Numpy continuation

Reshaping

```
import numpy as np


xs = np.arange(3, 17)
xs
[ 3  4  5  6  7  8  9 10 11 12 13 14 15 16]

xs = xs.reshape(2, 7)
xs
[[ 3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16]]
```

reshape

Very useful method for changing form of arrays, returns array with the previous content in a new shape

`array.reshape(new_shape, order='C')`

`new_shape` can be in one of the following formats

- tuple with lengths of dimensions
- each dimension as an argument

`order` corresponds to the index reading order - Fortran or C liike

np.reshape

Analogous function, but with 2 minor input differences

`np.reshape(array, new_shape)`

- requires `array` to be passed as a first argument
- `new_shape` can be only a tuple

For both variants new shape can contain one -1 value to automatically compute corresponding dimension size

```
# Change shape to 3 rows X 5 columns
xs.reshape(3, 5)

# Same
xs.reshape((3, 5))

# Same
np.reshape(xs, (3, 5))

# Same too, but usually easier
xs.reshape(3, -1)

# And this one
xs.reshape(-1, 5)
```

Some methods and common arguments

numpy has quite a big number of array methods for computing something. Many of them have these arguments

- `axis` - on which elements result should be computed. By default this is usually `None`, which means - compute result on all elements. Other variants are 0 - compute for each row, 1 - compute for each column, and so on

```
xs = np.arange(12).reshape(3, 4)
xs
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

xs.max()
11

xs.max(axis=0)
[ 8  9 10 11]

xs.max(axis=1)
[ 3  7 11]
```

out

- `out` - array where result should be placed, it should match dimensions of the result

```
to_be_filled = np.empty(xs.shape[1])
```

```
xs.max(axis=0, out=to_be_filled)
```

```
to_be_filled
[ 8.  9. 10. 11.]
```

keepdims

- `keepdims` - whether to keep redundant dimensions in the result to make it compliant with original array

```
xs.ndim
2

xs.max(axis=0)
[ 8  9 10 11]

xs.max(axis=0, keepdims=True)
[[ 8  9 10 11]]
```

More methods

We came to them)

- `min`, `max` - everything is obvious - minimum and maximum values
- `argmin`, `argmax` - indices of minima and maxima in the array, if it is used on the whole array (`axis=None`) index corresponds to the flatten array
- `std`, `var` - standard deviation and variance, have `ddof` argument, corresponding to the number subtracted from the number of observations in the denominator in the formula

```
xs.argmax()
11

xs.argmax(axis=0)
array([2, 2, 2, 2])

xs.std()
3.452052529534663

xs.var()
11.916666666666666
xs.var(ddof=1)
13.0
```

Logical indexing

There are many ways to index an array, couple of them we have already learnt (regular list-like and fancy indexing)

Another one is a logical indexing where we pass in `[]` list with boolean values for each element. Element is picked if it has `True` in a boolean list

```
xs = np.arange(10)

# Boolean array
xs > 5
array([False, False, False, False, False, False,
True,  True,  True, True])

# Boolean indexing
xs[xs > 5]

array([6, 7, 8, 9])
```

Indexing with specifying corresponding coordinates

Sometimes it is useful when you need elements from the coordinates without clear pattern

In this case we can pass lists with corresponding indices for several first necessary axes of the array

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

xs
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])


xs[[0, 1, 1], [1, 2, 3]]

array([1, 6, 7])
```

So we have picked elements from the xs with coordinates [0, 1], [1, 2], [1, 3]

where

Function to replace elements depending on the condition

```
# Substitute elems less than 5 with 0 and others
with 1
np.where(xs < 5, 0, 1)
```

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

Other usage

It is possible to invoke `where` only with condition, in this case it will return a tuple with arrays of indexes in axis for compliant elements. It is similar to `nonzero` method

```
np.where(xs < 5)
```

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

Concatenation

There are several methods to create greater arrays by stacking others. All of these methods takes a tuple with smaller arrays for concatenation

- concatenate - general method which should be the best, but ... requires same ndim of all arrays. Take axis to understand how stack them
- hstack - stack arrays horizontally
- vstack - stack arrays vertically
- dstack - stack arrays in depth

```
a = np.arange(3)
b = np.arange(3, 6)

np.concatenate((a, b))
```

array([0, 1, 2, 3, 4, 5])

```
np.concatenate((a, b), axis=1)
```

```
AxisError: axis 1 is out of bounds for array of
dimension 1
```

```
a = a.reshape((-1, a.size))
b = b.reshape((-1, b.size))
# Also change here axis to 0
np.concatenate((a, b), axis=0)
```

array([[0, 1, 2],
       [3, 4, 5]])

```
a = np.arange(3)
b = np.arange(3, 6)

np.hstack((a, b))

array([0, 1, 2, 3, 4, 5])

np.vstack((a, b))
array([[0, 1, 2],
       [3, 4, 5]])

np.dstack((a, b))
array([[[0, 3],
        [1, 4],
        [2, 5]]])
```