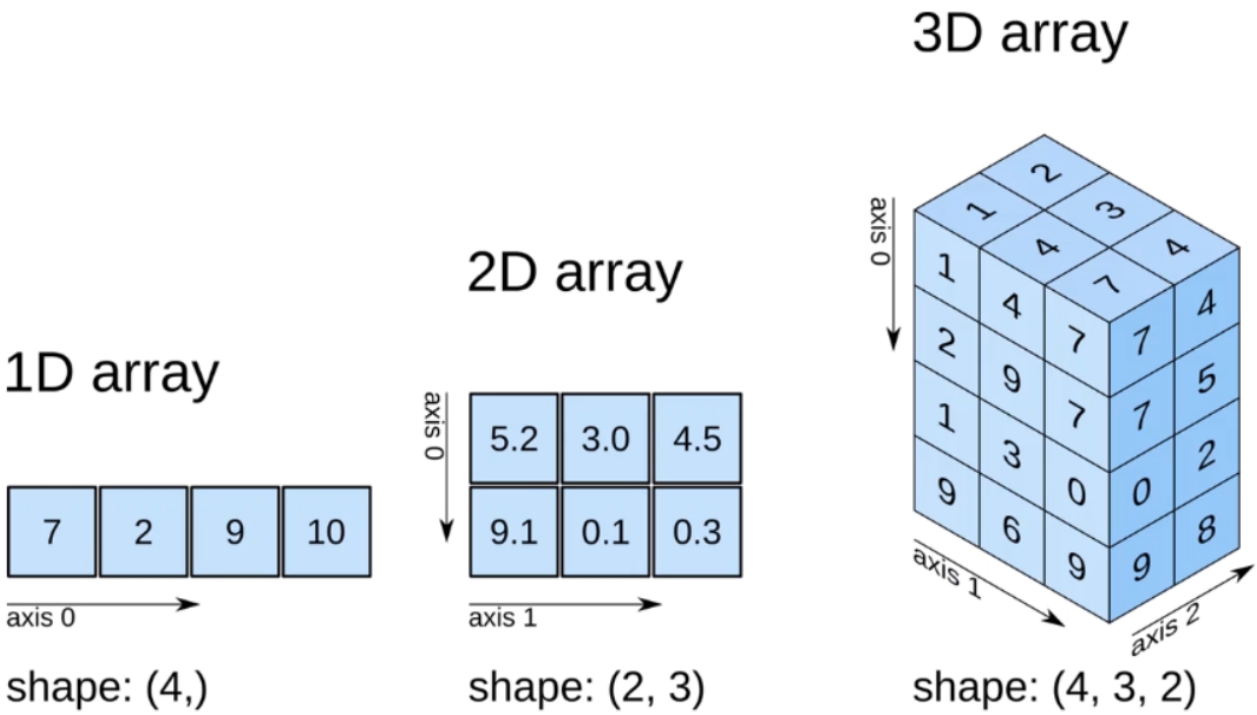
Numpy

Je knjižnjica za pomoč pri računanju z večdimenzijskimi listi in matrikami.

Numpy v glavnem operira s homogenimi večdimenzijskimi array-ji (listi). To so tabele array-jev, katerih elementi so vsi istega tipa. Reče se jim **ndarray**.

V numpy se dimenzijam reče **axis**.

Za primer: koordinate 3D točke bi lahko zapisali kot `[1,4,3]` . Naša točka ima *1 axis* in ta axis ima *3 elemente*.



Zakaj je numpy boljši kot navadni python list-i.

Na splošno - **numpy je hitrejši in optimiziran**.

> V numpy je integer lahko zapisana z 8biti. V python-u je integer objekt, kar pomeni, da bi za isto 8bitno številko dodatno potrebovali še nekaj dodanih bitov za opis te številke kot objekta.

> Numpy omogoča uporabo SIMD (Single Instruction Multiple Data) funkcionalnosti. To pomeni, da lahko isti ukaz naenkrat izvršimo nad večimi podatki. Primer: vsem elementov s array-ju lahko prištejemo isto vrednost naenkrat. Medtem, ko bi v python listu prištevanje mogli izvršiti nad vsakim elementom posebi.

Installing

```
pip install numpy
```

In [1]:

```
import numpy as np
```

Creating ndarray

Obstaja več načinov kako ustvarimo ndarray.

Lahko ga ustvarimo iz že obstoječih listov ali tuplov.

In [2]:

```
a = np.array([1,2,3])
print(a)
```

```
[1 2 3]
```

In [3]:

```
a = np.array([(1,2,3), (4,5,6)])
print(a)
print(a.shape)
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

Glavni atributi ndarray

ndarray.ndim

> število dimenzija

ndarray.shape

> oblika. Dobimo jo kot tupel števil, ki opisujejo obliko našega ndarray. Če imamo matriko z r vrsticami in c stolpci bi dobili obliko (r, c).

ndarray.size

> število vseh elementov v ndarray.

ndarray.dtype

> data tip elementov v ndarray

ndarray.itemsize

> velikost posameznega elementa v ndarray, podana v bytes. int32 = 4 (32/8)

In [5]:

```
a = np.array([(1,2,3), (4,5,6)])
print(a)
print("ndim: ", a.ndim)
print("shape:", a.shape)
print("size: ", a.size)
print("dtype: ", a.dtype)
print("itemsize; ", a.itemsize)
```

```
[[1 2 3]
 [4 5 6]]
ndim:  2
shape: (2, 3)
size:  6
dtype:  int64
itemsize;  8
```

In [ ]:

```

```

Numpy sam določi skupn tip elementov.

In [8]:

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

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

In [11]:

```
a = np.array([1,2,3.234,4])
print(a)
print(a.dtype)
```

```
[1.    2.    3.234 4.   ]
float64
```

Lahko pa data tip določimo sami.

In [12]:

```
a = np.array([1,2,3,4], dtype = "complex")
print(a)
print(a.dtype)
```

```
[1.+0.j 2.+0.j 3.+0.j 4.+0.j]
complex128
```

Velikokrat ne vemo točnih podatkov za naš array, vemo pa njegovo obliko. Numpy omogoča kreacijo array-a določene velikosti, ki vsebuje "placeholder" vrednost.

```
np.zeros(shape)
```

Nam vrne array oblike **shape**, katerega vrednosti so 0.

In [13]:

```
a = np.zeros((2,4))
print(a)
print(a.dtype)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
float64
```

```
np.ones(shape)
```

Nam vrne array oblike **shape**, katerega vrednosti so 1.

In [14]:

```
a = np.ones((2,4))
print(a)
print(a.dtype)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
float64
```

```
np.empty(shape)
```

Nam vrne array oblike **shape**, katerega vrednosti so random vrednosti odvisne od stanja pomnilnika.

In [21]:

```
a = np.empty((2,10))
print(a)
print(a.dtype)
```

```
[[0.00000000e+000 0.00000000e+000 0.00000000e+000 0.00000000e+000
  0.00000000e+000 5.30276956e+180 1.02941932e-071 4.57576743e-071
  3.85833328e-086 3.35709296e-143]
 [6.01433264e+175 6.93885958e+218 5.56218858e+180 3.94356143e+180
  7.93454752e-067 5.19650107e+170 5.18590542e+170 1.45251932e+165
  4.66905018e-143 1.50008929e+248]]
float64
```

```
np.full(shape, number)
```

Nam vrne array oblike **shape**, katerega vrednosti so **number**.

In [22]:

```
np.full((2,3),44)
```

Out[22]:

```
array([[44, 44, 44],
       [44, 44, 44]])
```

Podobno kot s python funkcijo **range(start,stop,step)** lahko tudi v numpy ustvarimo podoben array, ki se začne pri željeni vrednosti, konča pri željeni vrednosti, in izbira elemente s določenim korakom.

```
np.arange(start, stop, step)
```

Funkcija dovoljuje float vrednosti.

In [23]:

```
a = np.arange(10,45,5)
print(a)
```

```
[10 15 20 25 30 35 40]
```

In [24]:

```
a = np.arange(10,12,0.2)
print(a)
```

```
[10.  10.2 10.4 10.6 10.8 11.  11.2 11.4 11.6 11.8]
```

S pomočjo druge funkcije lahko določimo začetno in končno vrednost, ter število elementov.

```
np.linspace(start, stop, num_of_elements)
```

In [27]:

```
a = np.linspace(5,10,6)
print(a)
```

```
[ 5.  6.  7.  8.  9. 10.]
```

Basic operations

Aritmetične operacije so opravljene na vsakem elementu hkrati in shranjene v nov array.

In [28]:

```
a = np.array([[1,2,3], [4,5,6]])
b = a + 2
print(a)
print()

print(b)
```

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

[[3 4 5]
 [6 7 8]]
```

In [29]:

```
a = np.array([[1,2,3], [4,5,6]])
b = a - 10
print(a)
print()

print(b)
```

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

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

In [30]:

```
a = np.array([[1,2,3], [4,5,6]])
b = a*2
print(a)
print()

print(b)
```

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

[[ 2  4  6]
 [ 8 10 12]]
```

In [31]:

```
a = np.array([[1,2,3], [4,5,6]])
b = a**3
print(a)
print()

print(b)
```

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

[[  1   8  27]
 [ 64 125 216]]
```

Isti princip je pri operacijah med dvema ndarray.

In [32]:

```
a = np.array([[1,2,3], [4,5,6]])
b = a + a
print(a)
print()

print(b)
```

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

[[ 2  4  6]
 [ 8 10 12]]
```

In [33]:

```
a = np.array([[1,2,3], [4,5,6]])
b = a * a
print(a)
print()

print(b)
```

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

[[ 1  4  9]
 [16 25 36]]
```

In [35]:

```
a = np.array([[1,2,3], [4,5,6]])
b = a ** a
print(a)
print()

print(b)
```

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

[[    1     4    27]
 [  256  3125 46656]]
```

Matrični produkt dosežemo z @ ali z **.dot()** funkcijo.

In [36]:

```
a = np.array([[1,0,3],
              [0,0,3]])

b = np.array([[2,5],
              [3,0],
              [0,1]])

print(a@b)
# [ (1*2 + 0*3 + 3*0),     (1*5 + 0*0 + 3*1)]
# [ (0*2 + 0*3 + 3*0),     (0*5 + 0*0 + 3*1)]
print()

print(a.dot(b))
```

```
[[2 8]
 [0 3]]

[[2 8]
 [0 3]]
```

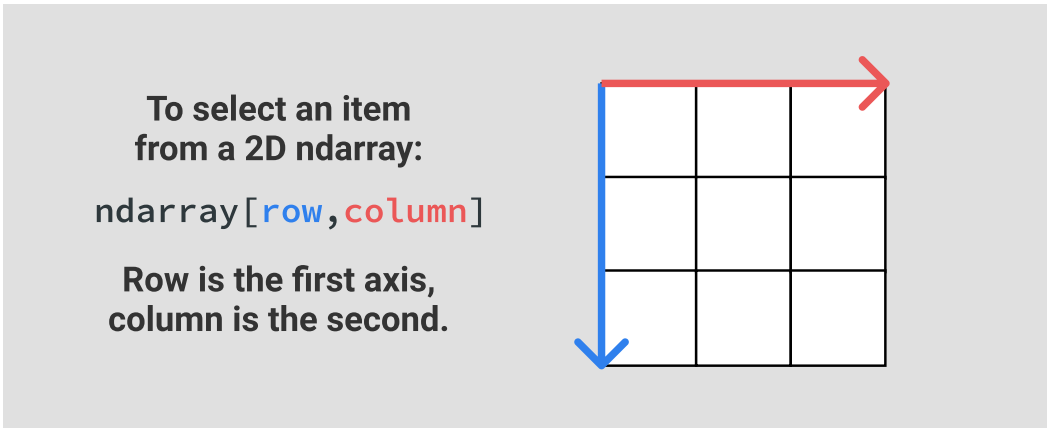
ndarray ima implementirane določene osnovne funkcije, kot so *min, max, sum...*.

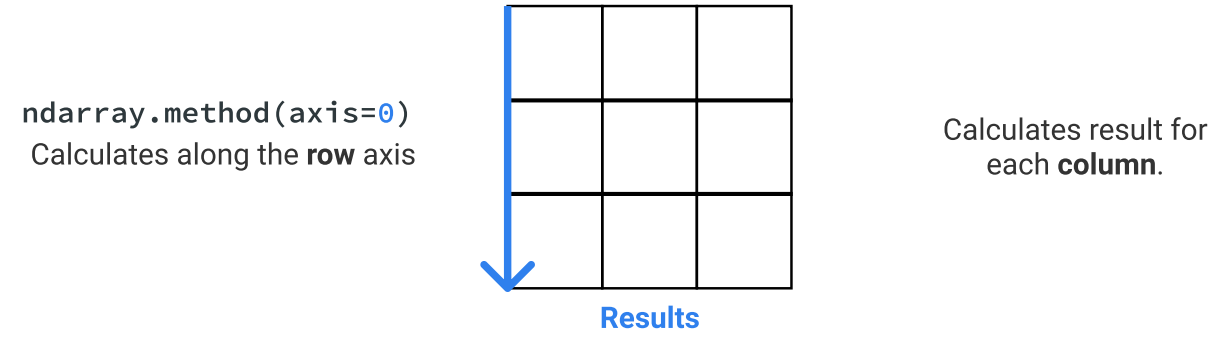
In [37]:

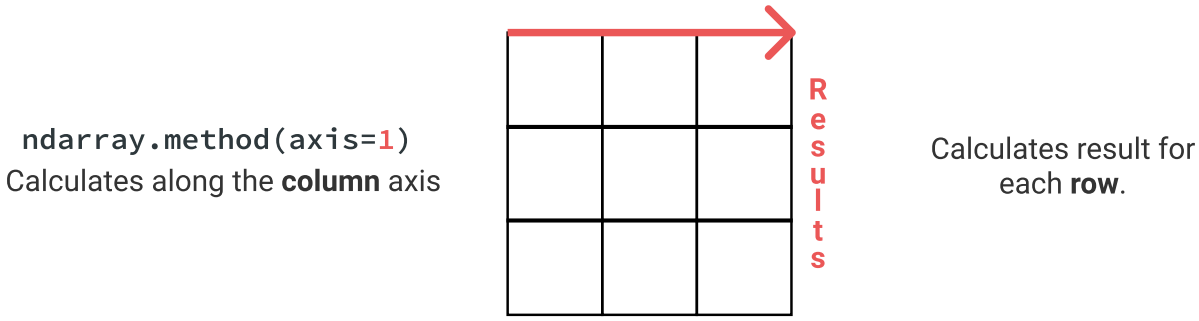
```
a = np.array([[1,2,3], [4,5,6]])

print(a.sum())
print(a.min())
print(a.max())
```

```
21
1
6
```

V funkcijah lahko tudi specificiramo katero *axis* gledamo.







In [38]:

```
a = np.array([[3,9,1],
              [4,7,6]])

print(a.sum(axis=0)) # izračuna vsoto stolpcev
print(a.min(axis=1)) # najde min vrednost vsake vrstice
print(a.max(axis=0)) # najde max vrednost vsakega stolpca
```

```
[ 7 16  7]
[1 4]
[4 9 6]
```

Indexin, slicing

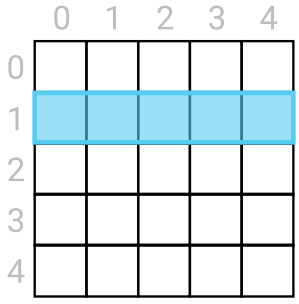
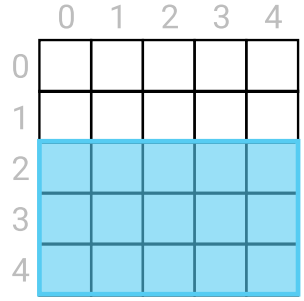
Tako kot v python list-u se lahko po ndarray premikamo z indexi in slicing.

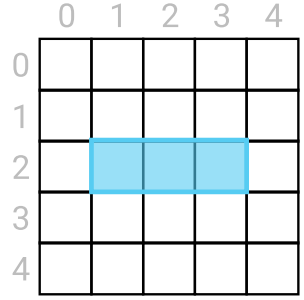
In [39]:

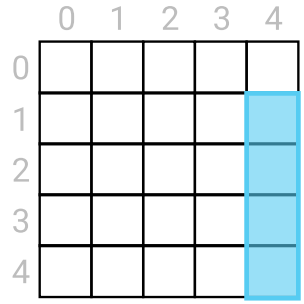
```
a = np.array([0,1,2,3,4])

print(a[2])
print(a[1:-2])
```

```
2
[1 2]
```

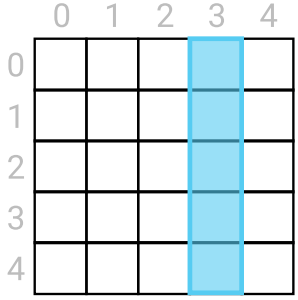
Če ima ndarray več dimenzij njihove indexe ločimo znotraj **[ ]** z vejico.

	List of lists method	*NumPy method*
Selecting a single row 	`sel_lol = data_lol[1]`	`sel_np = data_np[1]` *Same syntax as list of lists. Produces a 1D ndarray.*
Selecting multiple rows 	`sel_lol = data_lol[2:]`	`sel_np = data_np[2:]` *Same syntax as list of lists. Produces a 2D ndarray.*

	List of lists method	*NumPy method*
Selecting a 1D slice (row) 	`sel_lol = data_lol[2][1:4]`	`sel_np = data_np[2,1:4]` *Comma separated row location and column slice. Produces a 1D ndarray*

Selecting a 1D slice (column) 	`sel_lol = []`
	
	`rows = data_lol[1:]`
	`for r in rows:`
	`  col5 = r[4]`
	`  sel_lol.append(col5)`
		`sel_np = data_np[1:,4]`
		Comma separated row slice and column location. Produces a 1D ndarray

Selecting a single column



List of lists method

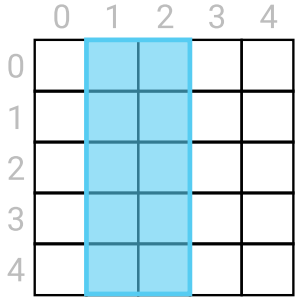
```
sel_lol = []

for row in data_lol:
    col4 = row[3]
    sel_lol.append(col4)
```

NumPy method

```
sel_np = data_np[:,3]
```

Comma separated row wildcard and column location. Produces a 1D ndarray

Selecting multiple columns



List of lists method

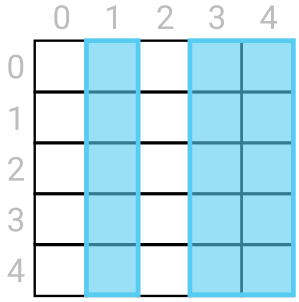
```
sel_lol = []

for row in data_lol:
    col23 = row[1:3]
    sel_lol.append(col23)
```

NumPy method

```
sel_np = data_np[:,1:3]
```

Comma separated row wildcard and column slice location. Produces a 2D ndarray

Selecting multiple, specific columns



List of lists method

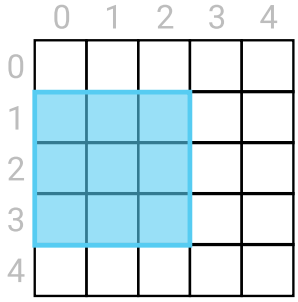
```
sel_lol = []

for row in data_lol:
    cols = [row[1],
            row[3],row[4]]
    sel_lol.append(cols)
```

NumPy method

```
cols = [1,3,4]
sel_np = data_np[:,cols]
```

Comma separated row wildcard and list of column locations. Produces a 2D ndarray

Selecting a 2D slice



List of lists method

```
sel_lol = []

rows = data_lol[1:4]
for r in rows:
    new_row = r[:3]
    sel_lol.append(new_row)
```

NumPy method

```
sel_np = data_np[1:4,:3]
```

Comma separated row/column slice locations. Returns a 2D ndarray

In [43]:

```
a = np.array([[0,1,2,3,4],
              [5,6,7,8,9],
              [10,11,12,13,14],
              [15,16,17,18,19]])

print(a[1,3]) # vrstica z indexom 1, stolpec z indexom 3
print()

print(a[(1,3), 1:3]) # vrstice z indexom 1 in 3, stolpci z indexom 1, 2
```

```
8

[[ 6  7]
 [16 17]]
```

Copies and Views

Pri manipuliranju z array-ji je potrebno paziti kaj je v kateri spremenljivki shranjeno.

V naslednjem primeru a kaže na določen del v spominu, v katerm je shranjen naš array. Ko naredimo **a=b** tudi b prične kazati na isto mesto v spominu. Tako da, če naredimo spremembo pri enem se ta pojavi tudi pri drugem ndarray.

In [44]:

```
a = np.array([[1,2,3,4],
              [5,6,7,8]])
print(f"A: {a}")
print()

b = a
print(f"B: {b}")
print()

b[1,1] = 100
print(f"A: {a}")
print()
print(f"B: {b}")
```

```
A: [[1 2 3 4]
 [5 6 7 8]]

B: [[1 2 3 4]
 [5 6 7 8]]

A: [[  1   2   3   4]
 [  5 100   7   8]]

B: [[  1   2   3   4]
 [  5 100   7   8]]
```

Isto velja, če ndarray pošiljam v funkcije.

In [45]:

```
def fun(x):
    x[0,0] = 100

a = np.array([[1,2,3,4],
              [5,6,7,8]])
print(a)
fun(a)
print(a)
```

```
[[1 2 3 4]
 [5 6 7 8]]
[[100   2   3   4]
 [  5   6   7   8]]
```

In isto velja pri slicingu.

In [47]:

```
a = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12],
              [13,14,15,16]])
print(a)

b = a[1:-1, 1:-1]
print(b)

print()
b[0,0] = 100

print(f"A: {a}")
print()
print(f"B: {b}")
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
[[ 6  7]
 [10 11]]

A: [[  1   2   3   4]
 [  5 100   7   8]
 [  9  10  11  12]
 [ 13  14  15  16]]

B: [[100   7]
 [ 10  11]]
```

Če želimo dobiti kopijo ndarray-ja, kličemo posebno funkcijo.

In [48]:

```
a = np.array([[1,2,3,4],
              [5,6,7,8]])

b = a.copy()

print(f"A: {a}")
print()
print(f"B: {b}")
print("---------")

b[0,1] = 100
print(f"A: {a}")
print()
print(f"B: {b}")
```

```
A: [[1 2 3 4]
 [5 6 7 8]]

B: [[1 2 3 4]
 [5 6 7 8]]
---------
A: [[1 2 3 4]
 [5 6 7 8]]

B: [[  1 100   3   4]
 [  5   6   7   8]]
```

Adding data

Za dodajanje podatkov lahko uporabimo funkcijo **numpy.concatenate()**.

In [49]:

```
a = np.ones((2,3))
print(a)
print(a.shape)
```

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

In [50]:

```
b = np.zeros(3)
print(b)
print(b.shape)
```

```
[0. 0. 0.]
(3,)
```

In [51]:

```
c = np.concatenate((a,b), axis=0)
```

```
---------------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
 last)
<ipython-input-51-47afc4f72d12> in <module>
----> 1 c = np.concatenate((a,b), axis=0)

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input arrays must have same number of dimensions,
 but the array at index 0 has 2 dimension(s) and the array at index 1
 has 1 dimension(s)
```

Da spremenimo obliko ndarray uporabimo funkcijo **reshape**. Potrebno je le, da je število elementov v prvotnem in spremenjenem ndarray enako.

In [52]:

```
a = np.array([[1,2,3,4,5],
              [6,7,8,9,10],
              [11,12,13,14,15],
              [16,17,18,19,20]])

print(a)
print(a.shape)

print()

a = a.reshape((2,10))
print(a)
print(a.shape)

print()
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
(4, 5)

[[ 1  2  3  4  5  6  7  8  9 10]
 [11 12 13 14 15 16 17 18 19 20]]
(2, 10)
```

Če dodamo -1 v našo obliko, se bo ta dimenzija samodejno izračunala na podlagi ostalih dimenzij in skupnega števila elementov.

In [53]:

```
a = np.ones((2,3))
print(a)
print(a.shape)

b = np.zeros(3)
print(b)
print(b.shape)
b = b.reshape((1,3))

c = np.concatenate((a,b), axis=0)
print("C")
print(c)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
(2, 3)
[0. 0. 0.]
(3,)
C
[[1. 1. 1.]
 [1. 1. 1.]
 [0. 0. 0.]]
```

Za dodajanje ndarray-ev lahko uporabimo funkcijo **vstack** oziroma **hstack**.

`np.vstack((ndarry1, ndarray2))` je isto kot klicanje `np.concatenate((ndarray1, ndarray2), axis=0)`

`np.hstack((ndarry1, ndarray2))` je isto kot klicanje `np.concatenate((ndarray1, ndarray2), axis=1)`

In [54]:

```
a = np.array([[1,2,3],
              [4,5,6]])
print(a)
print(a.shape)

b = np.array([[7,8,9]])
print(b)
print(b.shape)

c = np.vstack((a, b))
print(c)
```

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

In [55]:

```
a = np.array([[1,2,3],
              [4,5,6]])
print(a)
print(a.shape)

b = np.array([[7], [8]])
print(b)
print(b.shape)

c = np.hstack((a, b))
print(c)
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
[[7]
 [8]]
(2, 1)
[[1 2 3 7]
 [4 5 6 8]]
```

Boolean array and boolean indexing

Boolean array so arrays katerih elementi imajo boolean vrednost (True ali False).

Do boolean array lahko preprosti pridemo z logičnimi operacijami nad celotnim ndarray:

```
[1,2,3,4,5] < 3
```

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

Z boolean array lahko izberemo točno določene podatke iz naših ndarray. Kjer je vrednost True, ta podatek bomo izbrali.

In [65]:

```
a = np.random.randint(0,10, size=(6, 4))
print(a)
```

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

Lahko izberemo točno določene celice oziroma stolpce.

In [71]:

```
print(a[(True, True, False, False, True, True), :]) # izbere 0, 1, 4, 5 vrstico
```

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

Oziroma, izberemo lahko točno določene elemente.

In [72]:

```
filter_ = a < 6 # naredimo boolean array, kjer je vrednost True, če je element bil
print(filter_)
```

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

In [73]:

```
print(a[filter_]) # izberemo vse elemente na podlagi našega filtra
```

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

Vaja

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

Določite največjo vrednost, ki se pojavi v ndarray. Pridobite vse vrstice, kjer se ta vrednost pojavi in izpišite zadnji element v taki vrstici.

In [8]:

```
a = np.array([[4, 8, 0, 1, 3],
       [1, 2, 9, 2, 5],
       [4, 1, 3, 9, 7],
       [6, 0, 3, 5, 8]])

for vrstica in a:
    if a.max() in (vrstica):
        print(vrstica[-1])
```

```
5
7
```

In [ ]: