Лекция 1: Библиотека Numpy

Автор: Сергей Вячеславович Макрушин, 2022 г.

e-mail: s-makrushin@yandex.ru (mailto:s-makrushin@yandex.ru)

При подготовке лекции использованы материалы:

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

V 0.9 25.08.2022

In [90]:

```
# загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v2.css")
HTML(html.read().decode('utf-8'))
```

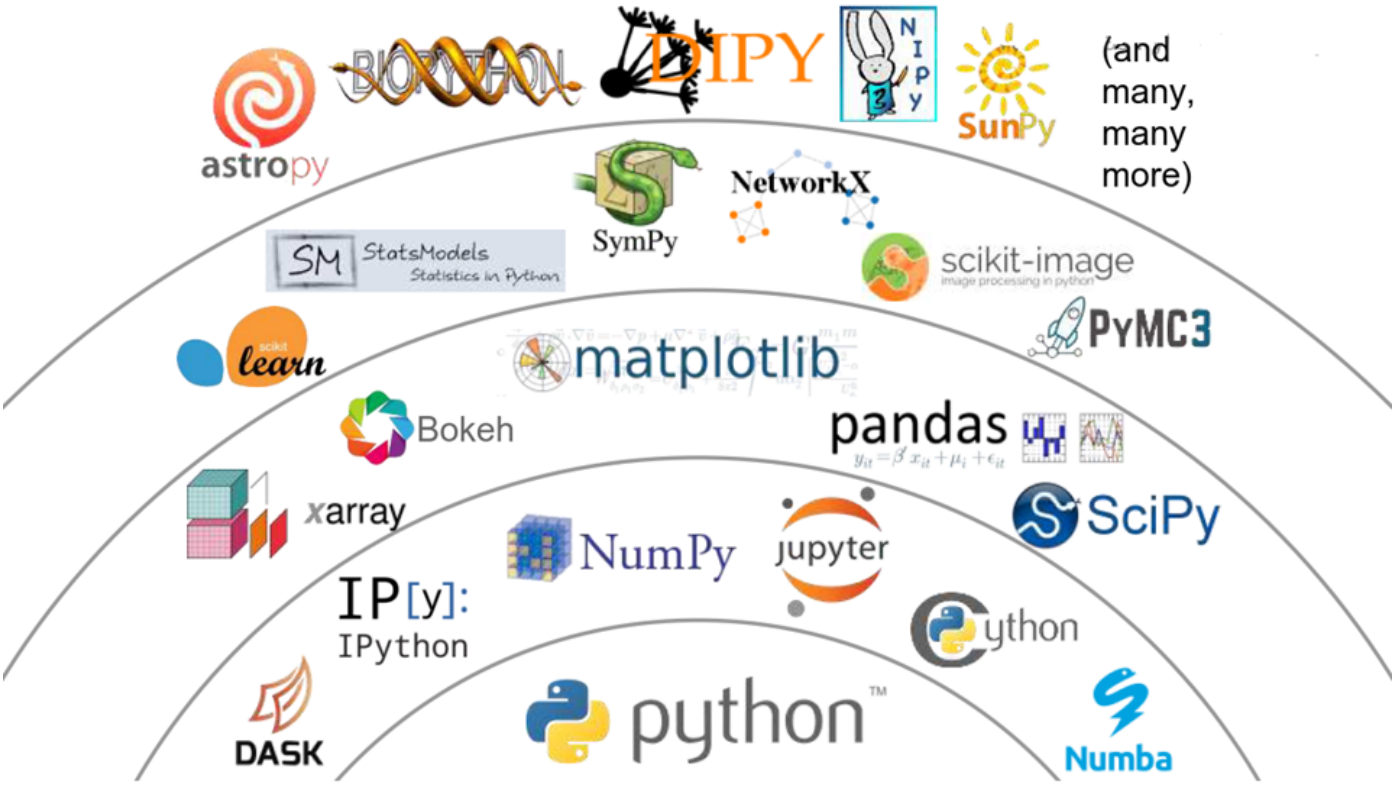
Out[90]:

Оглавление ¶

- к оглавлению

Знакомство с NumPy

- к оглавлению



Стек технологий Python для обработки данных и научных расчетов

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

NumPy - это краеугольный камень технологического стека Python для научных расчетов и обработки данных. NumPy - открытая библиотека, поставляемая с базовым дистрибутивом Python.

NumPy используется практически во всех вычислительных приложениях, использующих Python. Сочетание реализации векторных функций на C 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 a glue language".

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 включает ряд высокоуровневых пакетов ориентированных на определенные задачи работы с данными:

- numpy.linalg - Linear algebra

- numpy.fft - Discrete Fourier Transform
- numpy.matlib - Matrix library
- numpy.random - Random sampling

In [94]:

```
# общепринятый способ импорта библиотеки NumPy:
import numpy as np
```

Устройство ndarray и базовые операции с ним

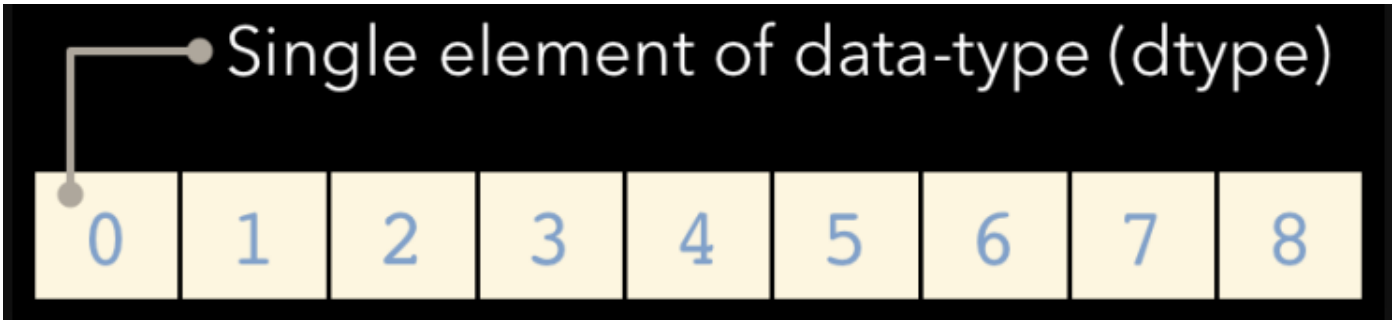
- к оглавлению

Создание и форма массивов ndarray

- к оглавлению

Создать массив numpy можно тремя способами:

- из списков или кортежей Python
- с помощью функций, которые предназначены для генерации массивов numpy
- из данных, хранящихся в файле



Пример массива ndarray

In [191]:

```
# создание ndarray на базе списка Python (не эффективный способ создания ndarray!)
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
a
```

Out[191]:

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

In [96]:

```
type(a)
```

Out[96]:

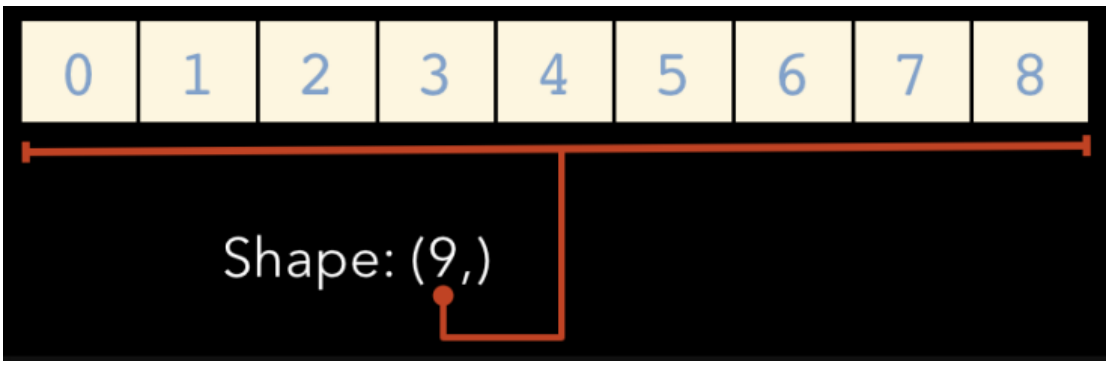
```
numpy.ndarray
```

In [97]:

```
# размер (количество элементов) массива:
a.size
```

Out[97]:

```
9
```



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

In [8]:

```
a
```

Out[8]:

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

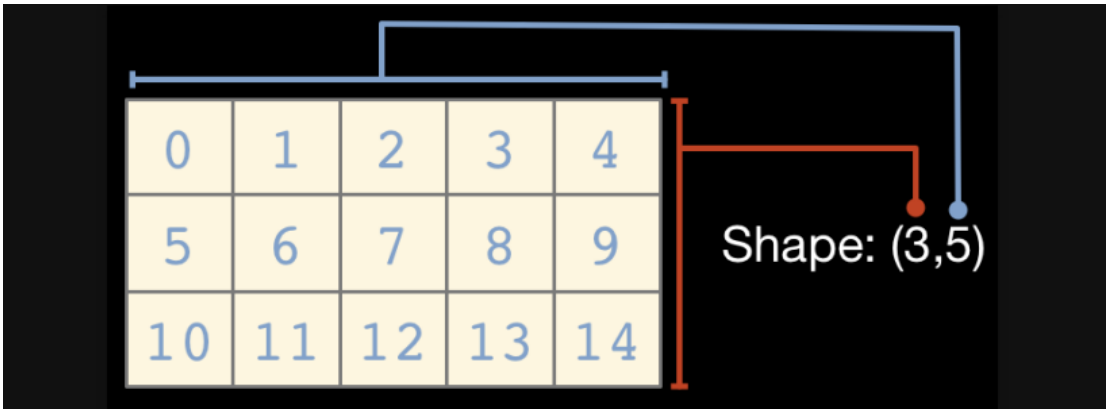
In [98]:

```
# форма массива a:
a.shape
```

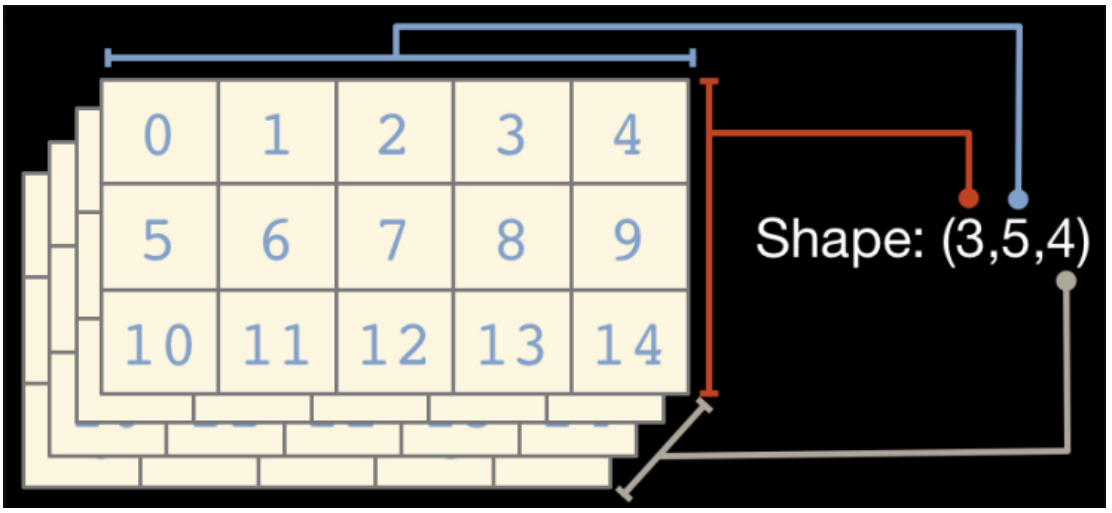
Out[98]:

```
(9,)
```

Массивы numpy могут быть многомерными.



Пример двухмерного массива NumPy. Shape в виде кортежа из двух элементов.



Многомерный массив. Количество измерений не ограничено и соответствует количеству элементов в кортеже shape.

In [179]:

```
# 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[179]:

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

In [100]:

```
# форма массива b:
b.shape
```

Out[100]:

```
(3, 5)
```

In [101]:

```
# количество измерений a:
a.ndim
```

Out[101]:

```
1
```

In [102]:

```
len(a.shape)
```

Out[102]:

```
1
```

In [103]:

```
# количество измерений b:
b.ndim
```

Out[103]:

```
2
```

In [104]:

```
len(b.shape)
```

Out[104]:

```
2
```

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 [19]:

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

In [105]:

```
# размерность по "внешнему измерению"
len(l_non_rect)
```

Out[105]:

```
3
```

In [106]:

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

Out[106]:

```
[3, 2, 4]
```

In [107]:

```
l_non_rect[2][2]
```

Out[107]:

```
3
```

In [108]:

```
# 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[108]:

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

In [109]:

```
a_l_non_rect.shape
```

Out[109]:

```
(3,)
```

In [110]:

```
a_l_non_rect.size
```

Out[110]:

```
3
```

In [111]:

```
# 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[111]:

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

In [112]:

```
b.size
```

Out[112]:

```
15
```

In [113]:

```
b.shape
```

Out[113]:

```
(3, 5)
```

У массивов разной размерности один и тот же тип - ndarray:

In [114]:

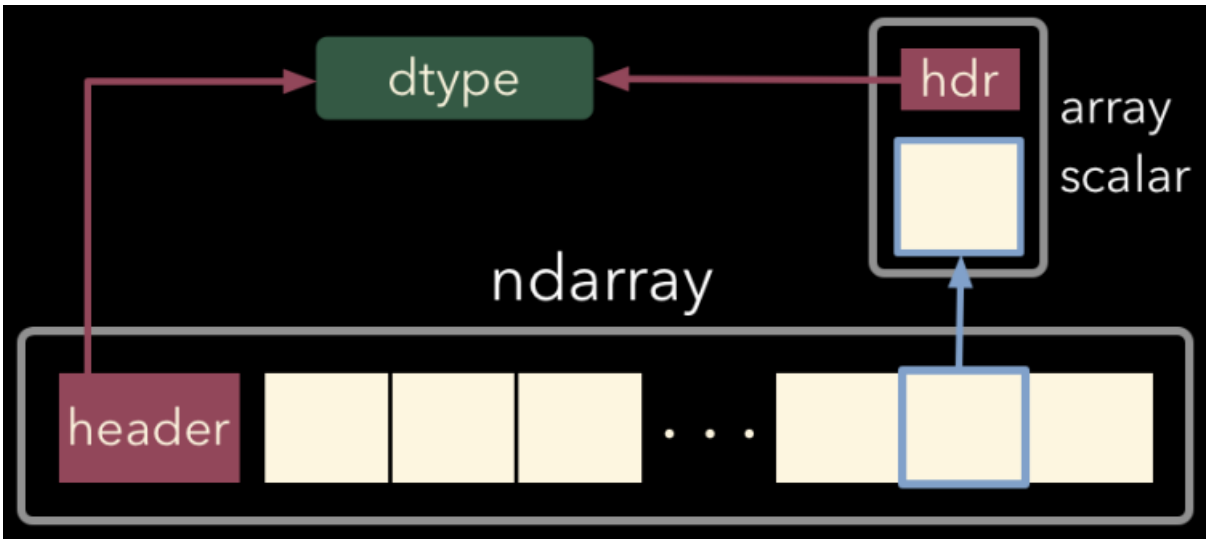
```
# тип объектов ndarray:
type(a), type(b)
```

Out[114]:

```
(numpy.ndarray, numpy.ndarray)
```

Устройство массивов ndarray и изменение формы

- к оглавлению



Устройство массива numpy

In [115]:

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

In [116]:

```
c.size
```

Out[116]:

```
8
```

In [117]:

```
c.shape
```

Out[117]:

```
(8,)
```

In [118]:

```
# функция reshape создает новое представление массива с другой размерностью и теми же данны
c2 = c.reshape((2, 4))
c2
```

Out[118]:

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

In [119]:

```
c2.size
```

Out[119]:

```
8
```

In [120]:

```
c2.shape
```

Out[120]:

```
(2, 4)
```

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

Идеология NumPy предполагает, что **массив не копируется, кроме случаев, где это явно не определено**. Эта логика продиктована тем, что библиотека предназначена для обработки больших объемов данных. Неявное копирование данных (особенно в случае их большого объема) при выполнении операций приведет к снижению производительности и возникновению проблем с доступной оперативной памятью.

In [121]:

```
c3 = c.reshape((4, 2))
c3
```

Out[121]:

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

In [122]:

```
c[0] = 10
c
```

Out[122]:

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

In [123]:

```
c2
```

Out[123]:

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

In [124]:

```
c3
```

Out[124]:

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

Все три массива ndarray используют одну область памяти для хранения значений.

In [125]:

```
c4cpy = c3.copy() # явно определенное копирование массива
c4cpy
```

Out[125]:

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

In [126]:

```
c4cpy[0, 0] = 100
c4cpy
```

Out[126]:

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

In [127]:

```
c3 # изменения в копии не приводят к изменениям в оригинале
```

Out[127]:

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

Типизация массивов ndarray

- к оглавлению

В отличие от списков Python многомерные массивы ndarray строго типизированы.

In [129]:

```
a
```

Out[129]:

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

In [130]:

```
# определение типа элементов массива numpy:
a.dtype
```

Out[130]:

```
dtype('int32')
```

Основные числовые типы dtype:

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long` ; normally either `int64` or `int32`)
intc	Identical to C `int` (normally `int32` or `int64`)
intp	Integer used for indexing (same as C `ssize_t` ; normally either `int32` or `int64`)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64` .

Data type	Description
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128` .
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	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 [131]:

```
b.dtype
```

Out[131]:

```
dtype('int32')
```

In [132]:

```
# При создании массива можно явно объявить тип его элементов, иначе питру выполнит автомати
d1 = np.array([[1, 2], [3, 4]], dtype=np.float)
d1, d1.dtype
```

Out[132]:

```
(array([[1., 2.],
        [3., 4.]]),
 dtype('float64'))
```

In [133]:

```
# 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[133]:

```
(array([[1, 2],
        [3, 4]]),
 dtype('int32'))
```

In [134]:

```
# даже одно значение более сложного типа потребует хранения всего массива с испольованием э
d2 = np.array([[1, 2], [3, 4.0]])
d2, d2.dtype
```

Out[134]:

```
(array([[1., 2.],
        [3., 4.]]),
 dtype('float64'))
```

In [36]:

```
d2_dt = d2.dtype
```

In [135]:

```
# размер (в байтах) элемента этого типа:
d2_dt.itemsize
```

Out[135]:

```
4
```

In [136]:

```
# размер (в байтах) элемента массива:
d2.itemsize
```

Out[136]:

```
8
```

In [137]:

```
# размер массива в байтах:
d2.nbytes
```

Out[137]:

```
32
```

In [139]:

```
d2.size
```

Out[139]:

```
4
```

In [138]:

```
# размер массива в байтах:
d2.itemsize * d2.size
```

Out[138]:

```
32
```

In [30]:

```
d2_dt.type, d2_dt.name
```

Out[30]:

```
(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 массивs numpy:

- **+ эффективно хранятся в памяти** (для хранения значений используется непрерывная область памяти с простой индексацией, как это принято в C или Fortran)
- **+** 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** ("не прямоугольные" вложенные списки, разнотипные элементы в списках)
- **-** прежде всего ориентированы на работу с числовой информацией (т.е. **имеют ограничения по типам используемой информации**)

Создание массивов с помощью функций для генерации массивов

- к оглавлению

In [141]:

```
list(range(10))
```

Out[141]:

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

In [142]:

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

ar1
```

Out[142]:

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

In [145]:

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

ar1
```

Out[145]:

```
array([100, 102, 104, 106, 108])
```

In [48]:

```
ar2 = np.arange(-1, 1, 0.1, dtype=np.float64)
ar2, ar2.dtype
```

Out[48]:

```
(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 [363]:

```
# 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[363]:

```
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 [49]:

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

Out[49]:

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

In [365]:

```
# 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[365]:

```
array([[0.29184894, 0.8769098 , 0.4093358 , 0.62439337, 0.35760498],
       [0.97726654, 0.59505163, 0.38649252, 0.83020082, 0.45807283],
       [0.99220724, 0.16106952, 0.68238018, 0.82392631, 0.73270889],
       [0.4285355 , 0.78899409, 0.60849969, 0.19200771, 0.78950572],
       [0.89999712, 0.67135811, 0.85580464, 0.99606939, 0.00224674]])
```

In [146]:

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

Out[146]:

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

In [147]:

```
# матрица из нулей
np.zeros((3, 3))
```

Out[147]:

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

In [148]:

```
np.zeros((3, 3), dtype=np.int)
```

Out[148]:

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

Полный список функций для создания массивов ndarray:
https://numpy.org/doc/stable/reference/routines.array-creation.html
(https://numpy.org/doc/stable/reference/routines.array-creation.html)

- From shape or value - по определенной формой, с заданным значением (например: zeros)
- From existing data - на основе существующих данных (например: array, copy)
- Creating record arrays - создание массивов записей
- Creating character arrays - создание строковых массивов (устарело, сохраняется для совместимости)
- Numerical ranges - числовые последовательности (например: arange, linspace)
- Building matrices - создание матриц (например: diag)
- The Matrix class - создание специализированны массивов-матриц

Сохранение ndarray в файл и загрузка из файла

- к оглавлению

NumPy предлагает два основных формата для хранения массивов ndarray:

- **npy** - стандартный формат двоичного файла в NumPy для сохранения одного массива NumPy. Формат npy разработан так, чтобы быть максимально простым при достижении ограниченных целей.
- **npz** - простой способ объединить несколько массивов в один файл, который использует zip архив (по умолчанию - не сжатый) для хранения нескольких файлов npy. Для этих архивов рекомендуется использовать расширение ".npz".

В NumPy предлагаются различные функции для работы с указанными бинарными форматами файлов:

- `np.save` - сохраняет единичный ndarray в бинарный файл формата npy.
- `np.savez` - сохраняет **несколько ndarray** в несжатый архив формата npz.
- `np.savez_compressed` - сохраняет несколько ndarray в **сжатый архив формата npz**.
- `np.load` - загружает массивы или объекты, сохраненные с помощью pickle из npy, npz или файлов pickle./

Сохранение и загрузка одного массива в npy:

In [149]:

```
a
```

Out[149]:

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

In [150]:

```
np.save('a_ndarr.npy', a)
```

In [151]:

```
a_ld = np.load('a_ndarr.npy')
a_ld
```

Out[151]:

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

In [152]:

```
a == a_ld
```

Out[152]:

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

Сохранение и загрузка нескольких массивов в npz:

In [153]:

```
a
```

Out[153]:

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

In [154]:

```
b
```

Out[154]:

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

In [155]:

```
np.savez('ab_ndarr.npz', a=a, b=b)
```

In [156]:

```
npzfile = np.load('ab_ndarr.npz')
npzfile.files
```

Out[156]:

```
['a', 'b']
```

In [157]:

```
npzfile['b']
```

Out[157]:

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

In [158]:

```
npzfile['a']
```

Out[158]:

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

Сохранение и загрузка нескольких анонимных массивов в npz:

In [160]:

```
np.savez('xx_ndarr.npz', a, b)
npzfile2 = np.load('xx_ndarr.npz')
npzfile2.files
```

Out[160]:

```
['arr_0', 'arr_1']
```

NumPy поддерживат сохранение и загрузку массивов в текстовом формате с помощью функций:

- `savetxt` - поддерживает различные опции форматирования сохраняемого массива (единственного, имеющего размерность 1 или 2). Документация: https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html (https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html)

- `loadtxt` - поддерживает большое количество вариантов загрузки массива ndarray из текстового файла (в том числе формата CSV). Документация: https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html (https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html)

In [78]:

```
b
```

Out[78]:

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

In [161]:

```
np.savetxt('b_s1.txt', b, delimiter=',')
```

In [162]:

```
with open("b_s1.txt", "r") as file:
    for line in file:
        print(line)
```

```
0.0000000000000000e+00,1.0000000000000000e+00,2.0000000000000000e+00,3.
0000000000000000e+00,4.0000000000000000e+00

5.0000000000000000e+00,6.0000000000000000e+00,7.0000000000000000e+00,8.
0000000000000000e+00,9.0000000000000000e+00

1.0000000000000000e+01,1.1000000000000000e+01,1.2000000000000000e+01,1.
3000000000000000e+01,1.4000000000000000e+01
```

In [163]:

```
np.loadtxt('b_s1.txt', delimiter=',')
```

Out[163]:

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

In [164]:

```
np.loadtxt('b_s1.txt', delimiter=',', dtype=np.int)
```

Out[164]:

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

In [165]:

```
np.savetxt('b_s2.txt', b, fmt='%1.4e')
```

In [166]:

```
with open("b_s2.txt", "r") as file:
    for line in file:
        print(line)
```

```
0.0000e+00 1.0000e+00 2.0000e+00 3.0000e+00 4.0000e+00

5.0000e+00 6.0000e+00 7.0000e+00 8.0000e+00 9.0000e+00

1.0000e+01 1.1000e+01 1.2000e+01 1.3000e+01 1.4000e+01
```

In [167]:

```
np.loadtxt('b_s2.txt')
```

Out[167]:

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

Для чтения данных из CSV файлов так же можно использовать функции `numpy.genfromtxt` или использовать `pandas.read_csv` .

Обращение к массивам ndarray

- к оглавлению

Индексация

- к оглавлению

In [192]:

```
a
```

Out[192]:

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

In [193]:

```
a[0]
```

Out[193]:

```
0
```

In [194]:

```
a[1]
```

Out[194]:

```
1
```

In [195]:

```
a[1] = 20
a
```

Out[195]:

```
array([ 0, 20,  2,  3,  4,  5,  6,  7,  8])
```

In [196]:

```
a[-1]
```

Out[196]:

```
8
```

In [197]:

```
a.shape
```

Out[197]:

```
(9,)
```

In [198]:

```
a[8]
```

Out[198]:

```
8
```

In [199]:

```
a[9]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-199-044f1ccc0778> in <module>
----> 1 a[9]

IndexError: index 9 is out of bounds for axis 0 with size 9
```

In [200]:

```
a[-8]
```

Out[200]:

```
20
```

In [201]:

```
a[-9]
```

Out[201]:

```
0
```

In [202]:

```
a[-10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-202-b1ddb357c14a> in <module>
----> 1 a[-10]

IndexError: index -10 is out of bounds for axis 0 with size 9
```

In [203]:

```
b
```

Out[203]:

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

In [204]:

```
# индексация элементов многомерного массива питру производится иначе, нежели для вложенных
b[1, 2] # размерность индекса должна совпадать с размерностью массива
```

Out[204]:

```
7
```

In [205]:

```
b.shape
```

Out[205]:

```
(3, 5)
```

In [207]:

```
# если количество переданных индексов меньше размерности массива,
# то считается, что для последних (по порядку) измерений индекс
# и будет возвращена соответствующая проекция массива:
b[1]
```

Out[207]:

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

In [208]:

```
# на основе этого механизма работает индексация в стиле многомерных списков Python
# она функционирует как последовательная индексация по одному индексу:
b[1][2]
```

Out[208]:

```
7
```

In [209]:

```
b[1, 2]
```

Out[209]:

```
7
```

In [210]:

```
b[1, 2] = 70
b
```

Out[210]:

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

In [211]:

```
b[1, 2] = 7
b
```

Out[211]:

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

In [212]:

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

Out[212]:

```
13
```

Срезы

- к оглавлению

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

In [213]:

```
b
```

Out[213]:

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

In [214]:

```
b.shape
```

Out[214]:

```
(3, 5)
```

In [215]:

```
b[1, :]
```

Out[215]:

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

In [216]:

```
b[1]
```

Out[216]:

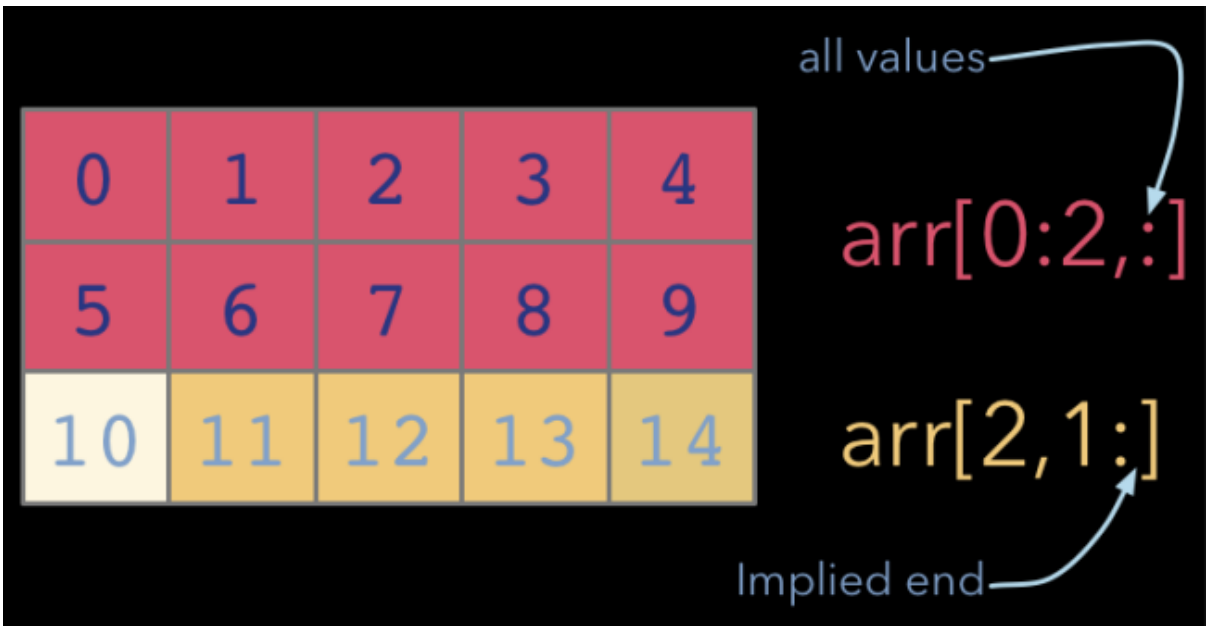
```
array([5, 6, 7, 8, 9])
```

In [217]:

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

Out[217]:

```
array([ 1,  6, 11])
```



Пример выполнения среза

In [218]:

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

Out[218]:

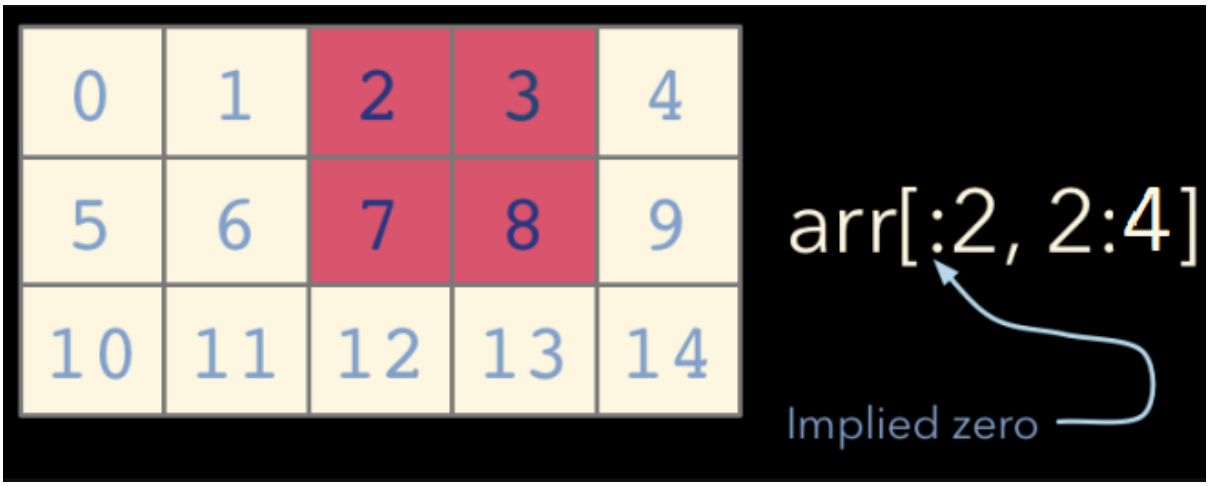
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

In [219]:

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

Out[219]:

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



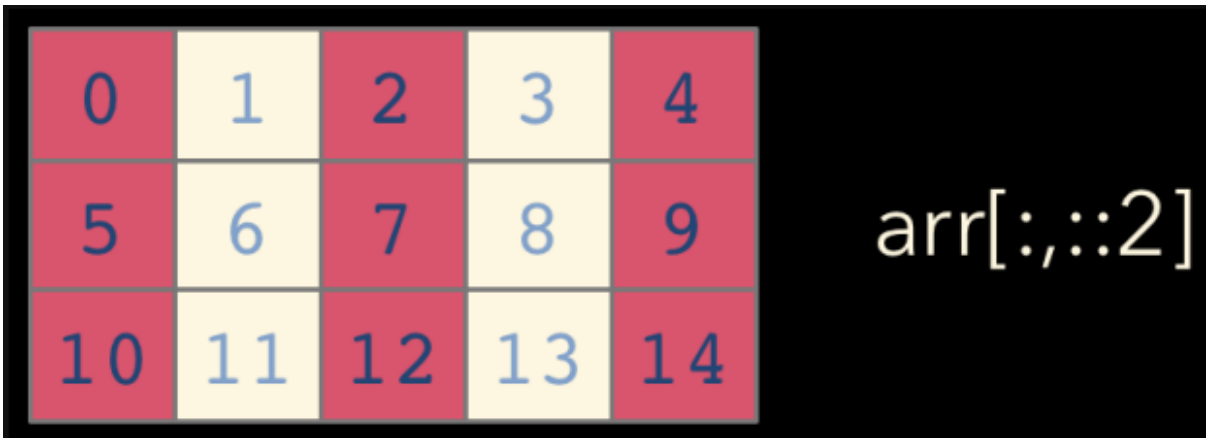
Пример выполнения среза

In [220]:

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

Out[220]:

```
array([[2, 3],
       [7, 8]])
```



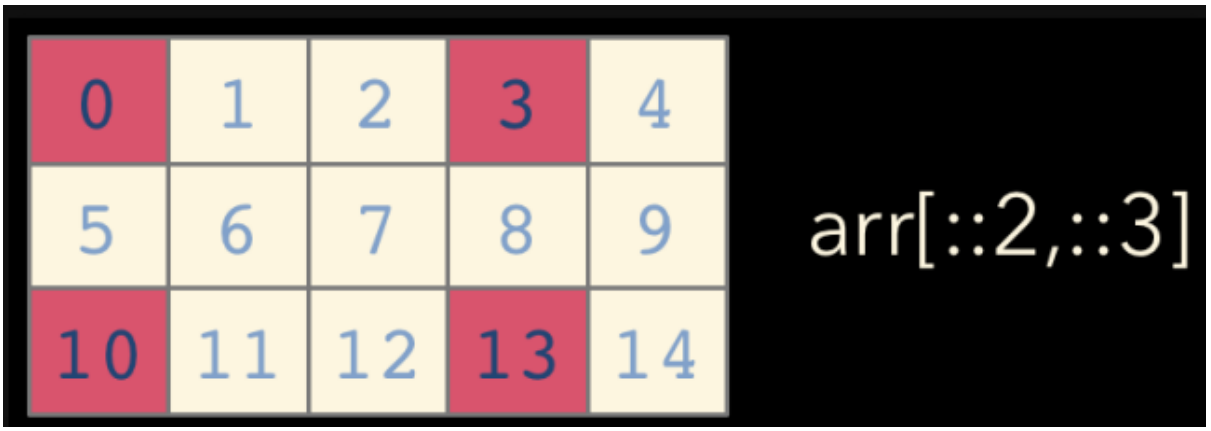
Пример выполнения среза

In [221]:

```
b[:, ::2]
```

Out[221]:

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



Пример выполнения среза

In [222]:

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

Out[222]:

```
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 [223]:

```
# определение, содержит ли объект данные или является представлением
b.flags.owndata, b_s2.flags.owndata
```

Out[223]:

```
(True, False)
```

In [224]:

```
b_s2[0, 0]
```

Out[224]:

```
0
```

In [225]:

```
b[0, 0] = 10
```

In [226]:

```
b_s2[0, 0]
```

Out[226]:

```
10
```

In [227]:

```
b_s2[0, 0] = 100
```

In [228]:

```
b[0, 0]
```

Out[228]:

```
100
```

In [229]:

```
b
```

Out[229]:

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

Срезам массивов можно присваивать новые значения

In [230]:

```
b2 = b.copy()
b2
```

Out[230]:

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

In [231]:

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

Out[231]:

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

In [232]:

```
b2[::2, ::3] = [[-1, -2], [-4, -5]] # присвоение срезу многомерной структуры совпадающей ра
b2
```

Out[232]:

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

In [233]:

```
b2[2, 1:]
```

Out[233]:

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

In [234]:

```
b2[2, 1:] = 110 # присвоение срезу скалярного значения за счет распространения (broadcastin
b2
```

Out[234]:

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

Работа с функциями NumPy

- к оглавлению

Универсальные функции

- к оглавлению

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, acsin, arccos, atctan
- гиперболические функции: sinh, cosh, tanh, acsinh, 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 [235]:

```
b
```

Out[235]:

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

In [236]:

```
b < 7
```

Out[236]:

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

In [237]:

```
(3 < b) & (b < 7)
```

Out[237]:

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

In [238]:

```
b + 10
```

Out[238]:

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

In [239]:

```
b * 10
```

Out[239]:

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

In [240]:

```
b + b
```

Out[240]:

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

In [241]:

```
b * b # поэлементное умножение!
```

Out[241]:

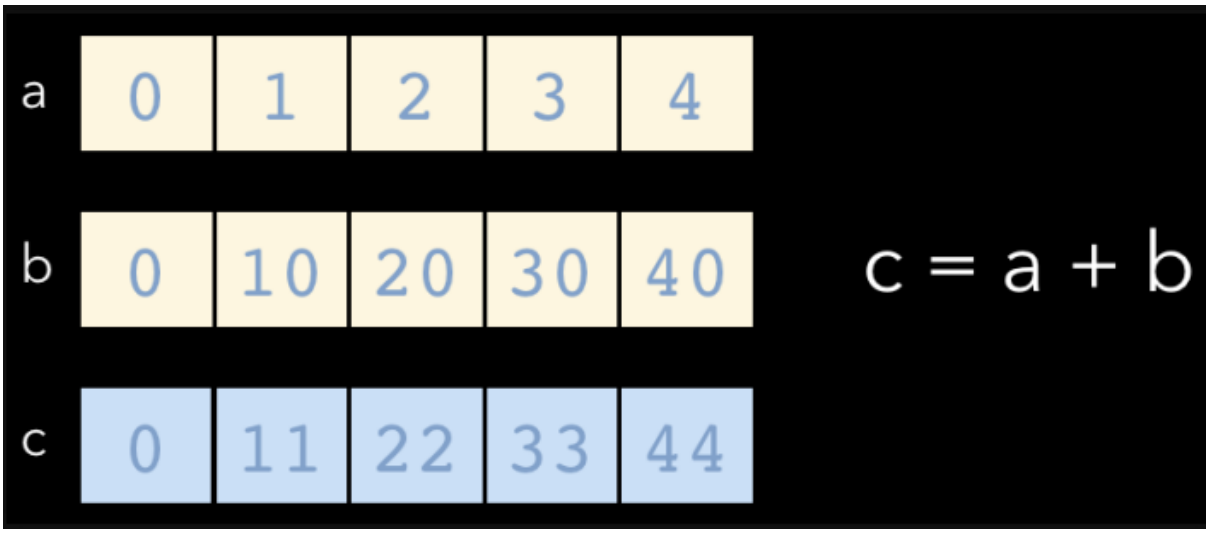
```
array([[10000,     1,     4,     9,    16],
       [   25,    36,    49,    64,    81],
       [  100,   121,   144,   169,   196]])
```

In [242]:

```
np.exp(b)
```

Out[242]:

```
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

In [243]:

```
a0 = np.arange(5)
a0
```

Out[243]:

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

In [244]:

```
b0 = np.arange(0, 50, 10)
b0
```

Out[244]:

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

In [245]:

```
c0 = a0 + b0
c0
```

Out[245]:

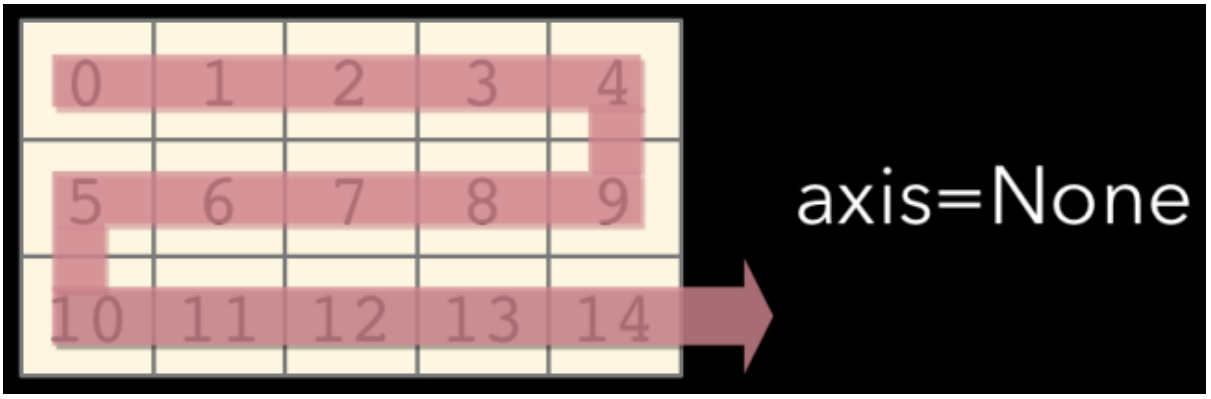
```
array([ 0, 11, 22, 33, 44])
```

Оси и агрегирующие функции

- к оглавлению

Основные типы векторных функций:

- Агрегирующие функциии: 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

In [246]:

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

Out[246]:

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

In [247]:

```
ar1.shape
```

Out[247]:

```
(3, 5)
```

In [248]:

```
ar1.sum()
```

Out[248]:

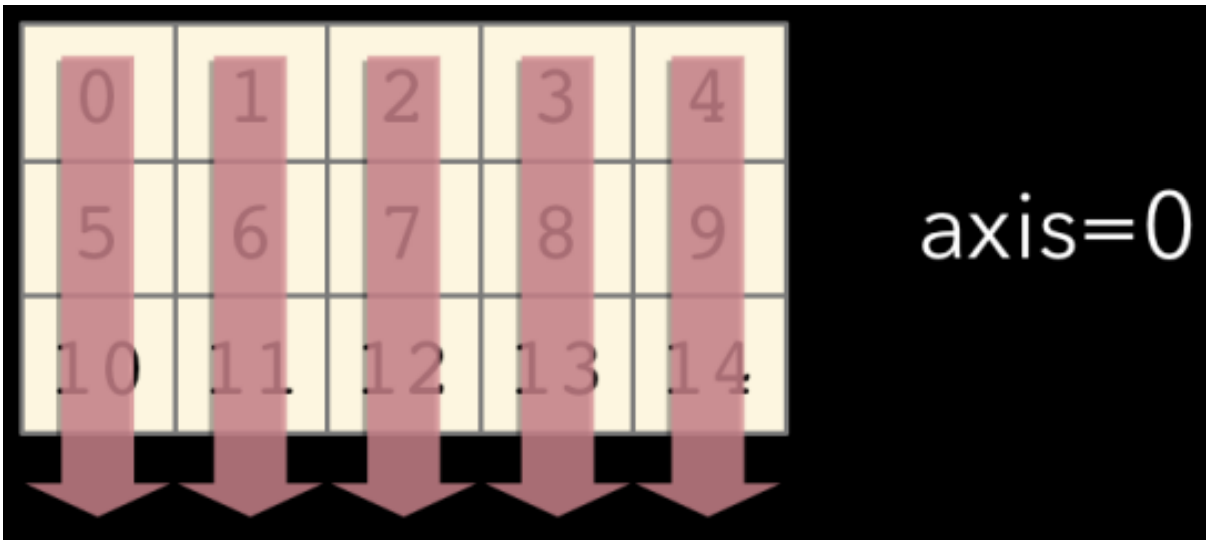
```
105
```

In [249]:

```
ar1.sum(axis=None)
```

Out[249]:

```
105
```



Обход элементов массива по axis=0

In [412]:

```
ar1.shape
```

Out[412]:

```
(3, 5)
```

In [250]:

```
ar1.sum(axis=0)
```

Out[250]:

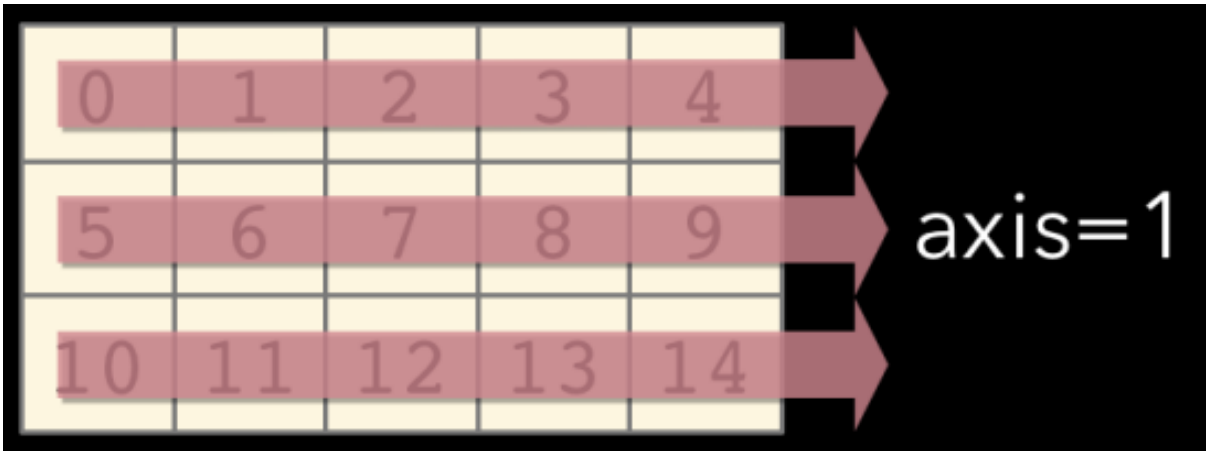
```
array([15, 18, 21, 24, 27])
```

In [413]:

```
ar1.sum(axis=0).shape
```

Out[413]:

```
(5,)
```



Обход элементов массива по axis=0

In [251]:

```
ar1.sum(axis=1)
```

Out[251]:

```
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.

Линейная алгебра в Numpy

- к оглавлению

Арифметические операции с массивами NumPy выполняются на поэлементной основе.

In [252]:

```
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 [253]:

```
e * 10
```

Out[253]:

```
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 [183]:

```
e * e
```

Out[183]:

```
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 [254]:

```
e / 2
```

Out[254]:

```
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 [255]:

```
# 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 [256]:

```
m3 = np.dot(m1, m2)
m3
```

Out[256]:

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

In [257]:

```
m1.shape, m2.shape, m3.shape
```

Out[257]:

```
((3, 3), (3, 2), (3, 2))
```

In [258]:

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

Out[258]:

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

In [259]:

```
m2
```

Out[259]:

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

In [260]:

```
m2.T # транспонирование
```

Out[260]:

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

In [261]:

```
m2_1 = m2[:,1]
m2_1, m2_1.shape # одномерный массив, а не столбец!
```

Out[261]:

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

In [262]:

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

Out[262]:

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

In [263]:

```
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 [264]:

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

Out[264]:

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

In [265]:

```
m1
```

Out[265]:

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

In [266]:

```
m2.T
```

Out[266]:

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

In [267]:

```
m2.T @ m1
```

Out[267]:

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

In [126]:

```
np.arange(10, 40, 10).T
```

Out[126]:

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

In [268]:

```
np.linalg.det(m1) # определитель
```

Out[268]:

```
0.0
```

In [269]:

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

Out[269]:

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

In [130]:

```
np.linalg.det(m3)
```

Out[130]:

```
265.00000000000017
```

In [270]:

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

Out[270]:

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

In [271]:

```
m3 @ m3i
```

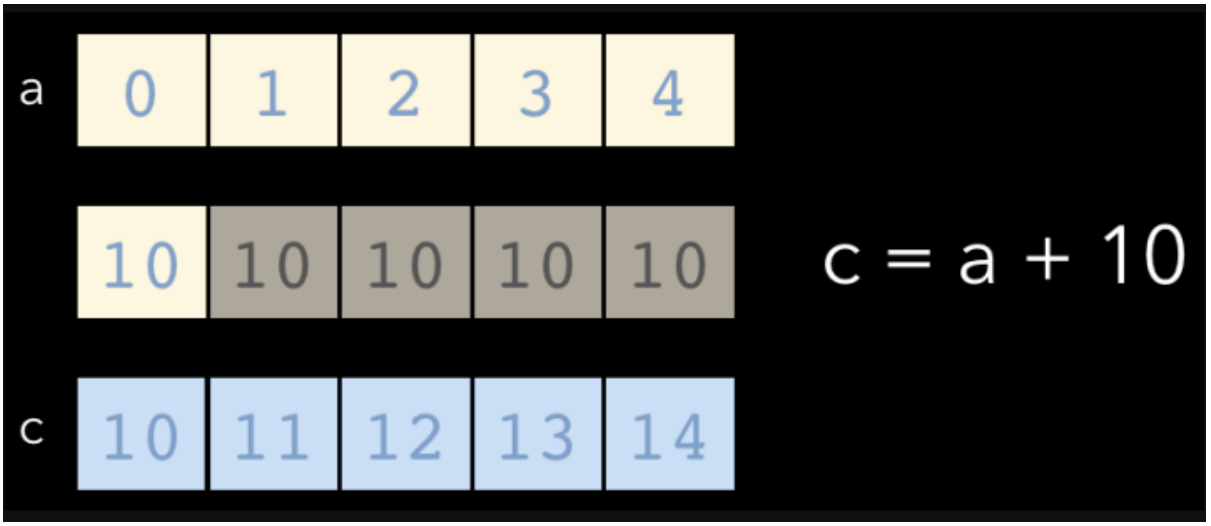
Out[271]:

```
array([[ 1.00000000e+00,  2.77555756e-17,  0.00000000e+00],
       [-1.66533454e-16,  1.00000000e+00, -2.22044605e-16],
       [ 0.00000000e+00,  1.38777878e-17,  1.00000000e+00]])
```

Распространение (broadcasting)

- к оглавлению

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



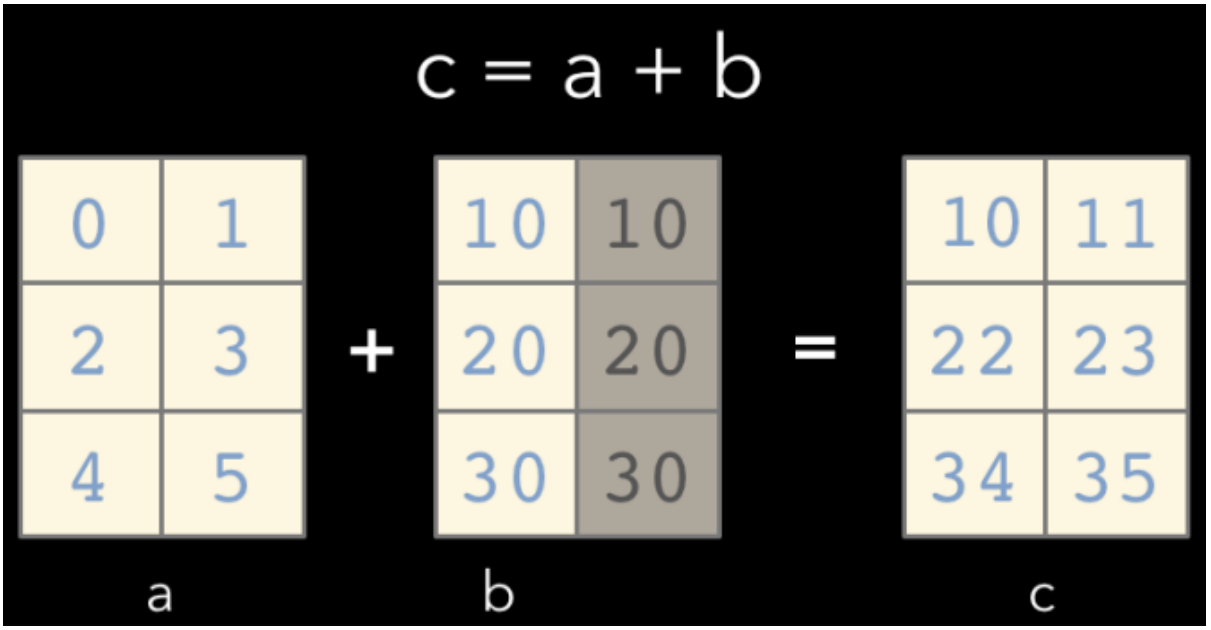
В примере скаляр распространяется до массива размерности (5,)

In [272]:

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

Out[272]:

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



Пример распространения для протяженных массивов разной размерности

In [273]:

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

Out[273]:

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

In [274]:

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

Out[274]:

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

In [275]:

```
a2 + b2
```

Out[275]:

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

In [103]:

```
a2.shape
```

Out[103]:

```
(3, 2)
```

In [276]:

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

Out[276]:

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

In [277]:

```
a2 + b3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-277-f3eec681a16e> in <module>
----> 1 a2 + b3

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

In [278]:

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

Out[278]:

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

In [279]:

```
a2 + b4
```

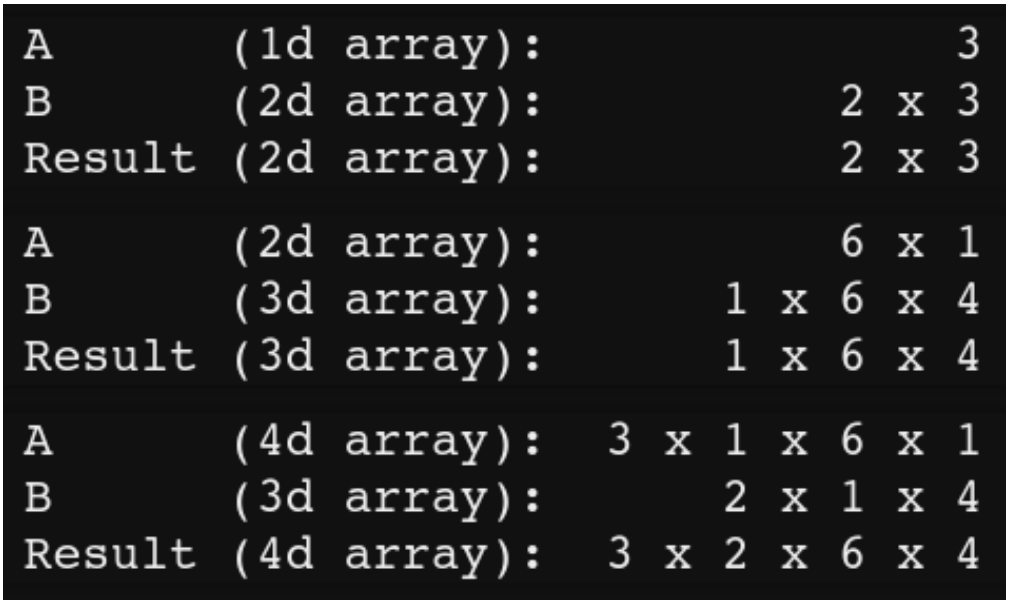
Out[279]:

```
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 в корректных операциях распространения

In [284]:

```
a2, a2.shape
```

Out[284]:

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

In [285]:

```
b3, b3.shape
```

Out[285]:

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

In [286]:

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

Out[286]:

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

In [287]:

```
a2 + b3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-287-f3eec681a16e> in <module>
----> 1 a2 + b3

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

In [288]:

```
a2 + b3t
```

Out[288]:

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

Продвинуте индексирование и операции с ndarray

- к оглавлению

Прихотливое индексирование (fancy indexing)

- к оглавлению

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

In [289]:

```
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[289]:

```
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 [290]:

```
e[1]
```

Out[290]:

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

In [291]:

```
row_indices = [3, 2, 1]
e[row_indices]
```

Out[291]:

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

In [135]:

```
col_indices = [1, 2, -1]
e[row_indices, col_indices]
```

Out[135]:

```
array([31, 22, 14])
```

In [446]:

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

Out[446]:

```
array([ 1, 12, 23, 34])
```

Маскирование ndarray

- к оглавлению

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

In [447]:

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

Out[447]:

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

In [448]:

```
f[fb]
```

Out[448]:

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

In [449]:

```
f % 2
```

Out[449]:

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

In [314]:

```
f % 2 == 0
```

Out[314]:

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

In [450]:

```
f[f % 2 == 0]
```

Out[450]:

```
array([0, 2, 4])
```

In [451]:

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

Out[451]:

```
6
```

Изменение формы и объединение ndarray

- к оглавлению

In [292]:

```
b
```

Out[292]:

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

In [293]:

```
b.shape
```

Out[293]:

```
(3, 5)
```

In [138]:

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

Out[138]:

```
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 больший массив из массивов меньших размеров.

In [294]:

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

Out[294]:

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

In [295]:

```
np.repeat(a5, 3)
```

Out[295]:

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

In [141]:

```
np.tile(a5, (3, 2))
```

Out[141]:

```
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 [296]:

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

Out[296]:

```
array([[5, 6]])
```

In [158]:

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

Out[158]:

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

In [143]:

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

Out[143]:

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

In [144]:

```
np.vstack((a5, b5, b5))
```

Out[144]:

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

In [145]:

```
np.hstack((a5, b5.T))
```

Out[145]:

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

Спасибо за внимание!