

# Нескучный нумпур

## Содержание

[Что такое NumPy?](#)

[Создание массива](#)

[Доступ к элементам, срезы](#)

[Форма массива и ее изменение](#)

[Перестановка осей и транспонирование](#)

[Объединение массивов](#)

[Клонирование данных](#)

[Математические операции над элементами массива](#)

[Матричное умножение](#)

[Агрегаторы](#)

[Вместо заключения – пример](#)

## Что такое NumPy?

Это библиотека с открытым исходным кодом, некогда отделившаяся от проекта SciPy. NumPy является наследником Numeric и NumArray. Основан NumPy на библиотеке LAPAC, которая написана на Fortran. Не-python альтернативой для NumPy является Matlab.

В силу того, что NumPy базируется на Fortran это быстрая библиотека. А в силу того, что поддерживает векторные операции с многомерными массивами – крайне удобная.

Кроме базового варианта (многомерные массивы в базовом варианте) NumPy включает в себя набор пакетов для решения специализированных задач, например:

- `numpy.linalg` – реализует операции линейной алгебры (простое умножение векторов и матриц есть в базовом варианте);
- `numpy.random` – реализует функции для работы со

случайными величинами;

- `numpy.fft` – реализует прямое и обратное преобразование Фурье.

Итак, я предлагаю рассмотреть подробно всего несколько возможностей NumPy и примеров их использования, которых будет достаточно, чтобы вы поняли, на сколько мощный этот инструмент!

[<наверх>](#)

## Создание массива

Создать массив можно несколькими способами:

1. преобразовать список в массив:

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

2. скопировать массив (копия и глубокая копия обязательна!!!):

```
B = A.copy()
B
Out:
array([ [1, 2, 3],
       [4, 5, 6] ])
```

3. создать нулевой или единичный массив заданного размера:

```
A = np.zeros((2, 3))
A
Out:
array([ [0., 0., 0.],
       [0., 0., 0.] ])
```

```
B = np.ones((3, 2))
B
Out:
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

```
B = np.ones((3, 2))
B
Out:
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

Либо взять размеры уже существующего массива:

```
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.zeros_like(A)
B
Out:
array([[0, 0, 0],
       [0, 0, 0]])
```

```
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.ones_like(A)
B
Out:
array([[1, 1, 1],
       [1, 1, 1]])
```

4. при создании двумерного квадратного массива можете сделать его единичной диагональной матрицей:

```
A = np.eye(3)
A
Out:
```

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

5. построить массив чисел от From (включая) до To (не включая) с шагом Step:

```
From = 2.5
To = 7
Step = 0.5
A = np.arange(From, To, Step)
A
Out:
array([2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5])
```

По умолчанию from = 0, step = 1, поэтому возможен вариант с одним параметром, интерпретируемым как To:

```
A = np.arange(5)
A
Out:
array([0, 1, 2, 3, 4])
```

Либо с двумя – как From и To:

```
A = np.arange(10, 15)
A
Out:
array([10, 11, 12, 13, 14])
```

Обратите внимание, что в методе №3 размеры массива передавались в качестве **одного** параметра (кортеж размеров). Вторым параметром в способах №3 и №4 можно указать желаемый тип элементов массива:

```
A = np.zeros((2, 3), 'int')
```

```
A
Out:
array([ [0, 0, 0],
       [0, 0, 0] ])
```

```
B = np.ones((3, 2), 'complex')
B
Out:
array([ [1.+0.j, 1.+0.j],
       [1.+0.j, 1.+0.j],
       [1.+0.j, 1.+0.j] ])
```

Используя метод `astype`, можно привести массив к другому типу.  
В качестве параметра указывается желаемый тип:

```
A = np.ones((3, 2))
B = A.astype('str')
B
Out:
array([ ['1.0', '1.0'],
       ['1.0', '1.0'],
       ['1.0', '1.0'] ], dtype='<U32')
```

Все доступные типы можно найти в словаре `sctypes`:

```
np.sctypes
Out:
{'int': [numpy.int8, numpy.int16, numpy.int32, numpy.int64],
 'uint': [numpy.uint8, numpy.uint16, numpy.uint32,
          numpy.uint64],
 'float': [numpy.float16, numpy.float32, numpy.float64,
           numpy.float128],
 'complex': [numpy.complex64, numpy.complex128,
             numpy.complex256],
 'others': [bool, object, bytes, str, numpy.void]}
```

[<наверх>](#)

# Доступ к элементам, срезы

Доступ к элементам массива осуществляется по целочисленным индексами, начинается отсчет с 0:

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

Если представить многомерный массив как систему вложенных одномерных массивов (линейный массив, элементы которого могут быть линейными массивами), становится очевидной возможность получать доступ к подмассивам с использованием неполного набора индексов:

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

С учетом этой парадигмы, можем переписать пример доступа к одному элементу:

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

При использовании неполного набора индексов, недостающие индексы неявно заменяются списком всех возможных индексов вдоль соответствующей оси. Сделать это явным образом можно, поставив «:». Предыдущий пример с одним индексом можно переписать в следующем виде:

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

```
Out:  
array([4, 5, 6])
```

«Пропустить» индекс можно вдоль любой оси или осей, если за «пропущенной» осью последуют оси с индексацией, то «`:`» обязательно:

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

Индексы могут принимать отрицательные целые значения. В этом случае отсчет ведется от конца массива:

```
A = np.arange(5)  
print(A)  
A[-1]  
Out:  
[0 1 2 3 4]  
4
```

Можно использовать не одиночные индексы, а списки индексов вдоль каждой оси:

```
A = np.arange(5)  
print(A)  
A[ [0, 1, -1] ]  
Out:  
[0 1 2 3 4]  
array([0, 1, 4])
```

Либо диапазоны индексов в виде «From:To:Step». Такая конструкция называется срезом. Выбираются все элементы по списку индексов начиная с индекса From включительно, до индекса To не **включая** с шагом Step:

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

```
print(A)
A[0:4:2]
Out:
[0 1 2 3 4]
array([0, 2])
```

Шаг индекса имеет значение по умолчанию 1 и может быть пропущен:

```
A = np.arange(5)
print(A)
A[0:4]
Out:
[0 1 2 3 4]
array([0, 1, 2, 3])
```

Значения From и To тоже имеют дефолтные значения: 0 и размер массива по оси индексации соответственно:

```
A = np.arange(5)
print(A)
A[:4]
Out:
[0 1 2 3 4]
array([0, 1, 2, 3])
```

```
A = np.arange(5)
print(A)
A[-3:]
Out:
[0 1 2 3 4]
array([2, 3, 4])
```

Если вы хотите использовать From и To по умолчанию (все индексы по данной оси) а шаг отличный от 1, то вам необходимо использовать две пары двоеточий, чтобы интерпретатор смог идентифицировать единственный параметр как Step. Следующий код «разворачивает» массив вдоль второй оси, а вдоль первой не меняет:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
B = A[:, ::-1]
print("A", A)
print("B", B)
Out:
A [ [1 2 3]
 [4 5 6] ]
B [ [3 2 1]
 [6 5 4] ]
```

А теперь выполним

```
print(A)
B[0, 0] = 0
print(A)
Out:
[ [1 2 3]
 [4 5 6] ]
[ [1 2 0]
 [4 5 6] ]
```

Как видите, через B мы изменили данные в A. Вот почему в реальных задачах важно использовать копии. Пример выше должен был бы выглядеть так:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
B = A.copy()[:, ::-1]
print("A", A)
print("B", B)
Out:
A [ [1 2 3]
 [4 5 6] ]
B [ [3 2 1]
 [6 5 4] ]
```

В NumPy также реализована возможность доступа ко множеству элементов массива через булев индексный массив. Индексный массив должен совпадать по форме с индексируемым.

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
I = np.array([ [False, False, True], [ True, False, True] ])
A[I]
Out:
array([3, 4, 6])
```

Как видите, такая конструкция возвращает плоский массив, состоящий из элементов индексируемого массива, соответствующих истинным индексам. Однако, если мы используем такой доступ к элементам массива для изменения их значений, то форма массива сохранится:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
I = np.array([ [False, False, True], [ True, False, True] ])
A[I] = 0
print(A)
Out:
[[1 2 0]
 [0 5 0]]
```

Над индексирующими булевыми массивами определены логические операции `logical_and`, `logical_or` и `logical_not` выполняющие логические операции И, ИЛИ и НЕ поэлементно:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
I1 = np.array([ [False, False, True], [True, False, True] ])
I2 = np.array([ [False, True, False], [False, False, True] ])
B = A.copy()
C = A.copy()
D = A.copy()
B[np.logical_and(I1, I2)] = 0
C[np.logical_or(I1, I2)] = 0
D[np.logical_not(I1)] = 0
print('B\n', B)
print('\nC\n', C)
print('\nD\n', D)
Out:
B
[[1 2 3]
 [4 5 0]]
```

```
C
[ [1 0 0]
[0 5 0] ]
D
[ [0 0 3]
[4 0 6] ]
```

`logical_and` и `logical_or` принимают 2 операнда, `logical_not` – один. Можно использовать операторы `&`, `|` и `~` для выполнения И, ИЛИ и НЕ соответственно с любым количеством operandов:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
I1 = np.array([ [False, False, True], [True, False, True] ])
I2 = np.array([ [False, True, False], [False, False, True] ])
A[I1 & (I1 | ~ I2)] = 0
print(A)
Out:
[ [1 2 0]
[0 5 0] ]
```

Что эквивалентно применению только `I1`.

Получить индексирующий логический массив, соответствующий по форме массиву значений можно, записав логическое условие с именем массива в качестве операнда. Булево значение индекса будет рассчитано как истинность выражения для соответствующего элемента массива.

Найдем индексирующий массив `I` элементов, которые больше, чем 3, а элементы со значениями меньше чем 2 и больше 4 – обнулим:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
print('A before\n', A)
I = A > 3
print('I\n', I)
A[np.logical_or(A < 2, A > 4)] = 0
print('A after\n', A)
Out:
A before
[ [1 2 3]
```

```
[4 5 6] ]
I
[ [False False False]
 [ True  True  True] ]
A after
[ [0 2 3]
 [4 0 0] ]
```

[<наверх>](#)

## Форма массива и ее изменение

Многомерный массив можно представить как одномерный массив максимальной длины, нарезанный на фрагменты по длине самой последней оси и уложенный слоями по осям, начиная с последних.

Для наглядности рассмотрим пример:

```
A = np.arange(24)
B = A.reshape(4, 6)
C = A.reshape(4, 3, 2)
print('B\n', B)
print('\nC\n', C)
Out:
B
[ [ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23] ]
C
[ [ [ 0  1]
 [ 2  3]
 [ 4  5] ]
 [ [ 6  7]
 [ 8  9]
 [10 11] ]
 [ [12 13]
 [14 15]
 [16 17] ]
 [ [18 19]
```

```
[20 21]  
[22 23] ] ]
```

В этом примере мы из одномерного массива длиной 24 элемента сформировали 2 новых массива. Массив В, размером 4 на 6. Если посмотреть на порядок значений, то видно, что вдоль второго измерения идут цепочки последовательных значений.

В массиве С, размером 4 на 3 на 2, непрерывные значения идут вдоль последней оси. Вдоль второй оси идут последовательно блоки, объединение которых дало бы в результате строки вдоль второй оси массива В.

А учитывая, что мы не делали копии, становится понятно, что это разные формы представления одного и того же массива данных. Поэтому можно легко и быстро менять форму массива, не изменяя самих данных.

Чтобы узнать размерность массива (количество осей), можно использовать поле `ndim` (число), а чтобы узнать размер вдоль каждой оси – `shape` (кортеж). Размерность можно также узнать и по длине `shape`. Чтобы узнать полное количество элементов в массиве можно воспользоваться значением `size`:

```
A = np.arange(24)  
C = A.reshape(4, 3, 2)  
print(C.ndim, C.shape, len(C.shape), A.size)  
Out:  
3 (4, 3, 2) 3 24
```

Обратите внимание, что `ndim` и `shape` – это атрибуты, а не методы!

Чтобы увидеть массив одномерным, можно воспользоваться функцией `ravel`:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])  
A.ravel()  
Out:  
array([1, 2, 3, 4, 5, 6])
```

Чтобы поменять размеры вдоль осей или размерность используется метод `reshape`:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
A.reshape(3, 2)
Out:
array([ [1, 2],
       [3, 4],
       [5, 6] ])
```

Важно, чтобы количество элементов сохранилось. Иначе возникнет ошибка:

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
A.reshape(3, 3)
Out:
ValueError                                Traceback (most
recent call last)
<ipython-input-73-d204e18427d9> in <module>
      1 A = np.array([ [1, 2, 3], [4, 5, 6] ])
----> 2 A.reshape(3, 3)
ValueError: cannot reshape array of size 6 into shape (3,3)
```

Учитывая, что количество элементов постоянно, размер вдоль одной любой оси при выполнении `reshape` может быть вычислен из значений длины вдоль других осей. Размер вдоль одной оси можно обозначить `-1` и тогда он будет вычислен автоматически:

```
A = np.arange(24)
B = A.reshape(4, -1)
C = A.reshape(4, -1, 2)
print(B.shape, C.shape)
Out:
(4, 6) (4, 3, 2)
```

Можно `reshape` использовать вместо `ravel`:

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

```
B = A.reshape(-1)
print(B.shape)
Out:
(6,)
```

Рассмотрим практическое применение некоторых возможностей для обработки изображений. В качестве объекта исследования будем использовать фотографию:

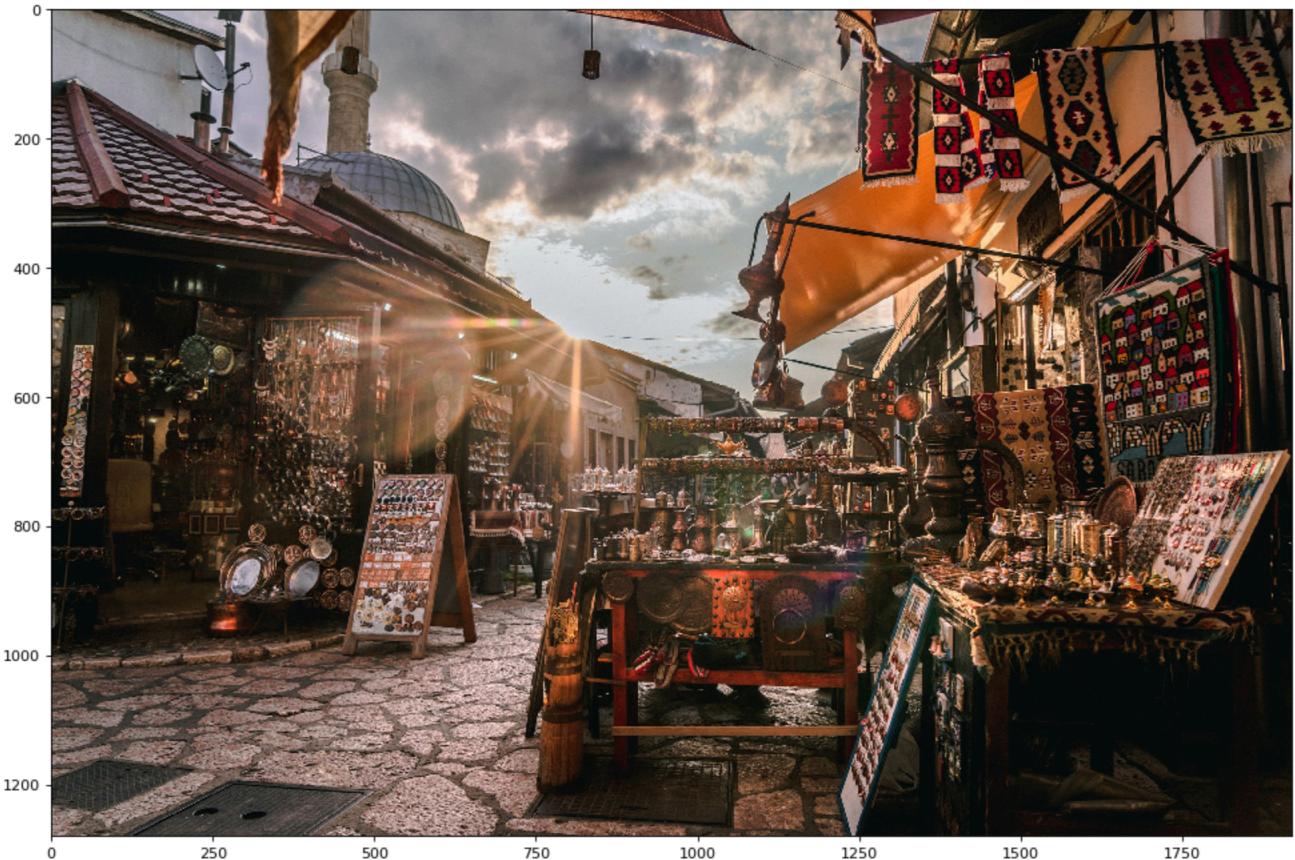


Исходная фотография для упражнений

Попробуем ее загрузить и визуализировать средствами Python. Для этого нам понадобятся OpenCV и Matplotlib:

```
import cv2
from matplotlib import pyplot as plt
I = cv2.imread('sarajevo.jpg')[::, ::-1]
plt.figure(num=None, figsize=(15, 15), dpi=80, facecolor='w',
edgecolor='k')
plt.imshow(I)
plt.show()
```

Результат будет такой:



Результат загрузки фото

Обратите внимание на строку загрузки:

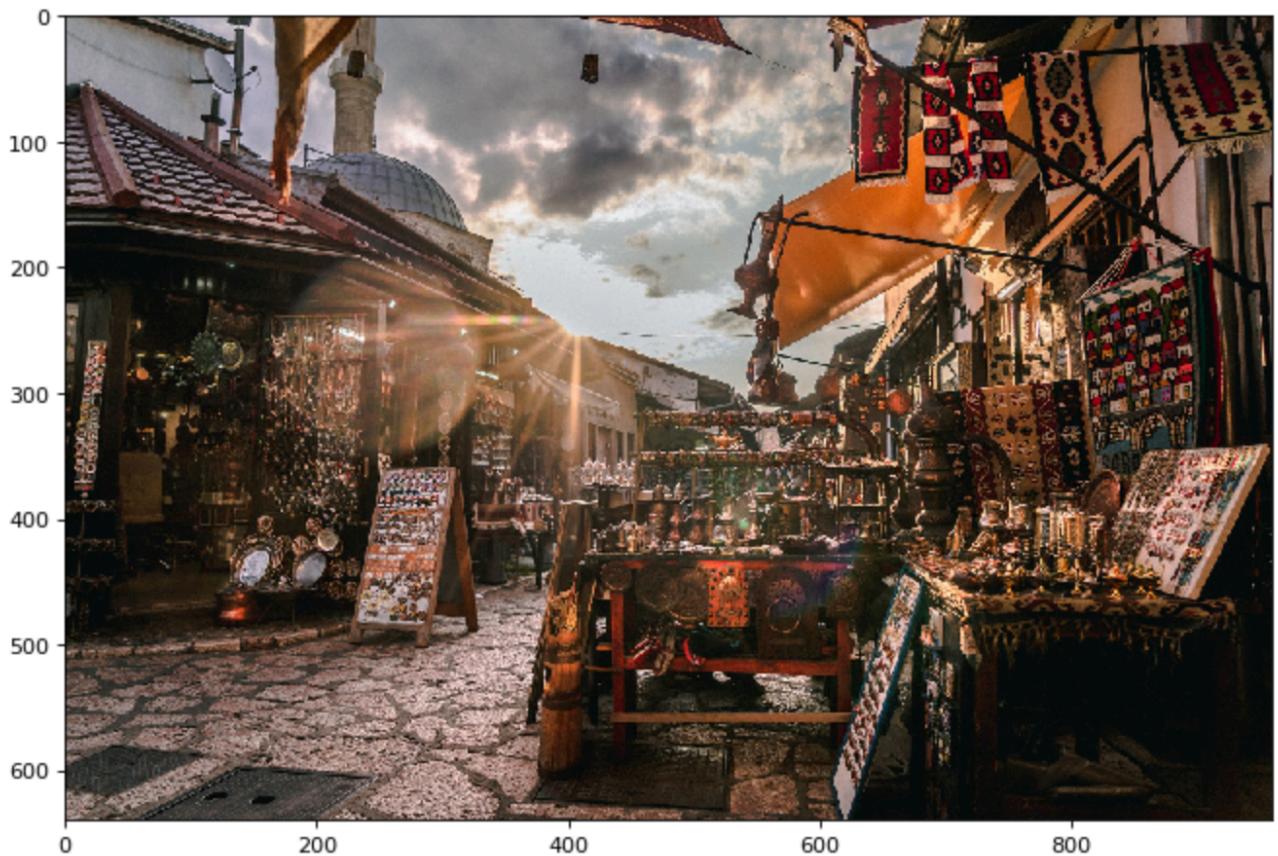
```
I = cv2.imread('sarajevo.jpg')[::, ::, ::-1]
print(I.shape)
Out:
(1280, 1920, 3)
```

OpenCV работает с изображениями в формате BGR, а нам привычен RGB. Мы меняем порядок байтов вдоль оси цвета без обращения к функциям OpenCV, используя конструкцию «`[::, ::, ::-1]`».

Уменьшим изображение в 2 раза по каждой оси. Наше изображение имеет четные размеры по осям, соответственно, может быть уменьшено без интерполяции:

```
I_ = I.reshape(I.shape[0] // 2, 2, I.shape[1] // 2, 2, -1)
print(I_.shape)
```

```
plt.figure(num=None, figsize=(10, 10), dpi=80, facecolor='w',
edgecolor='k')
plt.imshow(I_[:, 0, :, 0])
plt.show()
```



Уменьшим изображение в 2 раза по каждой оси

Поменяв форму массива, мы получили 2 новые оси, по 2 значения в каждой, им соответствуют кадры, составленные из нечетных и четных строк и столбцов исходного изображения.

Низкое качество связано с использованием Matplotlib, за то там видны размеры по осям. На самом деле, качество уменьшенного изображения такое:



Такое вот качество уменьшенного изображения

[<наверх>](#)

## Перестановка осей и транспонирование

В кроме изменения формы массива при неизменном порядке единиц данных, часто встречается необходимость изменить порядок следования осей, что естественным образом повлечет перестановки блоков данных.

Примером такого преобразования может быть транспонирование матрицы: взаимозамена строк и столбцов.

```
A = np.array([ [1, 2, 3], [4, 5, 6] ])
print('A\n', A)
print('\nA data\n', A.ravel())
B = A.T
```

```
print('\nB\n', B)
print('\nB data\n', B.ravel())
Out:
A
[[1 2 3]
 [4 5 6]]
A data
[1 2 3 4 5 6]
B
[[1 4]
 [2 5]
 [3 6]]
B data
[1 4 2 5 3 6]
```

В этом примере для транспонирования матрицы A использовалась конструкция A.T. Оператор транспонирования инвертирует порядок осей. Рассмотрим еще один пример с тремя осями:

```
C = np.arange(24).reshape(4, -1, 2)
print(C.shape, np.transpose(C).shape)
print()
print(C[0])
print()
print(C.T[:, :, 0])
Out:
[[0 1]
 [2 3]
 [4 5]]
[[0 2 4]
 [1 3 5]]
```

У этой короткой записи есть более длинный аналог: np.transpose(A). Это более универсальный инструмент для замены порядка осей. Вторым параметром можно задать кортеж номеров осей исходного массива, определяющий порядок их положения в результирующем массиве.

Для примера переставим первые две оси изображения. Картинка должна перевернуться, но цветовую ось оставим без изменения:

```
I_ = np.transpose(I, (1, 0, 2))
plt.figure(num=None, figsize=(15, 15), dpi=80, facecolor='w',
edgecolor='k')
plt.imshow(I_)
plt.show()
```



Для этого примера можно было применить другой инструмент `swapaxes`. Этот метод переставляет местами две оси, указанные в

параметрах. Пример выше можно было реализовать так:

```
I_ = np.swapaxes(I, 0, 1)
```

[<наверх>](#)

## Объединение массивов

Объединяемые массивы должны иметь одинаковое количество осей. Объединять массивы можно с образованием новой оси, либо вдоль уже существующей.

Для объединения с образованием новой оси исходные массивы должны иметь одинаковые размеры вдоль всех осей:

```
A = np.array([ [1, 2, 3, 4], [5, 6, 7, 8] ])
B = A[::-1]
C = A[:, ::-1]
D = np.stack((A, B, C))
print(D.shape)
D
Out:
(3, 2, 4)
array([ [ [1, 2, 3, 4],
          [5, 6, 7, 8] ],
         [ [5, 6, 7, 8],
          [1, 2, 3, 4] ],
         [ [4, 3, 2, 1],
          [8, 7, 6, 5] ] ])
```

Как видно из примера, массивы-операнды стали подмассивами нового объекта и выстроились вдоль новой оси, которая стоит самой первой по порядку.

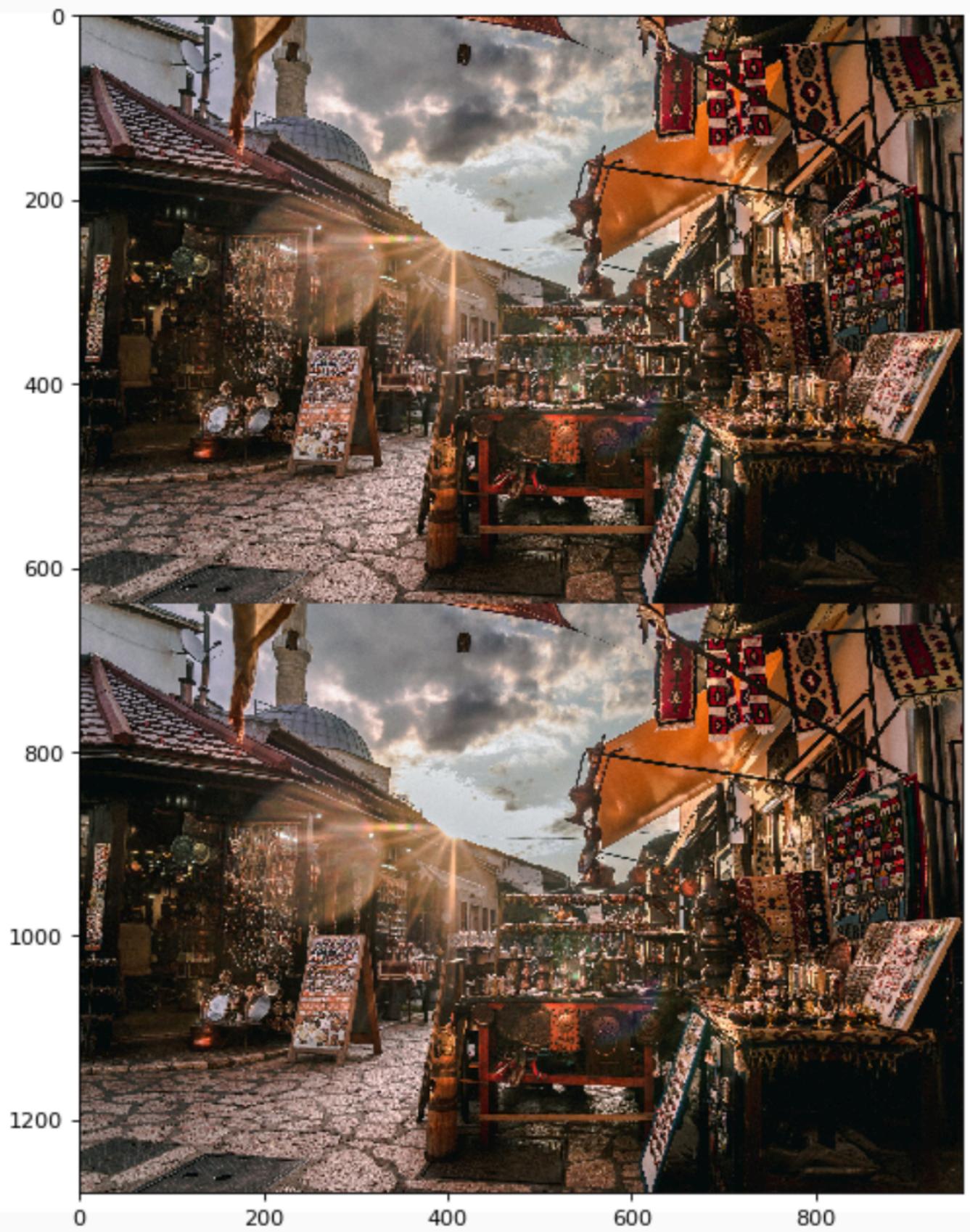
Для объединения массивов вдоль существующей оси, они должны иметь одинаковый размер по всем осям, кроме выбранной для объединения, а по ней могут иметь произвольные размеры:

```
A = np.ones((2, 1, 2))
B = np.zeros((2, 3, 2))
C = np.concatenate((A, B), 1)
print(C.shape)
C
Out:
(2, 4, 2)
array([ [ [1., 1.],
          [0., 0.],
          [0., 0.],
          [0., 0.] ],
         [ [1., 1.],
          [0., 0.],
          [0., 0.],
          [0., 0.] ] ])
```

Для объединения по первой или второй оси можно использовать методы `vstack` и `hstack` соответственно. Покажем это на примере изображений. `vstack` объединяет изображения одинаковой ширины по высоте, а `hsstack` объединяет одинаковые по высоте картинки в одно широкое:

```
I = cv2.imread('sarajevo.jpg')[:, :, ::-1]
I_ = I.reshape(I.shape[0] // 2, 2, I.shape[1] // 2, 2, -1)
Ih = np.hstack((I_[:, 0, :, 0], I_[:, 0, :, 1]))
Iv = np.vstack((I_[:, 0, :, 0], I_[:, 1, :, 0]))
plt.figure(num=None, figsize=(10, 10), dpi=80, facecolor='w',
edgecolor='k')
plt.imshow(Ih)
plt.show()
plt.figure(num=None, figsize=(10, 10), dpi=80, facecolor='w',
edgecolor='k')
plt.imshow(Iv)
plt.show()
```





Обратите внимание на то, что во всех примерах этого раздела объединяемые массивы передаются одним параметром (кортежем). Количество operandов может быть любым, а не обязательно только 2.

Также обратите внимание на то, что происходит с памятью, при объединении массивов:

```
A = np.array([ [1, 2, 3, 4], [5, 6, 7, 8] ])
B = A[::-1]
C = A[:, ::-1]
D = np.stack((A, B, C))
D[0, 0, 0] = 0
print(A)
Out:
[[1 2 3 4]
 [5 6 7 8]]
```

Так как создается новый объект, данные в него копируются из исходных массивов, поэтому изменения в новых данных не влияют на исходные.

[<наверх>](#)

## Клонирование данных

Оператор `np.repeat(A, n)` вернет одномерный массив с элементами массива A, каждый из которых будет повторен n раз.

```
A = np.array([ [1, 2, 3, 4], [5, 6, 7, 8] ])
print(np.repeat(A, 2))
Out:
[1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8]
```

После этого преобразования, можно перестроить геометрию массива и собрать повторяющиеся данные в одну ось:

```
A = np.array([ [1, 2, 3, 4], [5, 6, 7, 8] ])
B = np.repeat(A, 2).reshape(A.shape[0], A.shape[1], -1)
print(B)
Out:
[[[1 1]
 [2 2]]]
```

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

Этот вариант отличается от объединения массива с самим собой оператором `stack` только положением оси, вдоль которой стоят одинаковые данные. В примере выше это последняя ось, если использовать `stack` – первая:

```
A = np.array([ [1, 2, 3, 4], [5, 6, 7, 8] ])
B = np.stack((A, A))
print(B)
Out:
[ [ [1 2 3 4]
  [5 6 7 8] ]
[ [1 2 3 4]
  [5 6 7 8] ] ]
```

Как бы ни было выполнено клонирование данных, следующим шагом можно переместить ось, вдоль которой стоят одинаковые значения, в любую позицию с системе осей:

```
A = np.array([ [1, 2, 3, 4], [5, 6, 7, 8] ])
B = np.transpose(np.stack((A, A)), (1, 0, 2))
C = np.transpose(np.repeat(A, 2).reshape(A.shape[0],
A.shape[1], -1), (0, 2, 1))
print('B\n', B)
print('\nC\n', C)
Out:
B
[ [ [1 2 3 4]
  [1 2 3 4] ]
[ [5 6 7 8]
  [5 6 7 8] ] ]
C
[ [[1 2 3 4]
  [1 2 3 4] ] ]
```

```
[ [5 6 7 8]
  [5 6 7 8] ] ]
```

Если же мы хотим «растянуть» какую либо ось, используя повторение элементов, то ось с одинаковыми значениями надо поставить **после** растягиваемой (используя transpose), а затем объединить эти две оси (используя reshape). Рассмотрим пример с растяжением изображения вдоль вертикальной оси за счет дублирования строк:

```
I0 = cv2.imread('sarajevo.jpg')[:, :, ::-1]      # загрузили
большое изображение
I1 = I.reshape(I.shape[0] // 2, 2, I.shape[1] // 2, 2, -1)[:, ,
0, :, 0] # уменьшили вдвое по каждому измерению
I2 = np.repeat(I1, 2) # склонировали данные
I3 = I2.reshape(I1.shape[0], I1.shape[1], I1.shape[2], -1)
I4 = np.transpose(I3, (0, 3, 1, 2)) # поменяли порядок осей
I5 = I4.reshape(-1, I1.shape[1], I1.shape[2]) # объединили оси
print('I0', I0.shape)
print('I1', I1.shape)
print('I2', I2.shape)
print('I3', I3.shape)
print('I4', I4.shape)
print('I5', I5.shape)
plt.figure(num=None, figsize=(10, 10), dpi=80, facecolor='w',
edgecolor='k')
plt.imshow(I5)
plt.show()
Out:
I0 (1280, 1920, 3)
I1 (640, 960, 3)
I2 (3686400,)
I3 (640, 960, 3, 2)
I4 (640, 2, 960, 3)
I5 (1280, 960, 3)
```



Растяжение вдоль вертикальной оси

[<наверх>](#)

# Математические операции над элементами массива

Если A и B массивы одинакового размера, то их можно складывать, умножать, вычитать, делить и возводить в степень. Эти операции выполняются **поэлементно**, результирующий массив будет совпадать по геометрии с исходными массивами, а каждый его элемент будет результатом выполнения соответствующей операции над парой элементов из исходных массивов:

```
A = np.array([ [-1., 2., 3.], [4., 5., 6.], [7., 8., 9.] ])
B = np.array([ [1., -2., -3.], [7., 8., 9.], [4., 5., 6.] ])
C = A + B
D = A - B
E = A * B
F = A / B
G = A ** B
print('+\\n', C, '\\n')
print('-\\n', D, '\\n')
print('*\\n', E, '\\n')
print('/\\n', F, '\\n')
print('**\\n', G, '\\n')
Out:
+
[[ 0.  0.  0.]
 [11. 13. 15.]
 [11. 13. 15.]]
-
[[ -2.  4.  6.]
 [-3. -3. -3.]
 [ 3.  3.  3.]]
*
[[ -1. -4. -9.]
 [28. 40. 54.]
 [28. 40. 54.]]
/
[[ [-1.          -1.          -1.          ]
   [ 0.57142857  0.625        0.66666667]
   [ 1.75         1.6          1.5          ]]]
```

```
**
[ [-1.0000000e+00  2.5000000e-01  3.7037037e-02]
[ 1.6384000e+04  3.9062500e+05  1.0077696e+07]
[ 2.4010000e+03  3.2768000e+04  5.3144100e+05] ]
```

Можно выполнить любую операцию из приведенных выше над массивом и числом. В этом случае операция также выполнится над каждым из элементов массива:

```
A = np.array([ [-1., 2., 3.], [4., 5., 6.], [7., 8., 9.] ])
B = -2.

C = A + B
D = A - B
E = A * B
F = A / B
G = A ** B
print('+\\n', C, '\\n')
print('-\\n', D, '\\n')
print('*\\n', E, '\\n')
print('/\\n', F, '\\n')
print('**\\n', G, '\\n')
Out:
+
[ [-3.  0.  1.]
[ 2.  3.  4.]
[ 5.  6.  7.] ]

-
[ [ 1.  4.  5.]
[ 6.  7.  8.]
[ 9.  10. 11.] ]

*
[ [ 2.  -4.  -6.]
[ -8. -10. -12.]
[ -14. -16. -18.] ]

/
[ [ 0.5 -1.  -1.5]
[ -2.  -2.5 -3. ]
[ -3.5 -4.  -4.5] ]

**
[ [1.          0.25        0.11111111]
```

```
[0.0625    0.04    0.02777778]
[0.02040816 0.015625   0.01234568] ]
```

Учитывая, что многомерный массив можно рассматривать как плоский массив (первая ось), элементы которого – массивы (остальные оси), возможно выполнение рассматриваемых операций над массивами A и B в случае, когда геометрия B совпадает с геометрией подмассивов A при фиксированном значении по первой оси. Иными словами, при совпадающем количестве осей и размерах A[i] и B. Этот случай каждый из массивов A[i] и B будут operandами для операций, определенных над массивами.

```
A = np.array([ [1., 2., 3.], [4., 5., 6.], [7., 8., 9.] ])
B = np.array([-1.1, -1.2, -1.3])
C = A.T + B
D = A.T - B
E = A.T * B
F = A.T / B
G = A.T ** B
print('+\\n', C, '\\n')
print('-\\n', D, '\\n')
print('*\\n', E, '\\n')
print('/\\n', F, '\\n')
print('**\\n', G, '\\n')
Out:
+
[[ -0.1  2.8  5.7]
 [ 0.9  3.8  6.7]
 [ 1.9  4.8  7.7] ]

-
[[ 2.1  5.2  8.3]
 [ 3.1  6.2  9.3]
 [ 4.1  7.2 10.3] ]

*
[[ -1.1  -4.8  -9.1]
 [ -2.2  -6.   -10.4]
 [ -3.3  -7.2  -11.7] ]

/
[[ -0.90909091 -3.33333333 -5.38461538]
 [-1.81818182 -4.16666667 -6.15384615]]
```

```

[-2.72727273 -5.           -6.92307692] ]
**
[ [1.          0.18946457 0.07968426]
[0.4665165   0.14495593 0.06698584]
[0.29865282  0.11647119 0.05747576] ]

```

В этом примере массив  $B$  подвергается операции с каждой строкой массива  $A$ . При необходимости умножения/деления/сложения/вычитания/возведения степень подмассивов вдоль другой оси, необходимо использовать транспонирование, чтобы поставить нужную ось на место первой, а затем вернуть ее на свое место. Рассмотри пример выше, но с умножением на вектор  $B$  столбцов массива  $A$ :

```

A = np.array([ [1., 2., 3.], [4., 5., 6.], [7., 8., 9.] ])
B = np.array([-1.1, -1.2, -1.3])
C = (A.T + B).T
D = (A.T - B).T
E = (A.T * B).T
F = (A.T / B).T
G = (A.T ** B).T
print('+\\n', C, '\\n')
print('-\\n', D, '\\n')
print('*\\n', E, '\\n')
print('/\\n', F, '\\n')
print('**\\n', G, '\\n')
Out:
+
[ [-0.1  0.9  1.9]
[ 2.8  3.8  4.8]
[ 5.7  6.7  7.7] ]
-
[ [ 2.1  3.1  4.1]
[ 5.2  6.2  7.2]
[ 8.3  9.3  10.3] ]
*
[ [ -1.1 -2.2 -3.3]
[ -4.8  -6.    -7.2]
[ -9.1 -10.4 -11.7] ]
/

```

```

[ [-0.90909091 -1.81818182 -2.72727273]
[-3.33333333 -4.16666667 -5.          ]
[-5.38461538 -6.15384615 -6.92307692] ]
**
[[1.          0.4665165  0.29865282]
[0.18946457  0.14495593 0.11647119]
[0.07968426  0.06698584 0.05747576] ]

```

Для более сложных функций (например, для тригонометрических, экспоненты, логарифма, преобразования между градусами и радианами, модуля, корня квадратного и.д.) в NumPy есть реализация. Рассмотрим на примере экспоненты и логарифма:

```

A = np.array([ [1., 2., 3.], [4., 5., 6.], [7., 8., 9.] ])
B = np.exp(A)
C = np.log(B)
print('A', A, '\n')
print('B', B, '\n')
print('C', C, '\n')
Out:
A [ [1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.] ]
B [ [2.71828183e+00 7.38905610e+00 2.00855369e+01]
 [5.45981500e+01 1.48413159e+02 4.03428793e+02]
 [1.09663316e+03 2.98095799e+03 8.10308393e+03] ]
C [ [1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.] ]

```

С полным списком математических операций в NumPy можно ознакомиться [тут](#).

[<наверх>](#)

## Матричное умножение

Описанная выше операция произведения массивов выполняется поэлементно. А при необходимости выполнения операций по правилам линейной алгебры над массивами как над тензорами

можно воспользоваться методом `dot(A, B)`. В зависимости от вида операндов, функция выполнит:

- если аргументы скаляры (числа), то выполнится умножение;
- если аргументы вектор (одномерный массив) и скаляр, то выполнится умножение массива на число;
- если аргументы вектора, то выполнится скалярное умножение (сумма поэлементных произведений);
- если аргументы тензор (многомерный массив) и скаляр, то выполнится умножение вектора на число;
- если аргументы тензора, то выполнится произведение тензоров по последней оси первого аргумента и предпоследней – второго;
- если аргументы матрицы, то выполнится произведение матриц (это частный случай произведения тензоров);
- если аргументы матрица и вектор, то выполнится произведение матрицы и вектора (это тоже частный случай произведения тензоров).

Для выполнения операций должны совпадать соответствующие размеры: для векторов длины, для тензоров – длины вдоль осей, по которым будет происходить суммирование поэлементных произведений.

Рассмотрим примеры со скалярами и векторами:

```
# скаляры
A = 2
B = 3
print(np.dot(A, B), '\n')
# вектор и скаляр
A = np.array([2., 3., 4.])
B = 3
print(np.dot(A, B), '\n')
# вектора
A = np.array([2., 3., 4.])
B = np.array([-2., 1., -1.])
print(np.dot(A, B), '\n')
# тензор и скаляр
```

```
A = np.array([ [2., 3., 4.], [5., 6., 7.] ])
B = 2
print(np.dot(A, B), '\n')
Out:
6
[ 6.  9. 12.]
-5.0
[ [ 4.  6.  8.]
 [10. 12. 14.] ]
```

С тензорами посмотрим только на то, как меняется размер геометрия результирующего массива:

```
# матрица (тензор 2) и вектор (тензор 1)
A = np.ones((5, 6))
B = np.ones(6)
print('A:', A.shape, '\nB:', B.shape, '\nresult:', np.dot(A,
B).shape, '\n\n')
# матрицы (тензора 2)
A = np.ones((5, 6))
B = np.ones((6, 7))
print('A:', A.shape, '\nB:', B.shape, '\nresult:', np.dot(A,
B).shape, '\n\n')
# многомерные тензоры
A = np.ones((5, 6, 7, 8))
B = np.ones((1, 2, 3, 8, 4))
print('A:', A.shape, '\nB:', B.shape, '\nresult:', np.dot(A,
B).shape, '\n\n')
Out:
A: (5, 6)
B: (6,)
result: (5,)
A: (5, 6)
B: (6, 7)
result: (5, 7)
A: (5, 6, 7, 8)
B: (1, 2, 3, 8, 4)
result: (5, 6, 7, 1, 2, 3, 4)
```

Для выполнения произведения тензоров с использованием других осей, вместо определенных для dot можно воспользоваться

`tensordot` с явным указанием осей:

```
A = np.ones((1, 3, 7, 4))
B = np.ones((5, 7, 6, 7, 8))
print('A:', A.shape, '\nB:', B.shape, '\nresult:',
np.tensordot(A, B, [2, 1]).shape, '\n\n')
Out:
A: (1, 3, 7, 4)
B: (5, 7, 6, 7, 8)
result: (1, 3, 4, 5, 6, 7, 8)
```

Мы явно указали, используем третью ось первого массива и вторую – второго (размеры по этим осям должны совпадать).

[<наверх>](#)

## Агрегаторы

Агрегаторы – это методы NumPy позволяющие заменять данные интегральными характеристиками вдоль некоторых осей. Например, можно посчитать среднее значение, максимальное, минимальное, вариацию или еще какую-то характеристику вдоль какой-либо оси или осей и сформировать из этих данных новый массив. Форма нового массива будет содержать все оси исходного массива, кроме тех, вдоль которых подсчитывался агрегатор.

Для примера, сформируем массив со случайными значениями. Затем найдем минимальное, максимальное и среднее значение в его столбцах:

```
A = np.random.rand(4, 5)
print('A\n', A, '\n')
print('min\n', np.min(A, 0), '\n')
print('max\n', np.max(A, 0), '\n')
print('mean\n', np.mean(A, 0), '\n')
print('average\n', np.average(A, 0), '\n')
Out:
A
```

```
[ [0.58481838 0.32381665 0.53849901 0.32401355 0.05442121]
[0.34301843 0.38620863 0.52689694 0.93233065 0.73474868]
[0.09888225 0.03710514 0.17910721 0.05245685 0.00962319]
[0.74758173 0.73529492 0.58517879 0.11785686 0.81204847] ]
min
[0.09888225 0.03710514 0.17910721 0.05245685 0.00962319]
max
[0.74758173 0.73529492 0.58517879 0.93233065 0.81204847]
mean
[0.4435752 0.37060634 0.45742049 0.35666448 0.40271039]
average
[0.4435752 0.37060634 0.45742049 0.35666448 0.40271039]
```

При таком использовании [mean](#) и [average](#) выглядят синонимами. Но эти функции обладают разным набором дополнительных параметров. У них разные возможности по маскированию и взвешиванию усредняемых данных.

Можно подсчитать интегральные характеристики и по нескольким осям:

```
A = np.ones((10, 4, 5))
print('sum\n', np.sum(A, (0, 2)), '\n')
print('min\n', np.min(A, (0, 2)), '\n')
print('max\n', np.max(A, (0, 2)), '\n')
print('mean\n', np.mean(A, (0, 2)), '\n')
Out:
sum
[50. 50. 50. 50.]
min
[1. 1. 1. 1.]
max
[1. 1. 1. 1.]
mean
[1. 1. 1. 1.]
```

В этом примере рассмотрена еще одна интегральная характеристика `sum` – сумма.

Список агрегаторов выглядит примерно так:

- сумма: `sum` и `nansum` – вариант корректно обходящийся с `nan`;
- произведение: `prod` и `nanprod`;
- среднее и матожидание: `average` и `mean` (`nanmean`), `nanaverage` нету;
- медиана: `median` и `nanmedian`;
- перцентиль: `percentile` и `nanpercentile`;
- вариация: `var` и `nanvar`;
- стандартное отклонение (квадратный корень из вариации): `std` и `nanstd`;
- минимальное значение: `min` и `nanmin`;
- максимальное значение: `max` и `nanmax`;
- индекс элемента, имеющего минимальное значение: `argmin` и `nanargmin`;
- индекс элемента, имеющего максимальное значение: `argmax` и `nanargmax`.

В случае использования `argmin` и `argmax` (соответственно, и `nanargmin`, и `nanargmax`) необходимо указывать одну ось, вдоль которой будет считаться характеристика.

Если не указать оси, то по умолчанию все рассматриваемые характеристики считаются **по всему** массиву. В этом случае `argmin` и `argmax` тоже корректно отработают и найдут индекс максимального или минимального элемента так, как буд-то все данные в массиве вытянуты вдоль одной оси командой `ravel()`.

Еще следует отметить, агрегирующие методы определены не только как методы модуля NumPy, но и для самих массивов: запись `np.aggregator(A, axes)` эквивалентна записи `A.aggregator(axes)`, где под `aggregator` подразумевается одна из рассмотренных выше функций, а под `axes` – индексы осей.

```
A = np.ones((10, 4, 5))
print('sum\n', A.sum((0, 2)), '\n')
print('min\n', A.min((0, 2)), '\n')
print('max\n', A.max((0, 2)), '\n')
print('mean\n', A.mean((0, 2)), '\n')
```

```
Out:  
sum  
[50. 50. 50. 50.]  
min  
[1. 1. 1. 1.]  
max  
[1. 1. 1. 1.]  
mean  
[1. 1. 1. 1.]
```

[<наверх>](#)

## Вместо заключения – пример

Давайте построим алгоритм линейной низкочастотной фильтрации изображения.

Для начала загрузим зашумленное изображение.



Рассмотрим фрагмент изображения, чтобы увидеть шум:



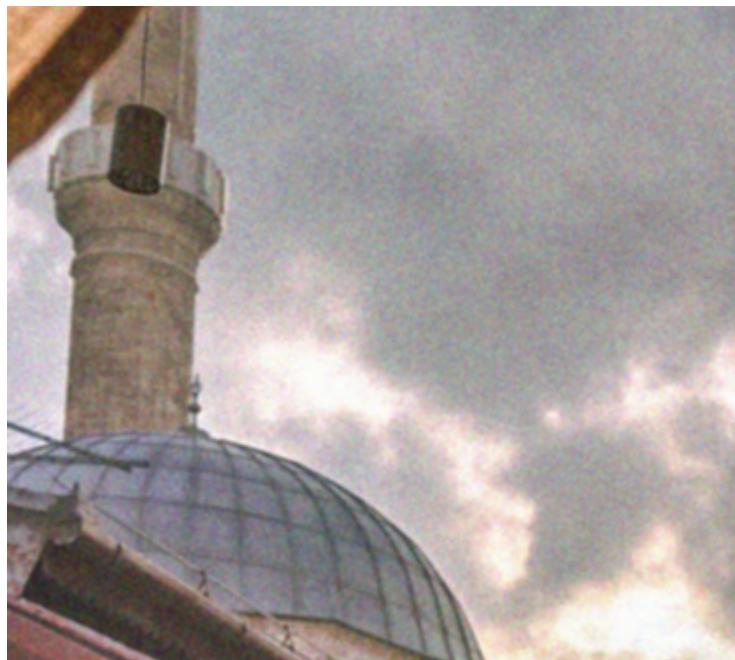
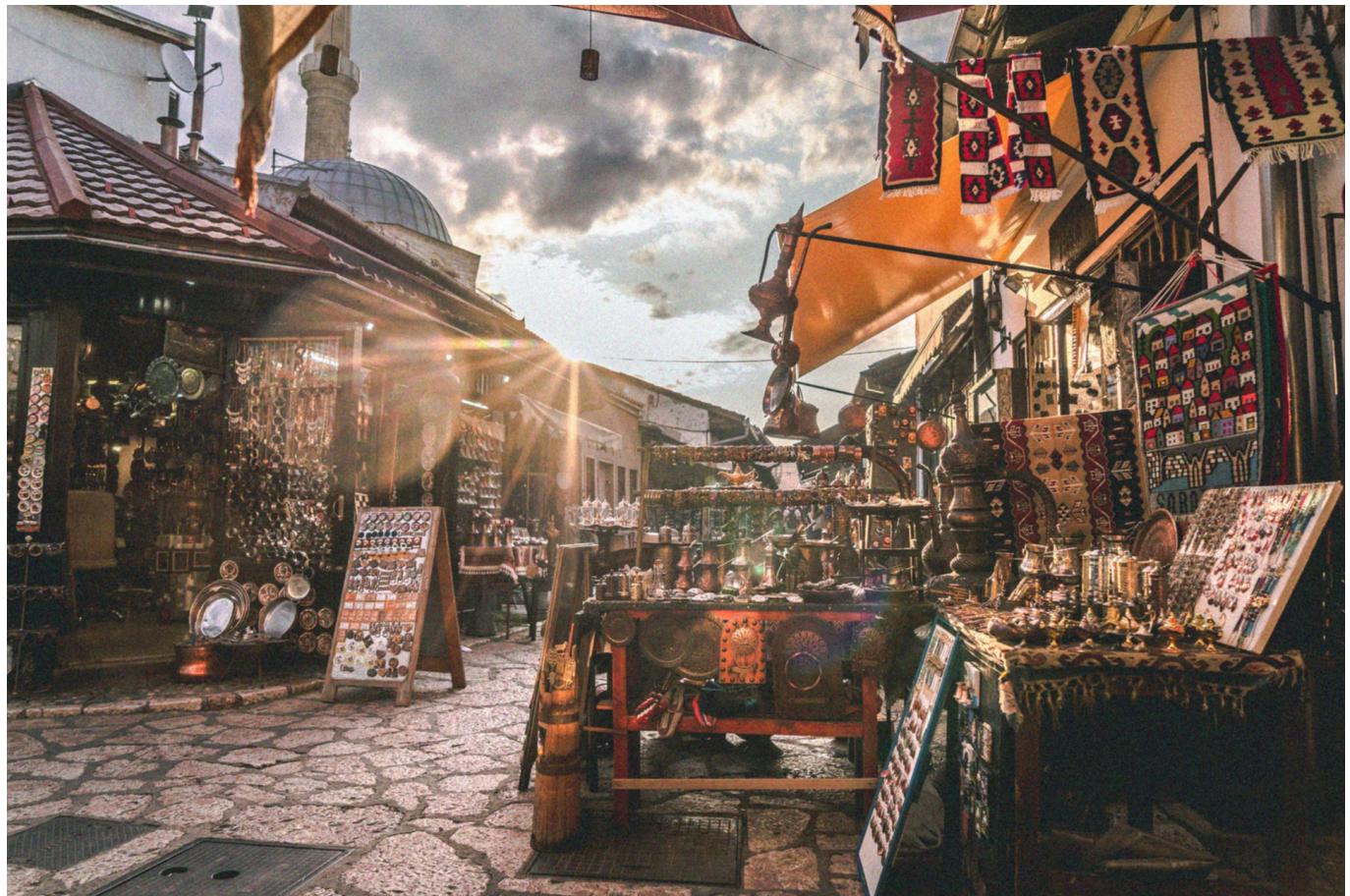
Фильтровать изображение будем с использованием гауссова фильтра. Но вместо выполнения свертки непосредственно (с итерированием), применим взвешенное усреднение срезов изображения, сдвинутых относительно друг друга:

```
def smooth(I):
    J = I.copy()
    J[1:-1] = (J[1:-1] // 2 + J[:-2] // 4 + J[2:] // 4)
    J[:, 1:-1] = (J[:, 1:-1] // 2 + J[:, :-2] // 4 + J[:, 2:] // 4)
    return J
```

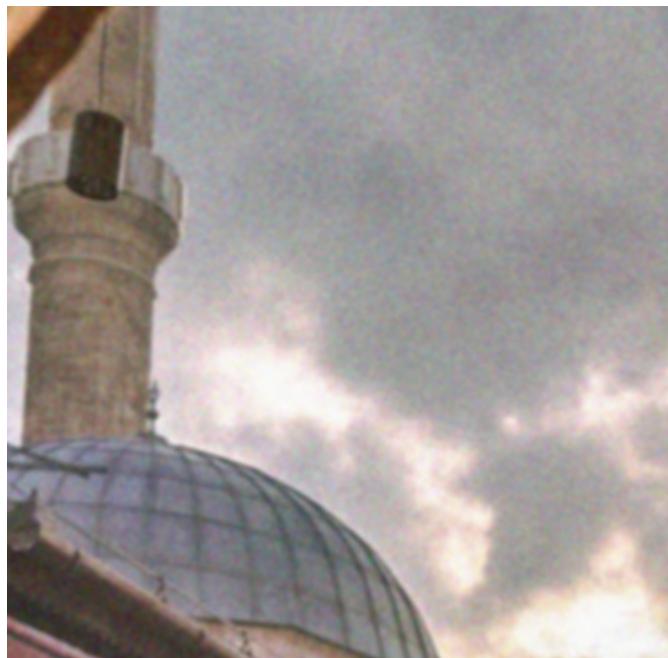
Применим эту функцию к нашему изображению единожды, дважды и трижды:

```
I_noise = cv2.imread('sarajevo_noise.jpg')
I_denoise_1 = smooth(I_noise)
I_denoise_2 = smooth(I_denoise_1)
I_denoise_3 = smooth(I_denoise_2)
cv2.imwrite('sarajevo_denoise_1.jpg', I_denoise_1)
cv2.imwrite('sarajevo_denoise_2.jpg', I_denoise_2)
cv2.imwrite('sarajevo_denoise_3.jpg', I_denoise_3)
```

Получаем следующие результаты:



при однократном применении фильтра;



при двукратном;



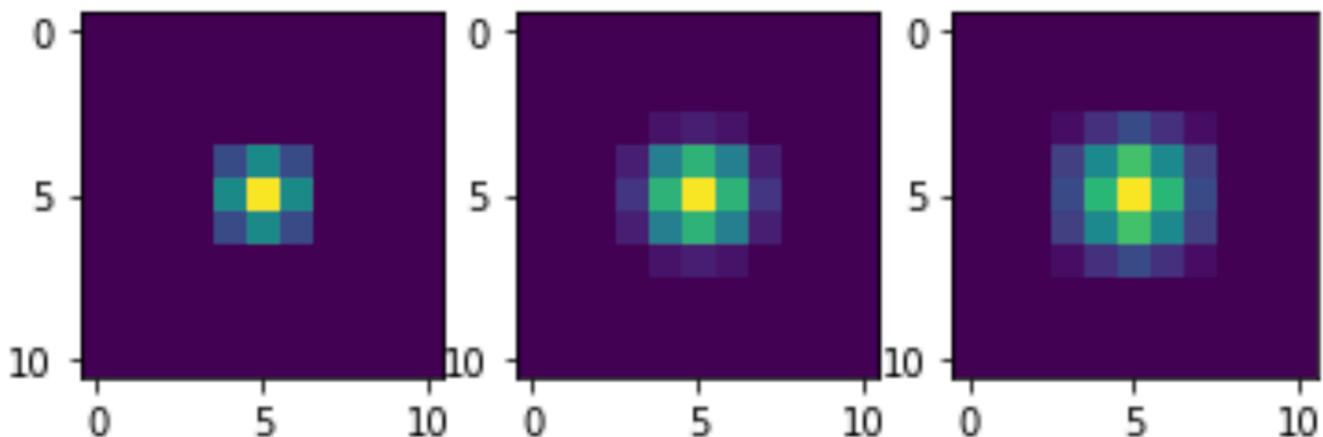
при трехкратном.

Видно, что с повышением количества проходов фильтра снижается уровень шума. Но при этом снижается и четкость изображения. Это известная проблема линейных фильтров. Но наш метод денойзинга изображения не претендует на оптимальность: это лишь демонстрация возможностей NumPy реализации свертки без

итераций.

Теперь давайте посмотрим, сверткам с какими ядрами эквивалентна наша фильтрация. Для этого подвергнем аналогичным преобразованиям одиночный единичный импульс и визуализируем. На самом деле импульс будет не единичным, а равным по амплитуде 255, так как само смещивание оптимизировано под целочисленные данные. Но это не мешает оценить общий вид ядер:

```
M = np.zeros((11, 11))
M[5, 5] = 255
M1 = smooth(M)
M2 = smooth(M1)
M3 = smooth(M2)
plt.subplot(1, 3, 1)
plt.imshow(M1)
plt.subplot(1, 3, 2)
plt.imshow(M2)
plt.subplot(1, 3, 3)
plt.imshow(M3)
plt.show()
```



Мы рассмотрели далеко не полный набор возможностей NumPy, надеюсь, этого было достаточно для демонстрации всей мощи и красоты этого инструмента!

Скрипты этой статьи, посвященные работе с картинками расположены на [Github](#)

Основано на материалах [Habr](#)