

# 08 – The basics of numpy

## 1 The basics of numpy

The basic element - the basic data type - in numpy is an array, which is a bit like a Python list. (In English, some people call it a list, because its official English name is list, but others call it a table, because that would actually be a better name for some reasons. Here, however, it comes in handy that the name tabela is still unused and can be used for numpy's arrays.) There are several important differences between numpy's tables and Python's lists. For our first encounter with numpy, we'll focus on two: the differences in the way they are indexed, and the differences between how different operations work on them. Let's prepare two Python lists

```
p = [3, 8, 9, 2]
r = [8, 0, 1, -3]
```

Indexing can be used to get to individual elements. The indexes must of course be integers (int).

```
p[2]
```

9

Lists can also be "aggregated". In fact, we use the word addition only because we use the + operator. It is not really addition (in the mathematical sense), but switching

```
p + r
```

```
[3, 8, 9, 2, 8, 0, 1, -3]
```

Since + is not really addition, subtracting lists makes no sense.

```
p - r
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[4], line 1
----> 1 p - r
```

```
TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

Prav tako nima smisla k seznamu prišteti 1, saj ne moremo stakniti seznama in števila.

```
p + 1
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[5], line 1
----> 1 p + 1
```

```
TypeError: can only concatenate list (not "int") to list
```

It also doesn't make sense to add 1 to the list, as we can't concatenate the list and the number.

```
p + 1
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[5], line 1  
----> 1 p + 1  
  
TypeError: can only concatenate list (not "int") to list
```

Alternatively, we can multiply the list by an integer; the result is the same as if we had added the list that many times.

```
p * 3
```

```
[3, 8, 9, 2, 3, 8, 9, 2, 3, 8, 9, 2]
```

## 1.1 Importing the numpy module

To use a module, you must first import it. We don't usually import the numpy module with `import numpy`, because we will use its functions so often that the whole program would be full of `numpy`.- the name `numpy` is simply too long. On the other hand, we don't import individual functions from `numpy` `import hstack, sum, max` because we typically need too many different functions, and some of them have the same names as the built-in Python functions (for example, `sum` and `max`; functions imported from `numpy` would displace them and cause problems because they expect different data, work slightly differently, and return different results than Python's functions). `numpy` is therefore usually imported with

```
import numpy as np
```

This imports the module, but then we don't see it under the name `numpy`, but under the shorter, friendlier name `np`. Now let's make two tables. It will be easiest to call the `array` function and give it a list of elements.

```
a = np.array([3, 8, 9, 2])  
b = np.array([8, 0, 1, -3])
```

So:

```
a
```

```
array([3, 8, 9, 2])
```

### 1.1.1 Indexing tables

Tables are indexed in the same way as lists. With indices that must be integers.

```
a[2]
```

Slices work too, and everything we've learned about them

```
a[2:4]
```

```
array([9, 2])
```

```
a[1:]
```

```
array([8, 9, 2])
```

```
a[:-1]
```

```
array([3, 8, 9])
```

The first difference compared to lists: if you need more elements, you can provide more indexes. We can give more items, not just by listing them (that will achieve something else), but by giving a list or a table of indexes as the index. So if we have

```
a
```

```
array([3, 8, 9, 2])
```

Is

```
a[[2, 0, 1, 0, 0, 1]]
```

```
array([9, 3, 8, 3, 3, 8])
```

a table containing the second, null, first, then twice null and again the first element of table a. Instead of a list of int's, you can give a list (or table) of bool's. This list should be the same length as the table we are indexing, as it tells us which elements we want and which we don't want. For example, let's say we get the first and last element of a table

```
a[[True, False, False, True]]
```

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

Both - especially the last one - seem ... not very useful. In fact, it will be phenomenally useful. Let's wait, for example

## 1.2 Operations on tables

So we have:

```
a
```

```
array([3, 8, 9, 2])
```

```
b
```

```
array([ 8,  0,  1, -3])
```

Let's bring it down

```
a + b
```

```
array([11,  8, 10, -1])
```

Yes, it's really addition. Then we can also, in fact, subtract.

```
a - b
```

```
array([-5,  8,  8,  5])
```

And we multiply.

```
a * b
```

```
array([24,  0,  9, -6])
```

All operations on tables, work on an element by element basis. +, -, \* ... each operation is performed on each element separately. So we can also multiply tables by numbers, add **numbers to them...**

```
a
```

```
array([3, 8, 9, 2])
```

```
a + 1
```

```
array([ 4,  9, 10,  3])
```

```
a * 2.5
```

```
array([ 7.5, 20. , 22.5,  5. ])
```

```
np.array([2]) ** range(10)
```

```
array([ 1,  2,  4,  8, 16, 32, 64, 128, 256, 512])
```

What's more. Even operations like < operate on individual elements.

```
b
```

```
array([ 8,  0,  1, -3])
```

```
b > 0
```

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

And the functions too!

```
np.abs(b)
```

```
array([8, 0, 1, 3])
```

```
np.sqrt(a)
```

```
array([1.73205081, 2.82842712, 3.          , 1.41421356])
```

Where it makes sense, clearly. Functions such as sum and max will of course add and multiply.

```
np.sum(a)
```

```
22
```

```
np.max(a)
```

```
9
```

Here we used numpy's abs, sqrt, sum and max functions. sum can add anything it can get its hands on (more precisely: anything it can run a for loop over), and max and min are not picky.

```
sum(a)
```

```
22
```

However: when working with tables, it is always worth using numpy's equivalent functions. If nothing else, they will be faster, but sometimes the built-in Python functions will not work correctly or at all (sqrt(a) reports an error). In addition, numpy functions often have additional arguments specific to working with tables.

### 1.3 Vector operations

When working with numpy, we try to avoid writing loops in the first place. How would we get the sum of all positive elements of b? First, we construct a "mask", a table of bool's containing True where b has positive elements.

```
b > 0
```

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

With this mask, we can select positive elements.

```
b[b > 0]
```

```
array([8, 1])
```

Since we are interested in the sum, we add them together

```
np.sum(b[b > 0])
```

```
9
```

If we were only interested in how many positive elements *b* has, we would simply add the mask, since even in numpy *True* is as many as 1 and *False* is as many as 0.

```
np.sum(b > 0)
```

```
2
```

If we were even less sophisticated and just wanted to know if *b* had a positive element, we would use *any* (again, taking the numpy equivalent rather than Python's embedded *any*, which we learned about last hour):

```
np.any(b > 0)
```

```
True
```

That *b* does not have only positive elements tells us everything:

```
np.all(b > 0)
```

```
False
```

Similarly, it is easy to find (count, check) the even elements of *a*

```
a[a % 2 == 0]
```

```
array([8, 2])
```

This last example is quite interesting to read: *a % 2 == 0*. What this formula says about *a* essentially applies to every element of *a*. When we say *a % 2 == 0*, we get what we would

get in Python, with lists, with `[x % 2 == 0 for x in a]`. The discussion of derived lists was necessary-if for no other reason-to help us understand the idea of "vector operations"-operations that happen, exactly the same, on every element of a table. There is still a loop somewhere inside numpy, of course, but because of the way numpy is made, such a loop is much (where "much" can easily mean fifty or even a hundred times) faster than writing a loop in Python.

### 1.4 Example: height differences

A cyclist recorded his altitude for every hundred metres of his (what looks like quite eventful :) ride

```
h = np.array([345, 355, 360, 364, 378, 370, 360, 355, 360, 361])
```

Now, like any cyclist who does this kind of thing, he is interested in the overall uplift. To start with, it is necessary to get a list of the changes in height between successive pairs of measurements. With the lists we would write something like:

```
d = [x - y for x, y in zip(h[1:], h)]
```

```
d
```

```
[10, 5, 4, 14, -8, -10, -5, 5, 1]
```

But we want to avoid the loop. We want to use the operator- on the table, not on its individual elements. The above zip- or its argument- has already shown what needs to be subtracted.

```
print(h[1:])
print(h)
```

```
[355 360 364 378 370 360 355 360 361]
```

```
[345 355 360 364 378 370 360 355 360 361]
```

We simply need to subtract the second table from the first

```
h[1:] - h
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[44], line 1
----> 1 h[1:] - h

ValueError: operands could not be broadcast together with shapes (9,) (10,)
```

Yes, a small detail: because it works by elements, the two tables must be the same length. The second one is one element too long. The last element is therefore cut off.

```
d = h[1:] - h[:-1]
```

```
d
```

```
array([ 10,  5,  4, 14, -8, -10, -5,  5,  1])
```

Adding this up, of course, gives 0, because the rider finished where he started.

```
np.sum(d)
```

```
16
```

The total number of rides is equal to the total number of rides. The cyclists are interested in the two rides,

```
d[d > 0]
```

```
array([10,  5,  4, 14,  5,  1])
```

This is the only thing that can be counted. We are going to have to do more than two. And let's get it together, so that we can get it when programmes were written

```
h = np.array([345, 355, 360, 364, 378, 370, 360, 355, 360, 361, 345])
d = h[1:] - h[:-1]
print(np.max(d))
print(np.sum(d[d > 0]))
```

```
14
```

```
39
```

## 1.5 Primer: Numpynadražbi

Now I will remember home tasks: Auction - we solved a job with numpy. We will see how it will show the mice!

For the beginning, we load data. Regarding functions like `genfromtxt`, I will talk about it next time. Here, let's call it.

```
cene = np.genfromtxt("../domace-naloge/02-drazba/drazba.txt", dtype=int)
```

```
cene
```

```
array([11, 17, 24, 30, -1, 13, 27, 33, -1, 12, 27, 34, 40, -1,  9, -1,  8,
       20, 30, 31, -1])
```

Now, we can answer two questions - how many products were sold and which one was the most expensive.

They sold as many products as is -1. `cene == -1` will give us a table of False and True; True, which are -1.

```
cene == -1
```

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

Since True is equal to 1 and False is equal to 0, the elements of the list are simply summed.

```
np.sum(cene == -1)
```

```
5
```

The price of the most expensive product is the largest number in the table.

```
np.max(cene)
```

```
40
```

The answer to the last question is correct, but from an aesthetic point of view it is a bit of a mistake to have calculated the maximum over everything, including intermediate offers and even -1. We can easily solve the problem more elegantly by asking the following question: what is the sum of the prices of the products sold. The `np.flatnonzero` function returns a table with the indices of the non-zero elements. If it is a table of True's and False's, it returns the indexes of the True's. In our example, these will be the indexes of the elements with value -1

```
indeksi = np.flatnonzero(cene == -1)
```

```
indeksi
```

```
array([ 4,  8, 13, 15, 20])
```

We quickly see that there are flat-1s on these indices.

```
cene[indeksi]
```

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

This is uninteresting: it is the elements in front of them that interest us. So subtract 1 from the indices.

```
koncne = cene[indeksi - 1]
```

```
koncne
```

```
array([30, 33, 40,  9, 31])
```

Now we can solve the first two problems again, plus a third:

```
print(f"Število prodanih predmetov: {len(koncne)}")
print(f"Cena najdražjega predmeta: {np.max(koncne)}")
print(f"Vsota končnih cen: {np.sum(koncne)}")
```

```
Število prodanih predmetov: 5
Cena najdražjega predmeta: 40
Vsota končnih cen: 143
```

The last two questions ask how many items Anna and Berta bought and how much either of them spent. There are only two of them in the auction and the first bid is always Anna's. To answer these questions, we need to count the number of odd bids and the number of even bids. To do this, we need to start by counting the number of bids for each item. The correct answer is [4, 3, 4, 1, 4].

The number of bids is (almost) equal to the difference between the indices.

```
indeksi
array([ 4,  8, 13, 15, 20])
```

For the first item, four bids were received. Next, we look at the differences: the index of the second-1 is 8, and the index of the first is 4; in between there were  $8 - 4 - 1 = 3$  bids. (We have to subtract another 1 because in the table we have intermediate elements-1 in addition to the prices.) The next two indices are 13 and 8;  $13 - 8 - 1 = 4$ . We will find out the number of bids for a single object if we take from

```
indeksi
array([ 4,  8, 13, 15, 20])
```

Substract

```
np.hstack((-1, indeksi[:-1]))
array([-1,  4,  8, 13, 15])
```

and 1 more. We added -1 to the beginning. This way, after subtraction, we will get just the right number for the unlucky, special first object, because we will have  $4 - (-1) - 1 = 4$ . Don't overlook the double parentheses: we give the np.hstack function as an argument a table with the two tables we want to staple together.

```
ponudb = indeksi - np.hstack((-1, indeksi[:-1])) - 1
ponudb
array([4, 3, 4, 1, 4])
```

Exactly what we need. Anna bought those items for which there were an odd number of bids; Berta bought those for which there were even bids.

```
ana = ponudb % 2 == 1
berta = ponudb % 2 == 0
```

```
ana
```

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

```
print(f"Ana je kupila {np.sum(ana)}, Berta pa {np.sum(berta)} reči.")
```

Ana je kupila 2, Berta pa 3 reči.

And how much did which one spend? These are the prices of things that ended up at Anna's:

```
koncne[ana]
```

```
array([33,  9])
```

So, obviously,

```
print(f"Ana je zapravila {np.sum(koncne[ana])}, Berta pa {np.
↳sum(koncne[berta])}.")
```

Ana je zapravila 42, Berta pa 101.

### 1.5.1 All together

To realise how elegantly short it all is, let's write the whole programme in one piece.

```
import numpy as np

cene = np.genfromtxt("../domace-naloge/02-drazba/drazba.txt", dtype=int)
indeksi = np.flatnonzero(cene == -1)
koncne = cene[indeksi - 1]
ponudb = indeksi - np.hstack([-1], indeksi[:-1])) - 1
ana = ponudb % 2 == 1
berta = ponudb % 2 == 0

print(f"Število prodanih predmetov: {len(koncne)}")
print(f"Cena najdražjega predmeta: {np.max(koncne)}")
print(f"Vsota končnih cen: {np.sum(koncne)}")
print(f"Ana je kupila {np.sum(ana)}, Berta pa {np.sum(berta)} reči.")
print(f"Ana je zapravila {np.sum(koncne[ana])}, Berta pa {np.
↳sum(koncne[berta])}.")
```

```
Število prodanih predmetov: 5
Cena najdražjega predmeta: 40
Vsota končnih cen: 143
Ana je kupila 2, Berta pa 3 reči.
Ana je zapravila 42, Berta pa 101.
```

## 8b. Multidimensional tables

Tables can also be multi-dimensional. "Multi-" should be "two-" to start with. (Since two is the exact number of dimensions that modern screens and projectors have, two-dimensional tables are just fine. Three-dimensional ones would require special glasses, and four would require a different universe.)

```
import numpy as np
```

```
[3]:
```

```
a = np.array([[2, -7, 2], [5, 9, 1], [1, -0, 0], [-1, -2, -8]])
```

```
[4]:
```

```
a
```

```
[4]:
```

```
array([[ 2, -7,  2],
       [ 5,  9,  1],
       [ 1,  0,  0],
       [-1, -2, -8]])
```

In Python, this would be a list of lists. You would get to the third line by

```
a[3]
```

```
[5]:
```

```
array([-1, -2, -8])
```

and to another element in it by

```
a[3][2]
```

```
[6]:
```

```
-8
```

Numpy tables can be indexed with multiple indices in the same parentheses.

The same magic applies to both indexes as to Python indexes otherwise. So we can get all rows from the second to the fourth and columns to the last:

```
a[2:4, :-1]
```

```
[8]:
```

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

Or, say, all the rows of the second column - in other words, the second column.

```
a[:, 2]
```

```
[9]:
```

```
array([ 2,  1,  0, -8])
```

Since a column is a one-dimensional thing, Python just turned it into a one-dimensional table.

Speaking of rollovers, let's mention a general rollover: a table has a "method" `T` that returns a transposed table. The quotes are necessary because `T` is not really a method but something else, which we won't learn about in the course. `T` does not need to be called. A little because the parentheses would take up more space than just the name of the function, a little because `T` comes from mathematics where the transposed matrix  $A$  would be written as  $A^T$ .

So if `a` is such that

```
a
```

```
[10]:
```

```
array([[ 2, -7,  2],
       [ 5,  9,  1],
       [ 1,  0,  0],
       [-1, -2, -8]])
```

`a.T` is :

```
a.T
```

```
[11]:
```

```
array([[ 2,  5,  1, -1],
       [-7,  9,  0, -2],
       [ 2,  1,  0, -8]])
```

Columns became rows, rows became columns. Surprisingly often we will rotate the table like this, because certain things will be easier to do with rows than they would be with columns.

Three-dimensional tables behave similarly to two-dimensional tables. They have one more index, but they are more awkward to print out. And four- or five-dimensional ones the same. `.T` reverses the order of the dimensions. (But I haven't used it for such tables, at least not yet. Well, I haven't used such tables either.)

A few more ways to build tables

So far, we've built tables by calling `np.array` and passing a prepared list (or list of lists, if we wanted two dimensions) in Python. (Last week, we also called `np.genfromtxt`, which read the file into a table, and we've got more tables by various ways of indexing the table so read.)

Sometimes we need to make an empty table - a table of zeros, ones or something else. As an argument, we give a tuple with dimensions. If the tuple contains two elements, the table will be two-dimensional. If five, it will be five-dimensional.

```
np.zeros((2, 4))
```

```
[12]:
```

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

```
[13]:
```

```
np.ones((2, 4))
```

```
[13]:
```

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

```
[14]:
```

```
np.full((2, 4), 42)
```

```
[14]:
```

```
array([[42, 42, 42, 42],  
       [42, 42, 42, 42]])
```

We often also need a table of sequential numbers.

```
np.arange(12)
```

```
Out[14]:
```

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

Or, say, 15 equally spaced numbers between 0 and 3.5.

```
np.linspace(0, 3.5, 15)
```

```
Out[15]:
```

```
array([0.   , 0.25, 0.5  , 0.75, 1.   , 1.25, 1.5  , 1.75, 2.   , 2.25, 2.5  ,  
       2.75, 3.   , 3.25, 3.5  ])
```

By the way, let's learn another way to create a table: let's make a table with five rows and three columns of random numbers between -1 and 1.

```
r = np.random.uniform(-1, 1, (5, 2))  
  
r
```

```
Out[16]:  
array([[ 0.28957304, -0.35546214],  
       [-0.05671772,  0.26460191],  
       [ 0.69538276, -0.47511787],  
       [-0.97905431,  0.60535218],  
       [-0.07506103, -0.55767802]])
```

In the tables above, they were int or float; we can see when something happened from the output. If we build a table from a list or with `np.full`, the type depends on the argument. If there is a float in the list, it will be a table of floats, if it is int-i, a table of int-i.

But of course

We can decide this ourselves

numpy doesn't know only one int and one float.

Time for a new intertitle. :)

## Table properties

Let's recall a.

```
a
```

```
Out[17]:  
array([[ 2, -7,  2],  
       [ 5,  9,  1],  
       [ 1,  0,  0],  
       [-1, -2, -8]])
```

```
In [18]:
```

```
a.ndim
```

```
Out[18]:  
2
```

```
In [19]:
```

```
a.shape
```

```
Out[19]:  
(4, 3)
```

```
a.size
```

```
Out[20]:
```

```
12
```

```
In [21]:
```

```
a.dtype
```

```
Out[21]:
```

```
dtype('int64')
```

```
In [22]:
```

```
a.itemsize
```

```
Out[22]:
```

```
8
```

ndim said that the table is two-dimensional, shape tells its shape (the length of the shape is just ndim) and size tells the number of elements (which is just the product of what we have in the shape). Looking at it this way, it would be enough to have a shape, because everything else can be calculated from it. Yes, but sometimes we need size and sometimes we need to check ndim, so it's quite practical that they exist.

The last two things are more interesting. dtype tells the data type of the elements. Numpy has its own data types, both numeric and ... other. The elements of table a are of type int64, which means int, stored with 64 bits. itemsize is the number of bytes occupied by each element. Since at least a byte consists of 8 bits (which seems logical today, but the history of computing records other models), 64 bits is 8 bytes.

Why int64? Why not int40? (Because it doesn't exist. :) Why not int32? Basically, there are int's in the table, and which version (width) of int will be the default depends on, um, the operating system, or the behaviour of numpy on each operating system. On MS Windows, the default int was until recently (or is it still?) 32-bit, even though the operating system has long been 64-bit. On macOS and Linux they are 64-bit. (How do I know? Because it has given us joy programming programs that should have worked on MS Windows, but mysteriously crashed.)

It's best not to worry too much about this, because for your purposes it will be enough to pretend that the tables are either int or float.

What type the table will be depends on what we write to it.

```
np.array([2, 8, -1]).dtype
```

```
Out[23]:  
dtype('int64')
```

```
In [24]:
```

```
np.array([2, 1, 3.14, 8]).dtype
```

```
Out[24]:  
dtype('float64')
```

Tables constructed with zeros and ones are usually float unless otherwise requested. You can also explicitly request a different type when building a table with `np.array` or any other function.

```
np.ones(4)
```

```
Out[25]:  
array([1., 1., 1., 1.])
```

```
In [26]:
```

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

```
Out[26]:  
array([1, 1, 1, 1])
```

```
In [27]:
```

```
np.ones(4, dtype=float)
```

```
Out[27]:  
array([1., 1., 1., 1.])
```

```
np.ones(4, dtype=bool)
```

```
Out[28]:  
array([ True,  True,  True,  True])
```

```
In [29]:
```

```
np.array([1, 2, 3], dtype=float)
```

```
Out[29]:  
array([1., 2., 3.])
```

## Changing table properties

You can change the format of a table or its type. More precisely: we can make a table into another table that has a different shape or type of elements.

We change the shape using `reshape`.

```
a
```

```
Out[30]:  
array([[ 2, -7,  2],  
       [ 5,  9,  1],  
       [ 1,  0,  0],  
       [-1, -2, -8]])
```

```
In [31]:
```

```
b = a.reshape(2, 6)
```

```
b
```

```
Out[31]:  
array([[ 2, -7,  2,  5,  9,  1],  
       [ 1,  0,  0, -1, -2, -8]])
```

And that's where the numpy magic begins. `b` is just a different view of the memory where `a` is stored. `b` didn't copy the data, so we didn't waste any time or extra memory. But that's not particularly important to us - at least as long as we don't modify the table. Changing `b` would also change `a`.

When you format a table, the new table must have the same number of elements. We can change a 3x4 table to 2x6 or even 1x12, but not to 3x5.

One of the dimensions can be set to -1 and numpy will calculate what it should be. If we want to put `a` in two columns and we can't calculate today how much  $3 \times 4 / 2$  is, we write something like

```
a.reshape(-1, 2)
```

```
Out[32]:  
array([[ 2, -7],  
       [ 2,  5],  
       [ 9,  1],  
       [ 1,  0],  
       [ 0, -1],  
       [-2, -8]])
```

but, of course, it hasn't changed. This is a new table (or a new view of an existing table).

The type is changed with `astype`. Here we get a new table.

```
a
```

```
Out[33]:  
array([[ 2, -7,  2],  
       [ 5,  9,  1],  
       [ 1,  0,  0],  
       [-1, -2, -8]])
```

```
In [34]:
```

```
a.astype(float)
```

```
Out[34]:  
array([[ 2., -7.,  2.],  
       [ 5.,  9.,  1.],  
       [ 1.,  0.,  0.],  
       [-1., -2., -8.]])
```

```
a.astype(bool)
```

```
Out[35]:  
array([[ True,  True,  True],  
       [ True,  True,  True],  
       [ True, False, False],  
       [ True,  True,  True]])
```

```
In [36]:
```

```
a.astype("U5")
```

```
Out[36]:  
array([[ '2', '-7', '2'],  
       [ '5', '9', '1'],  
       [ '1', '0', '0'],  
       [ '-1', '-2', '-8']], dtype='<U5')
```

Oops, what kind of guy is U5? A string (U for Unicode) with up to five characters.

Enough enumeration, it's time for an example.

## Numpy at auction without anonymity

First, let's read the data.

```
zapisnik = np.genfromtxt(  
    "../domace-naloge/03-drazba-brez-anonimnosti/zapisnik.txt",  
    delimiter=",")
```

The `np.genfromtxt` function can be given either a filename or an open file. The latter is particularly practical if you are on Windows and still need to specify the encoding.

```
zapisnik = np.genfromtxt(  
    open("../domace-naloge/03-drazba-brez-anonimnosti/zapisnik.txt",  
          encoding="utf-8"),  
    delimiter=",")
```

The beginning of the table is as follows.

```
zapisnik[:5]
```

```
Out[39]:  
array([[nan, nan, 31.],  
       [nan, nan, 33.],  
       [nan, nan, 35.],  
       [nan, nan, 37.],  
       [nan, nan, 40.]])
```

nan? nan means not a number. Yes, of course, the first two columns contain names of objects and people, not a number.

Let's specify dtype to be a string.

```
zapisnik = np.genfromtxt(  
    "../domace-naloge/03-drazba-brez-anonimnosti/zapisnik.txt",  
    delimiter="," ,  
    dtype=str)  
  
zapisnik[:5]
```

```
Out[40]:  
array([[ 'slika', 'Berta', '31'],  
       [ 'slika', 'Ana', '33'],  
       [ 'slika', 'Berta', '35'],  
       [ 'slika', 'Fanči', '37'],  
       [ 'slika', 'Ana', '40']], dtype='<U21')
```

(Before we continue: numpy says dtype='U12'. This means a string with a maximum of 20 characters.)

And we have a two-dimensional table. But it would actually be more practical to have three variables, objects, persons and prices, each with its own column.

We could go

```
predmeti = zapisnik[:, 0]  
osebe = zapisnik[:, 1]  
cene = zapisnik[:, 2]
```

but there is a handy trick: turn a table with three columns and God knows how many rows into a table with three rows and God knows how many columns.

```
zapisnik.T[:, :5]
```

```
Out[42]:  
array([[ 'slika', 'slika', 'slika', 'slika', 'slika'],  
       [ 'Berta', 'Ana', 'Berta', 'Fanči', 'Ana'],  
       [ '31', '33', '35', '37', '40']], dtype='<U21')
```

To keep it simple, as before we have five rows, but now we have only five columns. (See how! The first index is : to get all three rows, the second :5 to get only the first five columns.

In this inverted table, record.T[0] is the first row, record.T[1] the second, and record.T[2] the third. This is great because we can easily unpack these three rows into three variables!

```
predmeti, osebe, cene = zapisnik.T
```

```
In [44]:
```

```
predmeti[:15]
```

```
Out[44]:  
array([ 'slika', 'slika', 'slika', 'slika', 'slika', 'slika',  
       'pozlačen dežnik', 'Meldrumove vaze', 'Meldrumove vaze',  
       'Meldrumove vaze', 'Meldrumove vaze', 'Meldrumove vaze',  
       'Meldrumove vaze', 'Meldrumove vaze', 'Meldrumove vaze'],  
      dtype='<U21')
```

```
In [45]:
```

```
osebe[:15]
```

```
Out[45]:  
array([ 'Berta', 'Ana', 'Berta', 'Fanči', 'Ana', 'Fanči', 'Ema', 'Greta',  
       'Ana', 'Greta', 'Ana', 'Fanči', 'Ana', 'Greta', 'Ana'],  
      dtype='<U21')
```

```
cene[:15]
```

```
Out[46]:  
array([ '31', '33', '35', '37', '40', '45', '29', '44', '46', '48', '53',  
       '57', '60', '61', '63'], dtype='<U21')
```

Oh, the prices are strings, we'll have to turn them into numbers. We wave the magic wand of numpy, and there it is.

```
cene = cene.astype(int)
```

In [48]:

```
cene
```

Out[48]:

```
array([[ 31,  33,  35,  37,  40,  45,  29,  44,  46,  48,  53,  57,  60,
         61,  63,  67,  71,  76,  78,  50,  55,  60,  61,  62,  65,  68,
         70,  74,  76,  80,  83,  30,  32,  37,  39,  43,  44,  45,  50,
         53,  55,  58,  61,  63,  68,  72,  76,  77,  81,  85,  86,  90,
         92,  94,  97,  98,  99, 100, 103, 107,  15,  27,  30,  35,  39,
         40,  45,  47,  49,  53,  55,  58,  59,  62,  63,  16,  21])
```

Now, just do your homework.

## 1. Write down which item fetched the highest price, what price it fetched and who bought it.

It is trivial to find the highest price,

```
np.max(cene)
```

Out[49]:

```
107
```

We also want to know which item it is and who bought it. So we want to know what is in the table of objects and persons in the place where the number 107 is in the price. So we do not need (or at least: we do not need only) the maximum number in the price but also (and above all) its index.

Remember how we used to write the argmax function? Python doesn't have it, Numpy does.

```
np.argmax(cene)
```

Out[50]:

```
59
```

Then there is no problem:

```
naj_i = np.argmax(cene)
print(f"Najdražji predmet, {predmeti[naj_i]}, je za {cene[naj_i]} kupila .
```

Najdražji predmet, kip, je za 107 kupila Dani.

## 2. Print the final prices of all items

Last week was easy: we just had a list of prices and the prices of the different items were separated by -1. Now we have a list of items and we need to find out which rows the item is replaced on.

We know the trick from last time: last time we calculated the differences between consecutive rows (when we were interested in how much a rider goes up or down between two consecutive measurements). Now we are simply interested in whether two consecutive rows are different.

```
predmeti[1:] != predmeti[:-1]
```

Out[52]:

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

Let's find the indices where changes occur.

```
spremembe = np.flatnonzero(predmeti[1:] != predmeti[:-1])
spremembe
```

Out[53]:

```
array([ 5,  6, 18, 30, 59, 60, 74])
```

Since we have omitted the first line, these are essentially the indices of the last lines of each object. Only the last index is missing; it is just `len(objects) - 1`.

Adding 1 to the changes gives the first rows for each object. Now we run out of the first row of the first object; it is simply 0.

Prepare a table with the indexes of the starting (which we will need later) and ending (because we will need them now) rows describing each object. The missing first and last elements are appended with `np.hstack`. Again, let's not overlook: `hstack` expects as an argument a tuple with the tables to be stacked, hence the double brackets.

```
zacetni = np.hstack(([0], spremembe + 1))
koncni = np.hstack((spremembe, [len(predmeti) - 1]))
```

Now that we have the final indices, we know that the final prices are

```
cene[koncni]
```

```
Out[55]:  
array([ 45,  29,  78,  83, 107,  15,  63,  21])
```

And the objects belonging to

```
predmeti[koncni]
```

```
Out[56]:  
array(['slika', 'pozlačen dežnik', 'Meldrumove vaze', 'skodelice', 'kip',  
      'čajnik', 'srebrn jedilni servis', 'perzijska preproga'],  
      dtype='<U21')
```

So we can write out:

```
for predmet, cena in zip(predmeti[koncni], cene[koncni]):  
    print(f"{predmet:25} {cena:3}")
```

slika	45
pozlačen dežnik	29
Meldrumove vaze	78
skodelice	83
kip	107
čajnik	15
srebrn jedilni servis	63
perzijska preproga	21

Of course, a table of final items and prices could be prepared in advance.

```
konpred = predmeti[koncni]  
kconcene = cene[koncni]
```

To malo poenostavi zip:

```
for predmet, cena in zip(konpred, kconcene):  
    print(f"{predmet:25} {cena:3}")
```

slika	45
pozlačen dežnik	29
Meldrumove vaze	78
skodelice	83
kip	107
čajnik	15
srebrn jedilni servis	63
perzijska preproga	21

What if you wanted to price-regulate these items?

We can price-regulate:

```
np.sort(koncene)
```

```
Out[60]:  
array([ 15,  21,  29,  45,  63,  78,  83, 107])
```

But the problem is that now we would like to rearrange the names of the items in the same way as the prices are rearranged here. That is to say, we would have to, um, know the indices that map these numbers from the original table, ending in a table rearranged like this, in order to then map the names of the objects in the same way.

Don't we understand the above sentence? No? OK, let's go slowly.

Remember our famous argmax function, which returns the index of the largest element instead of its index. Equally - and even more - nominal is argsort, which instead of an ordered table returns the indices that order the table

```
red = np.argsort(koncene)
```

```
In [62]:
```

```
red
```

```
Out[62]:  
array([5, 7, 1, 0, 6, 2, 3, 4])
```

Compare this with

```
koncene
```

```
Out[63]:  
array([ 45,  29,  78,  83, 107,  15,  63,  21])
```

the order tells us this: if you want to end a list with ordered elements, you have to take the element with index 5 first,

```
koncene[5]
```

```
Out[64]:  
15
```

then the one with index 7,

```
koncene[7]
```

```
Out[65]:  
21
```

then the one with index 1,

```
koncene[1]
```

```
Out[66]:  
29
```

... or, to cut a long story short, by remembering that you can index a table with indexes from another table:

```
koncene[red]
```

```
Out[67]:  
array([ 15,  21,  29,  45,  63,  78,  83, 107])
```

Since the items in the konpred are listed in such a way that they have the same prices, we have to pick up the items in the same order.

```
konpred[red]
```

```
Out[68]:  
array(['čajnik', 'perzijska preproga', 'pozlačen dežnik', 'slika',  
      'srebrn jedilni servis', 'Meldrumove vaze', 'skodelice', 'kip'],  
      dtype='<U21')
```

If we want to arrange a descending one, we just reverse the order.

And there we have it:

```
red = np.argsort(koncene)[::-1]  
  
for predmet, cena in zip(konpred[red], koncene[red]):  
    print(f"{predmet:25} {cena:3}")
```

kip	107
skodelice	83
Meldrumove vaze	78
srebrn jedilni servis	63
slika	45
pozlačen dežnik	29
perzijska preproga	21
čajnik	15

### 3. List the number of bids received for each of the items

Let's add to the exercise that the items should be listed in descending order of the number of bids.

If we know that the first rows relating to an object are in the ending row and the starting rows are in the beginning row,

```
print(koncni)
print(zacetni)
```

```
[ 5  6 18 30 59 60 74 76]
[ 0  6  7 19 31 60 61 75]
```

all we have to do is subtract these two tables and add 1, and we have the number of bids for each item. We argue the number of bids and then proceed as before.

```
ponudb = koncni - zacetni + 1

red = np.argsort(ponudb)[::-1]

for predmet, pon in zip(konpred[red], ponudb[red]):
    print(f"{predmet:25} {pon:3}")
```

```
kip                29
srebrn jedilni servis 14
skodelice          12
Meldrumove vaze     12
slika               6
perzijska preproga  2
čajnik              1
pozlačen dežnik     1
```

#### 4. Please indicate the subject for which there were the highest number of bids. If more than one item shares the first place, list them all.

We've almost done this, but there's more to come. Only the part about listing multiple items sharing first place could be added. But now we'll do it differently: we'll find the object with the most bids and print out all the objects that have that many bids. This will only be one, but it will be obvious from the program that there could be more.

```
ponudbe = koncni - zacetni + 1

maska = ponudbe == np.max(ponudbe)
```

Now the mask contains True for all elements that have as many bids as np.max(bids).

```
ponudbe
```

```
Out[73]:
array([ 6,  1, 12, 12, 29,  1, 14,  2])
```

```
In [74]:
```

```
maska
```

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

There is only one True (where there are 29), but there could be more.

Now let's print out these objects: we apply the mask we made from the quote to the objects.

```
for predmet in konpred[maska]:  
    print(predmet)
```

kip

## A little more numpy: axes

Back to the two-dimensional matrix a.

```
a
```

```
Out[76]:  
array([[ 2, -7,  2],  
       [ 5,  9,  1],  
       [ 1,  0,  0],  
       [-1, -2, -8]])
```

What do functions like np.max and np.sum return?

```
np.max(a)
```

```
Out[77]:  
9
```

```
In [78]:
```

```
np.sum(a)
```

```
Out[78]:  
2
```

This is sometimes really useful, but rarely. More often we want the sum by rows or by columns.

A matrix has two axes. The first goes down, by rows (because: the first index is rows), the second by columns (because: the second index). The np.max, np.sum and other functions accept an additional axis argument where this makes sense (it doesn't for argsort, say).

```
np.max(a, axis=0)
```

```
Out[79]:  
array([5, 9, 2])
```

```
In [80]:
```

```
np.max(a, axis=1)
```

```
Out[80]:  
array([ 2,  9,  1, -1])
```

The first, axis=0, finds the largest element along index 0, so returns the table with the largest element in each column. The second one looks for the largest element of each row, i.e. along index 1.

Same sum:

```
np.sum(a, axis=0)
```

```
Out[81]:  
array([ 7,  0, -5])
```

```
In [82]:
```

```
np.sum(a, axis=1)
```

```
Out[82]:  
array([-3, 15,  1, -11])
```

How about this: we only want to keep lines that contain only non-negative elements.

```
a
```

```
Out[83]:  
array([[ 2, -7,  2],  
       [ 5,  9,  1],  
       [ 1,  0,  0],  
       [-1, -2, -8]])
```

```
In [84]:
```

```
a >= 0
```

```
Out[84]:  
array([[ True, False,  True],  
       [ True,  True,  True],  
       [ True,  True,  True],  
       [False, False, False]])
```

np.all(a >= 0) tells us whether all elements of a are non-negative.(They are not.)

```
np.all(a >= 0)
```

```
Out[85]:  
False
```

But we are interested in the direction of the columns, in direction 1.

```
np.all(a >= 0, axis=1)
```

```
Out[86]:  
array([False,  True,  True, False])
```

This is used as a mask for the lines of a.

```
a[np.all(a >= 0, axis=1)]
```

```
Out[87]:  
array([[5, 9, 1],  
       [1, 0, 0]])
```

Bi znali prešteti, koliko pozitivnih elementov vsebuje posamični stolpec?

```
np.sum(a > 0, axis=0)
```

```
Out[88]:  
array([3, 1, 2])
```

## Numpy, how much is $\pi$ ?

If we happen to forget the value of  $\pi$ , it's easy to calculate it.

We imagine a circle with its center at (0, 0) and a radius of 1. The area of this circle is  $\pi r^2 = \pi 1^2 = \pi$ .

Next, we imagine a square that is drawn around this circle. We place it parallel to the coordinate axes, in other words, the square extends from -1 to 1 along both the x and y axes. Its side length is 2, and its area is clearly 4.

Now we randomly generate  $NNN$  (let's say 1000) points inside the square. What fraction of these points falls within the circle? Since the ratio of the areas of the circle and the square is  $\pi/4$ , we expect  $k = N\pi/4$  points in the circle.

So, if we forget  $\pi$ , we can program an experiment like this. We randomly generate coordinates for  $NNN$  points, count how many of them are inside the circle (we'll denote this count as  $k$ ), and then calculate  $\pi$  from the formula  $\pi = 4k/N$ .

Let's go. We'll generate points using `np.random.uniform(-1, 1, (N, 2))`, which will return an array of size `(N,2)` with random numbers between -1 and 1.

To make things clearer, we'll start with just ten points.

```
N = 10

tocke = np.random.uniform(-1, 1, (N, 2))

tocke
```

```
Out[89]:
array([[ 0.11995016,  0.09098891],
       [-0.81012809, -0.79357372],
       [-0.85570538,  0.35303725],
       [ 0.85267372, -0.36440988],
       [ 0.27940679, -0.63779546],
       [-0.91855011, -0.89444881],
       [-0.85184532, -0.74527486],
       [-0.9410507 , -0.09879938],
       [ 0.54732715, -0.13532082],
       [-0.98596715, -0.71856324]])
```

The first column is the x-coordinates, the second the y-coordinates. The distance from a point to the centre of the coordinate system (and hence of the circle) is obtained by adding the squares of x and y and taking the square of this. Thank you, Pythagoras.

```
np.sum(tocke ** 2, axis=1)
```

```
Out[90]:
array([0.02266702, 1.28606678, 0.856867 , 0.85984703, 0.48485121,
       1.64377297, 1.28107507, 0.89533775, 0.31787873, 1.48846435])
```

We see that `np.sum(..., axis=1)` adds "horizontally", so we get a flat sum (squares) of the first and second columns.

Let's root this.

```
np.sqrt(np.sum(tocke ** 2, axis=1))
```

```
Out[91]:
array([0.15055571, 1.13404884, 0.92567111, 0.92727937, 0.69631258,
       1.2820971 , 1.13184587, 0.94622288, 0.56380735, 1.22002637])
```

A point is inside a circle of radius 1 if its distance from the origin is less than 1.

```
np.sqrt(np.sum(tocke ** 2, axis=1)) < 1
```

```
Out[92]:
array([ True, False,  True,  True,  True, False, False,  True,  True,
       False])
```

(Note at this point that the rooting is unnecessary. The root is less than 1 exactly when the number itself is less than 1.)

And now just count how many True's we have.

```
np.sum(np.sum(tocke ** 2, axis=1) < 1)
```

Out[93]:

6

This is our k. Let's repeat on 1000 points and calculate Pi

```
N = 1000  
  
k = np.sum(np.sum(np.random.uniform(-1, 1, (N, 2)) ** 2, axis=1) < 1)  
  
4 * k / N
```

Out[94]:

3.188

Well, let's say. We can easily repeat the multipoint thing.

```
N = 1000000  
  
k = np.sum(np.sum(np.random.uniform(-1, 1, (N, 2)) ** 2, axis=1) < 1)  
  
4 * k / N
```

Out[95]:

3.143676

## Unique elements

Let's learn about another interesting function: `np.unique` returns a table of all the different elements of a table. For example, let's say we get a table of all the items that participated in an auction.

```
np.unique(osebe)
```

Out[96]:

```
array(['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta', 'Helga'],  
      dtype='<U21')
```

The function is particularly interesting because of what it can return in addition to this table. If you add the argument `return_counts=True`, it returns the number of times each of these names has appeared - that is, the number of times the individual has raised the price (including the first bid).

```
np.unique(osebe, return_counts=True)
```

```
Out[97]:  
(array(['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta', 'Helga'],  
      dtype='<U21'),  
 array([ 6, 11, 13, 13, 11,  5, 15,  3]))
```

As it returns two things, it is most practical to unpack this into two variables. And we can already tell who was the most enthusiastic.

```
imena, visanja = np.unique(osebe, return_counts=True)  
  
imena[np.argmax(visanja)]
```

```
Out[98]:  
'Greta'
```

Another interesting additional argument is `return_inverse=True`. Calling `np.unique(a, return_inverse=True)` will return, in addition to a table of uniques, a table with as many elements as `a`. Each element tells where in the unique table the individual element of `a` is mapped.

```
imena, indeksi = np.unique(osebe, return_inverse=True)
```

```
osebe
```

```
Out[100]:  
array(['Berta', 'Ana', 'Berta', 'Fanči', 'Ana', 'Fanči', 'Ema', 'Greta',  
      'Ana', 'Greta', 'Ana', 'Fanči', 'Ana', 'Greta', 'Ana', 'Cilka',  
      'Greta', 'Fanči', 'Cilka', 'Dani', 'Berta', 'Dani', 'Berta',  
      'Dani', 'Berta', 'Dani', 'Berta', 'Dani', 'Berta', 'Dani', 'Berta',  
      'Cilka', 'Ema', 'Berta', 'Ema', 'Cilka', 'Berta', 'Cilka', 'Dani',  
      'Cilka', 'Greta', 'Dani', 'Cilka', 'Dani', 'Greta', 'Cilka',  
      'Greta', 'Ema', 'Dani', 'Greta', 'Cilka', 'Dani', 'Greta', 'Ema',  
      'Dani', 'Ema', 'Greta', 'Ema', 'Greta', 'Dani', 'Berta', 'Ema',  
      'Helga', 'Ema', 'Cilka', 'Helga', 'Greta', 'Ema', 'Cilka', 'Ema',  
      'Greta', 'Cilka', 'Greta', 'Cilka', 'Greta', 'Fanči', 'Helga'],  
      dtype='<U21')
```

```
In [101]:
```

```
indeksi
```

```
Out[101]:  
array([1, 0, 1, 5, 0, 5, 4, 6, 0, 6, 0, 5, 0, 6, 0, 2, 6, 5, 2, 3, 1, 3,  
      1, 3, 1, 3, 1, 3, 1, 3, 1, 2, 4, 1, 4, 2, 1, 2, 3, 2, 6, 3, 2, 3,  
      6, 2, 6, 4, 3, 6, 2, 3, 6, 4, 3, 4, 6, 4, 6, 3, 1, 4, 7, 4, 2, 7,  
      6, 4, 2, 4, 6, 2, 6, 2, 6, 5, 7])
```

```
imena
```

```
Out[102]:  
array(['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta', 'Helga'],  
      dtype='<U21')
```

We can (if we wanted to, but of course we don't have to) reconstruct persons with names and indexes. The first three elements of the index are 1, 0, 1, 5. The corresponding elements of the name are (first) Berta, (zero) Anna, first (Berta) and (fifth) Fanci - which are exactly the first elements of the person. So

```

imena[indeksi]

Out[103]:
array(['Berta', 'Ana', 'Berta', 'Fanči', 'Ana', 'Fanči', 'Ema', 'Greta',
      'Ana', 'Greta', 'Ana', 'Fanči', 'Ana', 'Greta', 'Ana', 'Cilka',
      'Greta', 'Fanči', 'Cilka', 'Dani', 'Berta', 'Dani', 'Berta',
      'Dani', 'Berta', 'Dani', 'Berta', 'Dani', 'Berta', 'Dani', 'Berta',
      'Cilka', 'Ema', 'Berta', 'Ema', 'Cilka', 'Berta', 'Cilka', 'Dani',
      'Cilka', 'Greta', 'Dani', 'Cilka', 'Dani', 'Greta', 'Cilka',
      'Greta', 'Ema', 'Dani', 'Greta', 'Cilka', 'Dani', 'Greta', 'Ema',
      'Dani', 'Ema', 'Greta', 'Ema', 'Greta', 'Dani', 'Berta', 'Ema',
      'Helga', 'Ema', 'Cilka', 'Helga', 'Greta', 'Ema', 'Cilka', 'Ema',
      'Greta', 'Cilka', 'Greta', 'Cilka', 'Greta', 'Fanči', 'Helga'],
      dtype='<U21')

```

Of course, we don't do that because we already have persons. Let's use the indices for something else.

## Numpy at the additional auction

### 5. For each person, write down how much they spent at the auction

This will require some crazy indexing. Crazy enough - and, above all, very wasteful - to solve this problem in practice with dictionaries and a Python loop. But it will be instructive from a numpy point of view.

Let's make a table of zeros with as many rows as there are items sold and as many columns as there are persons. So: a different column for each person.

```

imena, indeksi = np.unique(osebe, return_inverse=True)

nakupi = np.zeros((len(koncni), len(imena)), dtype=int)
nakupi

Out[104]:
array([[0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0]])

```

And now the crazy part. Unique has assigned an index to each person. Anna is 0, Berta is 1 and so on. The indexes for each row of data tells what the index of the person the row refers to is. `names[indices]` are, we know, just the names of the persons. The table of ends contains the indices of those rows which refer to the end prices. `indices[ends]` are then the indices of the names of the persons who actually bought the individual item.

```
indeksi[koncni]
```

```
Out[105]:  
array([5, 4, 2, 1, 3, 1, 6, 7])
```

The first item was bought by person 5, the second by person 4 and so on. Person 1 bought two items. Who is it? Berta.

```
imena[1]
```

```
Out[106]:  
'Berta'
```

Otherwise, they are the final buyers, in order by subject,

```
imena[indeksi[koncni]]
```

```
Out[107]:  
array(['Fanči', 'Ema', 'Cilka', 'Berta', 'Dani', 'Berta', 'Greta',  
      'Helga'], dtype='<U21')
```

In prices[final] we find, as we have known for a long time, the final prices of objects.

```
cene[koncni]
```

```
Out[108]:  
array([ 45,  29,  78,  83, 107,  15,  63,  21])
```

Now let's fill in the Purchases table. Each row relates to an item. These are unique and are numbered from 0 to len(end) (without len(end)). Each column refers to a person. For each number from 0 to len(final) and person in index[final] (indexes of persons who have bought an object), enter the price in the corresponding cell.

```
nakupi[np.arange(len(koncni)), indeksi[koncni]] = cene[koncni]
```

```
nakupi
```

```
Out[109]:  
array([[ 0,  0,  0,  0,  0, 45,  0,  0],  
       [ 0,  0,  0,  0, 29,  0,  0,  0],  
       [ 0,  0, 78,  0,  0,  0,  0,  0],  
       [ 0, 83,  0,  0,  0,  0,  0,  0],  
       [ 0,  0,  0, 107,  0,  0,  0,  0],  
       [ 0, 15,  0,  0,  0,  0,  0,  0],  
       [ 0,  0,  0,  0,  0,  0, 63,  0],  
       [ 0,  0,  0,  0,  0,  0,  0, 21]])
```

The first column is Anna's. She bought nothing. The second is Berta's: she bought something for 83 and something for 15. And so on. We find out how much someone spent by adding up the columns.

```
poraba = np.sum(nakupi, axis=1)

poraba
```

```
Out[110]:
array([ 45,  29,  78,  83, 107,  15,  63,  21])
```

The names belonging to the columns are given in the names. And we can print them out.

```
for ime, pora in zip(imena, poraba):
    print(f"{ime:8}{pora:4}")
```

```
Ana      45
Berta    29
Cilka    78
Dani     83
Ema     107
Fanči    15
Greta    63
Helga    21
```

Once again: using such a large table for so little work is nonsense. If each person bought several things, and if each product could be sold to several people, it would make sense. But we have learnt this double, parallel indexing because it will come in handy later.

## 6. For each item, work out by how much the final price was higher than the first price.

After the gymnastics of the previous section, this is trivial. The starting prices are in `price[starting]`, the ending prices in `price[ending]`. Subtract them, and you get the differences the problem asks for.

```
print(cene[koncni])
print(cene[zacetni])

cene[koncni] - cene[zacetni]
```

```
[ 45  29  78  83 107  15  63  21]
[31 29 44 50 30 15 27 16]
Out[112]:
array([14,  0, 34, 33, 77,  0, 36,  5])
```

The names of the objects are found in `objects[start]` or `objects[end]` (both will be the same). Since a good vaga helps to get to heaven, we will add the number of offers and repeat how we format the printout.

```

ponudb = koncni - zacetni + 1
zvisanja = cene[koncni] - cene[zacetni]

print(f"{'Predmet':25}{'ponudb':6}{'zvisanje':>12}")
print("-" * (25 + 6 + 12))
for predmet, pon, zvis in zip(predmeti[koncni], ponudb, zvisanja):
    print(f"{'predmet':25}{'pon':6}{'zvis':12}")

```

Predmet	ponudb	zvisanje
-----	-----	-----
slika	6	14
pozlačen dežnik	1	0
Meldrumove vaze	12	34
skodelice	12	33
kip	29	77
čajnik	1	0
srebrn jedilni servis	14	36
perzijska preproga	2	5

## Who is the most frequent Berta supporter?!

In one problem, Berta realised that she probably has an enemy who is always raising her bid. We had to help her unmask him. Basically, it would go like this:

First, we find out which lines Berta is advertising in. Let's remove the last line: if Berta happens to be there, it would be a nuisance for us later (we'll see when).

```
osebe[:-1] == "Berta"
```

```

Out[114]:
array([ True, False,  True, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False,  True, False,  True, False,  True, False,  True,
        False,  True, False,  True, False, False,  True, 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,  True, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False])

```

```

In [115]:
np.flatnonzero(osebe[:-1] == "Berta")

```

```

Out[115]:
array([ 0,  2, 20, 22, 24, 26, 28, 30, 33, 36, 60])

```

We will be interested in the names on the lines after those lines. So we add 1 to the indexes (and this is the reason we removed the last line - if Berta happened to be in the last line, we would have over-indexed).

```
np.flatnonzero(osebe[:-1] == "Berta") + 1
```

```
Out[116]:  
array([ 1,  3, 21, 23, 25, 27, 29, 31, 34, 37, 61])
```

```
In [117]:
```

```
osebe[np.flatnonzero(osebe[:-1] == "Berta") + 1]
```

```
Out[117]:  
array(['Ana', 'Fanči', 'Dani', 'Dani', 'Dani', 'Dani', 'Dani', 'Cilka',  
      'Ema', 'Cilka', 'Ema'], dtype='<U21')
```

And now we just need to count the number of times a single name appears on that list.

```
imena, za_berto = np.unique(osebe[np.flatnonzero(osebe[:-1] == "Berta") +
```

```
In [119]:
```

```
imena
```

```
Out[119]:
```

```
array(['Ana', 'Cilka', 'Dani', 'Ema', 'Fanči'], dtype='<U21')
```

```
In [120]:
```

```
za_berto
```

```
Out[120]:
```

```
array([1, 2, 5, 2, 1])
```

The largest counter is on the index

```
np.argmax(za_berto)
```

```
np.argmax(za_berto)
```

```
Out[121]:
```

```
2
```

And the name that goes with it - ha, there she is, the Bertha-hater – is

```
np.argmax(za_berto)
```

```
imena[np.argmax(za_berto)]
```

```
Out[122]:
```

```
'Dani'
```

## Logical operations on the tables and correction: who is really Berta's enemy?

But unfortunately, it's not that simple. Berta bought two products. After them, someone offered the first price for the next product, and we added him to Berta's enemies. We will have to go back a little.

This is a mask representing the lines in which Berta advertised (without the last one).

```
(osebe == "Berta")[:-1]
```

Out[123]:

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

These are lines that are not the last offer for a product - again, without the last offer.

```
predmeti[1:] == predmeti[:-1]
```

Out[124]:

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

We can see that we have done the right thing by looking at the minutes: they start with

```
slika,Berta,31
slika,Ana,33
slika,Berta,35
slika,Fanči,37
slika,Ana,40
slika,Fanči,45
pozlačen dežnik,Ema,29
Meldrumove vase,Greta,44
Meldrumove vase,Ana,46
```

The sixth and seventh elements of the above table are False; the corresponding rows are the last bids, i.e. the ones where Berta's row should not be considered as a bid that was subsequently increased.

We are therefore only interested in the rows where Berta is advertised and they are non-last bids, so, in effect, `(persons == "Berta")[:-1]` and `objects[1:] == objects[:-1]`.

Unfortunately, the and above tables do not work by element; the reasons are technical and historical. We have to use the `&` operator, which we know from sets. The role of or and not is taken over by `|` and `~`. Let's quickly check them with a simple example.

```
a = np.array([True, True, False, False])
b = np.array([True, False, False, True])

a & b
```

```
Out[125]:
array([ True, False, False, False])
```

```
In [126]:
```

```
a | b
```

```
Out[126]:
array([ True,  True, False,  True])
```

```
In [127]:
```

```
~b
```

```
Out[127]:
array([False,  True,  True, False])
```

If it ever comes in handy, there's also "exclusive or", which returns True if exactly one of the elements is True, but not both.

```
a ^ b
```

```
Out[128]:
array([False,  True, False,  True])
```

There is another complication with these operators: they are too powerful. They have too high a priority. The expression `a == b and c == d` would naturally mean `(a == b) and (c == d)`, since `==` binds stronger than `and`. But the `&` operator is stronger `a == b & c == d` would mean `a == (b & c) == d`. (Why didn't they make them weaker? Because they come from a different vocabulary. If we were interested in whether the intersection of the sets `a` and `b` is equal to `c`, we would write `a & b == c`, and it would seem logical that this would mean `(a & b) == c`. But `&` is not even originally an operation on sets, but on numbers. We haven't and won't talk about it, but that (and not sets) is the reason we use `&` between tables in numpy.)

So let's find out all about how to compute the conjunction of tables, right now let's build the right mask:

```
(osebe == "Berta")[:-1] & (predmeti[1:] == predmeti[:-1])
```

```
Out[129]:
array([ True, False,  True, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False,  True, False, True, False, True, False, True,
        False,  True, False, False, False, False,  True, 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,
        False, False, False, False, False, False, False, False, False,
         False, False, False, False])
```

And from there, everything goes back to the way it was.

```

imena, za_berto = np.unique(
    osebe[np.flatnonzero((osebe == "Berta")[:-1] & (predmeti[1:] == predm
return_counts=True)

```

In [131]:

```

za_berto

```

Out[131]:

```

array([1, 1, 5, 1, 1])

```

Yes, yes, it was [1, 2, 5, 2, 1]. But the worst one is still Dani.

## The highest increase in the last seven bids

In a homework assignment, it was necessary to find out how much the price of a single item had risen over the last seven bids; if there were fewer bids, then over the last as many bids as there were.

If the indices of the rows with the last bids in

```

koncni

```

Out[132]:

```

array([ 5,  6, 18, 30, 59, 60, 74, 76])

```

then we have to go back seven lines for each one,

```

koncni - 7

```

Out[133]:

```

array([-2, -1, 11, 23, 52, 53, 67, 69])

```

Of course, this is wrong, because for some items we don't even have seven bids. We are particularly sharply reminded that this is the case by the first two products, which even push us into negative indices. We need to "cap" the indexes of the final bid lines with the indexes of the first bid lines. We must take whichever of the two is larger.

```

print(koncni - 7)
print(zacetni)

```

```

[-2 -1 11 23 52 53 67 69]

```

```

[ 0  6  7 19 31 60 61 75]

```

How to do it? Simple; in fact, quite obvious from the way we've plotted them in the cell above: we put them in a single table and calculate the maximum.

```
np.array([koncni - 7, zacetni])
```

```
Out[135]:  
array([[ -2,  -1, 11, 23, 52, 53, 67, 69],  
       [  0,   6,  7, 19, 31, 60, 61, 75]])
```

```
In [136]:  
  
koncni_7 = np.max([koncni - 7, zacetni], axis=0)
```

```
In [137]:  
  
koncni_7
```

```
Out[137]:  
array([ 0,  6, 11, 23, 52, 60, 67, 75])
```

Just in case, we check that the rows end and end\_7 really refer to the same objects.

```
predmeti[koncni]
```

```
Out[138]:  
array(['slika', 'pozlačen dežnik', 'Meldrumove vaze', 'skodelice', 'kip',  
       'čajnik', 'srebrn jedilni servis', 'perzijska preproga'],  
      dtype='<U21')
```

```
In [139]:  
  
predmeti[koncni_7]
```

```
Out[139]:  
array(['slika', 'pozlačen dežnik', 'Meldrumove vaze', 'skodelice', 'kip',  
       'čajnik', 'srebrn jedilni servis', 'perzijska preproga'],  
      dtype='<U21')
```

Ah, why compare it to yourself! Let numpy do it for us.

```
predmeti[koncni] == predmeti[koncni_7]
```

```
Out[140]:  
array([ True,  True,  True,  True,  True,  True,  True,  True])
```

If we're even lazier, we can

```
np.all(predmeti[koncni] == predmeti[koncni_7])
```

```
Out[141]:  
True
```

Now prices[final] tells us the final prices, and prices[final-7] tells us the prices for the seven offers before. We want to know the difference.

```
cene[koncni] - cene[koncni_7]
```

```
Out[142]:  
array([14,  0, 21, 21, 15,  0, 16,  5])
```

And let's write it out.

```
for predmet, razlika in zip(predmeti[koncni], cene[koncni] - cene[koncni_1]):
    print(f"{predmet:25}{razlika:3}")
```

```
slika          14
pozlačen dežnik 0
Meldrumove vaze 21
skodelice      21
kip            15
čajnik         0
srebrn jedilni servis 16
perzijska preproga 5
```

## Function like that: clip

This is for impression. When I programmed the above example, I did as I did. But when I was converting it into notes, I remembered the clip function. This receives a table and a border; it sets all the elements below the lower border to the lower border and pushes those above the upper border to the upper border.

A professor calculates the grade by dividing the percentages obtained in the written exam by 10 and adding 1. Percentages can be greater than 100 because students can solve additional problems.

```
odstotki = np.array([63, 82, 45, 25, 125, 89])

ocene = odstotki // 10 + 1
ocene
```

```
Out[144]:
array([ 7,  9,  5,  3, 13,  9])
```

The UL rules state that we do not give students grades lower than 5 and higher than 10. And here we use np.clip.

```
np.clip(ocene, 5, 10)
```

```
Out[145]:
array([ 7,  9,  5,  5, 10,  9])
```

I knew about the function, but it had not occurred to me that the tables could also be the borders. Suppose that students also did homework and for some reason the final grade could not be higher than the homework grade. And, as before, it cannot be lower than 5.

```
domace = np.array([8, 6, 5, 5, 9, 10])
```

In [147]:

```
print(ocene)
print(domace)
print(np.clip(ocene, 5, domace))
```

```
[ 7  9  5  3 13  9]
[ 8  6  5  5  9 10]
[7 6 5 5 9 9]
```

Then, instead of

```
np.max([koncni - 7, zacetni], axis=0)
```

Out[148]:

```
array([ 0,  6, 11, 23, 52, 60, 67, 75])
```

They wrote

```
np.clip(koncni - 7, zacetni, None)
```

Out[149]:

```
array([ 0,  6, 11, 23, 52, 60, 67, 75])
```

There is no upper limit, so *None*.

Is this significantly shorter, better? If the tables were large and we cared, we could save a lot of memory. Besides, the name of the `np.clip` function makes it clearer what we're doing here; `np.max` is just ... a maximum.

Anyway, I'm writing this for another purpose: to show that numpy does indeed have a lot of functions, which we may or may not know and remember, or may or may not remember. The longer we use it, the more proficient and elegant our programs will be. Incidentally, the same is true of Python.