intro_numpy

June 1, 2016

```
In [1]: # Just to know last time this was run:
        import time
        print time.ctime()
```

```
Wed Jun  1 16:51:54 2016
```

1 B Numpy

This is part of the Python lecture given by Christophe Morisset at IA-UNAM. More informations at: http://python-astro.blogspot.mx/

1.0.1 Import numpy first

```
In [2]: # You need first to import the numpy library (must be installed on your con
        # As it will be widely used, better to give it a nickname, or an alias. Tra
        import numpy as np
```

```
In [3]: print np.__version__
```

```
1.11.0
```

1.0.2 Tutorials

http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-2-Numpy.ipynb AND http://nbviewer.ipython.org/gist/rpmuller/5920182 AND http://www.astro.washington.edu/users/vanderplas/Astr599/notebooks/11_EfficientNumpy

1.0.3 The ARRAY class

Create an array

```
In [4]: # Easy to create a numpy array (the basic class) from a list
        l = [1,2,3,4,5,6]
        print(l)
        a = np.array([1,2,3,4,5,6])
        print(a)
        print(type(a))
        # Works with tuples also:
        b = np.array((1,2,3))
        print(b)
```

```
[1, 2, 3, 4, 5, 6]
[1 2 3 4 5 6]
<type 'numpy.ndarray'>
[1 2 3]
```

Numpy arrays are efficiently connected to the computer:

```
In [5]: L = range(1000)
        %timeit L2 = [i**2 for i in L] # Notice the use of timeit, a magic function
        A = np.arange(1000)
        %timeit A2 = A**2
```

```
10000 loops, best of 3: 78.4 µs per loop
The slowest run took 11.17 times longer than the fastest. This could mean that an i
1000000 loops, best of 3: 1.24 µs per loop
```

```
In [6]: L = [1, 2, 3, 4]
        a = np.array(L)
        print a.dtype
        print a
```

```
int64
[1 2 3 4]
```

```
In [7]: L = [1,2,3,4.]
        a = np.array(L)
        print a.dtype
        print a
```

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

```
In [8]: L = [1,2,3,4.,'a']
        a = np.array(L)
        print L # Different types can coexist in a python list
        print a.dtype
        print a # NOT in a numpy array. The array is re-typed to the highest type,
```

```
[1, 2, 3, 4.0, 'a']
|S32
['1' '2' '3' '4.0' 'a']
```

Once the type of an array is defined, one can insert values of type that can be transformed to the type of the array

```
In [9]: a = np.array([1,2,3,4,5,6])
        print a
        a[4] = 2.56 # will be transformed to int(2.56)
        print a
        a[3] = '20' # will be tranformed to int('20')
        print a
```

```
[1 2 3 4 5 6]
[1 2 3 4 2 6]
[ 1  2  3 20  2  6]
```

```
In [10]: a[2] = '3.2'
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-10-2af1cc391cb1> in <module>()
    ----> 1 a[2] = '3.2'


        ValueError: invalid literal for long() with base 10: '3.2'
```

```
In [11]: a[2] = 'tralala'
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-11-f6467d624e31> in <module>()
    ----> 1 a[2] = 'tralala'


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

1D, 2D, 3D, . . .

```
In [12]: a = np.array([1,2,3,4,5,6])
         b = np.array([[1,2],[1,4]])
         c = np.array([[[1], [2]], [[3], [4]]])
         print a.shape, b.shape, c.shape
         print a[0] # no error
```

```
(6,) (2, 2) (2, 2, 1)
1
```

```
In [13]: print len(a), len(b), len(c) # size of the first dimension
```

```
6 2 2
```

```
In [14]: b.size
```

```
Out[14]: 4
```

```
In [15]: print a.ndim, b.ndim, c.ndim
```

```
1 2 3
```

```
In [16]: a = np.array([1,2,3,4,5,6])
         print('mean: {0}, max: {1}, shape: {2}'.format(a.mean(), a.max(), a.shape)
```

```
mean: 3.5, max: 6, shape: (6,)
```

mean and max are methods (functions) of the array class, they need ()s. shape is an atribute (like a variable).

```
In [17]: print(a.mean) # this is printing information about the function, NOT the
```

```
<built-in method mean of numpy.ndarray object at 0x10c9cff30>
```

```
In [18]: mm = a.mean # We assign to mn the function. Then we can call it directly,
         print(mm())
```

```
3.5
```

```
In [19]: print b
         print b.mean() # mean over the whole array
         print b.mean(axis=0) # mean over the first axis (columns)
         print b.mean(1) # mean over the raws
         print np.mean(b)
```

```
[[1 2]
 [1 4]]
2.0
[ 1.  3.]
[ 1.5  2.5]
2.0
```

Creating arrays from scratch

```
In [20]: print np.arange(10)
```

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

```
In [21]: print np.linspace(0, 1, 10) # start, stop (included), number of points
         print '---------------------------------'
         print np.linspace(0, 1, 11) # start, stop (included), number of points
         print '---------------------------------'
         print np.linspace(0, 1, 10, endpoint=False) # Not including the stop point
```

```
[ 0.          0.11111111  0.22222222  0.33333333  0.44444444  0.55555556
  0.66666667  0.77777778  0.88888889  1.        ]
---------------------------------
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
---------------------------------
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
```

```
In [22]: print np.logspace(0, 2, 10) # from 10**start to 10**stop, with 10 values
```

```
[   1.            1.66810054    2.7825594     4.64158883    7.74263683
   12.91549665   21.5443469    35.93813664   59.94842503  100.        ]
```

```
In [23]: print np.zeros(2) # Filled with 0.0
         print '---------------------------------'
         print np.zeros((2,3)) # a 2D array, also filled with 0.0
         print '---------------------------------'
         print np.ones_like(a) # This is very usefull: using an already created arr
         print '---------------------------------'
         print np.zeros_like(a, dtype=float)+3 # Can define a value to fille the ar
         print '---------------------------------'
         print np.ones_like([1,2,3])
```

```
[ 0.  0.]
---------------------------------
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
---------------------------------
[1 1 1 1 1 1]
---------------------------------
[ 3.  3.  3.  3.  3.  3.]
---------------------------------
[1 1 1]
```

```
In [24]: b = a.reshape((3,2)) # This does NOT change the shape of a
         print a
         print('-------------')
         print b

[1 2 3 4 5 6]
-------------
[[1 2]
 [3 4]
 [5 6]]
```

```
In [25]: print(b.ravel())
         print(b.reshape(b.size))

[1 2 3 4 5 6]
[1 2 3 4 5 6]
```

```
In [26]: # create 2 2D arrays (coordinates matrices), one describing how x varies,
         x, y = np.mgrid[0:5, 0:10] # This is not a function!!! notice the []
         print x
         print '------------------------------------'
         print y

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

```
In [27]: # coordinates matrices using user-defined x- and y-vectors
         x, y = np.meshgrid([1,2,4,7], [0.1, 0.2, 0.3])
         print x
         print '--------------------------------'
         print y

[[1 2 4 7]
 [1 2 4 7]
 [1 2 4 7]]
--------------------------------
[[ 0.1  0.1  0.1  0.1]
```

```
 [ 0.2  0.2  0.2  0.2]
 [ 0.3  0.3  0.3  0.3]]
```

```
In [28]: x, y = np.meshgrid([1,2,4,7], [0.1, 0.2, 0.3], indexing='ij') # the other
         print x
         print '-----------------------------------------'
         print y
```

```
[[1 1 1]
 [2 2 2]
 [4 4 4]
 [7 7 7]]
-----------------------------------------
[[ 0.1  0.2  0.3]
 [ 0.1  0.2  0.3]
 [ 0.1  0.2  0.3]
 [ 0.1  0.2  0.3]]
```

WARNING arrays share memory

```
In [29]: b = a.reshape((3,2))
         print(a.shape, b.shape)
```

```
((6,), (3, 2))
```

```
In [30]: b[1,1] = 100 # modify a value in the array
         print b
```

```
[[  1   2]
 [  3 100]
 [  5   6]]
```

```
In [31]: print a # !!! a and b are sharing the same place in the memory, they are p
```

```
[  1   2   3 100   5   6]
```

```
In [32]: b[1,1], a[3] # same value
```

```
Out[32]: (100, 100)
```

```
In [33]: a is b # a and b are different
```

```
Out[33]: False
```

```
In [34]: print b[1,1] == a[3]
         print b[1,1] is a[3] # Even if the values are the same, the "is" does not
```

```
True
False
```

```
In [35]: c = a.reshape((2,3)).copy() # This is the solution.
```

```
In [36]: print a
         print '---------------'
         print c
```

```
[  1   2   3 100   5   6]
---------------
[[  1   2   3]
 [100   5   6]]
```

```
In [37]: c[0,0] = 8888
         print a
         print '---------------'
         print c
```

```
[  1   2   3 100   5   6]
---------------
[[8888    2    3]
 [ 100    5    6]]
```

1.0.4 Random

```
In [38]: ran_uniform = np.random.rand(5) # between 0 and 1
         ran_normal = np.random.randn(5) # Gaussian mean 0 variance 1
         print ran_uniform
         print '-----------------------------'
         print ran_normal
         print '-----------------------------'
         ran_normal_2D = np.random.randn(5,5) # Gaussian mean 0 variance 1
         print ran_normal_2D
```

```
[ 0.9818688   0.91764804  0.81160137  0.166043    0.92166132]
-----------------------------
[ 2.14470113  1.10794086  0.56957675  0.65652227  0.41462357]
-----------------------------
[[-0.23128756  0.52164277  0.44706089 -1.35432273  0.8632987 ]
 [-0.44912384 -0.39465552 -0.59289268 -1.59285917 -0.52614779]
 [ 0.80096167  2.00567484  0.72730418  1.24066153  0.79719522]
 [-1.43764059 -2.70682178  0.42993417  1.00279651  1.10350841]
 [ 1.18080409  0.917795    1.60039925  1.78526168 -0.800947  ]]
```

```
In [39]: np.random.seed(1)
         print np.random.rand(5)
         np.random.seed(1)
         print np.random.rand(5)
```

```
[  4.17022005e-01   7.20324493e-01   1.14374817e-04   3.02332573e-01
   1.46755891e-01]
[  4.17022005e-01   7.20324493e-01   1.14374817e-04   3.02332573e-01
   1.46755891e-01]
```

1.0.5 Timing on 2D array

```
In [40]: N = 100
         A = np.random.rand(N, N)
         B = np.zeros_like(A)
```

```
In [41]: %%timeit
         for i in range(N):
             for j in range(N):
                 B[i,j] = A[i,j]
```

```
100 loops, best of 3: 2.52 ms per loop
```

```
In [42]: %%timeit
         B = A # very faster ! It does NOT copy...
```

```
10000000 loops, best of 3: 35.1 ns per loop
```

```
In [43]: %%timeit
         B = (A.copy()) # Takes more time
```

```
The slowest run took 26.10 times longer than the fastest. This could mean that an i
100000 loops, best of 3: 3.56 µs per loop
```

```
In [44]: %%timeit
         for i in range(N):
             for j in range(N):
                 B[i,j] = A[i,j]**2
```

```
100 loops, best of 3: 4.67 ms per loop
```

```
In [45]: %%timeit
         B = A**2 # very faster ! Does a copy
```

```
The slowest run took 22.60 times longer than the fastest. This could mean that an i
100000 loops, best of 3: 4.34 µs per loop
```

```
In [46]: %timeit  B = (A.copy())**2 # Takes a little bit more time
```

```
The slowest run took 12.50 times longer than the fastest. This could mean that an i
100000 loops, best of 3: 10.2 µs per loop
```

1.0.6 Slicing

```
In [47]: a = np.arange(10)
         print a
         print a[1:8:3]
```

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

```
In [48]: print a[:7]
```

```
[0 1 2 3 4 5 6]
```

```
In [49]: print a[4:]
```

```
[4 5 6 7 8 9]
```

```
In [50]: print a[::2]
         print a[::2][2]
```

```
[0 2 4 6 8]
4
```

```
In [51]: # Revert the array:
         print a[::-1]
```

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

Assignment

```
In [52]: a[5:] = 999
         print a
```

```
[  0   1   2   3   4 999 999 999 999 999]
```

```
In [53]: a[5:] = a[4::-1]
         print a

[0 1 2 3 4 4 3 2 1 0]
```

```
In [54]: print a
         b = a[:, np.newaxis] # create a new empty dimension
         print b
         print a.shape, b.shape
         c = a[np.newaxis, :]
         print c, c.shape

[0 1 2 3 4 4 3 2 1 0]
[[0]
 [1]
 [2]
 [3]
 [4]
 [4]
 [3]
 [2]
 [1]
 [0]]
(10,) (10, 1)
[[0 1 2 3 4 4 3 2 1 0]] (1, 10)
```

```
In [55]: b*c # Cross product, see below (broadcasting)

Out[55]: array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
                [ 0,  1,  2,  3,  4,  4,  3,  2,  1,  0],
                [ 0,  2,  4,  6,  8,  8,  6,  4,  2,  0],
                [ 0,  3,  6,  9, 12, 12,  9,  6,  3,  0],
                [ 0,  4,  8, 12, 16, 16, 12,  8,  4,  0],
                [ 0,  4,  8, 12, 16, 16, 12,  8,  4,  0],
                [ 0,  3,  6,  9, 12, 12,  9,  6,  3,  0],
                [ 0,  2,  4,  6,  8,  8,  6,  4,  2,  0],
                [ 0,  1,  2,  3,  4,  4,  3,  2,  1,  0],
                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0]])
```

Using an array

```
In [56]: print a
         a[[2,4,6]] = -999
         print a

[0 1 2 3 4 4 3 2 1 0]
[   0    1 -999    3 -999    4 -999    2    1    0]
```

```
In [57]: # a = 1 would turn a to be 1, but if we want to assign 1 to every value in
         a[:] = 1
         print a
```

```
[1 1 1 1 1 1 1 1 1 1]
```

1.0.7 Using masks

```
In [58]: a = np.random.random_integers(0, 100, 20) # min, max, N
         print a
```

```
[ 81  86  52  39  52  13 100   9  98  78  46  26  63  86   2  96  45  13
  67  37]


/Users/christophemorisset/anaconda/lib/python2.7/site-packages/ipykernel/__main__.p
  if __name__ == '__main__':
```

```
In [59]: a < 50
```

```
Out[59]: array([False, False, False,  True, False,  True, False,  True, False,
                False,  True,  True, False, False,  True, False,  True,  True,
                False,  True], dtype=bool)
```

```
In [60]: mask = (a < 50)
```

```
In [61]: mask.sum()
```

```
Out[61]: 9
```

```
In [62]: a[mask]
```

```
Out[62]: array([39, 13,  9, 46, 26,  2, 45, 13, 37])
```

```
In [63]: b = a.copy()  # do NOT use b = a
         b[mask] = 50  #
         print a
         print b
```

```
[ 81  86  52  39  52  13 100   9  98  78  46  26  63  86   2  96  45  13
  67  37]
[ 81  86  52  50  52  50 100  50  98  78  50  50  63  86  50  96  50  50
  67  50]
```

```
In [64]: b = a.copy()
         b[b <= 50] = 0 # shortest way. Not matter if not even one element fit the
         print b
```

```
[ 81  86  52   0  52   0 100   0  98  78   0   0  63  86   0  96   0   0
  67   0]
```

```
In [65]: print a[mask]
         print a[~mask] # complementary
```

```
[39 13  9 46 26  2 45 13 37]
[ 81  86  52  52 100  98  78  63  86  96  67]
```

```
In [66]: mask
```

```
Out[66]: array([False, False, False,  True, False,  True, False,  True, False,
                False,  True,  True, False, False,  True, False,  True,  True,
                False,  True], dtype=bool)
```

```
In [67]: mask = np.zeros_like(a, dtype=bool)
         print mask
```

```
[False False False False False False False False False False False False
 False False False False False False False False]
```

```
In [68]: mask[[2,3,4]] = True

In [69]: mask
```

```
Out[69]: array([False, False,  True,  True,  True, False, False, False, False,
                False, False, False, False, False, False, False, False, False,
                False, False], dtype=bool)
```

```
In [70]: a[mask]
```

```
Out[70]: array([52, 39, 52])
```

```
In [71]: a[mask].sum()
```

```
Out[71]: 143
```

combining masks

```
In [72]: print a
         mask_low = a > 30
         mask_high = a < 70
         print '--------------------------------------'
         print a[mask_low & mask_high] # both conditions are filled
         print '--------------------------------------'
         print a[~mask_low | ~mask_high] # complementary, using the | for OR
```

```
[ 81  86  52  39  52  13 100   9  98  78  46  26  63  86   2  96  45  13
  67  37]
-------------------------------------
[52 39 52 46 63 45 67 37]
-------------------------------------
[ 81  86  13 100   9  98  78  26  86   2  96  13]
```

the where function

```
In [73]: tt = np.where(a > 30)
         print a
         print tt # tt is a tuple of arrays, one for each dimension of the conditi
         # containing the indices where the condition is filled in that dimension.
```

```
[ 81  86  52  39  52  13 100   9  98  78  46  26  63  86   2  96  45  13
  67  37]
(array([ 0,  1,  2,  3,  4,  6,  8,  9, 10, 12, 13, 15, 16, 18, 19]),)
```

```
In [74]: (a > 30).nonzero() # "where" is the same than condition.nonzero().

Out[74]: (array([ 0,  1,  2,  3,  4,  6,  8,  9, 10, 12, 13, 15, 16, 18, 19]),)

In [75]: # the indices where the condition is filled are in the first element of th

In [76]: tt[0]

Out[76]: array([ 0,  1,  2,  3,  4,  6,  8,  9, 10, 12, 13, 15, 16, 18, 19])

In [77]: # faster once you know that the condition is 1D
         tt = np.where(a > 30)[0]

In [78]: tt # the array containing the indices where the condition is filled

Out[78]: array([ 0,  1,  2,  3,  4,  6,  8,  9, 10, 12, 13, 15, 16, 18, 19])

In [79]: a[tt] # the values where the condition is filled

Out[79]: array([ 81,  86,  52,  39,  52, 100,  98,  78,  46,  63,  86,  96,  45,
                 67,  37])

In [80]: # The where function can take 3 arguments.
         b = np.where(a < 50, np.nan, a)
         print a
         print b
         print np.isfinite(b)
```

```
[ 81  86  52  39  52  13 100   9  98  78  46  26  63  86   2  96  45  13
  67  37]
[  81.   86.   52.   nan   52.   nan  100.   nan   98.   78.   nan   nan
   63.   86.   nan   96.   nan   nan   67.   nan]
[ True  True  True False  True False  True False  True  True False False
  True  True False  True False False  True False]
```

```
In [81]: b = np.where(a < 50, True, False)
         print a
         print b
```

```
[ 81  86  52  39  52  13 100   9  98  78  46  26  63  86   2  96  45  13
  67  37]
[False False False  True False  True False  True False False  True  True
 False False  True False  True  True False  True]
```

1.0.8 Some operations with arrays

```
In [82]: a
```

```
Out[82]: array([ 81,  86,  52,  39,  52,  13, 100,   9,  98,  78,  46,  26,  63,
                 86,   2,  96,  45,  13,  67,  37])
```

```
In [83]: a + 1
```

```
Out[83]: array([ 82,  87,  53,  40,  53,  14, 101,  10,  99,  79,  47,  27,  64,
                 87,   3,  97,  46,  14,  68,  38])
```

```
In [84]: a**2 + 3*a**3
```

```
Out[84]: array([1600884, 1915564,  424528,  179478,  424528,    6760, 3010000,
                   2268, 2833180, 1429740,  294124,   53404,  754110, 1915564,
                     28, 2663424,  275400,    6760,  906778,  153328])
```

```
In [85]: # look for the integers I so that i**2 + (i+1)**2 = (i+2)**2
         i = np.arange(30)
         b = i**2 + (i+1)**2
```

```
In [86]: c = (i+2)**2
```

```
In [87]: print b
         print c
```

```
[   1    5   13   25   41   61   85  113  145  181  221  265  313  365  421
  481  545  613  685  761  841  925 1013 1105 1201 1301 1405 1513 1625 1741]
[  4   9  16  25  36  49  64  81 100 121 144 169 196 225 256 289 324 361
 400 441 484 529 576 625 676 729 784 841 900 961]
```

```
In [88]: b == c

Out[88]: array([False, False, False,  True, False, False, False, False, False,
                False, False, False, False, False, False, False, False, False,
                False, False, False, False, False, False, False, False, False,
                False, False, False], dtype=bool)

In [89]: i[b==c]

Out[89]: array([3])

In [90]: i[b==c][0] # the result is an array. To obtain the first value (here the

Out[90]: 3
```

Numpy manages almost any mathematical operation. log, trigo, etc

```
In [91]: a = np.arange(18)
         print a
         print np.log10(a)

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17]
[       -inf  0.          0.30103     0.47712125  0.60205999  0.69897
  0.77815125  0.84509804  0.90308999  0.95424251  1.          1.04139269
  1.07918125  1.11394335  1.14612804  1.17609126  1.20411998  1.23044892]


/Users/christophemorisset/anaconda/lib/python2.7/site-packages/ipykernel/__main__.p
  app.launch_new_instance()


In [92]: for aa in a:
             print('{0:2} {1:4.2f} {2:5.2f} {3:8.2e}'.format(aa, np.log10(aa), np.s

 0 -inf  0.00 1.00e+00
 1 0.00  0.84 2.72e+00
 2 0.30  0.91 7.39e+00
 3 0.48  0.14 2.01e+01
 4 0.60 -0.76 5.46e+01
 5 0.70 -0.96 1.48e+02
 6 0.78 -0.28 4.03e+02
 7 0.85  0.66 1.10e+03
 8 0.90  0.99 2.98e+03
 9 0.95  0.41 8.10e+03
10 1.00 -0.54 2.20e+04
11 1.04 -1.00 5.99e+04
12 1.08 -0.54 1.63e+05
13 1.11  0.42 4.42e+05
14 1.15  0.99 1.20e+06
15 1.18  0.65 3.27e+06
16 1.20 -0.29 8.89e+06
17 1.23 -0.96 2.42e+07
```

```
/Users/christophemorisset/anaconda/lib/python2.7/site-packages/ipykernel/__main__.p
  from ipykernel import kernelapp as app
```

sum

```
In [93]: print a.sum()
         print 17*18/2

153
153
```

```
In [94]: a = np.random.rand(2, 4, 3)
         print a.shape
         print a.size

(2, 4, 3)
24
```

2 planes, 4 rows, 3 columns

A small comment on the order of the elements in arrays in Python: There is two ways arrays can be stored: row- or column major. It has a direct impact on the way one has to loop on the arrays. IDL is like Fortran (column major) and Python is like C (row major). It means that in Python, as you move linearly through the memory of an array, the second dimension (rightmost) changes the fastest, while in IDL the first (leftmost) dimension changes the fastest. Consequence on the loop order in Python:

```
In [95]: for plane in a:
             for row in plane:
                 for col in row:
                     print col
                     print '-----'

0.149114084695
-----
0.223501304492
-----
0.735745679176
-----
0.276853008806
-----
0.491054183994
-----
0.234895768153
-----
0.441966266774
-----
```

```
0.425988454064
-----
0.364991677396
-----
0.254841464865
-----
0.931163311426
-----
0.428392066464
-----
0.0761677167238
-----
0.428421906294
-----
0.0897867491632
-----
0.713721228987
-----
0.484034913772
-----
0.282853195565
-----
0.722888404661
-----
0.2881094042
-----
0.161742769482
-----
0.853032025579
-----
0.220469822607
-----
0.85508854459
-----
```

```
In [96]: print a[0,1,2] # a[p, r, c]
```

```
0.234895768153
```

```
In [97]: a.sum()
```

```
Out[97]: 10.134823951927267
```

```
In [98]: a.sum(0) # from 3D to 2D. Generate an "image" of the sum, i.e. the "projec
```

```
Out[98]: array([[ 0.2252818 ,  0.65192321,  0.82553243],
                [ 0.99057424,  0.9750891 ,  0.51774896],
                [ 1.16485467,  0.71409786,  0.52673445],
                [ 1.10787349,  1.15163313,  1.28348061]])

In [99]: a.sum(0).shape

Out[99]: (4, 3)

In [100]: a.sum(0).sum(0) # from 3D to 1D. From the image, make the sum in each row

Out[100]: array([ 3.4885842 ,  3.4927433 ,  3.15349645])

In [101]: a.min(0)

Out[101]: array([[ 0.07616772,  0.2235013 ,  0.08978675],
                 [ 0.27685301,  0.48403491,  0.23489577],
                 [ 0.44196627,  0.2881094 ,  0.16174277],
                 [ 0.25484146,  0.22046982,  0.42839207]])

In [102]: a.ravel()

Out[102]: array([ 0.14911408,  0.2235013 ,  0.73574568,  0.27685301,  0.49105418,
                  0.23489577,  0.44196627,  0.42598845,  0.36499168,  0.25484146,
                  0.93116331,  0.42839207,  0.07616772,  0.42842191,  0.08978675,
                  0.71372123,  0.48403491,  0.2828532 ,  0.7228884 ,  0.2881094 ,
                  0.16174277,  0.85303203,  0.22046982,  0.85508854])

In [103]: i_min = a.argmin() # return the index of where the minimum is. It uses th
          print i_min
          b = np.array([10,2,3,4,5,2])
          b.argmin() # only the first occurence

12


Out[103]: 1

In [104]: a.ravel().shape # 1D

Out[104]: (24,)

In [105]: a.ravel()[i_min] # Check where the minimum is.

Out[105]: 0.076167716723843704

In [106]: z = i_min/12
          y = (i_min - 12*z)/3
          x = i_min - 12*z - 3*y
          print z, y, x
          print a[z, y, x]
```

```
1 0 0
0.0761677167238
```

```
In [107]: def decompose_ravel(arr, i):
              shapes = arr.shape
              idx = i
              res = []
              for i in np.arange(arr.ndim):
                  subdims = np.prod(shapes[i+1:])
                  n = int(idx/subdims)
                  #print n, subdims, idx
                  idx = idx - subdims*n
                  res.append(n)
              return tuple(res)
```

```
In [108]: res = decompose_ravel(a, i_min)
          print a.min()
          print res
          print a[res]
```

```
0.0761677167238
(1, 0, 0)
0.0761677167238
```

```
In [109]: a.min(0).min(0)
```

```
Out[109]: array([ 0.07616772,  0.22046982,  0.08978675])
```

```
In [110]: print a[:,0,0]
          a[:,0,0].min()
```

```
[ 0.14911408  0.07616772]
```

```
Out[110]: 0.076167716723843704
```

```
In [111]: a.mean(0)
```

```
Out[111]: array([[ 0.1126409 ,  0.32596161,  0.41276621],
                 [ 0.49528712,  0.48754455,  0.25887448],
                 [ 0.58242734,  0.35704893,  0.26336722],
                 [ 0.55393675,  0.57581657,  0.64174031]])
```

```
In [112]: np.median(a, 1)
```

```
Out[112]: array([[ 0.26584724,  0.45852132,  0.39669187],
                 [ 0.71830482,  0.35826566,  0.22229798]])
```

```
In [113]: a.std()

Out[113]: 0.2493761347658443

In [114]: np.percentile(a, 25)

Out[114]: 0.23204715223749081

In [115]: print a[0:4,0]
          print np.cumsum(a[0:100,0]) # axis is a keyword. If absent, applied on th

[[ 0.14911408  0.2235013   0.73574568]
 [ 0.07616772  0.42842191  0.08978675]]
[ 0.14911408  0.37261539  1.10836107  1.18452879  1.61295069  1.70273744]


In [116]: b = np.arange(1000).reshape(10,10,10)

In [117]: b.shape

Out[117]: (10, 10, 10)

In [118]: b[4,:,:] # hundreds digits = 4

Out[118]: array([[400, 401, 402, 403, 404, 405, 406, 407, 408, 409],
                 [410, 411, 412, 413, 414, 415, 416, 417, 418, 419],
                 [420, 421, 422, 423, 424, 425, 426, 427, 428, 429],
                 [430, 431, 432, 433, 434, 435, 436, 437, 438, 439],
                 [440, 441, 442, 443, 444, 445, 446, 447, 448, 449],
                 [450, 451, 452, 453, 454, 455, 456, 457, 458, 459],
                 [460, 461, 462, 463, 464, 465, 466, 467, 468, 469],
                 [470, 471, 472, 473, 474, 475, 476, 477, 478, 479],
                 [480, 481, 482, 483, 484, 485, 486, 487, 488, 489],
                 [490, 491, 492, 493, 494, 495, 496, 497, 498, 499]])

In [119]: b[:,2,:] # tens digit = 2

Out[119]: array([[ 20,  21,  22,  23,  24,  25,  26,  27,  28,  29],
                 [120, 121, 122, 123, 124, 125, 126, 127, 128, 129],
                 [220, 221, 222, 223, 224, 225, 226, 227, 228, 229],
                 [320, 321, 322, 323, 324, 325, 326, 327, 328, 329],
                 [420, 421, 422, 423, 424, 425, 426, 427, 428, 429],
                 [520, 521, 522, 523, 524, 525, 526, 527, 528, 529],
                 [620, 621, 622, 623, 624, 625, 626, 627, 628, 629],
                 [720, 721, 722, 723, 724, 725, 726, 727, 728, 729],
                 [820, 821, 822, 823, 824, 825, 826, 827, 828, 829],
                 [920, 921, 922, 923, 924, 925, 926, 927, 928, 929]])

In [120]: b[:,:,7] # unity digit = 7
```

```
Out[120]: array([[  7,  17,  27,  37,  47,  57,  67,  77,  87,  97],
                 [107, 117, 127, 137, 147, 157, 167, 177, 187, 197],
                 [207, 217, 227, 237, 247, 257, 267, 277, 287, 297],
                 [307, 317, 327, 337, 347, 357, 367, 377, 387, 397],
                 [407, 417, 427, 437, 447, 457, 467, 477, 487, 497],
                 [507, 517, 527, 537, 547, 557, 567, 577, 587, 597],
                 [607, 617, 627, 637, 647, 657, 667, 677, 687, 697],
                 [707, 717, 727, 737, 747, 757, 767, 777, 787, 797],
                 [807, 817, 827, 837, 847, 857, 867, 877, 887, 897],
                 [907, 917, 927, 937, 947, 957, 967, 977, 987, 997]])
```

```
In [121]: b.min(0) # elements with the smallest value for the hundreds digit
```

```
Out[121]: 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, 25, 26, 27, 28, 29],
                 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
                 [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
                 [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
                 [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
                 [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
                 [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
                 [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

```
In [122]: b.min(2) # smallest value for the unity digit
```

```
Out[122]: array([[  0,  10,  20,  30,  40,  50,  60,  70,  80,  90],
                 [100, 110, 120, 130, 140, 150, 160, 170, 180, 190],
                 [200, 210, 220, 230, 240, 250, 260, 270, 280, 290],
                 [300, 310, 320, 330, 340, 350, 360, 370, 380, 390],
                 [400, 410, 420, 430, 440, 450, 460, 470, 480, 490],
                 [500, 510, 520, 530, 540, 550, 560, 570, 580, 590],
                 [600, 610, 620, 630, 640, 650, 660, 670, 680, 690],
                 [700, 710, 720, 730, 740, 750, 760, 770, 780, 790],
                 [800, 810, 820, 830, 840, 850, 860, 870, 880, 890],
                 [900, 910, 920, 930, 940, 950, 960, 970, 980, 990]])
```

```
In [123]: b.min(2).shape
```

```
Out[123]: (10, 10)
```

```
In [124]: np.median(b)
```

```
Out[124]: 499.5
```

```
In [125]: np.median(b, axis=0)
```

```
Out[125]: array([[ 450.,  451.,  452.,  453.,  454.,  455.,  456.,  457.,  458.,
                   459.],
                 [ 460.,  461.,  462.,  463.,  464.,  465.,  466.,  467.,  468.,
                   469.],
                 [ 470.,  471.,  472.,  473.,  474.,  475.,  476.,  477.,  478.,
                   479.],
                 [ 480.,  481.,  482.,  483.,  484.,  485.,  486.,  487.,  488.,
                   489.],
                 [ 490.,  491.,  492.,  493.,  494.,  495.,  496.,  497.,  498.,
                   499.],
                 [ 500.,  501.,  502.,  503.,  504.,  505.,  506.,  507.,  508.,
                   509.],
                 [ 510.,  511.,  512.,  513.,  514.,  515.,  516.,  517.,  518.,
                   519.],
                 [ 520.,  521.,  522.,  523.,  524.,  525.,  526.,  527.,  528.,
                   529.],
                 [ 530.,  531.,  532.,  533.,  534.,  535.,  536.,  537.,  538.,
                   539.],
                 [ 540.,  541.,  542.,  543.,  544.,  545.,  546.,  547.,  548.,
                   549.]])
```

```
In [126]: x = 2 * np.random.rand(100,100,100) - 1.
          print np.min(x), np.max(x)
```

```
-0.999998652501 0.999998456548
```

```
In [127]: y = 2 * np.random.rand(100,100,100) - 1.
          z = 2 * np.random.rand(100,100,100) - 1.
```

```
In [128]: r = np.sqrt(x**2 + y**2 + z**2)
          print np.min(r), np.max(r)
          print np.sqrt(3)
```

```
0.00457859854772 1.72283773156
1.73205080757
```

```
In [129]: print np.mean(r)
          print r.mean()
```

```
0.960691278556
0.960691278556
```

```
In [130]: np.median(r)
```

```
Out[130]: 0.98507975493085209
```

1.0.9 Broadcasting

http://arxiv.org/pdf/1102.1523.pdf

```
If the two arrays differ in their number of dimensions, the shape of the array wit
If the shape of the two arrays does not match in any dimension, the array with shap
If in any dimension the sizes disagree and neither is equal to 1, an error is raise
```

```
In [131]: x1 = np.array((1,2,3,4,5))
          y1 = np.array((1,2,3,4,5))
          z1 = np.array((1,2,3,4,5))
          r1 = x1 * y1 * z1
          print r1.shape
```

```
(5,)
```

```
In [132]: x = np.array((1,2,3,4,5)).reshape(5,1,1)

In [133]: x

Out[133]: array([[[1]],

                 [[2]],

                 [[3]],

                 [[4]],

                 [[5]]])

In [134]: x.shape

Out[134]: (5, 1, 1)

In [135]: x.ndim

Out[135]: 3

In [136]: y = np.array((1,2,3,4,5)).reshape(1,5,1)
          z = np.array((1,2,3,4,5)).reshape(1,1,5)
          print y
          print z
```

```
[[[1]
  [2]
  [3]
  [4]
  [5]]]
[[[1 2 3 4 5]]]
```

```
In [137]: r = x * y * z

In [138]: print r.shape

(5, 5, 5)


In [139]: r

Out[139]: array([[[  1,   2,   3,   4,   5],
                  [  2,   4,   6,   8,  10],
                  [  3,   6,   9,  12,  15],
                  [  4,   8,  12,  16,  20],
                  [  5,  10,  15,  20,  25]],

                 [[  2,   4,   6,   8,  10],
                  [  4,   8,  12,  16,  20],
                  [  6,  12,  18,  24,  30],
                  [  8,  16,  24,  32,  40],
                  [ 10,  20,  30,  40,  50]],

                 [[  3,   6,   9,  12,  15],
                  [  6,  12,  18,  24,  30],
                  [  9,  18,  27,  36,  45],
                  [ 12,  24,  36,  48,  60],
                  [ 15,  30,  45,  60,  75]],

                 [[  4,   8,  12,  16,  20],
                  [  8,  16,  24,  32,  40],
                  [ 12,  24,  36,  48,  60],
                  [ 16,  32,  48,  64,  80],
                  [ 20,  40,  60,  80, 100]],

                 [[  5,  10,  15,  20,  25],
                  [ 10,  20,  30,  40,  50],
                  [ 15,  30,  45,  60,  75],
                  [ 20,  40,  60,  80, 100],
                  [ 25,  50,  75, 100, 125]]])

In [140]: a = np.ones((10,10))
          b = np.arange(10).reshape(10,1)
          print a
          print b
          print b.shape

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

```
In [141]: a * b
```

```
Out[141]: array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                 [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
                 [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
                 [ 3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.],
                 [ 4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.],
                 [ 5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.],
                 [ 6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.],
                 [ 7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.],
                 [ 8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.],
                 [ 9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.]])
```

```
In [142]: a * b.reshape(1,10)
```

```
Out[142]: array([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
                 [ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.]])
```

1.0.10 Structured arrays and RecArrays

See here: http://docs.scipy.org/doc/numpy/user/basics.rec.html

A structured array in numpy is an array of records. Each record can contain one or more items which can be of different types.

```
In [143]: a = np.array([(1.5, 2), (3.0, 4)]) # Classical numpy array
          print a

[[ 1.5  2. ]
 [ 3.   4. ]]
```

```
In [144]: astru = np.array([(1.5, 2), (3.0, 4)], dtype=[('x', float), ('y', int)])
          astru

Out[144]: array([(1.5, 2), (3.0, 4)],
                dtype=[('x', '<f8'), ('y', '<i8')])
```

```
In [145]: print astru['x']
          print astru['y']

[ 1.5  3. ]
[2 4]
```

```
In [146]: arec = astru.view(np.recarray)
          print type(a), type(astru), type(arec)
          print '----------------------------------'
          print a
          print astru
          print arec
          print '----------------------------------'
          print a.size, astru.size, arec.size # not even the same sixe
          print '----------------------------------'
          print a.dtype, astru.dtype, arec.dtype # types tell us that ar has column
          print '----------------------------------'
          print a[1,1], astru[1][1], arec[1][1] # one is 2D, the other is a collect
          print '----------------------------------'
          print astru['y'] # acces by name (a litle like dictionnaries)
          print '----------------------------------'
          print arec.x

<type 'numpy.ndarray'> <type 'numpy.ndarray'> <class 'numpy.recarray'>
----------------------------------
[[ 1.5  2. ]
 [ 3.   4. ]]
[(1.5, 2) (3.0, 4)]
[(1.5, 2) (3.0, 4)]
----------------------------------
4 2 2
----------------------------------
float64 [('x', '<f8'), ('y', '<i8')] (numpy.record, [('x', '<f8'), ('y', '<i8')])
----------------------------------
4.0 4 4
```

```
----------------------------------
[2 4]
----------------------------------
[ 1.5  3. ]
```

```
In [147]: %timeit astru2 = np.append(astru, np.array([(5.0, 6)], dtype=astru.dtype)

The slowest run took 18.33 times longer than the fastest. This could mean that an i
100000 loops, best of 3: 3.88 µs per loop
```

```
In [148]: %timeit astru3 = np.concatenate((astru, np.array([(5.0, 6)], dtype=astru.

The slowest run took 65.26 times longer than the fastest. This could mean that an i
1000000 loops, best of 3: 1.92 µs per loop
```

```
In [149]: %timeit arec2 = np.append(arec, np.array([(5.0, 6)], dtype=astru.dtype).v

100000 loops, best of 3: 17.2 µs per loop
```

```
In [150]: %timeit arec3 = np.concatenate((arec, np.array([(5.0, 6)], dtype=astru.dt

The slowest run took 8.93 times longer than the fastest. This could mean that an in
100000 loops, best of 3: 10.6 µs per loop
```

```
In [151]: arec4 = np.rec.fromrecords([(456,'dbe',1.2),(2,'de',1.3)],names='col1,col
          print arec4
          print type(arec4)
          print arec4.col1[1]
          print arec4[1].col1

[(456, 'dbe', 1.2) (2, 'de', 1.3)]
<class 'numpy.recarray'>
2
2
```

```
In [152]: arec4 = np.rec.fromrecords([('etoile_15', 30.015, -0.752, 10.722),
                                      ('etoile_11', 31.163, -9.109, 10.761),
                                      ('etoile_16', 39.789, -7.716, 11.071),
                                      ('etoile_14', 35.110, 6.785, 11.176),
                                      ('etoile_31', 33.530, 9.306, 11.823),
                                      ('etoile_04', 33.480, 5.568, 11.978)
                                      ],
                                     names='name,ra,dec, mag')
```

```
In [153]: mask = arec4.mag > 11.
          print arec4[mask]
          print '-------------------------'
          for star in arec4[mask]:
              print('name: {0} ra = {1} dec = {2} magnitude = {3}'.format(star.name
          print '-------------------------'
          for star in arec4[mask]:
              print('name: {0[name]} ra = {0[ra]} dec = {0[dec]} magnitude = {0[mag
```

```
[('etoile_16', 39.789, -7.716, 11.071) ('etoile_14', 35.11, 6.785, 11.176)
 ('etoile_31', 33.53, 9.306, 11.823) ('etoile_04', 33.48, 5.568, 11.978)]
-------------------------
name: etoile_16 ra = 39.789 dec = -7.716 magnitude = 11.071
name: etoile_14 ra = 35.11 dec = 6.785 magnitude = 11.176
name: etoile_31 ra = 33.53 dec = 9.306 magnitude = 11.823
name: etoile_04 ra = 33.48 dec = 5.568 magnitude = 11.978
-------------------------
name: etoile_16 ra = 39.789 dec = -7.716 magnitude = 11.071
name: etoile_14 ra = 35.11 dec = 6.785 magnitude = 11.176
name: etoile_31 ra = 33.53 dec = 9.306 magnitude = 11.823
name: etoile_04 ra = 33.48 dec = 5.568 magnitude = 11.978
```

1.0.11 NaN and other ANSI values

```
In [154]: a = np.array([-3, -2., -1., 0., 1., 2.])
          b = 1./a
          print b
```

```
[-0.33333333 -0.5        -1.                 inf  1.          0.5       ]


/Users/christophemorisset/anaconda/lib/python2.7/site-packages/ipykernel/__main__.p
  from ipykernel import kernelapp as app
```

```
In [155]: print a.sum()
          print b.sum()  # NaN and others are absorbant elements
```

```
-3.0
inf
```

```
In [156]: mask = np.isfinite(b)
          print mask
          print b[mask].sum()
```

```
[ True  True  True False  True  True]
-0.333333333333
```

```
In [157]: for elem in b:
              print np.isinf(elem)

False
False
False
True
False
False
```

1.0.12 Roundish values of floats

```
In [158]: import math
          res = []
          for i in range(100):
              res.append(math.log(2 ** i, 2)) # The second argument is the base of
          print res
          # We can see that sometimes the value of log2(2**i) is NOT i.

[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15


In [159]: res2 = []
          for i in range(100):
              res2.append(float(round(math.log(2**i, 2))) == math.log(2 ** i, 2))
          print res2
          # An equivalent result is obtained when comparing the round value. This s

[True, True, True, True, True, True, True, True, True, True, True, True, True, True


In [160]: res = []
          for i in range(100):
              res.append(np.log2(2.**i)) # The second argument is the base of the l
          print res

          res_np = []
          for i in range(100):
              res_np.append(float(round(np.log2(2.**i))) == np.log2(2.**i))
          print res_np
          # No problemes with the numpy log function.

[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15
[True, True, True, True, True, True, True, True, True, True, True, True, True, True
```

In case of doubdts, one can use the close function from numpy:

```
In [161]: res_np2 = []
          for i in range(100):
              res_np2.append(np.isclose(float(round(math.log(2 ** i, 2))), math.log
          print res_np2
          # The isclose

[True, True, True, True, True, True, True, True, True, True, True, True, True, True


In [162]: np.isclose?
```