],
       [1., 0., 1., 0., 1., 0., 1., 0.],
       [0., 1., 0., 1., 0., 1., 0., 1.],
       [1., 0., 1., 0., 1., 0., 1., 0.],
       [0., 1., 0., 1., 0., 1., 0., 1.],
       [1., 0., 1., 0., 1., 0., 1., 0.],
       [0., 1., 0., 1., 0., 1., 0., 1.],
       [1., 0., 1., 0., 1., 0., 1., 0.]])
```

6 Need for speed?

Python carries you in relative confort where you want

[85]:
```
Image(filename="camel.jpg")
```

[85]:

Is that enough?

No. As mentioned in the first slide it is not enough to made it feasible even for basic scientific computing. We want something with this performance:

[86]:
```
Image(filename="cheetah.jpg")
```

[86]:

[87]:
```
def listcomp():
    [i**2 for i in range(1000)]
```

[88]:
```
def usenumpy():
    a = np.arange(1000)
    b=a**2
```

[89]:
```
%timeit listcomp()
```

```
396 µs ± 1.92 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[90]:
```
%timeit usenumpy()
```

```
7.42 µs ± 34.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

numpy is about 20 times faster … but maybe not a cheetah in a future class we will how much fast we can run

7 Universal Functions

Numpy provides **Universal Functions** (ufunc) that operate on all elements of an array without using explicit loops in the interpreter, yielding a huge performance gain.

They are actually *wrappers* written directly in C or Fortran. arange, for instance, is one such *ufunc*. When typing simple operations between arrays, you

actually use the corresponding ufunc:

a*b is np.multiply(a,b)

there *ufunc* for many common tasks such as sum, mean (you already used it), floor, …, see the Section ??

[91]:
```
a = np.random.random(10)
a
```

[91]:
```
array([0.8484555 , 0.38287014, 0.09648073, 0.62925162, 0.9915193 ,
       0.67763188, 0.31550309, 0.16953502, 0.12814705, 0.76055568])
```

[92]:
```
np.square(a)
```

[92]:
```
array([0.71987674, 0.14658954, 0.00930853, 0.39595761, 0.98311052,
       0.45918496, 0.0995422 , 0.02874212, 0.01642167, 0.57844494])
```

[93]:
```
np.exp(a)
```

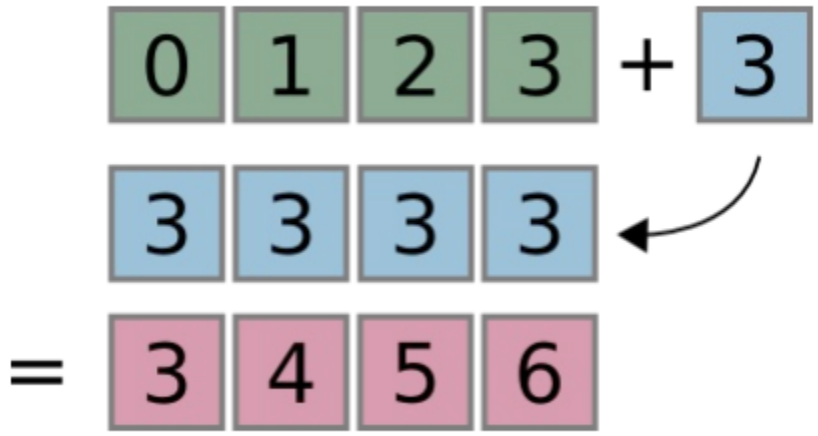
[93]:
```
array([2.33603606, 1.46648758, 1.10128836, 1.87620595, 2.69532637,
       1.96920888, 1.37094885, 1.18475383, 1.13672014, 2.13946474])
```

8 Broadcasting

Operations between numpy arrays are easy to manage as long as the arrays have the same shape: they take place elementwise.
However, arrays with diferent shape may be combined as long as is possible to convert them into new arrays with the same shape. This is called **broadcasting** and a **broadcasting error** is raised when not possible.

[94]:
```
Image(filename="broad.png")
```

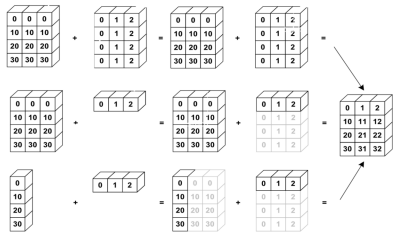
[94]:



Broadcasting takes place for each array involved in the operation, expanding the smaller axis dimension to match that of the other array. *Expand* means repeat the array over that axis.

[95]:
```
Image(filename="broad2.png")
```

[95]:



[96]:
```
a = np.array([1,2,3])
b = np.array([[1,2,3],[4,5,6]])
c = a + b
print(a.shape,b.shape,c.shape)
c
```

```
(3,) (2, 3) (2, 3)
```

[96]:
```
array([[2, 4, 6],
       [5, 7, 9]])
```

[97]:
```
a = np.arange(6)
a = a.reshape((2,1,3))
print(a.shape)
a
```

```
(2, 1, 3)
```

[97]:
```
array([[[0, 1, 2]],

       [[3, 4, 5]]])
```

[98]:
```
b = np.arange(8)
b = b.reshape((2,4,1))
print(b.shape)
b
```

```
(2, 4, 1)
```

[98]:
```
array([[[0],
        [1],
        [2],
        [3]],

       [[4],
        [5],
        [6],
        [7]]])
```

[99]:
```
c = a + b
print(c.shape)
c
```

```
(2, 4, 3)
```

[99]:
```
array([[[ 0,  1,  2],
        [ 1,  2,  3],
        [ 2,  3,  4],
        [ 3,  4,  5]],

       [[ 7,  8,  9],
        [ 8,  9, 10],
        [ 9, 10, 11],
        [10, 11, 12]]])
```

Broadcasting is not always possible:

[100]:
```
a = np.arange(15).reshape(((3,5))) #reshape method
a.shape
```

[100]:
```
(3, 5)
```

[101]:
```
b = np.arange(7)
b.shape
```

[101]:
```
(7,)
```

[102]:
```
try:
    print(a+b)
except ValueError as e:
    print(e)
```

```
operands could not be broadcast together with shapes (3,5) (7,)
```

8.1 Exercise

Consider an array of dimension (4,4,2), how to mulitply it by an array with dimensions (4,4). Hint: read the documentation of **np.newaxis**

[103]:
```
A = np.ones((4,4,2))
B = np.random.random(16); B.shape = (4,4)
```

[104]:
```
try:
    A*B
except ValueError as e:
    print(e)
```

operands could not be broadcast together with shapes (4,4,2) (4,4)

[105]:
```
A*B[:,:,np.newaxis]
```

[105]:
```
array([[[0.41318343, 0.41318343],
        [0.21881177, 0.21881177],
        [0.20163238, 0.20163238],
        [0.63585193, 0.63585193]],

       [[0.71030447, 0.71030447],
        [0.79367304, 0.79367304],
        [0.50266481, 0.50266481],
        [0.13772021, 0.13772021]],

       [[0.00818023, 0.00818023],
        [0.56128662, 0.56128662],
        [0.71375525, 0.71375525],
        [0.63420334, 0.63420334]],

       [[0.19361677, 0.19361677],
        [0.59183153, 0.59183153],
        [0.24574404, 0.24574404],
        [0.71320434, 0.71320434]]])
```

9 Advanced indexing

The syntax [start:stop:step] creates a slice object that can be used for *basic slicing*. Basic slicing can also be performed with **non array** sequences such as lists or tuples:

[106]:
```
a = np.arange(25)
a.shape = (5,5)
a
```

[106]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

[107]:
```
b = list((1,2,3))
c = a[b]
a[:] -= 1
c
```

[107]:
```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

[108]:
```
id(a),id(c)
```

[108]:
```
(140182843453040, 140182843453136)
```

[109]:
```
a
```

[109]:
```
array([[-1,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  8],
       [ 9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23]])
```

[110]:
```
c
```

[110]:
```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

Indexing can also be performed using other numpy arrays (not lists or tuples!). This however creates **copies** of arrays, **not views** !!

[111]:
```
a=np.array(range(-8,8))
a.shape=(4,4)
a
```

[111]:
```
array([[-8, -7, -6, -5],
       [-4, -3, -2, -1],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7]])
```

[112]:
```
b = np.array(b); b
```

[112]:
```
array([1, 2, 3])
```

c is a copy:

[113]:
```
c = a[b]
a *= 10
c
```

[113]:
```
array([[-4, -3, -2, -1],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7]])
```

[114]:
```
a
```

[114]:
```
array([[-80, -70, -60, -50],
       [-40, -30, -20, -10],
       [  0,  10,  20,  30],
       [ 40,  50,  60,  70]])
```

You can use multiple slicing objects in combination:

[115]:
```
a=np.array(range(-8,8))
a.shape=(4,4)
a
```

[115]:
```
array([[-8, -7, -6, -5],
       [-4, -3, -2, -1],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7]])
```

[116]:
```
a[[1,2,3],1]
```

[116]:
```
array([-3,  1,  5])
```

In addition to slicing it is possible to refer to a selection of elements in a numpy array using another array as a pattern or mask.

[117]:
```
a = np.array(range(-8,8))
a.shape = (4,4)
```

[118]:
```
c = a > 0
c
```

[118]:
```
array([[False, False, False, False],
       [False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
```

[119]:
```
g0 = a[c]
a += 1
```

```
g0
```

[119]: array([1, 2, 3, 4, 5, 6, 7])

with multiple slicing lists, elements are evaluated at the same time along all axes;

to take element (1,1) or (2,2) or (3,3) (note how order is dictated by the slicing lists):

```
[120]: a[[3,2],[3,2]]
```

[120]: array([8, 3])

All elements have to be **explicitly** selected. To obtain the lower right 2x2 array:

```
[121]: a[[2,3]][:,[2,3]]
```

[121]: array([[3, 4],
       [7, 8]])

We can also use a *list of lists*

```
[122]: rows = [[2],[3]]
cols = [2,3]
a[rows,cols]
```

[122]: array([[3, 4],
       [7, 8]])

this is like taking

rows[0]*cols[0]

etc … But rows[0] is a list, or, has shape (1,), thus cols is expanded from a scalar to [2], with shape (1,); instead of taking element (2,2) we take row 2 by column 2.

Numpy provides a huge number of functions and/or array methods that allow to directly search, sort or reduce the arrays. These methods ofen call directly optimized C code and are fastter than native Python code.

```
[123]: a = np.arange(100)
a.shape = (10,10)
a = a[1,::-1]
a
```

[123]: array([19, 18, 17, 16, 15, 14, 13, 12, 11, 10])

```
[124]: np.argsort(a)
```

[124]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

[125]:
```
b =  np.argsort(a)
a[b]
```

[125]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

numpy.where

[126]:
```
a = np.arange(-8,8)
np.where(a>0)
```

[126]:
```
(array([ 9, 10, 11, 12, 13, 14, 15]),)
```

[127]:
```
a.shape = (4,4)
np.where(a>0)
```

[127]:
```
(array([2, 2, 2, 3, 3, 3, 3]), array([1, 2, 3, 0, 1, 2, 3]))
```

np.where returns a *tuple of arrays* satisfying the condition over all the axes given.

9.1 Exercise

The data in populations.txt describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years.

Compute and print, based on the data in populations.txt:
1. The mean and std of the populations of each species for the years in the period.
2. Which year each species had the largest population.
3. Which species has the largest population for each year.
4. Which years any of the populations is above 500000.

9.1.1 Hints

1 -- Use np.mean() and np.std()

2,3 -- np.argmax(), advanced indexing

4 -- comparisons and np.any

[128]:
```
data = np.loadtxt('populations.txt')
year, hares, lynxes, carrots = data.T
populations = data[:,1:]
print(" Hares, Lynxes, Carrots")
print("Mean:", populations.mean(axis=0))
print("Std:", populations.std(axis=0))
```

```
 Hares, Lynxes, Carrots
Mean: [34080.95238095 20166.66666667 42400.        ]
Std: [20897.90645809 16254.59153691  3322.50622558]
```

[129]:
```
j_max_years = np.argmax(populations, axis=0)
#this is fancy indexing
print("Max. year:", year[j_max_years])
```

```
Max. year: [1903. 1904. 1900.]
```

[130]:
```
max_species = np.argmax(populations, axis=1)
species = np.array(['Hare', 'Lynx', 'Carrot'])
print("Max species:")
print(year)
print(species[max_species])
above_50000 = np.any(populations > 50000, axis=1)
print("Any above 50000:", year[above_50000])
```

```
Max species:
[1900. 1901. 1902. 1903. 1904. 1905. 1906. 1907. 1908. 1909. 1910. 1911.
 1912. 1913. 1914. 1915. 1916. 1917. 1918. 1919. 1920.]
['Carrot' 'Carrot' 'Hare' 'Hare' 'Lynx' 'Lynx' 'Carrot' 'Carrot' 'Carrot'
 'Carrot' 'Carrot' 'Carrot' 'Hare' 'Hare' 'Hare' 'Lynx' 'Carrot' 'Carrot'
 'Carrot' 'Carrot' 'Carrot']
Any above 50000: [1902. 1903. 1904. 1912. 1913. 1914. 1915.]
```

10 Masked arrays

Often, we have to deal with missing data or run over an array several times, each time picking up a part of it. One way of handling this is to fill the array with **NaN**, and then write functions that ignores **NaNs**. For instance, we might want to sum over all not NaN. Numpy already provides:

nanmean, nanmax, nanmin, nanargmax, nanargmin, nanstd, nanvar, nansum

another approach is possible by using masked arrays:

[131]:
```
import numpy.ma as MA
a = np.arange(-8,8); a.shape = (4,4)
mask = a%2==0
a_masked = MA.array(a,mask=mask)
print("Shape: ",a_masked.shape)
a_masked
```

```
Shape:  (4, 4)
```

[131]:
```
masked_array(
  data=[[--, -7, --, -5],
        [--, -3, --, -1],
```

```
       [--, 1, --, 3],
       [--, 5, --, 7]],
  mask=[[ True, False,  True, False],
       [ True, False,  True, False],
       [ True, False,  True, False],
       [ True, False,  True, False]],
  fill_value=999999)
```

To deal only with *valid* values:

[132]:
```
a_masked.compressed()
```

[132]:
```
array([-7, -5, -3, -1,  1,  3,  5,  7])
```

To get the mask:

[133]:
```
MA.getmaskarray(a_masked)
```

[133]:
```
array([[ True, False,  True, False],
       [ True, False,  True, False],
       [ True, False,  True, False],
       [ True, False,  True, False]])
```

Note that operating with masked array gives you back a masked array:

[134]:
```
b = a_masked-np.ones(a_masked.shape[0])*10
b
```

[134]:
```
masked_array(
  data=[[--, -17.0, --, -15.0],
       [--, -13.0, --, -11.0],
       [--, -9.0, --, -7.0],
       [--, -5.0, --, -3.0]],
  mask=[[ True, False,  True, False],
       [ True, False,  True, False],
       [ True, False,  True, False],
       [ True, False,  True, False]],
  fill_value=999999)
```

[135]:
```
type(b)
```

[135]:
```
numpy.ma.core.MaskedArray
```

10.1 Exercise

Build a boolean vector and a square matrix with trailing dimension equal to the vector's lenght'. Then calculate the maximum sum along rows for each row. Finally, find the row number that yields the maximum sum. The sum should

consider only valid (non False) elements in rows and columns; valid elements are determined by matching the index in the vector.

[136]:
```
N = 100
vec = np.random.random(N) >= 0.5
print(vec.shape)
vec[:10]
```

```
(100,)
```

[136]:
```
array([False, False, False,  True,  True, False, False,  True, False,
       False])
```

[137]:
```
mat = np.random.random(N*N)
mat.shape = (N,N)
mat[0,:5]
```

[137]:
```
array([0.51595797, 0.74222418, 0.78336248, 0.45175504, 0.30532187])
```

[138]:
```
vect = vec[np.newaxis,:].T
print(vect.shape)
square_mask = vec*vect
square_mask.shape
```

```
(100, 1)
```

[138]:
```
(100, 100)
```

[139]:
```
mat_masked = MA.array(mat,mask=square_mask)
sum_max = np.max(mat_masked.sum(axis=1))
where_max = mat_masked.sum(axis=1).argmax()
sum_max, where_max, mat_masked[where_max].sum()
```

[139]:
```
(55.33545086355203, 46, 55.33545086355203)
```

11 Matrix Solvers

You can solve systems of linear equations using the solve command from the linear algebra submodule:

[140]:
```
import numpy.linalg as LA
```

[141]:
```
A = np.array([[1,1,1],[0,2,5],[2,5,-1]])
b = np.array([6,-4,27])
LA.solve(A,b)
```

[141]:
```
array([ 5.,  3., -2.])
```

There is a number of routines to compute eigenvalues and eigenvectors

- **eigvals** returns the eigenvalues of a matrix
- **eigvalsh** returns the eigenvalues of a Hermitian matrix
- **eig** returns the eigenvalues and eigenvectors of a matrix
- **eigh** returns the eigenvalues and eigenvectors of a Hermitian matrix.

[142]:
```
A = np.array([[13.,-4.],[-4.,7.]])
LA.eigvalsh(A)
```

[142]:
```
array([ 5., 15.])
```

[143]:
```
LA.eigh(A)
```

[143]:
```
(array([ 5., 15.]),
 array([[-0.4472136 , -0.89442719],
        [-0.89442719,  0.4472136 ]]))
```

12 Numpy internals

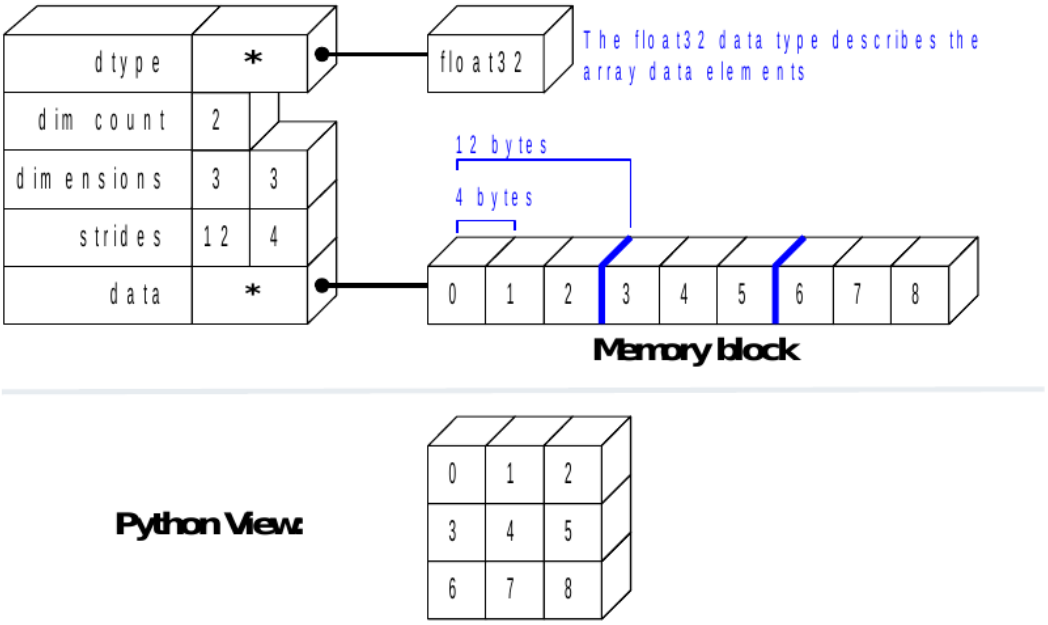
From the documentation:

NumPy arrays consist of two major components, the raw array data (from now on, referred to as the data buffer), and the information about the raw array data. The data buffer is typically what people think of as arrays in C or Fortran, a contiguous (and fixed) block of memory containing fixed sized data items. NumPy also contains a significant set of data that describes how to interpret the data in the data buffer. This extra information contains (among other things):

- The basic data element's size in bytes
- The start of the data within the data buffer (an offset relative to the beginning of the data buffer).
- The number of dimensions and the size of each dimension
- The separation between elements for each dimension (the `stride'). This does not have to be a multiple of the element size
- The byte order of the data (which may not be the native byte order)
- Whether the buffer is read-only
- Information (via the dtype object) about the interpretation of the basic data element. The basic data element may be as simple as a int or a float, or it may be a compound object (e.g., struct-like), a fixed character field, or- - Python object pointers.
- Whether the array is to interpreted as C-order or Fortran-order.

[144]:
```
Image(filename="array.png")
```

[144]:



Ndarray definition in numpy/core/include/numpy/ndarrayobject.h:

```
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data; // pointer to raw data buffer
    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;
    PyArray_Descr *descr; /* Pointer to type structure */
    int flags; /* Flags describing array
     */
    PyObject *weakreflist; /* For weakreferences */
} PyArrayObject;
```

13 The End!