intro_numpy

June 30, 2017

```
In [1]: # The following is to know when this notebook has been run and with which python version.
        import time, sys
        print(time.ctime())
        print(sys.version.split('|')[0])
```

```
Fri Jun 30 12:59:47 2017
3.6.1
```

1 B Numpy

This is part of the Python lecture given by Christophe Morisset at IA-UNAM.

1.0.1 Import numpy first

```
In [2]: # You need first to import the numpy library (must be installed on your computer ;-) )
        # As it will be widely used, better to give it a nickname, or an alias. Traditionnaly, it's "np
        import numpy as np
```

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

```
1.12.1
```

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]
<class '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 (starts with %)
        A = np.arange(1000)
        %timeit A2 = A**2
```

```
379 µs ± 3.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
2.25 µs ± 37.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
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, here string.
```

```
[1, 2, 3, 4.0, 'a']
<U32
['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 int() 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 int() 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 result of the functio
```

```
<built-in method mean of numpy.ndarray object at 0x7fa7a0455a30>
```

```
In [18]: mm = a.mean # We assign to mm the function. Then we can call it directly, but still need for t
         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 array (or list or tupl
         print('----------------------------------')
         print(np.zeros_like(a, dtype=float)+3) # Can define a value to fille the array when creating i
         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, the other for y.
         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 order...
         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 pointing to the same

[  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 tell it.

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)
```

```
[ 0.7882152   0.99241912  0.79943269  0.78337017  0.59978395]
-----------------------------
[-0.63970449 -0.57141457 -1.23728057  0.4390936   2.09181458]
-----------------------------
[[ 1.32569953 -0.21616863  0.87488409 -0.3893075  -0.38488923]
 [-1.10055984  1.15529923 -0.0193641   1.96576883 -0.63892173]
 [-1.58490005  0.43689955  0.66755566  0.50236483 -1.03943212]
 [-1.33942681 -0.35644385 -1.78784832  0.29479541  0.63027842]
 [ 0.89684588  0.63669925 -0.12266564 -2.58324896 -0.72706483]]
```

```
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]
```

```
4.19 ms ± 78.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

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

```
30.3 ns ± 0.244 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

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

```
5.02 µs ± 21 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

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

```
7.2 ms ± 38.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

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

```
7.94 µs ± 48.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

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

```
15.2 µs ± 203 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

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 one must do:
         a[:] = 1
         print(a)
```

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

1.0.7 Using masks

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

```
[81 86 52 39 52 13  9 98 78 46 26 63 86  2 96 45 13 67 37 36]
```

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

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

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

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

Out[61]: 10

In [62]: a[mask]

Out[62]: array([39, 13,  9, 46, 26,  2, 45, 13, 37, 36])

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

[81 86 52 39 52 13  9 98 78 46 26 63 86  2 96 45 13 67 37 36]
[81 86 52 50 52 50 50 98 78 50 50 63 86 50 96 50 50 67 50 50]

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

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

In [65]: print(a[mask])
         print(a[~mask]) # complementary

[39 13  9 46 26  2 45 13 37 36]
[81 86 52 52 98 78 63 86 96 67]

In [66]: mask

Out[66]: array([False, False, False,  True, False,  True,  True, False, False,
                True,  True, False, False,  True, False,  True,  True, False,
                True,  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  9 98 78 46 26 63 86  2 96 45 13 67 37 36]
-------------------------------------
[52 39 52 46 63 45 67 37 36]
-------------------------------------
[81 86 13  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 condition,
         # containing the indices where the condition is filled in that dimension.
```

```
[81 86 52 39 52 13  9 98 78 46 26 63 86  2 96 45 13 67 37 36]
(array([ 0,  1,  2,  3,  4,  7,  8,  9, 11, 12, 14, 15, 17, 18, 19]),)
```

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

```
Out[74]: (array([ 0,  1,  2,  3,  4,  7,  8,  9, 11, 12, 14, 15, 17, 18, 19]),)
```

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

```
In [76]: tt[0]
```

```
Out[76]: array([ 0,  1,  2,  3,  4,  7,  8,  9, 11, 12, 14, 15, 17, 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,  7,  8,  9, 11, 12, 14, 15, 17, 18, 19])
```

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

```
Out[79]: array([81, 86, 52, 39, 52, 98, 78, 46, 63, 86, 96, 45, 67, 37, 36])
```

```
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  9 98 78 46 26 63 86  2 96 45 13 67 37 36]
[ 81.  86.  52.  nan  52.  nan  nan  98.  78.  nan  nan  63.  86.  nan  96.
  nan  nan  67.  nan  nan]
[ True  True  True False  True False False  True  True False False  True
  True False  True False False  True False False]
```

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

```
[81 86 52 39 52 13  9 98 78 46 26 63 86  2 96 45 13 67 37 36]
[False False False  True False  True  True False False  True  True False
 False  True False  True  True False  True  True]
```

```
In [82]: b = np.where(a < 50, 0, 100)
         print(a)
         print(b)
```

```
[81 86 52 39 52 13  9 98 78 46 26 63 86  2 96 45 13 67 37 36]
[100 100 100   0 100   0   0 100 100   0   0 100 100   0 100   0   0 100
   0   0]
```

1.0.8 Some operations with arrays

```
In [83]: a
```

```
Out[83]: array([81, 86, 52, 39, 52, 13,  9, 98, 78, 46, 26, 63, 86,  2, 96, 45, 13,
                67, 37, 36])
```

```
In [84]: a + 1
```

```
Out[84]: array([82, 87, 53, 40, 53, 14, 10, 99, 79, 47, 27, 64, 87,  3, 97, 46, 14,
                68, 38, 37])
```

```
In [85]: a**2 + 3*a**3
```

```
Out[85]: array([1600884, 1915564,  424528,  179478,  424528,    6760,    2268,
                2833180, 1429740,  294124,   53404,  754110, 1915564,      28,
                2663424,  275400,    6760,  906778,  153328,  141264])
```

```
In [86]: # 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 [87]: c = (i+2)**2
```

```
In [88]: 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 [89]: b == c
```

```
Out[89]: 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 [90]: i[b==c]
```

```
Out[90]: array([3])
```

```
In [91]: i[b==c][0] # the result is an array. To obtain the first value (here the only one), use [0]
```

```
Out[91]: 3
```

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

```
In [92]: 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]

/home/morisset/anaconda2/envs/py3k6/lib/python3.6/site-packages/ipykernel_launcher.py:3: RuntimeWarning:
  This is separate from the ipykernel package so we can avoid doing imports until
```

```
In [93]: for aa in a:
             print('{0:2} {1:4.2f} {2:5.2f} {3:8.2e}'.format(aa, np.log10(aa), np.sin(aa), np.exp(aa)))
```

```
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

/home/morisset/anaconda2/envs/py3k6/lib/python3.6/site-packages/ipykernel_launcher.py:2: RuntimeWarning:
```

sum

```
In [94]: print(a.sum())
         print(17*18/2)
```

```
153
153.0
```

```
In [95]: 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 [96]: for plane in a:
             for row in plane:
                 for col in row:
                     print(col)
                     print('-----')
```

```
0.660518893775
-----
0.293911298631
-----
0.444763908363
-----
0.831897448682
-----
0.900511773191
-----
0.318277757182
-----
0.17908201684
-----
0.458741347231
-----
0.678980933538
-----
0.414421162986
-----
0.0565193550914
-----
0.854027011068
-----
0.885621908519
-----
0.88823776014
-----
0.477196646584
-----
0.779967013628
-----
0.399219321332
-----
0.266255324504
-----
0.271085651875
-----
0.647189178778
-----
0.0402554039138
-----
```

```
0.384413914535
-----
0.682927667015
-----
0.693636309673
-----
```

```
In [97]: print(a[0,1,2]) # a[p, r, c]
```

```
0.318277757182
```

```
In [98]: a.sum()
```

```
Out[98]: 12.507659007074615
```

```
In [99]: a.sum(0) # from 3D to 2D. Generate an "image" of the sum, i.e. the "projection" on the x-axis
```

```
Out[99]: array([[ 1.5461408 ,  1.18214906,  0.92196055],
                [ 1.61186446,  1.29973109,  0.58453308],
                [ 0.45016767,  1.10593053,  0.71923634],
                [ 0.79883508,  0.73944702,  1.54766332]])
```

```
In [100]: a.sum(0).shape
```

```
Out[100]: (4, 3)
```

```
In [101]: a.sum(0).sum(0) # from 3D to 1D. From the image, make the sum in each row.
```

```
Out[101]: array([ 4.40700801,  4.3272577 ,  3.77339329])
```

```
In [102]: a.min(0)
```

```
Out[102]: array([[ 0.66051889,  0.2939113 ,  0.44476391],
                 [ 0.77996701,  0.39921932,  0.26625532],
                 [ 0.17908202,  0.45874135,  0.0402554 ],
                 [ 0.38441391,  0.05651936,  0.69363631]])
```

```
In [103]: a.ravel()
```

```
Out[103]: array([ 0.66051889,  0.2939113 ,  0.44476391,  0.83189745,  0.90051177,
                  0.31827776,  0.17908202,  0.45874135,  0.67898093,  0.41442116,
                  0.05651936,  0.85402701,  0.88562191,  0.88823776,  0.47719665,
                  0.77996701,  0.39921932,  0.26625532,  0.27108565,  0.64718918,
                  0.0402554 ,  0.38441391,  0.68292767,  0.69363631])
```

```
In [104]: i_min = a.argmin() # return the index of where the minimum is. It uses the 1D index.
          print(i_min)
          b = np.array([10,2,3,4,5,2])
          b.argmin() # only the first occurence
```

```
20
```

```
Out[104]: 1
```

```
In [105]: a.ravel().shape # 1D
```

```
Out[105]: (24,)
```

```
In [106]: a.ravel()[i_min] # Check where the minimum is.
```

```
Out[106]: 0.040255403913796117

In [107]: 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 2 2
0.0402554039138

In [108]: 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 [109]: res = decompose_ravel(a, i_min)
          print(a.min())
          print(res)
          print(a[res])

0.0402554039138
(1, 2, 2)
0.0402554039138

In [110]: a.min(0).min(0)

Out[110]: array([ 0.17908202,  0.05651936,  0.0402554 ])

In [111]: print(a[:,0,0])
          a[:,0,0].min()

[ 0.66051889  0.88562191]

Out[111]: 0.66051889377460848

In [112]: a.mean(0)

Out[112]: array([[ 0.7730704 ,  0.59107453,  0.46098028],
                 [ 0.80593223,  0.64986555,  0.29226654],
                 [ 0.22508383,  0.55296526,  0.35961817],
                 [ 0.39941754,  0.36972351,  0.77383166]])

In [113]: np.median(a, 1)

Out[113]: array([[ 0.53747003,  0.37632632,  0.56187242],
                 [ 0.58219046,  0.66505842,  0.37172599]])

In [114]: a.std()

Out[114]: 0.26165147397684535
```

```
In [115]: np.percentile(a, 25)

Out[115]: 0.3121861425445312

In [116]: print(a[0:4,0])
          print(np.cumsum(a[0:100,0])) # axis is a keyword. If absent, applied on the ravel(), e.g. 1D

[[ 0.66051889  0.2939113   0.44476391]
 [ 0.88562191  0.88823776  0.47719665]]
[ 0.66051889  0.95443019  1.3991941   2.28481601  3.17305377  3.65025042]

In [117]: b = np.arange(1000).reshape(10,10,10)

In [118]: b.shape

Out[118]: (10, 10, 10)

In [119]: b[4,:,:] # hundreds digits = 4

Out[119]: 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 [120]: b[:,2,:] # tens digit = 2

Out[120]: 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 [121]: b[:,:,7] # unity digit = 7

Out[121]: 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 [122]: b.min(0) # elements with the smallest value for the hundreds digit
```

```
Out[122]: 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 [123]: b.min(2) # smallest value for the unity digit

Out[123]: 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 [124]: b.min(2).shape

Out[124]: (10, 10)

In [125]: np.median(b)

Out[125]: 499.5

In [126]: np.median(b, axis=0)

Out[126]: 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 [127]: x = 2 * np.random.rand(100,100,100) - 1.
          print(np.min(x), np.max(x))
```

```
-0.999999398446 0.99999911124
```

```
In [128]: y = 2 * np.random.rand(100,100,100) - 1.
          z = 2 * np.random.rand(100,100,100) - 1.
```

```
In [129]: r = np.sqrt(x**2 + y**2 + z**2)
          print(np.min(r), np.max(r))
          print(np.sqrt(3))
```

```
0.0126214002214 1.71795924809
1.73205080757
```

```
In [130]: print(np.mean(r))
          print(r.mean())
```

```
0.960795050103
0.960795050103
```

```
In [131]: np.median(r)
```

```
Out[131]: 0.98489862714318832
```

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 with fewer dimensions is
If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that
If in any dimension the sizes disagree and neither is equal to 1, an error is raised.
```

```
In [132]: 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 [133]: x = np.array((1,2,3,4,5)).reshape(5,1,1)
```

```
In [134]: x
```

```
Out[134]: array([[[1]],

                 [[2]],

                 [[3]],

                 [[4]],

                 [[5]]])
```

```
In [135]: x.shape
```

```
Out[135]: (5, 1, 1)
```

```
In [136]: x.ndim
```

```
Out[136]: 3

In [137]: 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 [138]: r = x * y * z

In [139]: print(r.shape)

(5, 5, 5)

In [140]: r

Out[140]: 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 [141]: 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 [142]: a * b
```

```
Out[142]: 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 [143]: a * b.reshape(1,10)
```

```
Out[143]: 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 [144]: a = np.array([(1.5, 2), (3.0, 4)]) # Classical numpy array
          print(a)
```

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

```
In [145]: astru = np.array([(1.5, 2), (3.0, 4)], dtype=[('x', float), ('y', int)]) # array with named a
          astru
```

```
Out[145]: array([( 1.5, 2), ( 3. , 4)],
                dtype=[('x', '<f8'), ('y', '<i8')])
```

```
In [146]: print(astru['x'])
          print(astru['y'])
```

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

```
In [147]: 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 size
          print('----------------------------------')
          print(a.dtype, astru.dtype, arec.dtype) # types tell us that ar has column names and types
          print('----------------------------------')
          print(a[1,1], astru[1][1], arec[1][1]) # one is 2D, the other is a collection of 1D
          print('----------------------------------')
          print(astru['y']) # acces by name (a litle like dictionnaries)
          print('----------------------------------')
          print(arec.x)
```

```
<class 'numpy.ndarray'> <class 'numpy.ndarray'> <class 'numpy.recarray'>
----------------------------------
[[ 1.5  2. ]
 [ 3.   4. ]]
[( 1.5, 2) ( 3. , 4)]
[( 1.5, 2) ( 3. , 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 [148]: %timeit astru2 = np.append(astru, np.array([(5.0, 6)], dtype=astru.dtype)) # Copied all the da
```

```
10.6 µs ± 101 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [149]: %timeit astru3 = np.concatenate((astru, np.array([(5.0, 6)], dtype=astru.dtype))) # A little l
```

```
5.08 µs ± 50.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [150]: %timeit arec2 = np.append(arec, np.array([(5.0, 6)], dtype=astru.dtype).view(np.recarray)) # (

43.7 µs ± 740 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

In [151]: %timeit arec3 = np.concatenate((arec, np.array([(5.0, 6)], dtype=astru.dtype).view(np.recarray

26.2 µs ± 299 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

In [152]: arec4 = np.rec.fromrecords([(456,'dbe',1.2),(2,'de',1.3)],names='col1,col2,col3') # direct fr
          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 [153]: 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 [154]: 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, star.ra, star.dec,
          print('-------------------------')
          for star in arec4[mask]:
              print('name: {0[name]} ra = {0[ra]} dec = {0[dec]} magnitude = {0[mag]}'.format(star)) #

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

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

/home/morisset/anaconda2/envs/py3k6/lib/python3.6/site-packages/ipykernel_launcher.py:2: RuntimeWarning:
```

```
In [156]: print(a.sum())
          print(b.sum()) # NaN and others are absorbant elements
```

```
-3.0
inf
```

```
In [157]: mask = np.isfinite(b)
          print(mask)
          print(b[mask].sum())
```

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

```
In [158]: for elem in b:
              print(np.isinf(elem))
```

```
False
False
False
True
False
False
```

```
In [159]: a = np.array([-2, -1, 1., 2, 3])
          b = np.log10(a)
          mask = np.isfinite(b)
          print(a)
          print(b)
          print(mask)
          print(a.mean())
          print(b.mean())
          print(b[mask].mean())
          print(np.nanmean(b))
```

```
[-2. -1.  1.  2.  3.]
[        nan         nan  0.          0.30103     0.47712125]
[False False  True  True  True]
0.6
nan
0.259383750128
0.259383750128
```

```
/home/morisset/anaconda2/envs/py3k6/lib/python3.6/site-packages/ipykernel_launcher.py:2: RuntimeWarning:
```

1.0.12 Roundish values of floats

```
In [160]: import math
          res = []
          for i in range(100):
              res.append(math.log(2 ** i, 2)) # The second argument is the base of the log. The result
          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, 15.0, 16.0, 17.0, 18.0
```

```
In [161]: 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 should be always True
```

```
[True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True,
```

```
In [162]: res = []
          for i in range(100):
              res.append(np.log2(2.**i)) # The second argument is the base of the log. The result shoul
          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, 15.0, 16.0, 17.0, 18.0
[True, True, 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 [163]: res_np2 = []
          for i in range(100):
              res_np2.append(np.isclose(float(round(math.log(2 ** i, 2))), math.log(2 ** i, 2)))
          print(res_np2)
          # The isclose
```

```
[True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True,
```

```
In [164]: np.isclose?
```

```
In [ ]:
```