https://dfedorov.spb.ru

Все рассмотренные нами примеры имеют дело с векторами в одном измерении. Ключевой частью красоты NumPy является его способность применять все, что мы рассмотрели до сих пор, к любому количеству измерений.

Создание матриц

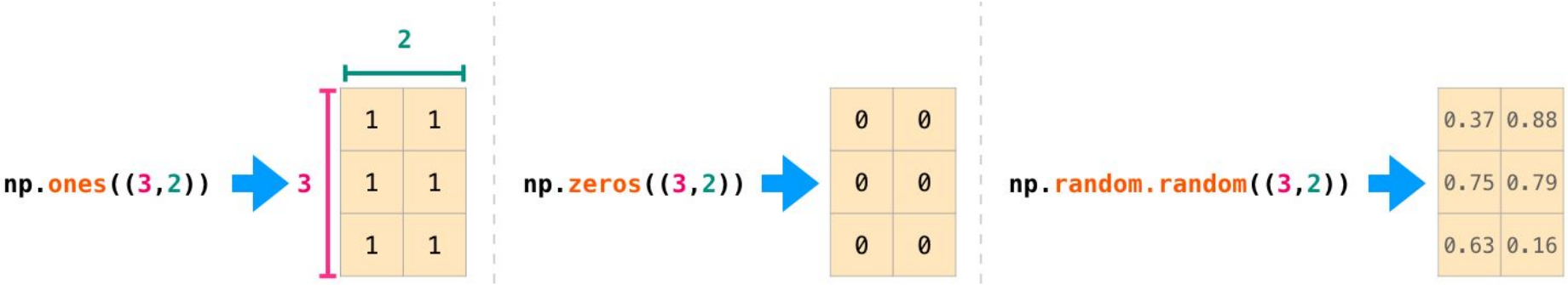
Мы можем передать списки списков Python в следующей форме, чтобы NumPy создал матрицу для их представления:

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

```
>>> np.array([range(i, i + 3) for i in [2, 4, 6]])
```

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

Мы также можем использовать те же методы, которые мы упомянули выше (**ones()**, **zeros()** и **random.random()**), если мы дадим им кортеж, описывающий размеры создаваемой матрицы:




```
>>> np.full((3, 5), 3.14)

>>> np.eye(3) # единичная матрица
```

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

a

1	2	3
4	5	6

```
.dtype == np.int32
.shape == (2, 3)
```

```
len(a) == a.shape[0]
```

```
np.zeros((3, 2))
```

2

3

0.	0.
0.	0.
0.	0.

```
np.full((3, 2), 7)
```

7	7
7	7
7	7

```
np.eye(3, 3)
```

1.	0.	0.
0.	1.	0.
0.	0.	1.

```
= np.eye(3)
```

```
np.ones((3, 2))
```

1.	1.
1.	1.
1.	1.

```
np.empty((3, 2))
```


Размерность матрицы

```
>>> np.random.random((3, 3))

>>> np.random.normal(0, 1, (3, 3))

>>> np.random.randint(0, 10, (3, 3))
```

```
np.random.randint(0, 10, (3, 2))
```

uniform, $x \in [0, 10)$

4	8
3	7
5	2

Careful!

```
random.randint(0, 10)
```

$x \in [0,10]$

```
np.random.rand(3, 2)
```

uniform, $x \in [0, 1)$

0.7	0.5
0.3	0.8
0.4	0.1

```
np.random.randn(3, 2)
```

normal, $\mu=0$, $\sigma=1$

0.4	-1.2
1.4	-0.3
0.8	0.7

```
np.random.uniform(1, 10, (3, 2))
```

uniform, $x \in [1, 10)$

9.6	8.7
3.8	2.6
6.0	9.4

```
np.random.normal(10, 2, (3, 2))
```

normal, $\mu=10$, $\sigma=2$

8.7	12.3
10.5	10.8
14.3	11.2

```
rng = np.random.default_rng()
```

```
rng.integers(0, 10, 3)
```

uniform, x ∈ [0, 10)

4	3	7

```
rng.integers(0, 10, 3, endpoint=True)
```

uniform, x ∈ [0, 10]

```
rng.random(3)
```

uniform, x ∈ [0, 1)

0.7	0.3	0.8

```
rng.standard_normal(3)
```

normal, μ=0, σ=1

0.4	-1.1	0.8

```
rng.uniform(1, 10, 3)
```

uniform, x ∈ [1, 10)

5.1	2.7	7.2

```
rng.normal(5, 2, 3)
```

normal, μ=5, σ=2

4.5	3.2	6.7

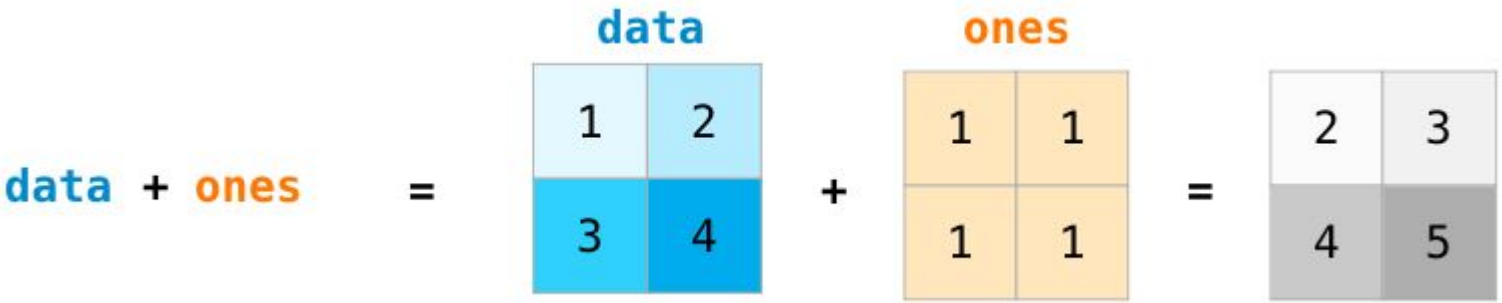
```
>>> import numpy as np
>>> np.random.seed(0)  # seed

>>> x3 = np.random.randint(10, size=(3, 4, 5))
```

```
>>> x3.ndim          # размерность: 3
>>> x3.shape         # размер каждого измерения: (3, 4, 5)
>>> x3.size          # общий размер массива: 60
>>> x3.dtype         # тип данных
>>> x3.itemsize      # размер в байтах каждого элемента
```

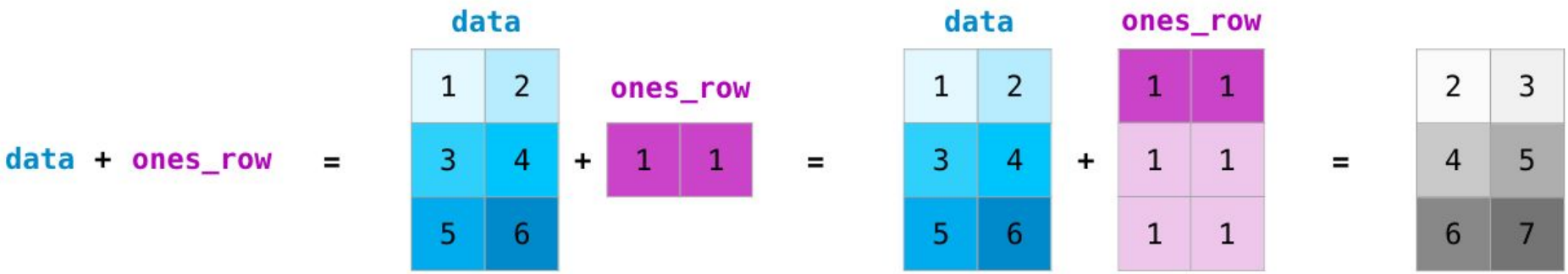
Функция	Описание
`array`	Преобразует входные данные (список, кортеж, массив или любую другую последовательность) в ndarray. Тип dtype задается явно или выводится неявно. Входные данные по умолчанию копируются
`asarray`	Преобразует входные данные в ndarray, но не копирует, если на вход уже подан ndarray
`arange`	Аналогична встроенной функции range, но возвращает массив, а не список
`ones`, `ones_like`	Порождает массив, состоящий из одних единиц, с заданными атрибутами shape и dtype. Функция `ones_like` принимает другой массив и порождает массив из одних единиц с такими же значениями shape и dtype
`zeros`, `zeros_like`	Аналогичны `ones` и `ones_like`, только порождаемый массив состоит из одних нулей
`empty`, `empty_like`	Создают новые массивы, выделяя под них память, но, в отличие от `ones` и `zeros`, не инициализируют ее
`full`, `full_like`	Создают массивы с заданными атрибутами shape и dtype, в которых все элементы равны заданному символу-заполнителю. `full_like` принимает массив и порождает заполненный массив с такими же значениями атрибутов shape и dtype
`eye`, `identity`	Создают единичную квадратную матрицу N×N (элементы на главной диагонали равны 1, все остальные – 0)

Матричная арифметика

Мы можем складывать и умножать матрицы с помощью арифметических операторов (+ - * /), если две матрицы имеют одинаковый размер. NumPy обрабатывает их как позиционные операции (*векторизация*):



Мы можем обойтись без выполнения этих арифметических операций с матрицами разного размера, только если другое измерение равно одному (например, матрица имеет только один столбец или одну строку), и в этом случае NumPy использует свои правила *широковещания (транслирования, broadcasting)* для этой операции:



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} / \begin{bmatrix} 9 & 9 & 9 \\ 9 & 9 & 9 \\ 9 & 9 & 9 \end{bmatrix} = \begin{bmatrix} .1 & .2 & .3 \\ .4 & .5 & .7 \\ .8 & .9 & 1. \end{bmatrix}$$

normalization

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 3 \\ -4 & 0 & 6 \\ -7 & 0 & 9 \end{bmatrix}$$

multiplying several columns at once

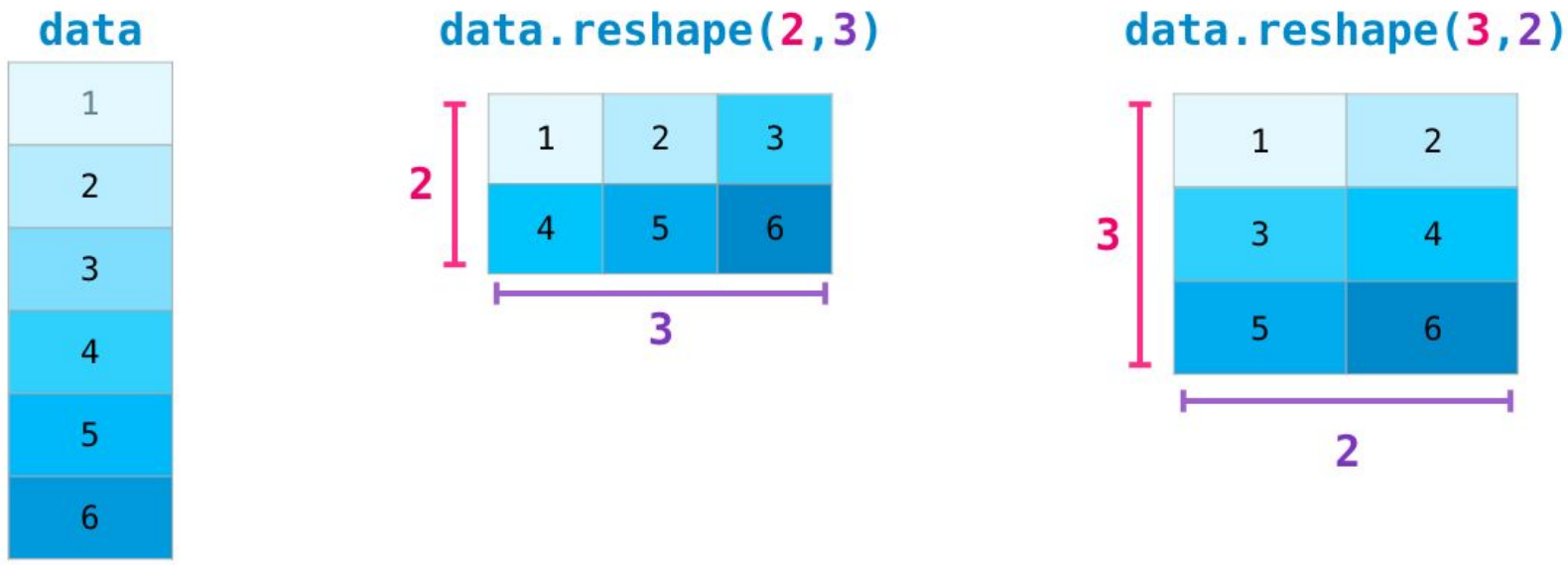
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} / \begin{bmatrix} 3 & 3 & 3 \\ 6 & 6 & 6 \\ 9 & 9 & 9 \end{bmatrix} = \begin{bmatrix} .3 & .7 & 1. \\ .6 & .8 & 1. \\ .8 & .9 & 1. \end{bmatrix}$$

row-wise normalization

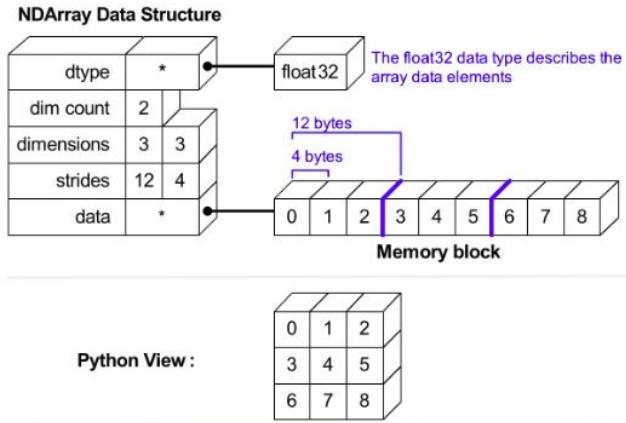
$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

outer product

В более продвинутом варианте использования вам может потребоваться переключить размеры определенной матрицы. Это часто имеет место в приложениях машинного обучения, где определенная модель ожидает определенной формы для входных данных, которая отличается от вашего набора данных. В этих случаях полезен метод NumPy **reshape()**. Вы просто передаете ей новые размеры, которые хотите для матрицы. Вы можете передать -1 для измерения, и NumPy может вывести правильное измерение на основе вашей матрицы:

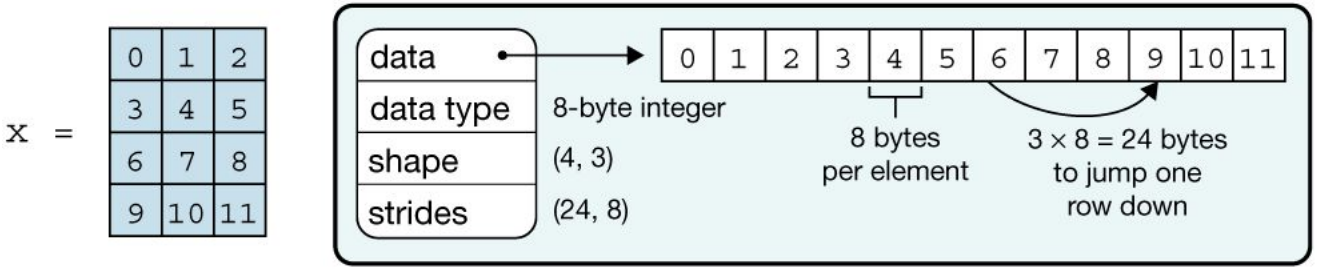


Array Data Structure




```
>>> np.arange(9).reshape((3, 3))
```

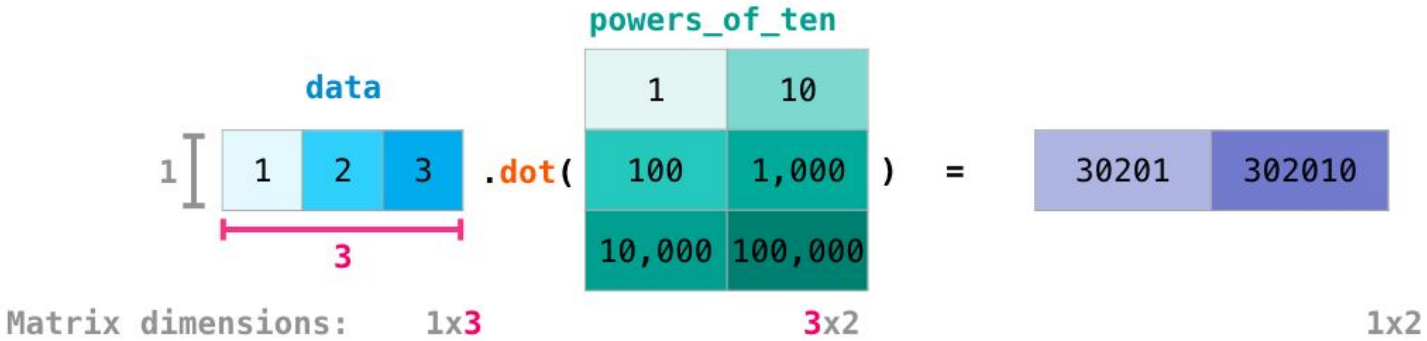
Data structure

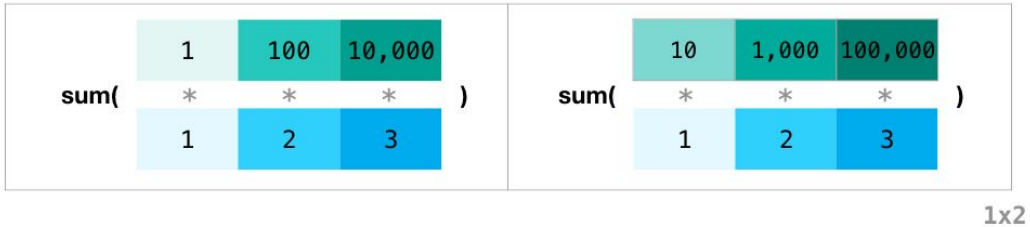


https://numpy.org/doc/stable/reference/arrays.html

Скалярное произведение

Ключевым отличием арифметики является умножение матриц с использованием скалярного произведения. NumPy предоставляет каждой матрице метод **dot()**, который мы можем использовать для выполнения операций скалярного произведения с другими матрицами:





1*1 + 2*100 + 3*10,000	1*10 + 2*1,000 + 3*100,000	=	30201 302010

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} @ \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

outer product

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} @ \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 14 \end{bmatrix}$$

inner (or dot) product

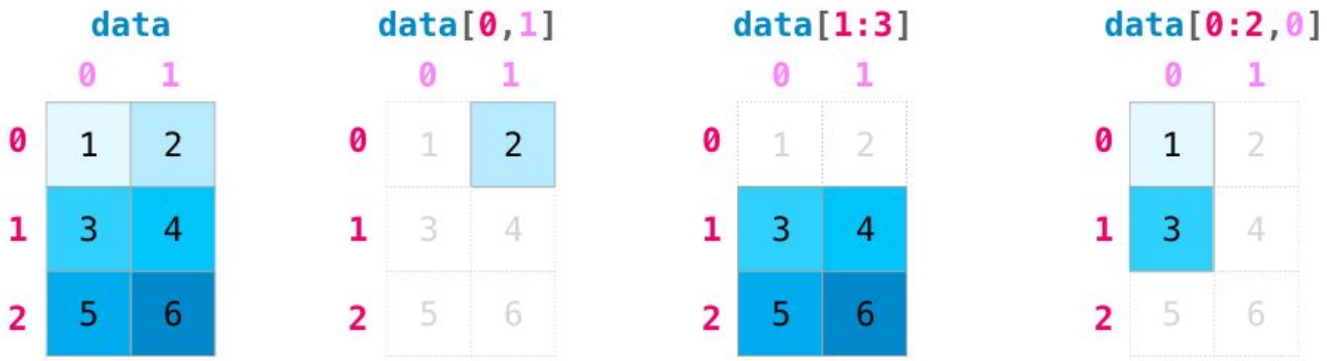
Наиболее употребительные функции из модуля numpy.linalg

Функция	Описание
diag	Возвращает диагональные элементы квадратной матрицы в виде одномерного массива или преобразует одномерный массив в квадратную матрицу, в которой все элементы, кроме находящихся на главной диагонали, равны нулю
dot	Вычисляет произведение матриц
trace	Вычисляет след матрицы – сумму диагональных элементов
det	Вычисляет определитель матрицы
eig	Вычисляет собственные значения и собственные векторы квадратной матрицы

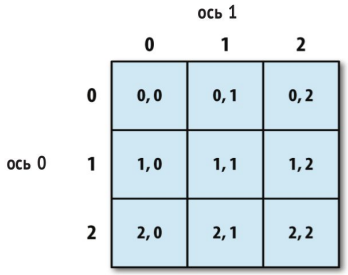
Функция	Описание
inv	Вычисляет обратную матрицу
pinv	Вычисляет псевдообратную матрицу Мура-Пенроуза для квадратной матрицы
qr	Вычисляет QR-разложение
svd	Вычисляет сингулярное разложение (SVD)
solve	Решает линейную систему Ax = b, где A – квадратная матрица
lstsq	Вычисляет решение уравнения y = Xb по методу наименьших квадратов

Индексирование матрицы

Операции индексирования и нарезки становятся еще более полезными, когда мы манипулируем матрицами:




```
>>> data[0][-1] = 5.4
>>> data[(0, -1)]
>>> data[0, -1]
```



Выражение	Форма
`arr[:2, 1:]`	`(2, 2)`
`arr[2]`	`(3,)`
`arr[2, :]`	`(3,)`
`arr[2:, :]`	`(1, 3)`
`arr[:, :2]`	`(3, 2)`
`arr[1, :2]`	`(2,)`
`arr[1:2, :2]`	`(1, 2)`

a

1	2	3	4
5	6	7	8
9	10	11	12

a[1,2]

1	2	3	4
5	6	7	8
9	10	11	12

a[1,:]

1	2	3	4
5	6	7	8
9	10	11	12

= a[1]

a[:,2]

1	2	3	4
5	6	7	8
9	10	11	12

a[:,1:3]

1	2	3	4
5	6	7	8
9	10	11	12

a[-2:,-2:]

1	2	3	4
5	6	7	8
9	10	11	12

a[::2,1::2]

1	2	3	4
5	6	7	8
9	10	11	12

```
>>> import numpy as np
>>> np.random.seed(0)  # seed
>>> x2 = np.random.randint(10, size=(3, 4))
>>> x2
>>> x2[:2, :3]        # two rows, three columns
>>> x2[:3, ::2]       # all rows, каждый второй столбец
>>> x2[::-1, ::-1]    # перевернуть
>>> x2[:, 0]          # first column of x2
>>> x2[0, :]          # first row of x2
>>> x2[0]             # equivalent to x2[0, :]
```

```
>>> import numpy as np
>>> np.random.seed(0)  # seed
>>> x2 = np.random.randint(10, size=(3, 4))
>>> x2
```

x2_sub

```
[[ 99  5  2  4]
 [  7  6  8  8]
 [  1  6  7  7]]
```

```
>>> x2_sub[0, 0] = 99
```

```
>>> x2_sub = x2[:2, :2] # представление, а не копия
>>> x2_sub
>>> x2_sub[0, 0] = 99
>>> x2_sub
>>> x2
```

Создание копии среза

```
>>> x2_sub_copy = x2[:2, :2].copy()
>>> x2_sub_copy
>>> x2_sub_copy[0, 0] = 42
>>> x2_sub_copy
>>> x2
```

x2

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

x2_sub_copy

```
[[12  5]
 [ 7  6]]
```

https://dfedorov.spb.ru