intro_numpy

January 10, 2021

[2]:
```
# 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])
```

```
Mon Oct 12 19:08:54 2020
3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)]
```

1 B Numpy

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

1.0.1 Import numpy first

[3]:
```
# 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
```

[4]:
```
print(np.__version__)
```

```
1.18.1
```

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

[5]:
```
# 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))
```

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

[6]:
```
# Works with tuples also:
b = np.array((1,2,3))
print(b)
```

```
[1 2 3]
```

Numpy arrays are efficiently connected to the computer:

[7]:
```
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
```

```
293 µs ± 19.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
842 ns ± 7.13 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

[8]:
```
L = [1, 2, 3, 4]
a = np.array(L)
print(a.dtype)
print(a)
```

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

[9]:
```
L = [1, 2, 3, 4.]
a = np.array(L)
print(a.dtype)
print(a)
```

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

[10]:
```
L = [1, 2, 3, 4.5, '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, 4.5, 'a']
<U32
['1' '2' '3' '4.5' '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

[11]:
```python
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]
```

[12]:
```python
a[2] = '3.2'
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-81ec24b7f2d9> in <module>
----> 1 a[2] = '3.2'

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

[15]:
```python
a[2] = 'tralala'
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-15-2774d8249152> in <module>
----> 1 a[2] = 'tralala'

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

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

[16]:
```python
a = np.array([1,2,3,4,5,6])
b = np.array([[1,2],[1,4], [4,7]])
c = np.array([[[1], [2]], [[3], [4]]])
print(a.shape, b.shape, c.shape)
print(b)
print(b[0]) # no error
```

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

```
 [1 4]
 [4 7]]
[1 2]
```

[17]:
```
print(len(a), len(b), len(c)) # size of the first dimension
```

```
6 3 2
```

[18]:
```
b.size
```

[18]:
```
6
```

[19]:
```
print(a.ndim, b.ndim, c.ndim)
```

```
1 2 3
```

[20]:
```
a = np.array([1,2,3,4,5,6])
print('mean: {0}, max: {1}, shape: {2}, median: {3}'.format(a.mean(), a.max(),␣
 ↪a.shape, np.median(a)))
```

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

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

[21]:
```
print(a.mean) # this is printing information about the function, NOT the result␣
 ↪of the function!
```

```
<built-in method mean of numpy.ndarray object at 0x7fd0f9fb3990>
```

[22]:
```
mm = a.mean # We assign to mn the function. Then we can call it directly, but␣
 ↪still need for the ()s:
print(mm())
```

```
3.5
```

[23]:
```
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, axis=1))
```

```
[[1 2]
 [1 4]
 [4 7]]
3.1666666666666665
[2.         4.33333333]
[1.5 2.5 5.5]
[1.5 2.5 5.5]
```

[24]:
```
np.mean?
```

```
Signature: np.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>)
Docstring:
Compute the arithmetic mean along the specified axis.

Returns the average of the array elements.  The average is taken over
the flattened array by default, otherwise over the specified axis.
`float64` intermediate and return values are used for integer inputs.

Parameters
----------
a : array_like
    Array containing numbers whose mean is desired. If `a` is not an
    array, a conversion is attempted.
axis : None or int or tuple of ints, optional
    Axis or axes along which the means are computed. The default is to
    compute the mean of the flattened array.

    .. versionadded:: 1.7.0

    If this is a tuple of ints, a mean is performed over multiple axes,
    instead of a single axis or all the axes as before.
dtype : data-type, optional
    Type to use in computing the mean.  For integer inputs, the default
    is `float64`; for floating point inputs, it is the same as the
    input dtype.
out : ndarray, optional
    Alternate output array in which to place the result.  The default
    is ``None``; if provided, it must have the same shape as the
    expected output, but the type will be cast if necessary.
    See `ufuncs-output-type` for more details.

keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the input array.

    If the default value is passed, then `keepdims` will not be
    passed through to the `mean` method of sub-classes of
    `ndarray`, however any non-default value will be.  If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.

Returns
-------
m : ndarray, see dtype parameter above
```

```
    If `out=None`, returns a new array containing the mean values,
    otherwise a reference to the output array is returned.

See Also
--------
average : Weighted average
std, var, nanmean, nanstd, nanvar

Notes
-----
The arithmetic mean is the sum of the elements along the axis divided
by the number of elements.

Note that for floating-point input, the mean is computed using the
same precision the input has.  Depending on the input data, this can
cause the results to be inaccurate, especially for `float32` (see
example below).  Specifying a higher-precision accumulator using the
`dtype` keyword can alleviate this issue.

By default, `float16` results are computed using `float32` intermediates
for extra precision.

Examples
--------
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])

In single precision, `mean` can be inaccurate:

>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924

Computing the mean in float64 is more accurate:

>>> np.mean(a, dtype=np.float64)
0.55000000074505806 # may vary
File:      ~/anaconda3/lib/python3.7/site-packages/numpy/core/fromnumeric.py
Type:      function
```

Creating arrays from scratch

[25]:
```
print(np.arange(10))
```

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

[26]:
```
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]
```

[27]:
```
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.        ]
```

[28]:
```
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(b)) # This is very usefull: using an already created array␣
↪(or list or tuple) as example for the shape of the new one.
print('--------------------------------')
print(b)
print(np.zeros_like(b, dtype=float) + 3) # Can define a value to fille the␣
↪array when creating it. Or latter...
print(np.ones_like(b, dtype=float) * 3) # Can define a value to fille the array␣
↪when creating it. Or latter...
```

```
[0. 0.]
--------------------------------
[[0. 0. 0.]
 [0. 0. 0.]]
--------------------------------
[[1 1]
 [1 1]
 [1 1]]
--------------------------------
[[1 2]
 [1 4]
 [4 7]]
```

```
[[3. 3.]
 [3. 3.]
 [3. 3.]]
[[3. 3.]
 [3. 3.]
 [3. 3.]]
```

[29]:
```python
new_a = a.reshape((3,2)) # This does NOT change the shape of a
print(a)
print('-------------')
print(new_a)
```

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

[30]:
```python
print(new_a.ravel())
print(new_a.reshape(new_a.size))
```

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

[31]:
```python
# 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.shape)
print(x)
print('------------------------------------')
print(y.shape)
print(y)
```

```
(5, 10)
[[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, 10)
[[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]]
```

[32]:
```python
# 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]]
```

[33]:
```python
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

[34]:
```python
b = a.reshape((3,2))
print(a.shape, b.shape)
```

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

[35]:
```python
b[1,1] = 100 # modify a value in the array
print(b)
```

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

[36]:
```python
print(a) # !!! a and b are sharing the same place in the memory, they are␣
 ↪pointing to the same values.
```

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

[37]:
```python
b[1,1], a[3] # same value
```

[37]: (100, 100)

[38]:
```python
a is b # a and b are different
```

[38]: False

[39]:
```python
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
```

[40]:
```python
c = a.reshape((2,3)).copy() # This is the solution.
```

[41]:
```python
print(a)
print('---------------')
print(c)
```

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

[42]:
```python
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

[43]:
```python
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)
```

```
[0.00493762 0.56611721 0.73756545 0.87137075 0.97132855]
-----------------------------
[-0.51035597 -0.22744195 -0.44674488 -2.11004404  0.03959021]
-----------------------------
[[-0.14487643  1.41711047 -0.19250961  2.72138627 -1.32575917]
 [ 1.308128    0.33684976  0.85683118  0.67287322 -0.13687007]
 [-2.37974623 -0.9063964  -1.1233723   1.42649559 -0.16433894]
 [-0.43420222 -0.77281828 -1.57142812  0.09280614 -1.19114424]
 [ 0.29052205 -0.19884749  0.1487806   1.48351285  1.95064135]]
```

[44]:
```python
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]
[0.09233859 0.18626021 0.34556073 0.39676747 0.53881673]
```

1.0.5 Timing on 2D array

[45]:
```python
N = 100
A = np.random.rand(N, N)
B = np.zeros_like(A)
```

[46]:
```python
%%timeit
for i in range(N):
    for j in range(N):
        B[i,j] = A[i,j]
```

```
2.38 ms ± 131 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[47]:
```python
%%timeit
B = A # very faster ! It does NOT copy...
```

```
19.3 ns ± 0.222 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

[48]:
```python
%%timeit
B = (A.copy()) # Takes more time
```

```
2.61 µs ± 210 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[49]:
```python
%%timeit
for i in range(N):
    for j in range(N):
        B[i,j] = A[i,j]**2
```

```
4.85 ms ± 39.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[50]:
```
%%timeit
B = A**2 # very faster ! Does a copy
```

```
5.69 µs ± 193 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[51]:
```
%timeit  B = (A.copy())**2 # Takes a little bit more time
```

```
8.75 µs ± 51 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

1.0.6 Slicing

[52]:
```
a = np.arange(10)
print(a)
print(a[1:8:3])
```

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

[53]:
```
print(a[:7])
```

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

[54]:
```
print(a[4:])
```

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

[55]:
```
print(a[::2])
print(a[::2][2])
```

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

[56]:
```
# Revert the array:
print(a[::-1])
```

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

Assignment

[57]:
```
a[5:] = 999
print(a)
```

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

[58]:
```
a[5:] = a[4::-1]
print(a)
```

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

[59]:
```python
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)
```

[60]:
```python
b * c # Cross product, see below (broadcasting)
```

[60]:
```
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

[61]:
```python
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]
```

[62]:
```python
# 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

[63]:
```python
a = np.random.randint(0, 100, 20) # min, max, N
print(a)
```

```
[78 46 26 63 86  2 96 45 13 67 37 36 54 63 65 58 49 48 59 26]
```

[64]:
```python
a < 50
```

[64]:
```
array([False,  True,  True, False, False,  True, False,  True,  True,
       False,  True,  True, False, False, False, False,  True,  True,
       False,  True])
```

[65]:
```python
mask = (a < 50)
```

[66]:
```python
mask.sum()
```

[66]:
```
10
```

[67]:
```python
a[mask]
```

[67]:
```
array([46, 26,  2, 45, 13, 37, 36, 49, 48, 26])
```

[68]:
```python
b = a.copy() # do NOT use b = a
b[mask] = 50 #
print(a)
print(b)
```

```
[78 46 26 63 86  2 96 45 13 67 37 36 54 63 65 58 49 48 59 26]
[78 50 50 63 86 50 96 50 50 67 50 50 54 63 65 58 50 50 59 50]
```

[69]:
```python
b = a.copy()
b[b <= 50] = -1 # shortest way. Not matter if not even one element fit the test
print(a)
print(b)
```

```
[78 46 26 63 86  2 96 45 13 67 37 36 54 63 65 58 49 48 59 26]
[78 -1 -1 63 86 -1 96 -1 -1 67 -1 -1 54 63 65 58 -1 -1 59 -1]
```

[70]:
```python
print(a[mask])
print(a[~mask]) # complementary
```

```
[46 26  2 45 13 37 36 49 48 26]
[78 63 86 96 67 54 63 65 58 59]
```

[71]:
```python
mask
```

[71]:
```
array([False,  True,  True, False, False,  True, False,  True,  True,
       False,  True,  True, False, False, False, False,  True,  True,
       False,  True])
```

[72]:
```
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]
```

[73]:
```
mask[[2,3,4]] = True
```

[74]:
```
mask
```

[74]:
```
array([False, False,  True,  True,  True, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False])
```

[75]:
```
a[mask]
```

[75]:
```
array([26, 63, 86])
```

[76]:
```
a[mask].sum()
```

[76]:
```
175
```

[77]:
```
a[mask].mean()
```

[77]:
```
58.333333333333336
```

combining masks

[78]:
```
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
```

```
[78 46 26 63 86  2 96 45 13 67 37 36 54 63 65 58 49 48 59 26]
------------------------------------
[46 63 45 67 37 36 54 63 65 58 49 48 59]
------------------------------------
[78 26 86  2 96 13 26]
```

the where function

[79]:
```python
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.
```

```
[78 46 26 63 86  2 96 45 13 67 37 36 54 63 65 58 49 48 59 26]
(array([ 0,  1,  3,  4,  6,  7,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18]),)
```

[80]:
```python
(a > 30).nonzero() # "where" is the same than condition.nonzero().
```

[80]:
```
(array([ 0,  1,  3,  4,  6,  7,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18]),)
```

[81]:
```python
# the indices where the condition is filled are in the first element of the
↪tuple
```

[82]:
```python
tt[0]
```

[82]:
```
array([ 0,  1,  3,  4,  6,  7,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
```

[83]:
```python
# faster once you know that the condition is 1D
tt = np.where(a > 30)[0]
```

[84]:
```python
tt # the array containing the indices where the condition is filled
```

[84]:
```
array([ 0,  1,  3,  4,  6,  7,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
```

[85]:
```python
a[tt] # the values where the condition is filled
```

[85]:
```
array([78, 46, 63, 86, 96, 45, 67, 37, 36, 54, 63, 65, 58, 49, 48, 59])
```

[86]:
```python
# The where function can take 3 arguments.
b = np.where(a < 50, np.nan, a)
print(a)
print(b)
print(np.isfinite(b))
```

```
[78 46 26 63 86  2 96 45 13 67 37 36 54 63 65 58 49 48 59 26]
[78. nan nan 63. 86. nan 96. nan nan 67. nan nan 54. 63. 65. 58. nan nan
 59. nan]
[ True False False  True  True False  True False False  True False False
  True  True  True  True False False  True False]
```

[87]:
```python
b = np.where(a < 50, True, False)
print(a)
print(b)
```

```
[78 46 26 63 86  2 96 45 13 67 37 36 54 63 65 58 49 48 59 26]
[False  True  True False False  True False  True  True False  True  True
```

```
 False False False False  True  True False  True]
```

[88]:
```
b = np.where(a < 50, 0, 100)
print(a)
print(b)
```

```
[78 46 26 63 86  2 96 45 13 67 37 36 54 63 65 58 49 48 59 26]
[100   0   0 100 100   0 100   0   0 100   0   0 100 100 100 100   0   0
 100   0]
```

1.0.8 Some operations with arrays

[90]:
```
a
```

[90]:
```
array([78, 46, 26, 63, 86,  2, 96, 45, 13, 67, 37, 36, 54, 63, 65, 58, 49,
       48, 59, 26])
```

[91]:
```
a + 1
```

[91]:
```
array([79, 47, 27, 64, 87,  3, 97, 46, 14, 68, 38, 37, 55, 64, 66, 59, 50,
       49, 60, 27])
```

[92]:
```
a**2 + 3*a**3
```

[92]:
```
array([1429740,  294124,   53404,  754110, 1915564,      28, 2663424,
        275400,    6760,  906778,  153328,  141264,  475308,  754110,
        828100,  588700,  355348,  334080,  619618,   53404])
```

[93]:
```
# 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
```

[94]:
```
c = (i+2)**2
```

[95]:
```
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]
```

[96]:
```
b == c
```

[96]:
```
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])
```

[97]:
```
i[b==c]
```

[97]:
```
array([3])
```

[98]:
```
i[b==c][0] # the result is an array. To obtain the first value (here the only␣
 ↪one), use [0]
```

[98]:
```
3
```

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

[99]:
```
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/anaconda3/lib/python3.7/site-
packages/ipykernel_launcher.py:3: RuntimeWarning: divide by zero encountered in
log10
  This is separate from the ipykernel package so we can avoid doing imports
until
```

[100]:
```
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
```

```
/Users/christophemorisset/anaconda3/lib/python3.7/site-
packages/ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in
log10
```

sum

[103]:
```
print(a.sum())
print(17*18/2)
```

```
153
153.0
```

[104]:
```
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:

[105]:
```
for plane in a:
    for row in plane:
        for col in row:
            print(col)
            print('-----')
```

```
0.9005117731906158
-----
0.318277757182466
-----
0.17908201684049108
-----
0.45874134723090876
-----
0.6789809335383882
-----
0.414421162986482
-----
```

```
0.056519355091393386
-----
0.8540270110679247
-----
0.8856219085188352
-----
0.888237760139903
-----
0.4771966465839834
-----
0.7799670136275508
-----
0.39921932133177807
-----
0.266255324504438
-----
0.27108565187535727
-----
0.6471891787778047
-----
0.04025540391379612
-----
0.38441391453496976
-----
0.6829276670149604
-----
0.6936363096727493
-----
0.8700864875082074
-----
0.7815463993013263
-----
0.17645130549021926
-----
0.240524011415837
-----
```

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

[106]:
```python
a.sum()
```

[106]: 12.345175661340384

[107]:
```python
a.sum(0) # from 3D to 2D. Generate an "image" of the sum, i.e. the "projection"␣
↪on the x-axis of the 3D array
```

[107]:
```
array([[1.29973109, 0.58453308, 0.45016767],
       [1.10593053, 0.71923634, 0.79883508],
       [0.73944702, 1.54766332, 1.7557084 ],
       [1.66978416, 0.65364795, 1.02049103]])
```

[108]:
```
a.sum(0).shape
```

[108]:
```
(4, 3)
```

[109]:
```
a.sum(0).sum(0) # from 3D to 1D. From the image, make the sum in each row.
```

[109]:
```
array([4.8148928 , 3.50508069, 4.02520217])
```

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

[110]:
```
array([[0.39921932, 0.26625532, 0.17908202],
       [0.45874135, 0.0402554 , 0.38441391],
       [0.05651936, 0.69363631, 0.87008649],
       [0.7815464 , 0.17645131, 0.24052401]])
```

[111]:
```
a.ravel()
```

[111]:
```
array([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,
       0.87008649, 0.7815464 , 0.17645131, 0.24052401])
```

[112]:
```
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
```

```
16
```

[112]:
```
1
```

[113]:
```
a.ravel().shape # 1D
```

[113]:
```
(24,)
```

[114]:
```
a.ravel()[i_min] # Check where the minimum is.
```

[114]:
```
0.04025540391379612
```

[115]:
```
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 1 1
0.040255540391379612
```

[116]:
```
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)
```

[117]:
```
res = decompose_ravel(a, i_min)
print(a.min())
print(res)
print(a[res])
```

```
0.040255540391379612
(1, 1, 1)
0.040255540391379612
```

[118]:
```
a.min(0).min(0)
```

[118]:
```
array([0.05651936, 0.0402554 , 0.17908202])
```

[119]:
```
print(a[:,0,0])
a[:,0,0].min()
```

```
[0.90051177 0.39921932]
```

[119]:
```
0.39921932133177807
```

[120]:
```
a.mean(0)
```

[120]:
```
array([[0.64986555, 0.29226654, 0.22508383],
       [0.55296526, 0.35961817, 0.39941754],
       [0.36972351, 0.77383166, 0.8778542 ],
       [0.83489208, 0.32682398, 0.51024551]])
```

[121]:
```python
np.median(a, 1)
```

[121]:
```
array([[0.67348955, 0.57808879, 0.59719409],
       [0.66505842, 0.22135331, 0.32774978]])
```

[124]:
```python
a.std(2)
```

[124]:
```
array([[0.31248694, 0.11569194, 0.38361261, 0.17395633],
       [0.0615729 , 0.24852108, 0.08581489, 0.27140593]])
```

[125]:
```python
np.percentile(a, 25)
```

[125]:
```
0.2698780700326274
```

[126]:
```python
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 array
```

```
[[0.90051177 0.31827776 0.17908202]
 [0.39921932 0.26625532 0.27108565]]
[0.90051177 1.21878953 1.39787155 1.79709087 2.06334619 2.33443184]
```

[127]:
```python
b = np.arange(1000).reshape(10,10,10)
```

[128]:
```python
b.shape
```

[128]:
```
(10, 10, 10)
```

[129]:
```python
b[4,:,:] # hundreds digits = 4
```

[129]:
```
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]])
```

[130]:
```python
b[:,2,:] # tens digit = 2
```

[130]:
```
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]])
```

[131]:
```
b[:,:,7] # unity digit = 7
```

[131]:
```
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]])
```

[132]:
```
b.min(0) # elements with the smallest value for the hundreds digit
```

[132]:
```
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]])
```

[133]:
```
b.min(2) # smallest value for the unity digit
```

[133]:
```
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]])
```

[134]:
```
b.min(2).shape
```

[134]: (10, 10)

[135]:
```
np.median(b)
```

[135]: 499.5

[136]:
```
np.median(b, axis=0)
```

[136]:
```
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.]])
```

[137]:
```
x = 2 * np.random.rand(100,100,100) - 1.
print(np.min(x), np.max(x))
```

```
-0.9999993984462574 0.9999991112403628
```

[138]:
```
y = 2 * np.random.rand(100,100,100) - 1.
z = 2 * np.random.rand(100,100,100) - 1.
```

[139]:
```
r = np.sqrt(x**2 + y**2 + z**2)
print(np.min(r), np.max(r))
print(np.sqrt(3))
```

```
0.01262140022137408 1.7179592480943136
1.7320508075688772
```

[140]:
```
print(np.mean(r))
print(r.mean())
```

```
0.9607950001733658
0.9607950001733658
```

[141]:
```
np.median(r)
```

[141]: 0.9848986271431883

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

[142]:
```python
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,)
```

[143]:
```python
x = np.array((1,2,3,4,5)).reshape(5,1,1)
```

[144]:
```python
x
```

[144]:
```
array([[[1]],

       [[2]],

       [[3]],

       [[4]],

       [[5]]])
```

[145]:
```python
x.shape
```

[145]:
```
(5, 1, 1)
```

[146]:
```python
x.ndim
```

[146]:
```
3
```

[147]:
```python
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]]]
```

[148]:
```python
r = x * y * z
```

[149]:
```python
print(r.shape)
```

```
(5, 5, 5)
```

[150]:
```python
r
```

[150]:
```
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]]])
```

[151]:
```python
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)
```

[153]:
```
a * b
```

[153]:
```
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.]])
```

[154]:
```
a * b.reshape(1,10)
```

[154]:
```
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.

[155]:
```python
a = np.array([(1.5, 2), (3.0, 4)]) # Classical numpy array
print(a)
```

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

[156]:
```python
astru = np.array([(1.5, 2), (3.0, 4)], dtype=[('x', float), ('y', int)]) #␣
↪array with named and typed columns
astru
```

[156]:
```
array([(1.5, 2), (3. , 4)], dtype=[('x', '<f8'), ('y', '<i8')])
```

[157]:
```python
print(astru['x'])
print(astru['y'])
```

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

[158]:
```python
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␣
↪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. ]
```

[ ]:
```
%timeit astru2 = np.append(astru, np.array([(5.0, 6)], dtype=astru.dtype)) #␣
↪Copied all the data, may be slow
```

[ ]:
```
%timeit astru3 = np.concatenate((astru, np.array([(5.0, 6)], dtype=astru.
↪dtype))) # A little bit faster
```

[ ]:
```
%timeit arec2 = np.append(arec, np.array([(5.0, 6)], dtype=astru.dtype).view(np.
↪recarray)) # Copied all the data, may be slow
```

[ ]:
```
%timeit arec3 = np.concatenate((arec, np.array([(5.0, 6)], dtype=astru.dtype).
↪view(np.recarray))) # A little bit faster
```

[159]:
```
arec4 = np.rec.fromrecords([(456,'dbe',1.2),(2,'de',1.
↪3)],names='col1,col2,col3') # direct from data.
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
```

[160]:
```
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')
```

[ ]:
```
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, star.mag))
print('-------------------------')
```

```
for star in arec4[mask]:
    print('name: {0[name]} ra = {0[ra]} dec = {0[dec]} magnitude = {0[mag]}'.
 ↪format(star)) # unse only one key in format
```

1.0.11 NaN and other ANSI values

[161]:
```
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/anaconda3/lib/python3.7/site-
packages/ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in
true_divide
```

[162]:
```
print(a.sum())
print(b.sum()) # NaN and others are absorbant elements
```

```
-3.0
inf
```

[163]:
```
mask = np.isfinite(b)
print(mask)
print(b[mask].sum())
```

```
[ True  True  True False  True  True]
-0.33333333333333326
```

[164]:
```
for elem in b:
    print(np.isinf(elem))
```

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

[165]:
```
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
0.25938375012788123
0.25938375012788123
```

```
/Users/christophemorisset/anaconda3/lib/python3.7/site-
packages/ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
log10
```

1.0.12 Roundish values of floats

[166]:
```
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 should be i.
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, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0,
28.0, 29.000000000000004, 30.0, 31.000000000000004, 32.0, 33.0, 34.0, 35.0,
36.0, 37.0, 38.0, 39.00000000000001, 40.0, 41.0, 42.0, 43.0, 44.0, 45.0, 46.0,
47.00000000000001, 48.0, 49.0, 50.0, 51.00000000000001, 52.0, 53.0, 54.0,
55.00000000000001, 56.0, 57.0, 58.00000000000001, 59.00000000000001, 60.0, 61.0,
62.00000000000001, 63.0, 64.0, 65.0, 66.0, 67.0, 68.0, 69.0, 70.0, 71.0, 72.0,
73.0, 74.0, 75.0, 76.0, 77.0, 78.00000000000001, 79.0, 80.0, 81.0, 82.0, 83.0,
84.0, 85.0, 86.0, 87.0, 88.0, 89.0, 90.0, 91.0, 92.0, 93.00000000000001,
94.00000000000001, 95.00000000000001, 96.0, 97.0, 98.0, 99.0]
```

[167]:
```
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, True, True, True, True, True, True, True, True, True, True,
True, True, True, False, True, False, True, True, True, True, True, True, True,
False, True, True, True, True, True, True, True, False, True, True, True, False,
```

```
True, True, True, False, True, True, False, False, True, True, False, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, False, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, False, False, False, True, True, True, True]
```

[168]:
```
res = []
for i in range(100):
    res.append(np.log2(2.**i)) # The second argument is the base of the log.␣
↪The result should be i.
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, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0,
28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40.0,
41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0, 48.0, 49.0, 50.0, 51.0, 52.0, 53.0,
54.0, 55.0, 56.0, 57.0, 58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0, 65.0, 66.0,
67.0, 68.0, 69.0, 70.0, 71.0, 72.0, 73.0, 74.0, 75.0, 76.0, 77.0, 78.0, 79.0,
80.0, 81.0, 82.0, 83.0, 84.0, 85.0, 86.0, 87.0, 88.0, 89.0, 90.0, 91.0, 92.0,
93.0, 94.0, 95.0, 96.0, 97.0, 98.0, 99.0]
[True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, 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:

[169]:
```
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, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
```

```
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True]
```

[170]:

```
np.isclose?
```

```
Signature: np.isclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
Docstring:
Returns a boolean array where two arrays are element-wise equal within a
tolerance.

The tolerance values are positive, typically very small numbers.  The
relative difference (`rtol` * abs(`b`)) and the absolute difference
`atol` are added together to compare against the absolute difference
between `a` and `b`.

.. warning:: The default `atol` is not appropriate for comparing numbers
             that are much smaller than one (see Notes).

Parameters
----------
a, b : array_like
    Input arrays to compare.
rtol : float
    The relative tolerance parameter (see Notes).
atol : float
    The absolute tolerance parameter (see Notes).
equal_nan : bool
    Whether to compare NaN's as equal.  If True, NaN's in `a` will be
    considered equal to NaN's in `b` in the output array.

Returns
-------
y : array_like
    Returns a boolean array of where `a` and `b` are equal within the
    given tolerance. If both `a` and `b` are scalars, returns a single
    boolean value.

See Also
--------
allclose

Notes
-----
.. versionadded:: 1.7.0

For finite values, isclose uses the following equation to test whether
two floating point values are equivalent.
```

```
 absolute(`a` - `b`) <= (`atol` + `rtol` * absolute(`b`))

Unlike the built-in `math.isclose`, the above equation is not symmetric
in `a` and `b` -- it assumes `b` is the reference value -- so that
`isclose(a, b)` might be different from `isclose(b, a)`. Furthermore,
the default value of atol is not zero, and is used to determine what
small values should be considered close to zero. The default value is
appropriate for expected values of order unity: if the expected values
are significantly smaller than one, it can result in false positives.
`atol` should be carefully selected for the use case at hand. A zero value
for `atol` will result in `False` if either `a` or `b` is zero.

Examples
--------
>>> np.isclose([1e10,1e-7], [1.00001e10,1e-8])
array([ True, False])
>>> np.isclose([1e10,1e-8], [1.00001e10,1e-9])
array([ True, True])
>>> np.isclose([1e10,1e-8], [1.0001e10,1e-9])
array([False,  True])
>>> np.isclose([1.0, np.nan], [1.0, np.nan])
array([ True, False])
>>> np.isclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
array([ True, True])
>>> np.isclose([1e-8, 1e-7], [0.0, 0.0])
array([ True, False])
>>> np.isclose([1e-100, 1e-7], [0.0, 0.0], atol=0.0)
array([False, False])
>>> np.isclose([1e-10, 1e-10], [1e-20, 0.0])
array([ True,  True])
>>> np.isclose([1e-10, 1e-10], [1e-20, 0.999999e-10], atol=0.0)
array([False,  True])
File:      ~/anaconda3/lib/python3.7/site-packages/numpy/core/numeric.py
Type:      function
```