numpy_intro

October 18, 2023

1 Numpy intro

Numpy is a Python library for scientific computing which provides routines to handle large data in a fast way, like operations on multidimensional arrays.

It contains a set of routines for different operations:

- read/write files
- mathematical
- linear algebraic
- logical
- random data
- statistical
- sorting
- selecting
- FFTs

with a large number of functions in the field of linear algebra, Fourier transform, random number capabilities, boolean masks, and efficient handling of N-dimensional arrays.

To get more informations about Numpy see https://numpy.org/.

1.1 Content:

- Import numpy
- Arrays
- Create an array
- Convert lists to arrays
- Array attributes
- Change type of the array
- Indexing and slicing
- Reshape an array

- Math operations
- Broadcasting
- Math operations
- Mathematical attributes and functions
- Creating arrays for masking
- Array stacking
- Shallow and deep copy
- Most useful functions
- Statistically functions
- Read and write data from file
- Read data from CSV file
- Masking
- Some hints
- Difference between Numpy and Pandas

1.2 Import numpy

First, we have to import the numpy library to our environment and, just to make it shorter, we want to use an abbreviation np instead of numpy.

`import numpy as np`

[1]:
```
import numpy as np
```

1.3 Arrays

Numpy's main object is an array, which is data structure containing the raw data, location and interpreting methods. Unlike Python lists, whose elements can have different data types, Numpy array elements must be all of the same data type.

Note: For the sake of simplicity, a Numpy array will be called only array in the following.

marks the output of a command

Numpy array wording:

- **Axes** another name for dimensions
- **length** is the number of elements
- **shape** shows the number of dimensions (axes)
- **data** the values of the elements

1.4 Create an array

There are different ways to create an array. You can use one of the Numpy functions to generate a new array or to convert a list to an array.

1.4.1 Create basic array with `np.array`

At the beginning, we want to show how easy it is to create an array with `np.array`.

1-dimensional array:

```
array1d = np.array([1,2])

print(array1d)
```

[1 2]

n-dimensional array:

```
array2d = np.array([[1,2],[3,4]])

print(array2d)
```

[[1 2] [3 4]]

[2]:
```
array1d = np.array([1,2])

print(array1d)
```

```
[1 2]
```

[3]:
```
array2d = np.array([[1,2],[3,4]])

print(array2d)
```

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

1.4.2 Alternative ways

Now, we want to show how to create an array with `np.arange`.

1. Generate an array with 5 elements

```
A = np.arange(5)
print(A)
```

[0 1 2 3 4]

2. Generate an array with 5 elements of type *float*

```
A = np.arange(5.0)
print(A)
```

[0. 1. 2. 3. 4.]

3. Generate an array with start value 2, end value 6

```
A = np.arange(2,6)
print(A)
```

[2 3 4 5]

4. Generate an array with start value 2, end value 6 with increment 2

```
A = np.arange(2,6,2)
print(A)
```

[2 4]

Note: If not set the increment is 1 and the value of the first element is 0.

If you want to create an array on n-elements without knowing the exact increment the `np.linspace` function is what you are looking for. You have to give the start and end value, and the number of elements to be generated.

For example

```
B = np.linspace(1, 2, num=11)

print(B)
```

[1. 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.]

Try it out

Just do it yourself.

[4]:
```
A = np.array([1,2,3])
print(A)

B = np.arange(2., 5., .5)
print(B)

C = np.linspace(1,2,num=11)
print(C)
```

```
[1 2 3]
[2.  2.5 3.  3.5 4.  4.5]
[1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2. ]
```

1.5 Convert lists to arrays

If you already have an list variable you can use the function `np.array`to convert it to an array.

```
x = [10, 12, 14, 29, 18, 21]

print(x, type(x))
```

[10, 12, 14, 29, 18, 21] <class 'list'>

```
B = np.array(x)

print(B, type(B))
```

[10 12 14 29 18 21] <class 'numpy.ndarray'>

Try it out

1. Create a list variable
2. Convert the list to an array

[5]:
```
# 1.
x = [10, 12, 14, 29, 18, 21]
```

[6]:
```
# 2.
B = np.array(x)
```

1.6 Array attributes

Numpy provides attributes for the arrays to retrieve information about the dimension size, shape, size, data type, and data itself.

```
ndim  = B.ndim
shape = B.shape
size  = B.size
dtype = B.dtype

print('--> n dimensions: ', ndim)
print('--> shape: ', shape)
print('--> size : ', size)
print('--> dtype: ', dtype)
```

–> n dimensions: 1 –> shape: (6,) –> size : 6 –> dtype: int64

[7]:
```
B = np.array([10, 12, 14, 29, 18, 21])
```

[8]:
```
ndim  = B.ndim
shape = B.shape
size  = B.size
dtype = B.dtype
```

[9]:
```
print('--> n dimensions: ', ndim)
print('--> shape: ', shape)
print('--> size : ', size)
print('--> dtype: ', dtype)
```

```
--> n dimensions:  1
--> shape:  (6,)
--> size :  6
--> dtype:  int64
```

1.7 Change type of the array

You can convert the array from the given data type to another. Here, we want to convert an array of type integer to float.

```
F = B.astype(float)

print('--> type of B: ', B.dtype)
print('--> type of F: ', F.dtype)
```

–> type of B: int64 –> type of F: float64

[10]:
```
F = B.astype(float)
print('--> type of B: ', B.dtype)
print('--> type of F: ', F.dtype)
```

```
--> type of B:  int64
--> type of F:  float64
```

1.8 Indexing and slicing

Indexing and sclicing of arrays is equivalent to indexing and slicing of Python lists.

```
print('--> B[:]:    ', B[:])
print('--> B[1]):   ', B[1])
print('--> B[0:3]:  ', B[0:3])
print('--> B[:3]:   ', B[:3])
print('--> B[3:]:   ', B[3:])
print('--> B[-1]:   ', B[-1])
print('--> B[-3:-1]:', B[-3:-1])
```

Output:

```
--> B[:]:     [10 12 14 29 18 21]
--> B[1]):    12
--> B[0:3]:   [10 12 14]
--> B[:3]:    [10 12 14]
--> B[3:]:    [29 18 21]
--> B[-1]:    21
--> B[-3:-1]: [29 18]
```

[11]:
```
print('--> B[:]:    ', B[:])
print('--> B[1]):   ', B[1])
print('--> B[0:3]:  ', B[0:3])
print('--> B[:3]:   ', B[:3])
print('--> B[3:]:   ', B[3:])
print('--> B[-1]:   ', B[-1])
print('--> B[-3:-1]:', B[-3:-1])
```

```
--> B[:]:     [10 12 14 29 18 21]
--> B[1]):    12
--> B[0:3]:   [10 12 14]
--> B[:3]:    [10 12 14]
--> B[3:]:    [29 18 21]
--> B[-1]:    21
--> B[-3:-1]: [29 18]
```

1.9 Reshape an array

So far we've only worked with 1-dimensional arrays in this section, and we'll see how easy it is to convert them into 2- or 3-dimensional arrays.

```
D = np.arange(0, 12, 1,)
print('D: ', D)
```

```
D:  [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

```
D_2d = D.reshape(4, 3)
print('D_2d: ', D_2d)
```

```
D_2d:  [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

```
D_3d = D.reshape(2, 3, 2)
print('D_3d: ', D_3d)
```

```
D_3d:  [[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]
```

[12]:
```
D = np.arange(0, 12, 1,)
print('D: ', D)

D_2d = D.reshape(4, 3)
print('D_2d: ', D_2d)

D_3d = D.reshape(2, 3, 2)
print('D_3d: ', D_3d)
```

```
D:  [ 0  1  2  3  4  5  6  7  8  9 10 11]
D_2d:  [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
D_3d:  [[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]
```

Of course, you can convert an n-dimensional array to an 1-dimensional array using the attribute `flatten`.

```
E_1d = D_3d.flatten()
print('E_1d: ', E_1d)
```

```
E_1d:  [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

There is a Numpy function doing this too, but it is called `ravel`.

```
F_1d = np.ravel(D_3d)
print('F_1d: ', F_1d)
```

```
F_1d:  [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Note:

- `flatten` always returns a copy of the input array
- `ravel` uses a shallow copy, every change you make have an impact to the input arary as well.

See the difference in the next example:

```
y = np.array(((1,2,3),(4,5,6),(7,8,9)))

a = y.ravel()       #-- returns a shallow copy of y
print(a)            #-- array([1, 2, 3, 4, 5, 6, 7, 8, 9])

b = y.flatten()     #-- returns a deep copy of y
print(b)            #-- [[1 2 3] [4 5 6] [7 8 9]]

b[0] = 99
print(y)            #-- [[1 2 3] [4 5 6] [7 8 9]]

a[0] = 99           #-- change first element
print(y)            #-- [[99  2  3] [ 4  5  6] [ 7  8  9]]
```

[13]:
```
y = np.array(((1,2,3),(4,5,6),(7,8,9)))

a = y.ravel()
print(a)
```

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

[14]:
```
b = y.flatten()
print(b)
```

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

[15]:
```
b[0] = 99
print(y)
```

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

[16]:
```
a[0] = 99
print(y)
```

```
[[99  2  3]
 [ 4  5  6]
 [ 7  8  9]]
```

1.10 Math operations

Adding or multiplying arrays of the same size is simple. We define two arrays, each has 5 elements, and compute the sum and the product of them.

```
m = np.array([2.1, 3.0, 4.7, 5.3, 6.2])
n = np.arange(5)
print('m = ', m)
print('n = ', n)
```

Output:

```
m =  [2.1 3.  4.7 5.3 6.2]
n =  [0 1 2 3 4]
```

Other math operation examples:

```
mn_add = m + n
mn_mul = m * n

print('m + n = ', mn_add)
print('m * n = ', mn_mul)
```

Output:

```
m + n =  [ 2.1  4.   6.7  8.3 10.2]
m * n =  [ 0.   3.   9.4 15.9 24.8]
```

[17]:
```
m = np.array([2.1, 3.0, 4.7, 5.3, 6.2])
n = np.arange(5)

print('m = ', m)
print('n = ', n)
```

```
m =  [2.1 3.  4.7 5.3 6.2]
n =  [0 1 2 3 4]
```

[18]:
```
mn_add = m + n
mn_mul = m * n

print('m + n = ', mn_add)
print('m * n = ', mn_mul)
```

```
m + n =  [ 2.1  4.   6.7  8.3 10.2]
m * n =  [ 0.   3.   9.4 15.9 24.8]
```

1.11 Broadcasting

Working with arrays of the same size is the same as working with matrices. Numpy provides a powerful **broadcasting** concept that allows you to easily work with arrays of different shapes. Broadcasting is a method that transforms the smaller array into a *fitting* form or applies this multiple times to the larger array. This avoids loops and makes arithmetic operations on arrays much faster.

Examples:

One of the simplest examples is the addition of a scalar to an array. For instance add the scalar 1 to the array [1,2,3,4] of shape (4,1).

Equation:

$$\begin{vmatrix}1 & 2 & 3 & 4\end{vmatrix} + 1 = \begin{vmatrix}1+1 & 2+1 & 3+1 & 4+1\end{vmatrix}$$

To add an array of shape (3,1) with an array of shape (1,3)

Equation:

$$\begin{vmatrix}1 & 2 & 3\end{vmatrix} + \begin{vmatrix}1\\2\\3\end{vmatrix} = \begin{vmatrix}1+1 & 2+1 & 3+1\\1+2 & 2+2 & 3+2\\1+3 & 2+3 & 3+3\end{vmatrix}$$

Create a 1D-array with 4 elements from a list:

```
a = np.array([1,2,3,4])
print(a.shape)
print(a)
print('dimension: ', len(a.shape))
```

Output:

```
(4,)
[1 2 3 4]
dimension:  1
```

[19]:
```
a = np.array([1,2,3,4])
print(a.shape)
print(a)
print('dimension: ', len(a.shape))
```

```
(4,)
[1 2 3 4]
dimension:  1
```

Create a 2D-array:

```
b = np.array([[1],[2],[3],[4]])
print(b.shape)
print(b)
print('dimension: ', len(b.shape))
```

```
(4, 1)
[[1]
 [2]
 [3]
 [4]]
dimension:  2
```

[20]:
```
b = np.array([[1],[2],[3],[4]])
print(b.shape)
print(b)
print('dimension: ', len(b.shape))
```

```
(4, 1)
[[1]
 [2]
 [3]
 [4]]
dimension:  2
```

Create the transposed array:

```
c = b.copy().T
print(c.shape)
print(c)
```

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

[21]:
```
c = b.copy().T
print(c.shape)
print(c)
print('dimension: ', len(c.shape))
```

```
(1, 4)
[[1 2 3 4]]
dimension:  2
```

Here is another example:

```
array_A = np.array([[1,2,3,4], [10,20,30,40], [100,200,300,400]])

print(array_A.shape)
(3, 4)

array_B = np.array([1,2,3,4])

print(array_B.shape)
(4,)

print(array_A + array_B)
```

```
[[  2   4   6   8]
 [ 11  22  33  44]
 [101 202 303 404]]
```

The calculations are now done by Numpy as if array_B is equal to

```
array_B = np.array([[1,2,3,4], [1,2,3,4], [1,2,3,4]])

print(array_A + array_B)
```

```
[[  2   4   6   8]
 [ 11  22  33  44]
 [101 202 303 404]]
```

Try it out

What is the result of array_A * array_B using broadcasting?

[22]:
```
array_A = np.array([[1,2,3,4], [10,20,30,40], [100,200,300,400]])
array_B = np.array([1,2,3,4])

array_A * array_B
```

[22]:
```
array([[   1,    4,    9,   16],
       [  10,   40,   90,  160],
       [ 100,  400,  900, 1600]])
```

For higher dimensioned arrays this also applies accordingly. The following example is identical with Numpy's broadcasting

```
array_C = np.array([[[ 1, 2, 3], [ 10, 20, 30], [ 100, 200, 300]],
                    [[11,22,33], [101,202,303], [1001,2002,3003]]])

array_D = np.array([[3,2,1]])

print(array_C + array_D)
```

```
[[[   4    4    4]
  [  13   22   31]
  [ 103  202  301]]

 [[  14   24   34]
  [ 104  204  304]
  [1004 2004 3004]]]
```

Rebuild the array_D to the same shape as array_C

```
array_D = array_D[np.newaxis, np.newaxis, :]
array_D = np.tile(array_D,(2,3,1))

print(array_C + array_D)
```

```
[[[[   4    4    4]
   [  13   22   31]
   [ 103  202  301]]
```

```
[[  14   24   34]
 [ 104  204  304]
 [1004 2004 3004]]]]
```

Try it out

What is the result of array_C * array_D using broadcasting?

[23]:
```
array_C = np.array([[[ 1, 2, 3], [ 10, 20, 30], [ 100, 200, 300]],
                    [[11,22,33], [101,202,303], [1001,2002,3003]]])

array_D = np.array([[3,2,1]])

array_C * array_D
```

[23]:
```
array([[[   3,    4,    3],
        [  30,   40,   30],
        [ 300,  400,  300]],

       [[  33,   44,   33],
        [ 303,  404,  303],
        [3003, 4004, 3003]]])
```

1.12 Mathematical attributes and functions

The attributes of a numpy array can be used to retrieve the minimum or maximum value, or to compute the sum, mean, or standard deviation of arrays.

```
m_min   = m.min()
m_max   = m.max()
m_sum   = m.sum()
m_mean  = m.mean()
m_std   = m.std()
m_round = m.round()

print('m_min:   ',m_min)
print('m_max:   ',m_max)
print('m_sum:   ',m_sum)
print('m_mean:  ',m_mean)
print('m_std:   ',m.std())
print('m_round: ',m.round())
```

m_min: 2.1 m_max: 6.2 m_sum: 21.3 m_mean: 4.26 m_std: 1.502797391533536
m_round: [2. 3. 5. 5. 6.]

[24]:
```
m_min   = m.min()
m_max   = m.max()
m_sum   = m.sum()
m_mean  = m.mean()
```

```
m_std   = m.std()
m_round = m.round()

print('m_min:   ',m_min)
print('m_max:   ',m_max)
print('m_sum:   ',m_sum)
print('m_mean:  ',m_mean)
print('m_std:   ',m.std())
print('m_round: ',m.round())
```

```
m_min:    2.1
m_max:    6.2
m_sum:    21.3
m_mean:   4.26
m_std:    1.502797391533536
m_round:  [2. 3. 5. 5. 6.]
```

Numpy also provides a bunch of mathematical routines, see https://docs.scipy.org/doc/numpy/reference/routines.math.html.

Here are some examples:

```
m_sqrt = np.sqrt(m)
m_exp  = np.exp(m)
mn_add = np.add(m,n)

print('m_sqrt: ', m_sqrt)
print('m_exp:  ', m_exp)
print('mn_add: ', mn_add)
```

m_sqrt: [1.44913767 1.73205081 2.16794834 2.30217289 2.48997992]

m_exp: [8.16616991 20.08553692 109.94717245 200.33680997 492.74904109]

mn_add: [2.1 4. 6.7 8.3 10.2]

[25]:
```
m_sqrt = np.sqrt(m)
m_exp  = np.exp(m)
mn_add = np.add(m,n)

print('m_sqrt: ', m_sqrt)
print('m_exp:  ', m_exp)
print('mn_add: ', mn_add)
```

```
m_sqrt:  [1.44913767 1.73205081 2.16794834 2.30217289 2.48997992]
m_exp:   [  8.16616991  20.08553692 109.94717245 200.33680997 492.74904109]
mn_add:  [ 2.1  4.   6.7  8.3 10.2]
```

More useful functions: `np.sin`, `np.cos`, `np.degrees`

```
data_radians = np.linspace(0., 6., 5)
data_sin     = np.sin(data_radians)
```

```
data_cos     = np.cos(data_radians)
data_degrees = np.degrees(data_sin)

print('data_radians: ', data_radians)
print('data_sin:     ', data_sin)
print('data_cos:     ', data_cos)
print('data_degrees: ', data_degrees)
```

data_radians: [0. 1.5 3. 4.5 6.] data_sin: [0. 0.99749499 0.14112001 -0.97753012 -0.2794155] data_cos: [1. 0.0707372 -0.9899925 -0.2107958 0.96017029] data_degrees: [0. 57.15225282 8.08558087 -56.00835009 -16.00932878]

[26]:
```
data_radians = np.linspace(0., 6., 5)
data_sin     = np.sin(data_radians)
data_cos     = np.cos(data_radians)
data_degrees = np.degrees(data_sin)

print('data_radians: ', data_radians)
print('data_sin:     ', data_sin)
print('data_cos:     ', data_cos)
print('data_degrees: ', data_degrees)
```

```
data_radians:  [0.  1.5 3.  4.5 6. ]
data_sin:      [ 0.          0.99749499  0.14112001 -0.97753012 -0.2794155 ]
data_cos:      [ 1.          0.0707372  -0.9899925  -0.2107958   0.96017029]
data_degrees:  [  0.          57.15225282   8.08558087 -56.00835009
-16.00932878]
```

1.13 Creating arrays for masking

Being able to create arrays containing only zeros or ones can be very helpful if you want to mask your data. Numpy provides the functions `np.zero` and `np.ones`.

```
zeros = np.zeros((3,4))
ones  = np.ones((3,4))

print('zeros: ',zeros)
print('ones: ',ones)
```

zeros: [[0. 0. 0. 0.] [0. 0. 0. 0.] [0. 0. 0. 0.]] ones: [[1. 1. 1. 1.] [1. 1. 1. 1.] [1. 1. 1. 1.]]

```
zeros = np.zeros((3,4),dtype=int)
ones  = np.ones((3,4),dtype=int)

print('zeros type integer: ',zeros)
print('ones type integer: ',ones)
```

zeros type integer: [[0 0 0 0] [0 0 0 0] [0 0 0 0]] ones type integer: [[1 1 1 1] [1 1 1 1] [1 1 1 1]]

Try it out

Create an arrays with elements equal 5.

[27]:
```
np.ones((3,4)) * 5
```

[27]:
```
array([[5., 5., 5., 5.],
       [5., 5., 5., 5.],
       [5., 5., 5., 5.]])
```

1.14 Array stacking

Numpy gives you the possibilities to stack arrays on different axes using the functions `np.vstack` and `hstac`.

```
v = np.vstack((zeros,ones))
h = np.hstack((zeros,ones))

print('np.vstack(zeros,ones): \n', v)
print('np.hstack(zeros,ones): \n', h)
```

np.vstack(zeros,ones): [[0 0 0 0] [0 0 0 0] [0 0 0 0] [1 1 1 1] [1 1 1 1] [1 1 1 1]]

np.hstack(zeros,ones): [[0 0 0 0 1 1 1 1] [0 0 0 0 1 1 1 1] [0 0 0 0 1 1 1 1]]

[28]:
```
zeros= np.zeros((3,4),dtype=int)
ones = np.ones((3,4),dtype=int)

v = np.vstack((zeros,ones))
h = np.hstack((zeros,ones))

print('np.vstack(zeros,ones): \n', v)
print('np.hstack(zeros,ones): \n', h)
```

```
np.vstack(zeros,ones):
 [[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
np.hstack(zeros,ones):
 [[0 0 0 0 1 1 1 1]
 [0 0 0 0 1 1 1 1]
 [0 0 0 0 1 1 1 1]]
```

1.15 Shallow and deep copy

Maybe you have recognized that copying an array a doesn't mean to get a physical copy. If you change your copy the origin array will be changed too because it isn't a physical copy, it is more or less a pointer to the origin array. It's called a **shallow copy**.

```
a_origin = np.arange(12).reshape(3,4)

b_copy_of_a = a_origin

b_copy_of_a[1,3] = 999

print('a_origin: \n', a_origin)
print('b_copy_of_a: ', b_copy_of_a)
print('a_origin:    ', a_origin)
```

```
a_origin: [[ 0 1 2 3] [ 4 5 6 7] [ 8 9 10 11]] b_copy_of_a: [[ 0 1 2 3] [ 4 5 6 999] [ 8 9
10 11]] a_origin: [[ 0 1 2 3] [ 4 5 6 999] [ 8 9 10 11]]
```

To create a physical copy, so called **deep copy**, you have to use numpy's `np.copy` function.

```
a_origin = np.arange(12).reshape(3,4)

c_deep_copy = a_origin.copy()

c_deep_copy[1,3] = 222

print('a_origin: \n', a_origin)
print('c_deep_copy: \n', c_deep_copy)
```

```
a_origin: [[ 0 1 2 3] [ 4 5 6 7] [ 8 9 10 11]] c_deep_copy: [[ 0 1 2 3] [ 4 5 6 222] [ 8 9 10
11]]
```

Try it out

Follow the steps above.

[29]:
```
# shallow copy

a_origin = np.arange(12).reshape(3,4)
print('a_origin: \n', a_origin)

b_copy_of_a = a_origin
b_copy_of_a[1,3] = 999

print('b_copy_of_a: ', b_copy_of_a)
print('a_origin:    ', a_origin)

# deep copy
a_origin = np.arange(12).reshape(3,4)
print('a_origin: \n', a_origin)

c_deep_copy = a_origin.copy()
c_deep_copy[1,3] = 222
```

```
print('c_deep_copy: \n', c_deep_copy)
print('a_origin: \n', a_origin)
```

```
a_origin:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
b_copy_of_a:  [[  0   1   2   3]
 [  4   5   6 999]
 [  8   9  10  11]]
a_origin:     [[  0   1   2   3]
 [  4   5   6 999]
 [  8   9  10  11]]
a_origin:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
c_deep_copy:
 [[  0   1   2   3]
 [  4   5   6 222]
 [  8   9  10  11]]
a_origin:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

1.16 Most useful functions

Working with arrays 1- or multiple-dimensional arrays makes the programs sometimes slow when you have to find values in given ranges or to change values to set missing values for invalid data. Usually, you would thinl about a for or while loop to go through all elements of an array, but numpy has efficient functions to do it in just one line.

Useful functions

`np.where` `np.argwhere` `np.all` `np.any`

The `np.where` function allows you to look at the array using a logical expression. If it is True than let the value untouched but when it is False change it to the given value, maybe a kind of a missing value, but this is NOT the same as a netCDF missing value (see below `masked arrays`).

```
x = np.array([-1, 2, 0, 5, -3, -2])

x_ge0 = np.where(x >= 0, x, -9999)

print('x: ', x)
print('x_ge0: ', x_ge0)
```

x: [-1 2 0 5 -3 -2] x_ge0: [-9999 2 0 5 -9999 -9999]

[30]:
```
x = np.array([-1, 2, 0, 5, -3, -2])

x_ge0 = np.where(x >= 0, x, -9999)

print('x: ', x)
print('x_ge0: ', x_ge0)
```

```
x:  [-1  2  0  5 -3 -2]
x_ge0:  [-9999     2     0     5 -9999 -9999]
```

In the upper example the values of the array were located and directly changed when the condition is False. But sometimes you want to retrieve the indices of the values instead the values themselves, because you need the same indices later again. Then use the `np.argwhere` function with a logical condition.

```
x_ind = np.argwhere(x < 0)

print('--> indices x >= 0: \n', x_ind)
```

–> indices x >= 0: [[0] [4] [5]]

```
y = x

y[x_ind] = -9999

print('y[x_ind] where x >= 0: \n', y)
```

y[x_ind] where x >= 0: [-9999 2 0 5 -9999 -9999]

[31]:
```
x_ind = np.argwhere(x < 0)

print('--> indices x >= 0: \n', x_ind)
```

```
--> indices x >= 0:
 [[0]
 [4]
 [5]]
```

[32]:
```
y = x

y[x_ind] = -9999

print('y[x_ind] where x >= 0: \n', y)
```

```
y[x_ind] where x >= 0:
 [-9999     2     0     5 -9999 -9999]
```

To see if the values of an array are less than 0 for instance you can't try do it like below - ah, no that would be too easy.

```
if(x < 0):
   print('some elements are less 0')
```

```
else:
   print('no values are less 0')

if(x > 0):
   print('all elements are greater than 0')
else:
   print('not all values are greater than 0')
```

The result would be the following error:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-6f0311fb54a3> in <module>
----> 1 if(x < 0):
      2    print('some elements are less 0')
      3 else:
      4    print('no values are less 0')
      5

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() o
```

The last line of the error message gives you the right hint to use the array functions `any` or `all`.

```
if((x < 0).any()):
   print('some elements are less 0')
else:
   print('no values are less 0')
```

some elements are less 0

```
if(x.all() > 0):
   print('all elements are greater than 0')
else:
   print('not all values are greater than 0')
```

not all values are greater than 0

Or you can use the numpy function `np.any` or `np.all`.

```
if(np.any(x < 0)):
   print('some elements are less 0')
else:
   print('no values are less 0')
```

some elements are less 0

```
if(np.all(x > 0)):
   print('all elements are greater than 0')
else:
   print('not all values are greater than 0')
```

not all values are greater than 0

[52]:
```
if((x < 0).any()):
   print('some elements are less 0')
else:
   print('no values are less 0')

if(x.all() > 0):
   print('all elements are greater than 0')
else:
   print('not all values are greater than 0')
```

```
some elements are less 0
not all values are greater than 0
```

1.17 Statistical functions

Numpy provides a large number of statistical functions. These are only briefly mentioned here, but it is recommended to take a closer look at them, because they make the programmer's life much easier.

https://numpy.org/doc/stable/reference/routines.statistics.html

- `np.average` - compute the weighted average
- `np.mean` - compute the arithmetic mean
- `np.std` - compute the standard deviation
- `np.var` - compute the variance
- `np.histogram` - compute the histogram
- `np.bincount` - compute the number of occurences of each value

For the special case that the data contains NaNs use the following functions instead:

- `np.nanmean` - compute the arithmetic mean ignoring NaNs
- `np.nanstd` - compute the standard deviation ignoring NaNs
- `np.nanvar` - compute the variance ignoring NaNs

Note: Define NaNs (missing values) using the function `np.nan`.

Try it out

1. create array A = np.array([1,1,2,1,2,2,1,1]).astype(float)
2. create array B = A.copy()
3. for array B set values equal 2. to NaN
4. What is the difference between np.sum(B) and np.nansum(B) or np.mean(B) and np.nanmean(B)?

[33]:
```
# 1.
A = np.array([1,1,2,1,2,2,1,1]).astype(float)
```

[34]:
```
# 2.
B = A.copy()
```

[35]:
```
# 3.
B[B == 2] = np.nan
```

[36]:
```
# 4.
print(np.sum(B))
print(np.nansum(B))
print(np.mean(B))
print(np.nanmean(B))
```

```
nan
5.0
nan
1.0
```

1.18 Read and write data from file

Numpy can be used to read data from a file or to write variables or values into a file. There are several methods to work with files, but only text files will be discussed here.

- Text file - read: `np.loadtxt`
- Text file - write: `np.savetxt`

Here are only briefly the methods for working with binary files:

- Binary file - read: `np.load`
- Binary file - write: `np.save`

In the next section the procedure for handling `np.loadtxt` will be shown using a CSV file. A CSV file (comma separated variables) is a text file in which data values are separated by a separator.

1.19 Read data from CSV file

Numpy also provides functions to read data from files. Our next example shows how to read data from a CSV file. The input CSV file ***pr.dat*** looks like

```
 ID      LAT      LON      PW
BLAC    36.75   -97.25    48.00
BREC    36.41   -97.69    46.30
BURB    36.63   -96.81    49.80
...
```

We want to read all values, and because the mixed data types in the file, we read all as strings. We don't need the header line so we skip it.

```
lines = np.loadtxt('../data/pr.dat', dtype='str', skiprows=1)
ID  = lines[:,0]
lat = lines[:,1]
lon = lines[:,2]
pw  = lines[:,3]

print('ID:  \n', ID)
print('lat: \n', lat)
print('lon: \n', lon)
print('pw:  \n', pw)
```

```
ID:
 ['BLAC' 'BREC' 'BURB' 'DQUA' 'FBYN' 'GUTH' 'HBRK' 'HKLO' 'JTNT' 'LMNO'
 'LTHM' 'MEDF' 'NDS1' 'OILT' 'PRCO' 'REDR' 'RWDN' 'SA14' 'SG01' 'SG09'
 'SG10' 'SG11' 'SG12' 'SG13' 'SG14' 'SG15' 'SG16' 'SG18' 'SG19' 'SG20'
 'SG22' 'VCIO']
lat:
 ['36.75' '36.41' '36.63' '34.11' '40.08' '35.85' '38.31' '35.68' '33.02'
 '36.69' '39.58' '36.79' '37.30' '36.03' '34.98' '36.36' '40.09' '36.56'
 '36.60' '36.43' '36.88' '37.33' '38.20' '38.12' '37.84' '38.20' '37.38'
 '34.88' '35.36' '35.56' '35.26' '36.07']
lon:
 ['-97.25' '-97.69' '-96.81' '-94.29' '-97.31' '-97.48' '-97.29' '-95.86'
 '-100.98' '-97.48' '-94.17' '-97.75' '-95.60' '-96.50' '-97.52' '-97.15'
 '-100.65' '-100.61' '-97.49' '-98.28' '-98.29' '-99.31' '-99.32' '-97.51'
 '-97.02' '-95.59' '-96.18' '-98.20' '-98.98' '-98.02' '-97.48' '-99.22']
pw:
 ['48.00' '46.30' '49.80' '45.00' '38.20' '46.60' '34.30' '50.20' '39.80'
 '47.10' '40.40' '46.70' '39.90' '48.80' '46.50' '50.40' '26.40' '22.40'
 '49.00' '43.20' '41.40' '33.60' '33.90' '36.30' '40.20' '31.70' '42.80'
 '45.90' '45.30' '47.90' '46.80' '38.60']
```

[37]:
```
lines = np.loadtxt('../data/pr.dat', dtype='str', skiprows=1)
ID  = lines[:,0]
lat = lines[:,1]
lon = lines[:,2]
pw  = lines[:,3]

print('ID:  \n', ID)
print('lat: \n', lat)
print('lon: \n', lon)
print('pw:  \n', pw)
```

```
ID:
 ['BLAC' 'BREC' 'BURB' 'DQUA' 'FBYN' 'GUTH' 'HBRK' 'HKLO' 'JTNT' 'LMNO'
 'LTHM' 'MEDF' 'NDS1' 'OILT' 'PRCO' 'REDR' 'RWDN' 'SA14' 'SG01' 'SG09'
 'SG10' 'SG11' 'SG12' 'SG13' 'SG14' 'SG15' 'SG16' 'SG18' 'SG19' 'SG20'
 'SG22' 'VCIO']
lat:
 ['36.75' '36.41' '36.63' '34.11' '40.08' '35.85' '38.31' '35.68' '33.02'
 '36.69' '39.58' '36.79' '37.30' '36.03' '34.98' '36.36' '40.09' '36.56'
 '36.60' '36.43' '36.88' '37.33' '38.20' '38.12' '37.84' '38.20' '37.38'
 '34.88' '35.36' '35.56' '35.26' '36.07']
lon:
 ['-97.25' '-97.69' '-96.81' '-94.29' '-97.31' '-97.48' '-97.29' '-95.86'
 '-100.98' '-97.48' '-94.17' '-97.75' '-95.60' '-96.50' '-97.52' '-97.15'
 '-100.65' '-100.61' '-97.49' '-98.28' '-98.29' '-99.31' '-99.32' '-97.51'
 '-97.02' '-95.59' '-96.18' '-98.20' '-98.98' '-98.02' '-97.48' '-99.22']
pw:
```

```
 ['48.00' '46.30' '49.80' '45.00' '38.20' '46.60' '34.30' '50.20' '39.80'
 '47.10' '40.40' '46.70' '39.90' '48.80' '46.50' '50.40' '26.40' '22.40'
 '49.00' '43.20' '41.40' '33.60' '33.90' '36.30' '40.20' '31.70' '42.80'
 '45.90' '45.30' '47.90' '46.80' '38.60']
```

If you don't need the IDs then you can directly read the lat, lon, and pw values using the `usecols` parameter. There is no need to use the `dtype` parameter anymore because the default data type is float, and that's what we want.

```
data = np.loadtxt('../data/pr.dat', usecols=(1,2,3), skiprows=1)

print()
print('--> data: \n', data)
print()
print('--> lon:  \n', data[:,0])
print('--> lon:  \n', data[:,1])
print('--> pw:   \n', data[:,2])
```

```
--> data:
 [[  36.75  -97.25   48.  ]
 [  36.41  -97.69   46.3 ]
 [  36.63  -96.81   49.8 ]
 [  34.11  -94.29   45.  ]
 [  40.08  -97.31   38.2 ]
 [  35.85  -97.48   46.6 ]
 [  38.31  -97.29   34.3 ]
 [  35.68  -95.86   50.2 ]
 [  33.02 -100.98   39.8 ]
 [  36.69  -97.48   47.1 ]
 [  39.58  -94.17   40.4 ]
 [  36.79  -97.75   46.7 ]
 [  37.3   -95.6    39.9 ]
 [  36.03  -96.5    48.8 ]
 [  34.98  -97.52   46.5 ]
 [  36.36  -97.15   50.4 ]
 [  40.09 -100.65   26.4 ]
 [  36.56 -100.61   22.4 ]
 [  36.6   -97.49   49.  ]
 [  36.43  -98.28   43.2 ]
 [  36.88  -98.29   41.4 ]
 [  37.33  -99.31   33.6 ]
 [  38.2   -99.32   33.9 ]
 [  38.12  -97.51   36.3 ]
 [  37.84  -97.02   40.2 ]
 [  38.2   -95.59   31.7 ]
 [  37.38  -96.18   42.8 ]
 [  34.88  -98.2    45.9 ]
 [  35.36  -98.98   45.3 ]
 [  35.56  -98.02   47.9 ]
```

```
 [  35.26  -97.48   46.8 ]
 [  36.07  -99.22   38.6 ]]

--> lon:
 [36.75 36.41 36.63 34.11 40.08 35.85 38.31 35.68 33.02 36.69 39.58 36.79
 37.3  36.03 34.98 36.36 40.09 36.56 36.6  36.43 36.88 37.33 38.2  38.12
 37.84 38.2  37.38 34.88 35.36 35.56 35.26 36.07]
--> lon:
 [ -97.25  -97.69  -96.81  -94.29  -97.31  -97.48  -97.29  -95.86 -100.98
  -97.48  -94.17  -97.75  -95.6   -96.5   -97.52  -97.15 -100.65 -100.61
  -97.49  -98.28  -98.29  -99.31  -99.32  -97.51  -97.02  -95.59  -96.18
  -98.2   -98.98  -98.02  -97.48  -99.22]
--> pw:
 [48.  46.3 49.8 45.  38.2 46.6 34.3 50.2 39.8 47.1 40.4 46.7 39.9 48.8
 46.5 50.4 26.4 22.4 49.  43.2 41.4 33.6 33.9 36.3 40.2 31.7 42.8 45.9
 45.3 47.9 46.8 38.6]
```

[38]:
```
data = np.loadtxt('../data/pr.dat', usecols=(1,2,3), skiprows=1)

print()
print('--> data: \n', data)
print()
print('--> lon:  \n', data[:,0])
print('--> lon:  \n', data[:,1])
print('--> pw:   \n', data[:,2])
```

```
--> data:
 [[  36.75  -97.25   48.  ]
 [  36.41  -97.69   46.3 ]
 [  36.63  -96.81   49.8 ]
 [  34.11  -94.29   45.  ]
 [  40.08  -97.31   38.2 ]
 [  35.85  -97.48   46.6 ]
 [  38.31  -97.29   34.3 ]
 [  35.68  -95.86   50.2 ]
 [  33.02 -100.98   39.8 ]
 [  36.69  -97.48   47.1 ]
 [  39.58  -94.17   40.4 ]
 [  36.79  -97.75   46.7 ]
 [  37.3   -95.6    39.9 ]
 [  36.03  -96.5    48.8 ]
 [  34.98  -97.52   46.5 ]
 [  36.36  -97.15   50.4 ]
 [  40.09 -100.65   26.4 ]
 [  36.56 -100.61   22.4 ]
 [  36.6   -97.49   49.  ]
 [  36.43  -98.28   43.2 ]
```

```
 [  36.88  -98.29   41.4 ]
 [  37.33  -99.31   33.6 ]
 [  38.2   -99.32   33.9 ]
 [  38.12  -97.51   36.3 ]
 [  37.84  -97.02   40.2 ]
 [  38.2   -95.59   31.7 ]
 [  37.38  -96.18   42.8 ]
 [  34.88  -98.2    45.9 ]
 [  35.36  -98.98   45.3 ]
 [  35.56  -98.02   47.9 ]
 [  35.26  -97.48   46.8 ]
 [  36.07  -99.22   38.6 ]]

--> lon:
 [36.75 36.41 36.63 34.11 40.08 35.85 38.31 35.68 33.02 36.69 39.58 36.79
 37.3  36.03 34.98 36.36 40.09 36.56 36.6  36.43 36.88 37.33 38.2  38.12
 37.84 38.2  37.38 34.88 35.36 35.56 35.26 36.07]
--> lon:
 [ -97.25  -97.69  -96.81  -94.29  -97.31  -97.48  -97.29  -95.86 -100.98
  -97.48  -94.17  -97.75  -95.6   -96.5   -97.52  -97.15 -100.65 -100.61
  -97.49  -98.28  -98.29  -99.31  -99.32  -97.51  -97.02  -95.59  -96.18
  -98.2   -98.98  -98.02  -97.48  -99.22]
--> pw:
 [48.  46.3 49.8 45.  38.2 46.6 34.3 50.2 39.8 47.1 40.4 46.7 39.9 48.8
 46.5 50.4 26.4 22.4 49.  43.2 41.4 33.6 33.9 36.3 40.2 31.7 42.8 45.9
 45.3 47.9 46.8 38.6]
```

Try it out

Read the file and print the 5. element of pw.

[39]:
```
data = np.loadtxt('../data/pr.dat', usecols=(1,2,3), skiprows=1)
print(data[4,2])
```

```
38.2
```

However, we want to have the IDs, too. It's never too late… But again, we have to tell numpy to use the data type string.

```
IDs = np.loadtxt('../data/pr.dat', dtype='str', usecols=(0), skiprows=1)

print('IDs: \n', IDs)
```

IDs: > ['BLAC' 'BREC' 'BURB' 'DQUA' 'FBYN' 'GUTH' 'HBRK' 'HKLO' 'JTNT' 'LMNO' > 'LTHM' 'MEDF' 'NDS1' 'OILT' 'PRCO' 'REDR' 'RWDN' 'SA14' 'SG01' 'SG09' > 'SG10' 'SG11' 'SG12' 'SG13' 'SG14' 'SG15' 'SG16' 'SG18' 'SG19' 'SG20' > 'SG22' 'VCIO']

[40]:
```
IDs = np.loadtxt('../data/pr.dat', dtype='str', usecols=(0), skiprows=1)

print('IDs: \n', IDs)
```

```
IDs:
 ['BLAC' 'BREC' 'BURB' 'DQUA' 'FBYN' 'GUTH' 'HBRK' 'HKLO' 'JTNT' 'LMNO'
 'LTHM' 'MEDF' 'NDS1' 'OILT' 'PRCO' 'REDR' 'RWDN' 'SA14' 'SG01' 'SG09'
 'SG10' 'SG11' 'SG12' 'SG13' 'SG14' 'SG15' 'SG16' 'SG18' 'SG19' 'SG20'
 'SG22' 'VCIO']
```

1.20 Masking

In our daily work we are confronted with data which we want to mask to see only the values in sections we need. Masking is sometimes tricky and you have to take care.

What does data masking mean? For example, you want to display only a certain range of values in an array or a certain range of values is to be hidden. Logically, masks consists of either [True, False] or [0, 1] values.

In the following example, we try to demonstrate how to mask a 2-dimensional array by a given mask array containing zeros and ones, where 0 means 'don't mask', and 1 means 'mask'.

```
field = np.arange(1,9,1).reshape((4,2))

mask = np.array([[0,0],[1,0],[1,1],[1,0]])

mask_field = np.ma.MaskedArray(field,mask)

print('field:      \n', field)
print('mask:       \n', mask)
print('mask_field: \n', mask_field)
```

field: [[1 2] [3 4] [5 6] [7 8]] mask: [[0 0] [1 0] [1 1] [1 0]] mask_field: [[1 2] [– 4] [– –] [– 8]]

[41]:
```
field = np.arange(1,9,1).reshape((4,2))

mask = np.array([[0,0],[1,0],[1,1],[1,0]])

mask_field = np.ma.MaskedArray(field,mask)

print('field:      \n', field)
print('mask:       \n', mask)
print('mask_field: \n', mask_field)
```

```
field:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]
mask:
 [[0 0]
 [1 0]
 [1 1]
```

```
 [1 0]]
mask_field:
 [[1 2]
 [-- 4]
 [-- --]
 [-- 8]]
```

For the next example we want to get data from one array depending on data of a second array.

To create two arrays with random data of type integer we use numpy's random generator. For reproducibility we set the random seed to the fixed value 0.

```
np.random.seed(0)

A = np.random.randint(-3, high=5, size=10)
B = np.random.randint(-4, high=4, size=10)

print('A: ', A)
print('B: ', B)
```

Output:

```
A: [ 1  4  2 -3  0  0  0  4 -2  0]
B: [ 1 -2  0  3  2 -4 -4  0 -2 -3]
```

[42]:
```
np.random.seed(0)    #-- set the random seed
```

[43]:
```
A = np.random.randint(-3, high=5, size=10)
print('A: ', A)

B = np.random.randint(-4, high=4, size=10)
print('B: ', B)
```

```
A:  [ 1  4  2 -3  0  0  0  4 -2  0]
B:  [ 1 -2  0  3  2 -4 -4  0 -2 -3]
```

Now, we want only the values of array A which are

1. greater equal array B values
2. less than array B values

First, we have to find the indices of those values. Numpy provides routines to do that for us, presupposed that both arrays are of the same shape.

```
ind_ge = list(np.greater_equal(A,B))
ind_lt = list(np.less(A,B))

print('ind_ge: ', ind_ge)
print('ind_lt: ', ind_lt)
```

Output:

```
ind_ge:  [True, True, True, False, False, True, True, True, True, True]
ind_lt:  [False, False, False, True, True, False, False, False, False, False]
```

[44]:
```
ind_ge = list(np.greater_equal(A,B))
ind_lt = list(np.less(A,B))

print('ind_ge: ', ind_ge)
print('ind_lt: ', ind_lt)
```

```
ind_ge:  [True, True, True, False, False, True, True, True, True, True]
ind_lt:  [False, False, False, True, True, False, False, False, False, False]
```

This is the same as

```
ind_ge = list(A>=B)
ind_lt = list(A<B)
```

Output:

```
ind_ge:  [True, True, True, False, False, True, True, True, True, True]
ind_lt:  [False, False, False, True, True, False, False, False, False, False]
```

[45]:
```
ind_ge = list(A>=B)
ind_lt = list(A<B)

print('ind_ge: ', ind_ge)
print('ind_lt: ', ind_lt)
```

```
ind_ge:  [True, True, True, False, False, True, True, True, True, True]
ind_lt:  [False, False, False, True, True, False, False, False, False, False]
```

Use these indices to get the data we want.

```
A_ge = A[ind_ge]
A_lt = A[ind_lt]

print('A_ge: ', A_ge)
print('A_lt: ', A_lt)
```

Output:

```
A_ge:  [ 1  4  2  0  0  4 -2  0]
A_lt:  [-3  0]
```

[46]:
```
A_ge = A[ind_ge]
A_lt = A[ind_lt]

print('A_ge: ', A_ge)
print('A_lt: ', A_lt)
```

```
A_ge:  [ 1  4  2  0  0  4 -2  0]
A_lt:  [-3  0]
```

In this case we get only the values of the array A which conforms the condition, and not the complete masked array. This has to be done with the numpy.ma.MaskedArray.

```
A_ge2 = np.ma.MaskedArray(A, ind_ge)
A_lt2 = np.ma.MaskedArray(A, ind_lt)

print('A_ge2: ', A_ge2)
print('A_lt2: ', A_lt2)
```

Output:

```
A_ge2:  [-- -- -- -3 0 -- -- -- -- --]
A_lt2:  [1 4 2 -- -- 0 0 4 -2 0]
```

[47]:
```
A_ge2 = np.ma.MaskedArray(A, ind_ge)
A_lt2 = np.ma.MaskedArray(A, ind_lt)

print('A_ge2: ', A_ge2)
print('A_lt2: ', A_lt2)
```

```
A_ge2:  [-- -- -- -3 0 -- -- -- -- --]
A_lt2:  [1 4 2 -- -- 0 0 4 -2 0]
```

Let's have a look at the different array types:

```
print(type(A_ge))
```

```
<class 'numpy.ndarray'>```
```

```
print(type(A_ge2))
```

```
<class 'numpy.ma.core.MaskedArray'>
```

[48]:
```
print(type(A_ge))
print(type(A_ge2))
```

```
<class 'numpy.ndarray'>
<class 'numpy.ma.core.MaskedArray'>
```

Here is just a brief example in order to locate all the values of an array that are not equal to zero. Of course, it also shows how to locate the values equal to zero.

Count and locate the non zero values of an array, select them, and mask the array to save the shape of the array.

```
C = np.random.randint(-2, high=1, size=10)

C_count_nonzero = np.count_nonzero(C)
C_nonzero_ind   = np.nonzero(C)

C2 = C[C_nonzero_ind]

C3 = np.ma.MaskedArray(C, C==0)
```

```
print('--> C:                ', C)
print('--> C_count_nonzero: ', C_count_nonzero)
print('--> C_nonzero_ind:   ', C_nonzero_ind)
print('--> C2:               ', C2)
print('--> C3:               ', C3)
```

Output:

```
C:                [-2 -1  0 -2  0 -2 -1 -1  0 -2]
C_count_nonzero:  7
C_nonzero_ind:    (array([0, 1, 3, 5, 6, 7, 9]),)
C2:               [-2 -1 -2 -2 -1 -1 -2]
C3:               [-2 -1 -- -2 -- -2 -1 -1 -- -2]
```

[49]:
```
C = np.random.randint(-2, high=1, size=10)

C_count_nonzero = np.count_nonzero(C)
C_nonzero_ind   = np.nonzero(C)

C2 = C[C_nonzero_ind]

C3 = np.ma.MaskedArray(C, C==0)

print('C:                ', C)
print('C_count_nonzero: ', C_count_nonzero)
print('C_nonzero_ind:   ', C_nonzero_ind)
print('C2:               ', C2)
print('C3:               ', C3)
```

```
C:                [ 0  0 -2 -1 -1 -1 -1 -2 -1 -2]
C_count_nonzero:  8
C_nonzero_ind:    (array([2, 3, 4, 5, 6, 7, 8, 9]),)
C2:               [-2 -1 -1 -1 -1 -2 -1 -2]
C3:               [-- -- -2 -1 -1 -1 -1 -2 -1 -2]
```

Now, we look at the zeros.

Count and locate the zero values of an array, select them, and mask the array to save the shape of the array.

```
Z_count_zero = np.count_nonzero(C==0)

Z_zero_ind = np.argwhere(C==0)

Z = C[Z_zero_ind]

Z2 = np.ma.MaskedArray(C, C!=0)

print('Z_count_zero: ', Z_count_zero)
print('Z_zero_ind:   ', Z_zero_ind.flatten())
```

```
print('Z:            ', Z.flatten())
print('Z2:           ', Z2)
```

Output:

```
Z_count_zero:  4
Z_zero_ind:    [4 6 7 9]
Z:             [0 0 0 0]
Z2:            [-- -- -- -- 0 -- 0 0 -- 0]
```

[50]:
```
Z_count_zero = np.count_nonzero(C==0)

Z_zero_ind = np.argwhere(C==0)

Z = C[Z_zero_ind]

Z2 = np.ma.MaskedArray(C, C!=0)

print('Z_count_zero: ', Z_count_zero)
print('Z_zero_ind:   ', Z_zero_ind.flatten())
print('Z:            ', Z.flatten())
print('Z2:           ', Z2)
```

```
Z_count_zero:  2
Z_zero_ind:    [0 1]
Z:             [0 0]
Z2:            [0 0 -- -- -- -- -- -- -- --]
```

1.21 Some hints

You've learned to generate random arrays, arrays of zeros or ones but one important array generation ist still missing - the array filled with with missing values.

To generate an array which contains only missing values use the numpy routine `np.full`.

```
missing = 1.0e20

empty_array = np.full(20, missing)

print(empty_array)
```

Output:

```
[1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20
 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20]
```

[51]:
```
missing = 1.0e20

empty_array = np.full(20, missing)

print(empty_array)
```

```
[1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20
 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20 1.e+20]
```

1.22 Difference between Numpy and Pandas

Both libraries contains a set of mathematical functions for arrays and series but with a different aim. Pandas held all data in a Pandas Series object called Dataframe in memory which slows down the program while using large datasets.

In general, Numpy is much faster than Pandas due to the fact that Pandas doing all work in Python but Numpy in C. In most cases Pandas are efficient enough for your work.