

Лекция 1: Библиотека NumPy

Автор: Сергей Вячеславович Макрушин e-mail: SVMakrushin@fa.ru

Финансовый университет, 2020 г.

При подготовке лекции использованы материалы:

- J.R. Johansson (jrjohansson at gmail.com) IPython notebook доступен на: <http://github.com/jrjohansson/scientific-python-lectures>.
- Bryan Van de Ven презентация: Introduction to NumPy
- Уэс Маккинли Python и анализ данных / Пер. с англ. Слипкин А.А. - М.: ДМК Пресс, 2015


V 0.8 03.09.2020

```
In [1]: # загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v1.css")
HTML(html.read().decode('utf-8'))
```

Out[1]:

Оглавление

Введение

 Стек технологий Python для обработки данных и научных расчетов **Стек технологий Python для обработки данных и научных расчетов**

Def: __NumPy__ (от Numerical Python) - библиотека (пакет) для Python, интегрированная с кодом на С и Fortran, решающая задачи математических расчетов и манипулирования массивами данных (в первую очередь - числовых).

В основе NumPy - тип массива **ndarray**:

- быстрый, потребляющий мало памяти, многомерный массив;
- для массива доступен широкий набор высокоэффективных математических и других операций для манипулирования информацией (в первую очередь - числовой).

NumPy - это краеугольный камень технологического стека Python для научных расчетов и обработки данных.

NumPy используется практически во всех вычислительных приложениях, использующих Python. Сочетание реализации векторных функций на С и Fortran и оперирования данных на Python позволяет совместить высокую производительность и гибкость и удобство использования библиотеки. В этом смысле NumPy является ярким примером использования концепции Python as "glue" language.

```
In [2]: # общепринятый способ импорта библиотеки NumPy:
import numpy as np
```

Создание массивов NumPy

Создать массив numpy можно тремя способами:

- из списков или кортежей Python
- с помощью функций, которые предназначены для генерации массивов numpy (например: `arange`, `linspace` и т.д.)
- из данных, хранящихся в файле (будет рассмотрено на семинарах)

 Массив NumPy **Пример массива NumPy**

```
In [8]: # создание ndarray на базе списка Python (не эффективный способ создания ndarray!)
a = np.array([1,2,3,4,5,6,7,8])
a
```

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

```
In [9]: type(a)
```

```
Out[9]: numpy.ndarray
```




```
In [10]: # размер (количество элементов) массива:
a.size
```

```
Out[10]: 8
```

Форма массивов NumPy

 Массив NumPy

Пример одномерного массива NumPy. Форма (shape) массива определяется в виде кортежа из одного элемента.

Массивы numpy могут быть многомерными.  Массив NumPy **Пример одномерного массива NumPy. Форма (shape) массива определяется в виде кортежа из одного элемента.**  Массив NumPy **Пример двумерного массива NumPy. Shape в виде кортежа из двух элементов.**  Массив NumPy **Многомерный массив. Количество измерений не ограничено и соответствует количеству элементов в кортеже shape.**

```
In [11]: # форма массива a:
a.shape
```

```
Out[11]: (8,)
```

В отличие от списков в Python массивы в NumPy строго "прямоугольные". Т.е. количество элементов по каждой из размерностей во всех частях массива должно строго совпадать.

```
In [12]: # Пример "не прямоугольного" вложенного списка:
l_non_rect = [[1, 2, 3], [1, 2], [1, 2, 3, 4]]
```

```
In [13]: # размерность по "внешнему измерению"
len(l_non_rect)
```

```
Out[13]: 3
```

```
In [16]: # размерность массива по вложенному измерению не совпадает (он "не прямоугольный"):  
[len(l) for l in l_non_rect]
```

```
Out[16]: [3, 2, 4]
```

```
In [17]: l_non_rect[2][2]
```

```
Out[17]: 3
```

```
In [19]: # ndarray из l_non_rect создается, но не является двухмерным массивом, как мы того ожидаем  
a_l_non_rect = np.array(l_non_rect)  
a_l_non_rect
```

```
Out[19]: array([list([1, 2, 3]), list([1, 2]), list([1, 2, 3, 4])], dtype=object)
```

```
In [20]: a_l_non_rect.shape
```

```
Out[20]: (3,)
```

```
In [24]: a_l_non_rect.size
```

```
Out[24]: 3
```

```
In [21]: # построение обычного двухмерного массива ndarray:  
b = np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])  
b
```

```
Out[21]: array([[ 0,  1,  2,  3,  4],  
               [ 5,  6,  7,  8,  9],  
               [10, 11, 12, 13, 14]])
```

```
In [25]: b.size
```

```
Out[25]: 15
```

```
In [26]: b.shape
```

```
Out[26]: (3, 5)
```

```
In [27]: # тип объектов ndarray:  
type(a), type(b)
```

```
Out[27]: (numpy.ndarray, numpy.ndarray)
```

```
In [28]: a.itemsize # размер элемента в байтах
```

```
Out[28]: 4
```

```
In [29]: a.nbytes # размер массива в байтах
```

Out[29]: 32

In [30]: `a.ndim` # количество измерений

Out[30]: 1

In [31]: `len(b.shape)`

Out[31]: 2

Типизация массивов NumPy

In [32]: `# определение типа элементов массива numpy:`
`a.dtype`

Out[32]: `dtype('int32')`

Основные числовые типы dtype:

| Data type | Description |
|-------------------------|--|
| <code>bool_</code> | Boolean (True or False) stored as a byte |
| <code>int_</code> | Default integer type (same as C <code>long</code> ; normally either <code>int64</code> or <code>int32</code>) |
| <code>intc</code> | Identical to C <code>int</code> (normally <code>int32</code> or <code>int64</code>) |
| <code>intp</code> | Integer used for indexing (same as C <code>ssize_t</code> ; normally either <code>int32</code> or <code>int64</code>) |
| <code>int8</code> | Byte (-128 to 127) |
| <code>int16</code> | Integer (-32768 to 32767) |
| <code>int32</code> | Integer (-2147483648 to 2147483647) |
| <code>int64</code> | Integer (-9223372036854775808 to 9223372036854775807) |
| <code>uint8</code> | Unsigned integer (0 to 255) |
| <code>uint16</code> | Unsigned integer (0 to 65535) |
| <code>uint32</code> | Unsigned integer (0 to 4294967295) |
| <code>uint64</code> | Unsigned integer (0 to 18446744073709551615) |
| <code>float_</code> | Shorthand for <code>float64</code> . |
| <code>float16</code> | Half precision float: sign bit, 5 bits exponent, 10 bits mantissa |
| <code>float32</code> | Single precision float: sign bit, 8 bits exponent, 23 bits mantissa |
| <code>float64</code> | Double precision float: sign bit, 11 bits exponent, 52 bits mantissa |
| <code>complex_</code> | Shorthand for <code>complex128</code> . |
| <code>complex64</code> | Complex number, represented by two 32-bit floats (real and imaginary components) |
| <code>complex128</code> | Complex number, represented by two 64-bit floats (real and imaginary components) |

Кроме `intc` имеются платформо-зависимые числовые типы: `short`, `long`, `float` и их беззнаковые версии.

Типы dtype доступны с помощью объявления в пространстве имен `numpy`, например: `np.bool_`, `np.float32` и т.д.

```

In [29]: b.dtype
Out[29]: dtype('int32')

In [33]: # При создании массива можно явно объявить тип его элементов, иначе numpy выполнит автоматическое определение типа
d1 = np.array([[1, 2], [3, 4]], dtype=np.float)
d1, d1.dtype
Out[33]: (array([[1., 2.],
                [3., 4.]]),
          dtype('float64'))

In [34]: # автоматическое определение типа выбирает самый простой тип,
# достаточный для хранения всех представленных при объявлении значений:
d2 = np.array([[1, 2], [3, 4]])
d2, d2.dtype
Out[34]: (array([[1, 2],
                [3, 4]]),
          dtype('int32'))

In [35]: d2_dt = d2.dtype

In [36]: d2_dt.itemsize # размер (в байтах) элемента этого типа
Out[36]: 4

In [37]: d2_dt.type, d2_dt.name
Out[37]: (numpy.int32, 'int32')

```

Итог: чем отличаются массивы numpy от списков (и вложенных списков) Python

Массивы numpy:

- **статически типизированы:** тип объектов массива определяется во время объявления массива и не может меняться
- **однородны:** все элементы массива имеют одинаковый тип
- **статичны:** размер массива неизменен, массивы должны быть "прямоугольными", но проекции массива по осям могут меняться

За счет этих свойств массивы numpy:

- **+ эффективно хранятся в памяти**
- **+ операции над массивами numpy могут быть реализованы на компилируемых языках (C, Fortran).** Это **на порядок повышает скорость выполнения операций.** Для массивов numpy в виде высокоэффективных функций реализованы основные математические операции.
- **- не обладают гибкостью списков Python**
- **- прежде всего ориентированы на работу с числовой информацией (т.е. имеют ограничения по типам используемой информации)**

Устройство массивов NumPy и изменение формы



Устройство массива numpy Устройство массива numpy

In [42]:

```
a
```

Out[42]:

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

In [43]:

```
a.size
```

Out[43]:

```
8
```

In [45]:

```
a.shape
```

Out[45]:

```
(8,)
```

In [43]:

```
# функция reshape создает новое представление массива с другой размерностью и теми же дан  
a2 = a.reshape((2, 4))  
a2
```

Out[43]:

```
array([[10, 2, 3, 4],  
       [ 5, 6, 7, 8]])
```

In [44]:

```
a2.size
```

Out[44]:

```
8
```

In [45]:

```
a2.shape
```

Out[45]:

```
(2, 4)
```

Функция `reshape` не копирует массив, а создает новый заголовок, работающий с теми же данными.

Идеология NumPy предполагает, что массив не копируется везде, где это явно не определено. Эта логика продиктована тем, что библиотека предназначена для обработки больших объемов данных. Неявное копирование этих данных при выполнении операций приведет к снижению производительности и возникновению проблем с доступной оперативной памятью.

In [46]:

```
a3 = a.reshape((4, 2))  
a3
```

Out[46]:

```
array([[10, 2],  
       [ 3, 4],  
       [ 5, 6],  
       [ 7, 8]])
```

In [47]:

```
a[0] = 10  
a
```

Out[47]:

```
array([10, 2, 3, 4, 5, 6, 7, 8])
```

In [48]:

```
a2
```

Out[48]:

```
array([[10, 2, 3, 4],  
       [ 5, 6, 7, 8]])
```

```
In [49]: a3
```

```
Out[49]: array([[10,  2],
               [ 3,  4],
               [ 5,  6],
               [ 7,  8]])
```

```
In [50]: a4c = a3.copy() # явно определенное копирование массива
a4c
```

```
Out[50]: array([[10,  2],
               [ 3,  4],
               [ 5,  6],
               [ 7,  8]])
```

```
In [51]: a4c[0, 0] = 100
a4c
```

```
Out[51]: array([[100,  2],
               [  3,  4],
               [  5,  6],
               [  7,  8]])
```

```
In [52]: a3 # изменения в копии не приводят к изменениям в оригинале
```

```
Out[52]: array([[10,  2],
               [ 3,  4],
               [ 5,  6],
               [ 7,  8]])
```

Создание массивов с помощью функций для генерации массивов

```
In [54]: list(range(10))
```

```
Out[54]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [55]: ar1 = np.arange(10) # аргументы: [start], stop, [step], dtype=None
ar1
```

```
Out[55]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [58]: # Функция arange, аналог встроенной функции range
ar1 = np.arange(0, 10, 1) # аргументы: [start], stop, [step], dtype=None
ar1
```

```
Out[58]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [56]: ar2 = np.arange(-1, 1, 0.1, dtype=np.float64)
ar2, ar2.dtype
```

```
Out[56]: (array([-1.00000000e+00, -9.00000000e-01, -8.00000000e-01, -7.00000000e-01,
               -6.00000000e-01, -5.00000000e-01, -4.00000000e-01, -3.00000000e-01,
               -2.00000000e-01, -1.00000000e-01, -2.22044605e-16,  1.00000000e-01,
               2.00000000e-01,  3.00000000e-01,  4.00000000e-01,  5.00000000e-01,
               6.00000000e-01,  7.00000000e-01,  8.00000000e-01,  9.00000000e-01]),
          dtype('float64'))
```

```
In [57]: # linspace - последовательность значений из заданного интервала с постоянным шагом
# np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
np.linspace(0, 10, 25)
```

```
Out[57]: array([ 0.         ,  0.41666667,  0.83333333,  1.25         ,  1.66666667,
        2.08333333,  2.5         ,  2.91666667,  3.33333333,  3.75         ,
        4.16666667,  4.58333333,  5.         ,  5.41666667,  5.83333333,
        6.25         ,  6.66666667,  7.08333333,  7.5         ,  7.91666667,
        8.33333333,  8.75         ,  9.16666667,  9.58333333, 10.         ])
```

```
In [59]: # geomspace - геометрическая последовательность значений из заданного интервала
# np.geomspace(start, stop, num=50, endpoint=True, dtype=None)
np.geomspace(1, 256, num=9)
```

```
Out[59]: array([ 1.,  2.,  4.,  8., 16., 32., 64., 128., 256.] )
```

```
In [66]: # в модуле np.random находятся функции для работы со случайными значениями
# равномерно распределенные случайные числа из диапазона [0,1]:
np.random.rand(5, 5) # аргументы - размерность получаемого массива
```

```
Out[66]: array([[0.15770997, 0.86162913, 0.50710942, 0.50636251, 0.07034829],
        [0.27554711, 0.56442661, 0.16217884, 0.49915686, 0.01048966],
        [0.81446749, 0.20009838, 0.20309595, 0.73554491, 0.42966668 ],
        [0.29705968, 0.37303607, 0.59897688, 0.4920559 , 0.29289069],
        [0.38007565, 0.85987236, 0.79461751, 0.1791864 , 0.25232482]])
```

```
In [67]: # диагональная матрица с заданными в аргументе значениями на диагонали
np.diag([1,2,3])
```

```
Out[67]: array([[1, 0, 0],
        [0, 2, 0],
        [0, 0, 3]])
```

```
In [68]: # матрица из нулей
np.zeros((3, 3))
```

```
Out[68]: array([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

```
In [69]: np.zeros((3, 3), dtype=np.int)
```

```
Out[69]: array([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

Работа с массивами

Индексация

```
In [60]: a
```

```
Out[60]: array([10,  2,  3,  4,  5,  6,  7,  8])
```



```
In [61]: a[1]
```

```
Out[61]: 2
```

```
In [62]: a[1] = 20
a
```

```
Out[62]: array([10, 20,  3,  4,  5,  6,  7,  8])
```

```
In [63]: a[-1]
```

```
Out[63]: 8
```

```
In [64]: b
```

```
Out[64]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])
```

```
In [65]: # индексация элементов многомерного массива numpy производится иначе, нежели для вложенных списков
b[1, 2] # размерность индекса должна совпадать с размерностью массива
```

```
Out[65]: 7
```

```
In [66]: # если при индексации не хватает измерений, то считается, что для последних (по порядку) измерений
# и по этим измерениям выбираются все значения:
b[1]
```

```
Out[66]: array([5, 6, 7, 8, 9])
```

```
In [67]: # работает как последовательная индексация по одному индексу:
b[1][2]
```

```
Out[67]: 7
```

```
In [68]: b[1, 2]
```

```
Out[68]: 7
```

```
In [69]: b[1, 2] = 70
b
```

```
Out[69]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6, 70,  8,  9],
               [10, 11, 12, 13, 14]])
```

```
In [70]: b[1, 2] = 7
b
```

```
Out[70]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])
```

```
In [71]: i = (2, 3)
         b[i] # аргумент индексации - кортеж

Out[71]: 13

In [72]: # если количество переданных индексов меньше размерности массива, то будет возвращена соо:
         # (считается, что опущены индексы для последних осей):
         b[1]

Out[72]: array([5, 6, 7, 8, 9])

In [73]: # на основе этого механизма работает индексация в стиле многомерных списков Python:
         b[1][2]

Out[73]: 7
```

Срезы

NumPy поддерживает работу со срезами, аналогичными срезам для списков Python.

```
In [87]: b

Out[87]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])

In [88]: b[1, :]

Out[88]: array([5, 6, 7, 8, 9])

In [89]: b[1]

Out[89]: array([5, 6, 7, 8, 9])

In [90]: # срез без определенных границ позволяет получать проекцию по любым осям:
         b[:, 1]

Out[90]: array([ 1,  6, 11])
```

 Устройство массива numpy **Пример выполнения среза**

```
In [82]: b[0:2, :]

Out[82]: array([[0, 1, 2, 3, 4],
               [5, 6, 7, 8, 9]])

In [83]: b[2, 1:]

Out[83]: array([11, 12, 13, 14])
```

 Устройство массива numpy **Пример выполнения среза**

```
In [84]: b[:, 2:4]
```

```
Out[84]: array([[2, 3],
              [7, 8]])
```



Устройство массива numpy **Пример выполнения среза**

```
In [85]: b[:, ::2]
```

```
Out[85]: array([[ 0,  2,  4],
              [ 5,  7,  9],
              [10, 12, 14]])
```



Устройство массива numpy **Пример выполнения среза**

```
In [92]: b_s2 = b[:, ::2, ::3]
b_s2
```

```
Out[92]: array([[ 0,  3],
              [10, 13]])
```

При получении среза массива создается объект-представление (array view), который работает с данными исходного массива (идеология numpy - избегание копирования данных), определяя для него специальный порядок обхода элементов.

```
In [93]: # определение, содержит ли объект данные или является представлением
b.flags.owndata, b_s2.flags.owndata
```

```
Out[93]: (True, False)
```

```
In [94]: b_s2[0, 0]
```

```
Out[94]: 0
```

```
In [95]: b[0, 0] = 10
```

```
In [96]: b_s2[0, 0]
```

```
Out[96]: 10
```

```
In [97]: b_s2[0, 0] = 100
```

```
In [98]: b[0, 0]
```

```
Out[98]: 100
```

Срезам массивов можно присваивать новые значения

```
In [99]: b2 = b.copy()
b2
```

```
Out[99]: array([[100,  1,  2,  3,  4],
              [  5,  6,  7,  8,  9],
              [ 10, 11, 12, 13, 14]])
```

```
In [100...
```

```
b2[:,2, ::3]
```

```
Out[100...] array([[100,    3],  
       [ 10,   13]])
```

```
In [101...] b2[:,2, ::3] = [[-1, -2], [-4, -5]] # присвоение срезу многомерной структуры совпадающей с  
b2
```

```
Out[101...] array([[ -1,    1,    2,  -2,    4],  
       [  5,    6,    7,    8,    9],  
       [- 4,   11,   12,  -5,   14]])
```

```
In [102...] b2[2, 1:]
```

```
Out[102...] array([11, 12, -5, 14])
```

```
In [103...] b2[2, 1:] = 110 # присвоение срезу скалярного значения за счет распространения (broadcasting)  
b2
```

```
Out[103...] array([[ -1,    1,    2,  -2,    4],  
       [  5,    6,    7,    8,    9],  
       [- 4,  110,  110,  110,  110]])
```

Работа с функциями NumPy

Универсальные функции

Def: __Универсальные функции__ (ufuncs) - функции, выполняющие поэлементные операции над данными, хранящимися в массиве. Это векторные операции на базе простых функций, работающих с одним или несколькими скалярными значениями и возвращающими скаляр.

Основные универсальные функции:

- операции сравнения: <, <=, ==, !=, >=, >
- арифметические операции: +, -, *, /, %, reciprocal, square
- экспоненциальные функции: exp, expm1, exp2, log, log10, log1p, log2, power, sqrt
- тригонометрические функции: sin, cos, tan, asin, arccos, atan
- гиперболические функции: sinh, cosh, tanh, asinh, arccosh, atctanh
- побитовые операции: &, |, ~, ^, left_shift, right_shift
- логические операции: and, logical_xor, not, or
- предикаты: isfinite, isinf, isnan, signbit
- другие функции: abs, ceil, floor, mod, modf, round, sinc, sign, trunc

```
In [104...] b
```

```
Out[104...] array([[100,    1,    2,    3,    4],  
       [  5,    6,    7,    8,    9],  
       [ 10,   11,   12,   13,   14]])
```

```
In [105...] b < 7
```

```
Out[105...] array([[False,  True,  True,  True,  True],  
       [ True,  True, False, False, False],  
       [False, False, False, False, False]])
```

```
In [106...] (3 < b) & (b < 7)
```

```
Out[106...] array([[False, False, False, False,  True],
       [ True,  True, False, False, False],
       [False, False, False, False, False]])
```

```
In [107...] b + 10
```

```
Out[107...] array([[110, 11, 12, 13, 14],
       [ 15, 16, 17, 18, 19],
       [ 20, 21, 22, 23, 24]])
```

```
In [108...] b * 10
```

```
Out[108...] array([[1000, 10, 20, 30, 40],
       [ 50, 60, 70, 80, 90],
       [ 100, 110, 120, 130, 140]])
```

```
In [109...] b + b
```

```
Out[109...] array([[200, 2, 4, 6, 8],
       [ 10, 12, 14, 16, 18],
       [ 20, 22, 24, 26, 28]])
```

```
In [110...] b * b # поэлементное умножение!
```

```
Out[110...] array([[10000, 1, 4, 9, 16],
       [ 25, 36, 49, 64, 81],
       [ 100, 121, 144, 169, 196]])
```

```
In [111...] np.exp(b)
```

```
Out[111...] array([[2.68811714e+43, 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],
       [2.20264658e+04, 5.98741417e+04, 1.62754791e+05, 4.42413392e+05,
       1.20260428e+06]])
```



Выполнение универсальной функции **Выполнение универсальной функции**

```
In [112...] a0 = np.arange(5)
a0
```

```
Out[112...] array([0, 1, 2, 3, 4])
```

```
In [113...] b0 = np.arange(0, 50, 10)
b0
```

```
Out[113...] array([ 0, 10, 20, 30, 40])
```

```
In [114...] c0 = a0 + b0
c0
```

```
Out[114...] array([ 0, 11, 22, 33, 44])
```

Оси и векторные функции

Основные типы векторных функций:

- Агрегирующие функции: `sum()`, `mean()`, `argmin()`, `argmax()`, `cumsum()`, `cumprod()`
- Предикаты `a.any()`, `a.all()`
- Манипуляция векторными данными: `argsort()`, `a.transpose()`, `trace()`, `reshape(...)`, `ravel()`, `fill(...)`, `clip(...)`

Обход элементов массива при незаданной оси __Обход элементов массива при незаданной оси__

```
In [115... ar1 = np.arange(15).reshape(3, 5)
ar1
```

```
Out[115... array([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]])
```

```
In [116... ar1.shape
```

```
Out[116... (3, 5)
```

```
In [117... ar1.sum()
```

```
Out[117... 105
```

```
In [118... ar1.sum(axis=None)
```

```
Out[118... 105
```

Обход элементов массива __Обход элементов массива по axis=0__

```
In [119... ar1.sum(axis=0)
```

```
Out[119... array([15, 18, 21, 24, 27])
```

```
In [120... ar1.sum(axis=0).shape
```

```
Out[120... (5,)
```

Обход элементов массива __Обход элементов массива по axis=1__

```
In [121... ar1.sum(axis=1)
```

```
Out[121... array([10, 35, 60])
```

Основные функции, которым может передаваться ось:

- `all([axis, out, keepdims])` Returns True if all elements evaluate to True.
- `all([axis, out, keepdims])` Returns True if all elements evaluate to True.
- `any([axis, out, keepdims])` Returns True if any of the elements of a evaluate to True.
- `argmax([axis, out])` Return indices of the maximum values along the given axis.
- `argmin([axis, out])` Return indices of the minimum values along the given axis of a.

- `argpartition(kth[, axis, kind, order])` Returns the indices that would partition this array.
- `argsort([axis, kind, order])` Returns the indices that would sort this array.
- `compress(condition[, axis, out])` Return selected slices of this array along given axis.
- `cumprod([axis, dtype, out])` Return the cumulative product of the elements along the given axis.
- `cumsum([axis, dtype, out])` Return the cumulative sum of the elements along the given axis.
- `diagonal([offset, axis1, axis2])` Return specified diagonals.
- `max([axis, out, keepdims])` Return the maximum along a given axis.
- `mean([axis, dtype, out, keepdims])` Returns the average of the array elements along given axis.
- `min([axis, out, keepdims])` Return the minimum along a given axis.
- `partition(kth[, axis, kind, order])` Rearranges the elements in the array in such a way that the value of the element in `kth * position` is in the position it would be in a sorted array.
- `prod([axis, dtype, out, keepdims])` Return the product of the array elements over the given axis
- `ptp([axis, out, keepdims])` Peak to peak (maximum - minimum) value along a given axis.
- `repeat(repeats[, axis])` Repeat elements of an array.
- `sort([axis, kind, order])` Sort an array, in-place.
- `squeeze([axis])` Remove single-dimensional entries from the shape of a.
- `std([axis, dtype, out, ddof, keepdims])` Returns the standard deviation of the array elements along given axis.
- `sum([axis, dtype, out, keepdims])` Return the sum of the array elements over the given axis.
- `swapaxes(axis1, axis2)` Return a view of the array with `axis1` and `axis2` interchanged.
- `take(indices[, axis, out, mode])` Return an array formed from the elements of a at the given indices.
- `trace([offset, axis1, axis2, dtype, out])` Return the sum along diagonals of the array.
- `var([axis, dtype, out, ddof, keepdims])` Returns the variance of the array elements, along given axis.

Распространение (broadcasting)

В качестве аргументов универсальных функций могут быть массивы с различной, но сравнимой формой. В этом случае применяется механизм распространения (broadcasting).


 Обход элементов массива В примере скаляр распространяется до массива размерности (5,)

In [122...

```
np.arange(5) + 10
```

Out[122...

```
array([10, 11, 12, 13, 14])
```

 Пример распространения Пример распространения для протяженных массивов разной размерности

In [123...

```
a2 = np.arange(6).reshape(3, 2)
a2
```

Out[123...

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

In [124...

```
b2 = np.arange(10, 40, 10).reshape(3,1)
b2, b2.shape
```

Out[124...

```
(array([[10],
       [20],
       [30]]),
 (3, 1))
```

In [125...

```
a2 + b2
```

Out[125...

```
array([[10, 11],
       [22, 23],
       [34, 35]])
```

In [126...

```
a2.shape
```

Out[126...

```
(3, 2)
```

In [127...

```
b3 = np.arange(10, 40, 10)
b3, b3.shape
```

Out[127...

```
(array([10, 20, 30]), (3,))
```

In [128...

```
a2 + b3
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-128-f3eec681a16e> in <module>
----> 1 a2 + b3
```

```
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

In [129...

```
b4 = np.arange(10, 30, 10)
b4, b4.shape
```

Out[129...

```
(array([10, 20]), (2,))
```

In [130...

```
a2 + b4
```

Out[130...

```
array([[10, 21],
       [12, 23],
       [14, 25]])
```

Правила выполнения распространения:

- соответствующие измерения двух массивов должны либо совпадать
- либо одно из них должно быть равно единице.

Если в одном из массивов не хватает измерений, то считается, что недостающее количество измерений - это младшие измерения (измерения с наименьшими номерами), которым приписывается размерность 1.



Пример работы с размерностями массивов Пример работы с размерностями массивов в корректных операциях распространения

In [131...

```
a2, a2.shape
```

Out[131...

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

In [132...

```
b3, b3.shape
```


Out[132... (array([[10, 20, 30]], (3,))

```
In [133... # для добавления измерения (оси) размерностью 1 можно использовать np.newaxis :
b3t = b3[:, np.newaxis]
b3t, b3t.shape
```

Out[133... (array([[10],
 [20],
 [30]]),
 (3, 1))

```
In [134... a2 + b3
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-134-f3eec681a16e> in <module>
----> 1 a2 + b3

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

```
In [135... a2 + b3t
```

Out[135... array([[10, 11],
 [22, 23],
 [34, 35]])

Линейная алгебра

Арифметические операции с массивами NumPy выполняются на поэлементной основе.

```
In [136... e = np.array([[ 0,  1,  2,  3,  4],
               [10, 11, 12, 13, 14],
               [20, 21, 22, 23, 24],
               [30, 31, 32, 33, 34],
               [40, 41, 42, 43, 44]])
```

```
In [137... e * 10
```

Out[137... array([[0, 10, 20, 30, 40],
 [100, 110, 120, 130, 140],
 [200, 210, 220, 230, 240],
 [300, 310, 320, 330, 340],
 [400, 410, 420, 430, 440]])

```
In [138... e * e
```

Out[138... array([[0, 1, 4, 9, 16],
 [100, 121, 144, 169, 196],
 [400, 441, 484, 529, 576],
 [900, 961, 1024, 1089, 1156],
 [1600, 1681, 1764, 1849, 1936]])

```
In [139... e / 2
```

Out[139... array([[0. , 0.5, 1. , 1.5, 2.],
 [5. , 5.5, 6. , 6.5, 7.],
 [10. , 10.5, 11. , 11.5, 12.],
 [15. , 15.5, 16. , 16.5, 17.],
 [20. , 20.5, 21. , 21.5, 22.]])

In [140...

```
# матричное умножение:
m1 = np.arange(9).reshape(3, 3)
m2 = np.arange(6).reshape(3, 2)
print(m1)
print(m2)
```

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

In [141...

```
np.dot(m1, m2)
```

Out[141...

```
array([[10, 13],
       [28, 40],
       [46, 67]])
```

In [142...

```
m1 @ m2 # бинарный оператор, аналогичный dot()
```

Out[142...

```
array([[10, 13],
       [28, 40],
       [46, 67]])
```

In [143...

```
m2
```

Out[143...

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

In [144...

```
m2.T # транспонирование
```

Out[144...

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

In [145...

```
m2_1 = m2[:,1]
m2_1, m2_1.shape # одномерный массив
```

Out[145...

```
(array([1, 3, 5]), (3,))
```

In [146...

```
m2_1.T # транспонирование одномерного массива не приводит к созданию вектора столбца!
```

Out[146...

```
array([1, 3, 5])
```

In [147...

```
m2_11 = m2_1[np.newaxis, :] # создаем "матрицу-строку"
print(m2_11, m2_11.shape, '\n')
print(m2_11.T, m2_11.T.shape) # транспонирование работает!
```

```
[[1 3 5]] (1, 3)
```

```
[[1]
 [3]
 [5]] (3, 1)
```

In [148...

```
m2_1[:, np.newaxis] # делаем "матрицу-столбец" напрямую
```

```
Out[148... array([[1],  
          [3],  
          [5]])
```

```
In [149... m1
```

```
Out[149... array([[0, 1, 2],  
          [3, 4, 5],  
          [6, 7, 8]])
```

```
In [150... m2.T
```

```
Out[150... array([[0, 2, 4],  
          [1, 3, 5]])
```

```
In [151... m2.T @ m1
```

```
Out[151... array([[30, 36, 42],  
          [39, 48, 57]])
```

```
In [152... np.arange(10, 40, 10).T
```

```
Out[152... array([10, 20, 30])
```

```
In [153... np.linalg.det(m1) # определитель
```

```
Out[153... 0.0
```

```
In [154... m3 = np.array([[3, 7, 4], [11, 2, 9], [4, 11, 2]])  
m3
```

```
Out[154... array([[ 3,  7,  4],  
          [11,  2,  9],  
          [ 4, 11,  2]])
```

```
In [155... np.linalg.det(m3)
```

```
Out[155... 265.000000000000017
```

```
In [156... m3i = np.linalg.inv(m3) # получение обратной матрицы  
m3i
```

```
Out[156... array([[ -0.35849057,  0.11320755,  0.20754717],  
          [ 0.05283019, -0.03773585,  0.06415094],  
          [ 0.42641509, -0.01886792, -0.26792453]])
```

```
In [157... m3i @ m3
```

```
Out[157... array([[ 1.00000000e+00, -5.27355937e-16, -5.55111512e-17],  
          [ 5.55111512e-17,  1.00000000e+00,  0.00000000e+00],  
          [ 2.22044605e-16,  2.22044605e-16,  1.00000000e+00]])
```

Маскирование и прихотливое индексирование

Def: `_`Прихотливым индексированием `_` (fancy indexing) называется использование массива или списка в качестве индекса.

```
In [158... e = np.array([[ 0,  1,  2,  3,  4],
               [10, 11, 12, 13, 14],
               [20, 21, 22, 23, 24],
               [30, 31, 32, 33, 34],
               [40, 41, 42, 43, 44]])
e
```

```
Out[158... array([[ 0,  1,  2,  3,  4],
        [10, 11, 12, 13, 14],
        [20, 21, 22, 23, 24],
        [30, 31, 32, 33, 34],
        [40, 41, 42, 43, 44]])
```

```
In [161... e[1]
```

```
Out[161... array([10, 11, 12, 13, 14])
```

```
In [162... row_indices = [3, 2, 1]
e[row_indices]
```

```
Out[162... array([[30, 31, 32, 33, 34],
        [20, 21, 22, 23, 24],
        [10, 11, 12, 13, 14]])
```

```
In [163... col_indices = [1, 2, -1]
e[row_indices, col_indices]
```

```
Out[163... array([31, 22, 14])
```

```
In [164... ind = np.arange(4)
e[ind, ind + 1]
```

```
Out[164... array([ 1, 12, 23, 34])
```

Для индексирования мы можем использовать маски (маскирование): если массив NumPy содержит элементы типа `bool`, то элемент выбирается в зависимости от булевого значения.

```
In [165... f = np.arange(5)
fb = np.array([True, False, True, False, False])
f, fb
```

```
Out[165... (array([0, 1, 2, 3, 4]), array([ True, False,  True, False, False]))
```

```
In [166... f[fb]
```

```
Out[166... array([0, 2])
```

```
In [167... f % 2
```

```
Out[167... array([0, 1, 0, 1, 0], dtype=int32)
```

```
In [168... f % 2 == 0
```

```
Out[168... array([ True, False,  True, False,  True])
```

```
In [169... f[f % 2 == 0]
```

```
Out[169... array([0, 2, 4])
```

```
In [170... f[f % 2 == 0].sum() # сумма всех четных чисел в массиве
```

```
Out[170... 6
```

Дополнительные операции с массивами

```
In [171... b
```

```
Out[171... array([[100,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]])
```

```
In [172... b.flatten() # операция создает копию массива!
```

```
Out[172... array([100,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,
        13, 14])
```

Используя функции `repeat`, `tile`, `vstack`, `hstack`, `concatenate`, можно создать большой массив из массивов меньших размеров.

```
In [173... a5 = np.array([[1, 2], [3, 4]])
a5
```

```
Out[173... array([[1, 2],
        [3, 4]])
```

```
In [174... np.repeat(a5, 3)
```

```
Out[174... array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
```

```
In [175... np.tile(a5, (3, 2))
```

```
Out[175... array([[1, 2, 1, 2],
        [3, 4, 3, 4],
        [1, 2, 1, 2],
        [3, 4, 3, 4],
        [1, 2, 1, 2],
        [3, 4, 3, 4]])
```

```
In [176... b5 = np.array([[5, 6]])
b5
```

```
Out[176... array([[5, 6]])
```

```
In [177... np.concatenate((a5, b5), axis=0)
```

```
Out[177... array([[1, 2],
        [3, 4],
        [5, 6]])
```

```
In [178... np.concatenate((a5, b5.T), axis=1)
```

```
Out[178... array([[1, 2, 5],  
        [3, 4, 6]])
```

```
In [179... np.vstack((a5, b5, b5))
```

```
Out[179... array([[1, 2],  
        [3, 4],  
        [5, 6],  
        [5, 6]])
```

```
In [180... np.hstack((a5, b5.T))
```

```
Out[180... array([[1, 2, 5],  
        [3, 4, 6]])
```