Глава 7 Библиотека NumPy

§ 7.1 Назначение библиотеки

При работе с предыдущими ноутбуками вы познакомились с основами языка Python. Узнали, что Python – довольно простой, понятный язык программирования, научились пользоваться *Jupyter notebook'ом* и оценили плюсы и возможности *Python+Jupyter* – написание кода в отдельных ячейках и последовательный запуск кода в ячейках.

Но есть и обратная сторона медали – Python очень медленный язык. Возможно, пока что это было не очень заметно, потому что мы не работали с большими массивами данных. Но в эпоху *Big data*, при работе с огромными массивами данных даже небольшое отставание языка программирования по скорости становится заметно и критично. Есть, конечно, языки программирования, которые работают намного быстрее (например, C++), но они гораздо сложнее в написании и не интерпретируемы, а компилируемы (то есть, в *Jupyter notebook'e* с такими языками работать бы не получилось).

Как же найти компромисс? Ответ прост: написать *Python*-библиотеку для работы с массивами данных, функции которой будут написаны на очень быстром языке C++, но которую можно было бы использовать из Python. Эта библиотека называется `NumPy` (НамПай).

> `NumPy` – это *open-source* (свободно распространяемое программное обеспечение с открытым исходным кодом) модуль для Python, который предоставляет общие математические и числовые операции в виде пре-скомпилированных, быстрых функций. Они объединяются в высокоуровневые пакеты и обеспечивают функционал, который можно сравнить с функционалом *MatLAB*. Библиотека `NumPy` предназначена для **вычислений на многомерных массивах**. Она используется в качестве базовой в большом количестве библиотек для научных вычислений, в том числе **для анализа данных и машинного обучения**.

`NumPy` (*Numeric Python*) – один из наиболее важных пакетов для выполнения высокопроизводительных научных расчётов; он позволяет эффективно работать с числами, одномерными и многомерными массивами: вычислять стандартные математические функции, реализовывать алгоритмы линейной алгебры, работать с генераторами случайных чисел.

Особенность `NumPy` – выполнение действий **с высокой скоростью и потребление небольшого количества ресурсов компьютера**, по сравнению с операциями, реализованными с помощью других модулей. `NumPy` – низкоуровневый модуль Python, на его основе построен `Pandas`. `NumPy` не содержит средств высокоуровневого анализа данных, но позволяет ускорить и упростить решение многих базовых задач, связанных с преобразованием матриц, получением базовой статистики, извлечением индексов элементов.

Можно выделить следующие **характерные черты** `NumPy`:

- использование типизированных, однородных n-мерных массивов фиксированной длины.
- операции применяются сразу ко многим элементам массивов.
- поддерживается векторизация, что позволяет повысить производительность вычислений.

Иногда использование `NumPy` необходимо; например, чтобы найти индекс максимального или минимального элемента, необходимо знать функцию модуля `NumPy`, которая позволит облегчить работу. Реализовывать эту простейшую задачу при помощи других модулей реально, но сложнее и дольше по времени. Внутри `NumPy` написана на языке Си, поэтому мы можем совместить скорость выполнения программ на языке Python со скоростью выполнения их на языке Си.

Основная структура библиотеки – это некоторый **многомерный массив**, который создается на основе наших данных. Основной тип данных `NumPy` – это `ndarray`, это многомерный массив с некоторым количеством осей (`NumPy` позволяет делать любое количество осей). Эти оси (*axis*) используются в `NumPy`, чтобы разделять структуры данных, которые в нем находятся. Количество осей в `ndarray` принято называть рангами: так у двумерного массива (матицы) – два ранга. Быстродействие кода Python с использованием `NumPy` в 50 раз быстрее кода на «чистом» Python. Компактный код вычислений на `NumPy`, в котором, в частности, отсутствуют циклы, может оказаться производительнее кода на обычном Python в десятки и даже сотни раз.

Ввод [43]:

```
import numpy as np
from timeit import timeit

count = 100000
num = 1000

def test1(x, y):
    return [a * b for a, b in zip(x, y)]

def test2(x, y): return x * y

x, y = list(range(0, count)), list(range(0, count))
t1 = timeit(lambda: test1(x, y), number=num)

nx, ny = np.array(x), np.array(y)
t2 = timeit(lambda: test2(nx, ny), number=num)

print(t1 / t2)
```

```
97.68147048453602
```

Замечание. Модуль `timeit` предоставляет простой способ замера времени выполнения небольших фрагментов Python-кода.

§ 7.2 Установка и импорт NumPy

Для того чтобы **установить эту библиотеку** необходимо зайти в терминал и выполнить команду:

```
python -m pip install numpy
```

Далее в терминале или в файле с программой необходимо **импортировать сам модуль**, написав:

```
import numpy as np
```

Это выражение позволяет нам получать доступ к `NumPy` объектам, используя `np.X` вместо `numpy.X`. Также можно импортировать `NumPy` прямо в используемое пространство имен, чтобы вообще не использовать функции через точку, а вызывать их напрямую:

```
from numpy import *
```

Однако этот вариант не приветствуется в программировании на Python, так как убирает некоторые полезные структуры, которые модуль предоставляет, поэтому будем использовать вариант импорта `import numpy as np`.

Варианты импорта библиотеки `NumPy` и последующего обращения к функции `array`, создающей массив, представлены в таблице.

Таблица. Варианты импорта библиотеки `NumPy`

	Импорт всей библиотеки	Импорт библиотеки с сокращенным названием	Импорт всех функций и переменных
Вариант импорта	`import numpy`	`import numpy as np`	`from numpy import *`
Пример использования	`v = numpy.array([1, 2])`	`v = np.array([1, 2])`	`v = array([1, 2])`

Посмотреть версию и конфигурацию `NumPy` можно командами

```
np.__version__
np.show_config()
```

Ввод [1]:

```
# принято, что numpy импортируют именно так
import numpy as np
```

Ввод [7]:

```
# print(np.__version__)
# print(np.show_config())
```

Можно воспользоваться документацией `NumPy` вызов команду `help(np)` – отобразится большой текстовый файл, предоставленный разработчиками `NumPy`, где описаны те команды, которые могут пригодиться. Документация есть на сайте.
Если необходима справка по конкретной команде, можно воспользоваться командой `np.lookfor('название команды')`, например:

Ввод [20]:

```
# help(np)
```

Ввод [21]:

```
# np.lookfor('binary representation')
```

§ 7.3 Массивы в NumPy, свойства массива

Тот факт, что код `Numpy` написан на C++, накладывает некоторые ограничения на массивы этой библиотеки: в любом `numpy` -массиве могут храниться элементы **только одного типа**: например, все `float` или все `string` (как вы помните с прошлого материала, в обычном python в массивах (`list`) могут быть элементы совершенно разных типов)

> **Массив** – это структура данных, содержащая упорядоченный набор значений (элементов) **одного типа**, идентифицируемых по индексу или набору индексов. **Размерностью массива** называют количество индексов, необходимых для однозначного определения его элементов. Первый элемент в массиве Python имеет индекс 0. Массивы могут быть одномерными (вектор), двумерными (матрица) и многомерными.

Index	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50

Основные **правила индексирования** массивов `NumPy` :

- индекс **первого** элемента массива равен 0;
- для обращения к элементу массива по индексу необходимо указать имя переменной, в которой хранится массив, и индекс в квадратных скобках: `vector[1]` , `matrix[0,5]` и т.д.;
- допускается использование отрицательных индексов;
- для обращения к нескольким идущим подряд элементам массива создаётся **срез**, в котором указывается индекс первого элемента среза и индекс элемента, следующего за последним, разделённые двоеточием;
- при создании среза возможно задание шага, в этом случае в срез будут включены не все элементы, а только отстоящие друг от друга на величину шага.

Индексация **двумерных** массивов имеет следующие особенности:

- при указании только одного индекса из массива будет выделена вся строка, соответствующая указанному индексу;
- можно указывать несколько индексов или срезов для каждой из осей.

Завести массив в numpy очень просто: надо всего лишь перевести обычный *python*- `list в np.array` :

Ввод [7]:

```
# Создим массив pupils, содержащий данные о школьниках
pupils =  np.array([[1, 135, 34, 4],
                    [2, 160, 43, 5],
                    [3, 163, 40, 4.3],
                    [4, 147, 44, 5],
                    [5, 138, 41, 4.7],
                    [6, 149, 54, 3.9],
                    [7, 136, 39, 4.2],
                    [8, 154, 48, 4.9],
                    [9, 137, 35, 3.7],
                    [10, 165, 60, 4.6]])
print(pupils)
```

```
[[  1.  135.   34.    4. ]
 [  2.  160.   43.    5. ]
 [  3.  163.   40.    4.3]
 [  4.  147.   44.    5. ]
 [  5.  138.   41.    4.7]
 [  6.  149.   54.    3.9]
 [  7.  136.   39.    4.2]
 [  8.  154.   48.    4.9]
 [  9.  137.   35.    3.7]
 [ 10.  165.   60.    4.6]]
```

Ввод [13]:

```
n = 1
pupils[n]    # содержимое n+1 строки
```

Out[13]:

```
array([  2., 160.,  43.,   5.])
```

Ввод [15]:

```
n = 3
pupils[:, n]   # содержимое n+1 столбца
```

Out[15]:

```
array([4. , 5. , 4.3, 5. , 4.7, 3.9, 4.2, 4.9, 3.7, 4.6])
```

Ввод [17]:

```
pupils[1::2, 1:3]  # данные по росту и весу школьников с четными индексами
```

Out[17]:

```
array([[160.,  43.],
       [147.,  44.],
       [149.,  54.],
       [154.,  48.],
       [165.,  60.]])
```

Ввод [34]:

```
print('Рост самого высокого школьника:', max(pupils[:, 1]))
print(f'Самый высокий ср. балл у %d ученика' % (np.argmax(pupils[:, 3]) + 1))
print(f'Самый низкий ср. балл у %d ученика' % (np.argmin(pupils[:, 3]) + 1))
print('Суммарный вес школьников:', sum(pupils[:, 2]))
print('Средний балл в классе:', round(sum(pupils[:, 3])/pupils.shape[0], 2))
print('Средний балл в классе:', round(np.mean(pupils[:, 3]), 2))
```

```
Рост самого высокого школьника: 165.0
Самый высокий ср. балл у 2 ученика
Самый низкий ср. балл у 9 ученика
Суммарный вес школьников: 438.0
Средний балл в классе: 4.43
Средний балл в классе: 4.43
```

Массивы, разумеется, можно использовать в `for` циклах. Но **при этом теряется** главное преимущество `numpy` – **быстродействие**. Всегда, когда это возможно, лучше использовать операции над массивами как едиными целыми.

К **свойствам массива** относится: тип его элементов, размерность массива, размер массива, число элементов в массиве, размер занимаемой памяти.

В отличие от чистого Python, в `numpy` есть **несколько типов для целых чисел** (`int16` , `int32` , `int64`) **и чисел с плавающей точкой** (`float32` , `float64`). Они отличаются тем, с какой точностью в памяти хранятся элементы массива.

Таблица. Просмотр свойств массива `ar`

Вызов метода	Описание метода
`ar.ndim`	Просмотр числа измерений массива или количество рангов (размерность)
`ar.shape`	Определение формы массива (числа элементов по осям)
`ar.size`	Определение количества элементов в массиве
`ar.dtype`	Определение типа данных, хранящихся в массиве
`ar.itemsize`	Просмотр размера элементов массива

Ввод [36]:

```
lst = [[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
       [10, 11, 12]]

# Тип lst
print(type(lst))       # => <class 'list'>

ar = np.array(lst)

# Тип ar
print(type(ar))       # => <class 'numpy.ndarray'>

# Размерность ar
print(ar.ndim)        # => 2

# Форма ar
print(ar.shape)       # => (4, 3)

# Количество элементов в ar
print(ar.size, ar.nbytes)      # => 12 48

# Тип элементов в ar
print(ar.dtype, ar.itemsize)   # => int32 4
```

```
<class 'list'>
<class 'numpy.ndarray'>
2
(4, 3)
12 48
int32 4
```

§ 7.4 Функции создания массива

Создать массив **из некоторой последовательности** можно функцией `array()`, которая преобразует вложенные элементы в массив. Создавать массив **из некоторой последовательности** можно **с указанием типа элементов** массива.

Ввод [37]:

```
vec = np.array([1, 2, 3, 4], dtype=np.int64)
print(vec, vec.shape)
```

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

Ввод [87]:

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

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

Создать вектор `vec` размера `n`, **заполненный нулями**.

Ввод [36]:

```
n = 8
vec = np.zeros((n,), dtype=int)
print(vec, vec.shape)
```

```
[0 0 0 0 0 0 0 0] (8,)
```

Создать матрицу, **заполненную нулями**.

Ввод [109]:

```
matr = np.zeros(shape=(3, 4), dtype=np.int8)
print(matr)
```

```
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Создать вектор `vec` размера `n`, **заполненный единицами**.

Ввод [4]:

```
n = 10
vec = np.ones(shape=(4,), dtype=float)
print(vec)
```

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

Создать матрицу, **заполненную единицами**.

Ввод [84]:

```
n = 10
matr = np.ones((4, 7))
print(matr)
```

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

Создать вектор `vec` размера `n`, **заполненный числом** `a`.

Ввод [32]:

```
n, a = 8, 13
vec = np.full(n, a)
print(vec)
```

```
[13 13 13 13 13 13 13 13]
```

Создать матрицу, **заполненную числом** `a`.

Ввод [10]:

```
a = 3
matr = np.full((4, 7), a)
print(matr)
```

```
[[3 3 3 3 3 3 3]
 [3 3 3 3 3 3 3]
 [3 3 3 3 3 3 3]
 [3 3 3 3 3 3 3]]
```

Создать вектор **со значениями от** `a` **до** `b` (`b` не войдет).

Функция `arange` подобна `range` . Аргументы могут быть с плавающей точкой. Следует избегать ситуаций, когда `(конец-начало)/шаг` – целое число, потому что в этом случае включение последнего элемента зависит от ошибок округления. Лучше, чтобы конец диапазона был где-то посредине шага.

Ввод [36]:

```
a, b, h = 0, 10, 2
vec = np.arange(a, b, h)
print(vec)
```

```
[0 2 4 6 8]
```

Ввод [37]:

```
a, b, h = 0., 10, 2
vec = np.arange(a, b, h)
print(vec)
```

```
[0. 2. 4. 6. 8.]
```

Ввод [122]:

```
tenzor = np.arange(20, 44).reshape(2, 3, 4)
print(tenzor)
```

```
[[[20 21 22 23]
  [24 25 26 27]
  [28 29 30 31]]

 [[32 33 34 35]
  [36 37 38 39]
  [40 41 42 43]]]
```

Создать **случайный вектор** размера 10.

Ввод [52]:

```
vec = np.random.random(10)
print(vec)
```

```
[0.71758036 0.28640869 0.13323778 0.53392348 0.69874418 0.38385635
 0.90427991 0.34465781 0.73891822 0.06807669]
```

Ввод [100]:

```
vec = np.random.randint(-5, 3, (5,))
print(vec)
```

```
[ 2  1 -1 -4  2]
```

Создать **массив** $10x10$ со **случайными значениями**

Ввод [74]:

```
matr = np.random.random((3, 4))
print(matr)
```

```
[[0.76735501 0.26550671 0.55894525 0.59326342]
 [0.76241415 0.62127854 0.37602239 0.51050028]
 [0.19855404 0.2277817  0.08657882 0.87925405]]
```

Ввод [41]:

```
matr = np.random.uniform(-3.0, 5.0, size=(3, 5))
print(matr)
```

```
[[-1.56004947 -0.95826281 -2.5529678  -1.40998338 -1.54888933]
 [ 0.9845089  -2.94465472  2.55459958  4.32131407 -0.49439201]
 [ 2.68809475  0.84695957 -1.25180823  2.70600474 -1.98845048]]
```

Ввод [79]:

```
matr = np.random.randint(-3, 5, (2,7))
print(matr)
```

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

Последовательность чисел с постоянным шагом можно также создавать функцией `linspace`. Функция `linspace` позволяет создавать `float` массив с известным начальным и конечным значением, а также массив, для которого известно, сколько элементов между начальным и конечным надо получить (начало и конец диапазона включаются; последний аргумент – число точек).

Ввод [43]:

```
a, b, h = 1., 2.5, 5
vec = np.linspace(a, b, h)
print(vec)
```

```
[1.    1.375 1.75  2.125 2.5  ]
```

Последовательность чисел с постоянным шагом по логарифмической шкале от 10^0 до 10^1.

Ввод [47]:

```
vec = np.logspace(0, 1, 5)
print(vec)
```

```
[ 1.          1.77827941  3.16227766  5.62341325 10.        ]
```

Для создания массива, в котором **элементы рассчитываются по формуле, как комбинации индексов** используется функция `fromfunction()` .

Ввод [12]:

```
def f(i, j):
    return 10*i + j

vec = np.fromfunction(f, (5, 4), dtype=int)
print(vec)
```

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
```

Создать **единичную матрицу** формата $3x3$

Ввод [48]:

```
n = 3
matr = np.eye(3)
print(matr)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Создать **диагональную матрицу**.

Ввод [97]:

```
matr = np.identity(4, dtype=int)
print(matr)
```

```
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

Создать **диагональную матрицу** матрицу с 1, 2, 3, 4 **под диагональю**.

Ввод [96]:

```
matr = np.diag(np.arange(1, 5), k=-1)
print(matr)
```

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

§ 7.5 Операции над массивами

7.5.1 Изменение и добавление элементов

Как и `list` в Python, массивы `np.array` – **изменяемые объекты**. Механика изменений значений в них такая же, как у `list` -ов Python. Давайте в этом убедимся:

Ввод [112]:

```
vec = np.array([3, 4 ,1])
vec[1] = -3
print(vec)
```

```
[ 3 -3  1]
```

Единственный (но логичный) **нюанс**: при изменении значения в массиве с элементами одного типа на элемент другого типа новый элемент будет приведен к типу массива:

Ввод [116]:

```
vec = np.array([8, 4 ,1]).astype(np.int64)

# значение 3.5 будет приведено к типу int64, т.е. станет 3
vec[1] = 3.5
print(vec)
```

```
[8 3 1]
```

Пример. Дан массив, поменять знак у элементов, значения которых между 3 и 8, после чего обнулить чётные элементы.

Ввод [8]:

```
vec = np.arange(11)
print(vec)

vec[(3 < vec) & (vec <= 8)] *= -1
print(vec)

vec[vec % 2 == 0] = 0
print(vec)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10]
[ 0  1  2  3 -4 -5 -6 -7 -8  9 10]
[ 0  1  0  3  0 -5  0 -7  0  9  0]
```

Пример (модификация на месте). Уменьшить все элементы массива в 2 раза.

Ввод [18]:

```
vec = np.arange(11, dtype=float)
print(vec)

vec /= 2
print(vec)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
```

В Numpy можно **сделать массив неизменяемым**:

Ввод [150]:

```
new_ar = np.zeros(10)
new_ar.flags.writeable = False
new_ar[0] = 1        # Ошибка
```

В Numpy можно **переназначить тип элементов** массива.

Ввод [17]:

```
# Преобразовать массив из float в int
vec = np.array([8.7, 4.9, -1.4])
print(vec, vec.dtype)

vec = vec.astype(np.int16, copy=False)
print(vec, vec.dtype)
```

```
[ 8.7  4.9 -1.4] float64
[ 8  4 -1] int16
```

Ввод [13]:

```
# Преобразовать массив из int во float
vec = np.arange(10, dtype=np.int32)
print(vec, vec.dtype)

vec = vec.astype(np.float32, copy=False)
print(vec, vec.dtype)
```

```
[0 1 2 3 4 5 6 7 8 9] int32
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.] float32
```

Ввод [24]:

```
# Преобразовать массив из float в str
vec = np.array([0, 2, 1], dtype=np.float64)
print(vec, vec.dtype)

vec = vec.astype(str)
print(vec, vec.dtype)
```

```
[0. 2. 1.] float64
['0.0' '2.0' '1.0'] <U32
```

А вот **добавить к массиву новый элемент** в конец чуть сложнее, чем у `list`. Напримним, в `list` это делалось с помощью метода `.append()`. В `numpy` это также делается с помощью `append`, но чуть по-другому:

Ввод [118]:

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

# вот так пишется append
vec = np.append(vec, 6)
print(vec)
```

```
[3 4 1 6]
```

Обратите внимание, что в `numpy` при `append` создается новый массив, а не происходит добавление элемента в уже существующий массив. Поэтому **не рекомендуется** создавать массивы с помощью `append` в `numpy`.

7.5.2 Изменение формы массива, объединение массивов

При работе с многомерными массивами, часто возникает потребность как-то **поменять форму**, либо **размерность данных**, например, если у нас была какая-то многомерная структура, мы хотим ее развернуть в один одномерный массив или наоборот. В `NumPy` это можно осуществить, используя следующие методы:

- `flatten()` – возвращает копию массива сжатую до одного измерения, т.е разворачивает все значения, хранящиеся в матрице, в одномерный массив. Вызов метода: `ar.flatten()`;
- `ravel()` – возвращает сжатый до одной оси массив. Вызов метода: `np.ravel(ar)` или `a.ravel()`;
- `reshape()` – изменяет форму массива без изменения его данных. В метод `reshape` необходимо передать новые значения по двум осям `X` и `У`, которые мы хотим увидеть. Вызов метода: `np.reshape(a, (4, 3))` или `ar.reshape(4, 3)`.

Пояснение. Функции `flatten` и `ravel` возвращают сплющенные массивы 1D, указывающие на новые структуры памяти. Правильное пространство имен для функций: `numpy.ndarray.flatten` и `numpy.ravel`. `flatten` является методом объекта `ndarray` и, следовательно, может быть вызван только для истинных массивов `numpy`. `ravel` является функцией библиотечного уровня и, следовательно, может быть вызвана для любого объекта, который может быть успешно проанализирован. Например, `ravel` будет работать со списком `ndarrays`, в то время как `flatten` недоступен для этого типа объектов.

Ввод [2]:

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

print(ar.flatten())
print(ar)
```

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

Ввод [3]:

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

print(ar.ravel())  # print(np.ravel(ar))
print(ar)
```

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

Ввод [10]:

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

new_ar = ar.reshape(3, 2)
print(new_ar)
```

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

Важно отметить, что при использовании `reshape` размер новой формы не должен по количеству элементов превышать размер нашей исходной матрицы, то есть если у нас в матрице было всего шесть элементов, мы никак не можем задать форму, в которой будет, например, 15 элементов, мы получим сразу же ошибку.

Также используя метод `resize`, можно поменять форму нашего исходного массива. Отличие `resize` от `reshape` заключается в том, что `resize` автоматически меняет исходный массив, в то время, как `reshape` просто изменяет его форму. Кроме того, если у нового массива число элементов должно оказаться больше, чем у исходного, то недостающие элементы заполняются нулями.

Ввод [7]:

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

ar.resize(3, 2)
print(ar)
print(ar.shape)
```

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

Ввод [11]:

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

ar.reshape(3, 2)
print(ar)
print(ar.shape)
```

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

Объединение массивов "по-горизонтали" (*horizontal stack*).

Ввод [5]:

```
a = np.array([1, 2, 3])
b = np.array([100, 200, 300])

print(np.hstack((a, b)))
print(np.hstack((b, [500])))
```

```
[  1   2   3 100 200 300]
[100 200 300 500]
```

Объединение массивов "по-вертикали" (*vertical stack*).

Ввод [21]:

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

```
[[  1   2   3]
 [100 200 300]]
```

Расщепление массива в позициях 3 и 6.

Ввод [24]:

```
a = np.random.random(10)
print(a)

a1, a2, a3 = np.hsplit(a, [3, 6])
print(a2)
```

```
[0.27050291 0.67514166 0.04852551 0.06924081 0.59810422 0.13413315
 0.94316399 0.64322534 0.44959668 0.12467934]
[0.06924081 0.59810422 0.13413315]
```

Функции `delete`, `insert` и `append` **не меняют** массив на месте, а возвращают новый массив, в котором удалены, вставлены в середину или добавлены в конец какие-то элементы.

Ввод [25]:

```
a = np.arange(10)

a = np.delete(a, [5, 7])
print(a)

a = np.insert(a, 2, [0, 0])
print(a)

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

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

Транспонирование также не приводит к изменению данных:

Ввод [16]:

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

print(a.T)
print(a)
```

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

§ 7.6 Способы индексации массивов

После создания массива взаимодействовать с ним и просматривать какие-то определенные значения можно, обратившись к элементу по индексу. **Индексация начинается с нуля**.

Если требуется «вытащить» какой-то элемент, находящийся в двумерном массиве `ar`, допустим, в первой строке и в третьем столбце, нам необходимо задать индексы сразу же по двум осям, то есть сначала первым аргументом в квадратных скобках указать индекс **по оси X**, а вторым аргументом указать индекс **по оси Y**: `ar[0, 2]`.

7.6.1 Срезы

При просмотре, подсчете и присвоении можно использовать срезы.

Простая индексация:

Ввод [21]:

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

# Запись производится по ссылке, то есть модифицируется исходный массив
a[3] = 42
print(a)

# Если типы элементов не совпадают, то значение "обрезается"
a[3] = 4.9
print(a, a.dtype)
```

```
1 4
[ 1  2  3 42]
[1 2 3 4] int32
```

Ввод [22]:

```
# Последний элемент
print(a[-1])

a1 = a.reshape(2, 2)
print(a1)

# Индексация производится от строк к столбцам
print(a1[0], a1[1, -1])
print(a1[1, 0], a1[0, 1])

# Так тоже можно
print(a1[1][0], a1[0][1])
```

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

Срезы работают аналогично стандартным в Python, однако модифицируются исходные данные:

Ввод [23]:

```
a = np.array([1, 2, 3, 4])
a1 = a.reshape(2, 2)

print(a[::2])

print(a1)
b = a1[:, 1:]
print(b)

# Заменили элемент и в исходных массивах
b[0] = 42
print(a)
print(a1)
```

```
[1 3]
[[1 2]
 [3 4]]
[[2]
 [4]]
[ 1 42  3  4]
[[ 1 42]
 [ 3  4]]
```

Индексация с помощью массивов:

Ввод [27]:

```
a = np.arange(0, 100, 10)
print(a)
print(a[[0, 4, 6, 7]])
print(a[[-4, -3, -2, -1]])
```

```
[ 0 10 20 30 40 50 60 70 80 90]
[ 0 40 60 70]
[60 70 80 90]
```

Подмассиву можно присвоить значение – массив правильного размера или скаляр.

Ввод [6]:

```
ar = np.arange(40).reshape(5, 8)
print('Было:\n', ar)

ar[1:4, 1:6:2] = 0
print('Стало:\n', ar)
```

```
Было:
 [[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]
 [32 33 34 35 36 37 38 39]]
Стало:
 [[ 0  1  2  3  4  5  6  7]
 [ 8  0 10  0 12  0 14 15]
 [16  0 18  0 20  0 22 23]
 [24  0 26  0 28  0 30 31]
 [32 33 34 35 36 37 38 39]]
```

Ввод [38]:

```
ar = np.delete(ar, [1,3], axis=1)
print(ar)
```

```
[[ 0  2  4  5  6  7]
 [ 8 10 12  0 14 15]
 [16 18 20  0 22 23]
 [24 26 28  0 30 31]
 [32 34 36 37 38 39]]
```

Встроенная функция `enumerate()` возвращает индекс и текущий элемент. Рассмотрим эквивалент функции `enumerate` для `numpy` массивов?

Ввод [6]:

```
a = np.arange(9).reshape(3,3)
for index, value in   np.ndenumerate(a):
    print(index, value)
for index in np.ndindex(a.shape):
    print(index, a[index])
```

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

7.6.2 Сравнения и маски

Также можно, используя некоторые условия, вытаскивать значения, которые удовлетворяют заданному порогу. Допустим у нас есть некоторый массив, мы хотим найти все значения, превышающие какого-либо заданного числа, например 2. Для начала необходимо задать условия, в результате этого возвращается некоторая маска.

> **Маска** – это массив того же размера, что наш исходный, но в качестве значений указывается `True` или `False` , что означает, что данное число в массиве превышает, либо не превышает заданного порога, и затем, для того чтобы вытащить нужное нам значение, мы эту маску передаем как индекс в наш массив, и, в результате этого, возвращается уже массив значений, удовлетворяющих условию.

Пример. Определить в массиве сумму элементов, принадлежащих отрезку $(-10; 10)$.

Ввод [16]:

```
a = np.array([[ 0,  -10,  2,  3,  40,  50,  0,  -70],
              [ 8,  9, -4, 11, 12, 13, -4, 15]])
print(a)
print([(-10 < a) & (a < 10)])
print(sum(a[(-10 < a) & (a < 10)]))
```

```
[[  0 -10   2   3  40  50   0 -70]
 [  8   9  -4  11  12  13  -4  15]]
[array([[ True, False,  True,  True, False, False,  True, False],
       [ True,  True,  True, False, False, False,  True, False]])]
14
```

Маска может использоваться для выбора только определенных строк или столбцов из всего массива и скрытия остальных. Маска в Python задаётся при помощи булевых 0 и 1, где 0 скрывает столбец или строку, а 1 оставляет ее на виду. Например, выведем элементы, которые расположены в нечетных столбцах массива:

Ввод [19]:

```
a = np.array([[ 0,  -10,  2,  3,  40,  50,  0,  -70],
              [ 8,  9, -4, 11, 12, 13, -4, 15]])
print(a)

mask = np.array([1,  0,  1,  0,  1,  0,  1,  0], dtype=bool)
print(a[:, mask])
```

```
[[  0 -10   2   3  40  50   0 -70]
 [  8   9  -4  11  12  13  -4  15]]
[[ 0  2 40  0]
 [ 8 -4 12 -4]]
```

§ 7.7 Математические и статистические операции

Особенностью работы с массивами в `NumPy` является то, что возможности библиотеки позволяют выполнять любые математические действия с массивами без использования циклов `for`, благодаря чему вычисления производятся с большой скоростью. В `NumPy` над массивами можно производить все стандартные арифметические операции.

Операции выполняются поэлементно, поэтому для получения корректного результата размерность массивов должна быть одинаковой. Также можно производить математические операции между массивом и числом.

Встроенные функции библиотеки `NumPy`:

- ***поэлементное сложение***: `res = v + w` или `res = np.add(v, w)`;
- ***поэлементная разность***: `res = v - w` или `res = np.subtract(v, w)`;
- ***поэлементное умножение***: `res = v * w` или `res = np.multiply(v, w)`;
- ***поэлементное частное***: `res = v / w`;
- ***поэлементное умножение на число***: `res = v * 2`;
- ***скалярное произведение*** между векторами. Для того чтобы посчитать какое-либо скалярное произведение между векторами, либо умножить вектор на матрицу, необходимо воспользоваться методом `dot`. Он на вход принимает два вектора, которые необходимо перемножить. Таким же способом, используя функцию `dot`, мы можем перемножать вектор с матрицей, либо матрицу с матрицей.

Ввод [32]:

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

print(a * 10)
print(a + b)
print(a * b)
print(a ** b)
print(np.sin(a))

print(np.negative(a))
print(a > 2)
print(a.dot(b))
```

```
[10 20 30 40]
[3 5 7 9]
[ 2  6 12 20]
[   1    8   81 1024]
[ 0.84147098  0.90929743  0.14112001 -0.7568025 ]
[-1 -2 -3 -4]
[False False  True  True]
40
```

7.7.1 Универсальные функции

Numpy содержит **элементарные функции**, которые тоже применяются к массивам поэлементно. Они называются универсальными функциями (ufunc).

> **Универсальными** называют **функции**, которые выполняют поэлементные операции над данными, хранящимися в объектах ndarray . Большинство универсальных функций относятся к унарным операциям и выполняются над каждым элементом массива по очереди. **Унарные операции** – это и есть операции, которые выполняются над каждым элементом массива по очереди.

Рассмотрим список часто используемых универсальных функций NumPy . При вызове каждой из этих функций необходимо указывать название библиотеки NumPy :

```
np.func_name(имя_массива)
```

Таблица. Универсальные функции NumPy

Функция	Описание
abs	Абсолютное значение целых, вещественных или комплексных элементов массива
sqrt	Квадратный корень каждого элемента массива
exp	Экспонента (ex) каждого элемента массива
log , log10 , log2 , log1p	Натуральный (по основанию e), десятичный, двоичный логарифм и функция log(1+x) соответственно
modf	Дробные и целые части массива в виде отдельных массивов
isnan	Массив логических (булевых) значений, показывающий, какие из элементов исходного массива являются NaN (не числами)
cos , sin , tan	Обычные тригонометрические функции

Функция	Описание
`arccos`, `arcsin`, `arctan`	Обратные тригонометрические функции

Ввод [26]:

```
np.sin, type(np.sin)
```

Out[26]:

```
(<ufunc 'sin'>, numpy.ufunc)
```

Ввод [41]:

```
a = np.array([[2, 5], [6, 7]])

print(np.sqrt(a))
print(np.exp(a))
print(np.log(a + 1))
print(np.sin(a))
print(np.e, np.pi)
```

```
[[1.41421356 2.23606798]
 [2.44948974 2.64575131]]
[[   7.3890561   148.4131591 ]
 [ 403.42879349 1096.63315843]]
[[1.09861229 1.79175947]
 [1.94591015 2.07944154]]
[[ 0.90929743 -0.95892427]
 [-0.2794155   0.6569866 ]]
2.718281828459045 3.141592653589793
```

Функции вычисления суммы элементов массива, поиска минимального и максимального элементов и многие другие **по умолчанию работают для всех элементов массива, не учитывая размерность**:

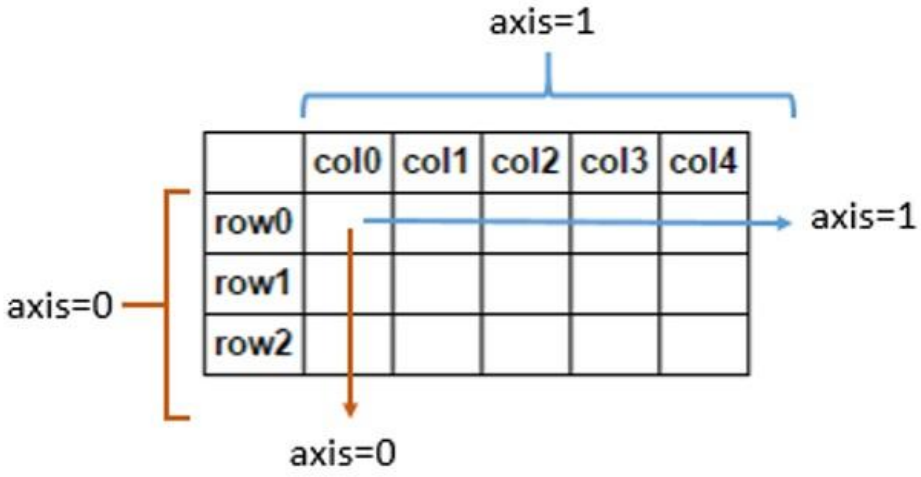
Ввод [1]:

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

print(a.sum())    # сумма всех элементов
print(a.prod())   # произведение всех элементов
print(a.max())    # максимальный из всех элементов
print(a.min())    # минимальный из всех элементов
```

```
[[ 2  5  4]
 [-6 -7  1]]
-1
1680
5
-7
```

Дополнительно в указанных функциях можно указать **номер оси (индексация с 0), на которой будет работать функция**:



Ввод [4]:

```
a = np.array([[2, 5, 4, -1], [-6, -7, 1, 4], [-5, 0, 2, -3]])
print(a)
print(a.shape)

print(a.min(axis=1))    # минимум в строках
print(a.max(axis=0))    # максимум в столбцах
print(a.sum(axis=1))    # суммы в строках
print(a.sum(axis=0))    # суммы в столбцах
```

```
[[ 2  5  4 -1]
 [-6 -7  1  4]
 [-5  0  2 -3]]
(3, 4)
[-1 -7 -5]
[2 5 4 4]
[10 -8 -6]
[-9 -2  7  0]
```

Пояснение. Например, если использовалась команда: `a.sum(axis=n)` , то измерение `n` будет свернуто и удалено, при этом каждое значение в новой матрице будет равно сумме соответствующих свернутых значений.

Ввод [7]:

```
a = np.array([[2.8, -5.9, 4.01], [-6.25, -7.77, 0.005]])
print(a)

print(np.modf(a))
```

```
[[ 2.80e+00 -5.90e+00  4.01e+00]
 [-6.25e+00 -7.77e+00  5.00e-03]]
(array([[ 0.8  , -0.9  ,  0.01 ],
       [-0.25 , -0.77 ,  0.005]]), array([[ 2., -5.,  4.],
       [-6., -7.,  0.]]))
```

В модуле `Numpy` имеются **кванторы "существует" и "для всех"**.

Пример. Проверьте, что в массиве:

- есть элемент, равный нулю;

- все элементы равны единице.

Ввод [75]:

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

print(c)
np.any(a == 0), np.all(c == 1)
```

```
[[1 1]
 [1 1]]
```

Out[75]:

```
(True, True)
```

7.7.2 Операции линейной алгебры

Матричное умножение – `@`

Ввод [24]:

```
a = np.array([[0, 1], [2, 3]])
b = np.array([[1, 2, 3],
              [4, 5, 6]])
c = a @ b
print(c)
```

```
[[ 4  5  6]
 [14 19 24]]
```

Вычисление определителя – `np.linalg.det(ar)`

Ввод [23]:

```
a = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]])
print(a)
print('Определитель =', np.linalg.det(a))
```

```
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Определитель = 27.0
```

Нахождение обратной матрицы – `np.linalg.inv(ar)`

Ввод [21]:

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

negativea= np.linalg.inv(a)
print(negativea)

print(a @ negativea)
```

```
[[0 1]
 [2 3]]
[[-1.5  0.5]
 [ 1.   0. ]]
[[1. 0.]
 [0. 1.]]
```

Определение ранга матрицы – `np.linalg.matrix_rank(ar)`

> **Рангом** называют максимальное число линейно независимых строк (столбцов) матрицы. **Линейная независимость** означает, что строки (столбцы) не могут быть линейно выражены через другие строки (столбцы).

Ранг матрицы можно найти через ее миноры, он равен наибольшему порядку минора, который не равен нулю. Существование ранга у матрицы не зависит от того квадратная она или нет.

Ввод [59]:

```
a = np.random.randint(-10, 10, (4, 4))
print(a)
rank = np.linalg.matrix_rank(a)
print(rank)
```

```
[[  7   2  -1   5]
 [  6 -10   5  -5]
 [  0  -1  -3  -9]
 [  6  -8   4   9]]
4
```

Ввод [58]:

```
a = np.array([[1, 0, 0, 0],
              [0, 1, 0, 0],
              [0, 0, 1, 0],
              [0, 0, 0, 0]])
rank = np.linalg.matrix_rank(a)
print(rank)
```

```
3
```

Решение линейной системы $AX = B$.

Ввод [33]:

```
a = np.array([[1, 4, 2, -1],
              [4, 3, 2, 3],
              [1, 2, 1, -1],
              [3, 7, 3, 2]])
b = np.array([3, 8, 1, 7])
if np.linalg.det(a) != 0:
    x = np.linalg.solve(a, b)
    print(x)
    print('Проверка:\n', a @ x - b)
else:
    print('Определитель матрицы A равен нулю.')
```

```
[ 1.94289029e-16 -1.00000000e+00  4.00000000e+00  1.00000000e+00]
Проверка:
 [-4.4408921e-16  0.0000000e+00  0.0000000e+00  0.0000000e+00]
```

Собственные значения и собственные векторы: $au_i = \lambda_i u_i$.

Для поиска собственных чисел и собственного вектора используется метод `eig`, который возвращает сразу пару значений, где первым значением являются собственные числа, а вторым – собственные векторы.

`l` – одномерный массив собственных значений λ_i, столбцы матрицы u – собственные векторы u_i.

Ввод [37]:

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

l, u = np.linalg.eig(a)
print(l)
print(u)
```

```
[-0.56155281  3.56155281]
[[-0.87192821 -0.27032301]
 [ 0.48963374 -0.96276969]]
```

Ввод [38]:

```
# Проверим
for i in range(2):
    print(a @ u[:, i] - l[i] * u[:, i])
```

```
[0.00000000e+00 1.66533454e-16]
[1.11022302e-16 0.00000000e+00]
```

Транспонированние матрицы – `a.T`

Ввод [35]:

```
a = np.array([[1, 4, 2, -1],
              [4, 3, 2, 3],
              [1, 2, 1, -1],
              [3, 7, 3, 2]])
print(a)
print(a.T)
```

```
[[ 1  4  2 -1]
 [ 4  3  2  3]
 [ 1  2  1 -1]
 [ 3  7  3  2]]
[[ 1  4  1  3]
 [ 4  3  2  7]
 [ 2  2  1  3]
 [-1  3 -1  2]]
```

Скалярное произведение двух векторов – `np.dot(a, b)`

Ввод [41]:

```
a = np.array([1, 4, 2, -1])
b = np.array([3, 7, 3, 2])
print(np.dot(a, b))
```

```
35
```

7.7.3 Статистические функциии

Модуль `NumPy` содержит множество базовых **статистических функций**, которые помогают описать имеющиеся данные.

Синтаксис вызова этих функций следующий:

```
np.func_name(имя_вектора)
```

Таблица. Некоторые базовые статистические функций

Вызов метода	Описание метода
`mean`	Среднее арифметическое
`median`	Медиана
`std`	Стандартное отклонение
`corrcoef`	Корреляция
`trace`	**След** - сумма диагональных элементов
`cumsum`	Кумулятивные суммы

Ввод [55]:

```
b = np.array([[4, 5, 6], [7, 8, 9], [1, 3, 0]])
print(b)
print(b.trace())    # сумма диагональных элементов
```

```
[[4 5 6]
 [7 8 9]
 [1 3 0]]
12
```

Ввод [63]:

```
b = np.array([2, -1, 4, -2, 0, 3])
b.mean(), b.std()
```

Out[63]:

```
(1.0, 2.160246899469287)
```

Иногда бывает нужно использовать **кумулятивные суммы**.

Ввод [65]:

```
b = np.array([2, -1, 4, -2, 0, 3])
print(b.cumsum())
```

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

Функция `sort` возвращает **отсортированную копию**, метод `sort` **сортирует на месте**.

Ввод [40]:

```
b = np.array([2, -1, 4, -2, 0, 3])
print(np.sort(b))
print(b)

b.sort()
print(b)
```

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

Для изучения статистических функций будем использовать набор данных, содержащих информацию об учениках шестого класса. Данные об учениках представлены в таблице

Ввод [42]:

```
# Создим массив pupils, содержащий данные о школьниках
pupils =  np.array([[1, 135, 34, 4],
                    [2, 160, 43, 5],
                    [3, 163, 40, 4.3],
                    [4, 147, 44, 5],
                    [5, 138, 41, 4.7],
                    [6, 149, 54, 3.9],
                    [7, 136, 39, 4.2],
                    [8, 154, 48, 4.9],
                    [9, 137, 35, 3.7],
                    [10, 165, 60, 4.6]])
```

Ввод [43]:

```
import pandas as pd

df = pd.DataFrame(pupils, columns=['Numder', 'Height', 'Weight', 'SrBall'])
df
```

Out[43]:

	Numder	Height	Weight	SrBall
0	1.0	135.0	34.0	4.0
1	2.0	160.0	43.0	5.0
2	3.0	163.0	40.0	4.3
3	4.0	147.0	44.0	5.0
4	5.0	138.0	41.0	4.7
5	6.0	149.0	54.0	3.9
6	7.0	136.0	39.0	4.2
7	8.0	154.0	48.0	4.9
8	9.0	137.0	35.0	3.7
9	10.0	165.0	60.0	4.6

7.7.3.1 Среднее арифметическое `mean`

> **Среднее арифметическое (`mean`)** – сумма всех значений, делённая на их количество, показывает общую тенденцию данных и описывает их одним числом.

> **Математическое ожидание** – среднее значение случайной величины при стремлении числа её измерений к бесконечности. Если число измерений конечно, то для оценки среднего значения величины используется среднее арифметическое.

Ввод [44]:

```
print('Средний балл по классу равен %.2f.' % np.mean(pupils[:, -1]))
print('Средний рост школьников равен %.2f.' % np.mean(pupils[:, 1]))
print('Средний вес школьников равен %.2f.' % np.mean(pupils[:, 2]))
```

```
Средний балл по классу равен 4.43.
Средний рост школьников равен 148.40.
Средний вес школьников равен 43.80.
```

7.7.3.2 Медиана `median`

> **Медиана (`median`)** – это такое число, что ровно половина элементов из выборки больше него, а другая половина меньше него. Если в выборке есть *выбросы* (значения, которые принимают величину существенно выше или ниже среднего и выделяются из всей выборки), то медиана будет лучше характеризовать всю выборку, чем среднее арифметическое.

Ввод [45]:

```
print('Балл, равный %.2f поделил учеников пополам.' % np.median(pupils[:, -1]))
```

```
Балл, равный 4.45 поделил учеников пополам.
```

Полученные результаты говорят нам о том, что в среднем школьники учатся хорошо, среди них много отличников (медианное значение среднего балла выше среднего арифметического). Заметим, что школьников в выборке всего 10, а медиана – это значение выборки, при котором ровно половина значений меньше неё, а половина – больше.

Python сначала упорядочит значения по возрастанию, а затем возьмет среднее средних элементов полученного ряда, то есть пятого и шестого. Проверим это, используя функцию `sort` для сортировки значений оценок:

Ввод [46]:

```
sort_lst = np.sort(pupils[:, -1])

half_size = len(sort_lst) // 2
if len(sort_lst) % 2 != 0:
    mediana = sort_lst[half_size]
else:
    mediana = (sort_lst[half_size - 1] + sort_lst[half_size]) / 2
print('Медиана = %.2f' % mediana)
```

```
Медиана = 4.45
```

Задача. Каково медианное значение роста школьников и их веса?

Ввод [49]:

```
print('Медианное значение роста равно %.2f.' % np.median(pupils[:, 1]))
print('Медианное значение веса равно %.2f.' % np.median(pupils[:, 2]))
```

```
Медианное значение роста равно 148.00.
Медианное значение веса равно 42.00.
```

7.7.3.3 Стандартное отклонение `std`

> **Дисперсия** – мера разброса случайной величины относительно её математического ожидания.

$$D(X) = M[(X - M[X])^2]$$

Как следует из формулы, дисперсия случайной величины X равна математическому ожиданию квадрата отклонения случайной величины от ее математического ожидания. То есть, если величина дисперсии небольшая, значит, все числа в выборке имеют близкие друг к другу значения, а чем она больше – тем значительнее разброс показателей.

> **Стандартное отклонение** равно квадратному корню из дисперсии. Низкое стандартное отклонение показывает, что все значения в выборке сгруппированы около среднего значения. Большой показатель этой величины говорит о том, что разброс значений большой.

$$\sigma = \sqrt{D(x)}$$

Стандартное отклонение – самый распространенный показатель рассеивания случайной величины относительно её математического ожидания.

Задача. Чему равно стандартное отклонение роста от среднего в классе?

Ввод [52]:

```
print('Стандартное отклонение роста от среднего \
в классе составляет  %.2f см.' % np.std(pupils[:, 1]))
```

```
Стандартное отклонение роста от среднего в классе составляет  11.08 см.
```

В результате разброс в росте большинства учеников составляет более 11 см от среднего арифметического. Полученный результат показывает, что в этом классе одни дети уже начали активно расти, а другие ещё не вступили в период полового созревания.

7.7.3.4 Корреляция `corrcoef`

> **Корреляция** — статистическая взаимосвязь случайных величин. Мерой корреляции служит одноименный коэффициент, который показывает, насколько сильно связаны величины, он может быть положительным или отрицательным, принимает значение от -1 до 1. Отрицательный коэффициент говорит о том, что случайные величины связаны, но при увеличении одной из них вторая уменьшается. Если коэффициент положительный, то величины изменяются в одном направлении.

Аналитики часто оперируют в своей работе данной величиной и ошибаются, делая ложные выводы о данных. Связано это с тем, что не всегда наличие корреляции между двумя показателями говорит о том, что между ними есть причинно-следственная связь.

Задача. Определим, как связаны рост и вес школьников, посчитав коэффициент корреляции столбцов матрицы с ростом и весом.

Ввод [53]:

```
corr_hw = np.corrcoef(pupils[:, 1], pupils[:, 2])
print(corr_hw)
```

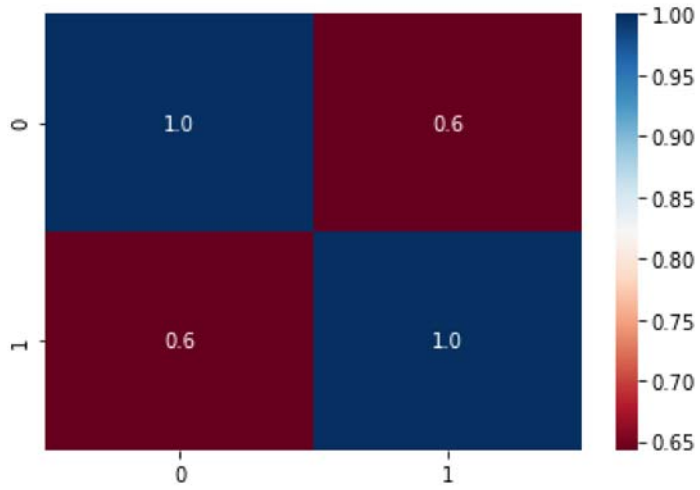
```
[[1.         0.64314431]
 [0.64314431 1.        ]]
```

Полученный нами коэффициент корреляции составляет чуть больше 0.64. Это говорит о том, что между ростом и весом школьников существует положительная связь: более рослые ученики обычно имеют более высокую массу тела.

С помощью библиотеки `seaborn` можно визуализировать корреляционную таблицу.

Ввод [94]:

```
import seaborn as sns
sns.heatmap(corr_hw, fmt='.1f', annot=True, cmap='RdBu');
```

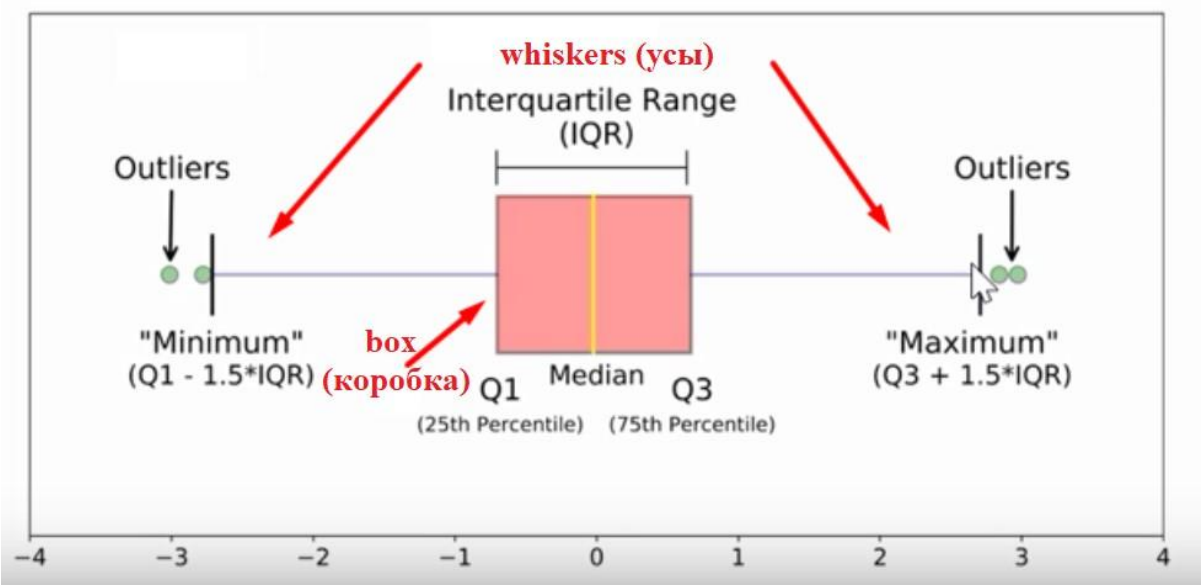


7.7.3.5 Визуализация числовых характеристик

> **Boxplot** (ящик с усами) - это график, который используется в описательной статистике, компактно изображающий одномерные статистики распределения переменный. Такой вид диаграммы в удобном формате показывает медиану, 25-процентный квантиль, 75-процентный квантиль (оба этих квантиля называют квартилями), минимальное значение и максимальное значение, а также выбросы. Расстояние между различными частями ящика позволяет определить степень разброса асимметрии данных и выявить выбросы.

Для визуализации распределения числовых признаков удобно использовать **ящик с усами** (или боксплот). Их преимущество состоит в том, что с помощью одной диаграммы можно представить основные описательные статистики: медиану, а также верхний (третий) и нижний (первый) квартили. Напомним, что данные показатели являются робастными (устойчивыми к выбросам), что позволяет

представлять с помощью этой диаграммы даже признаки с наличием выбросов. Причем, выбросы будут обозначены с помощью кружочков. То есть, по сути, с помощью построения боксплота можно даже выявить наличие выбросов в ваших данных.



Этот график показывает нам последовательное расположение по оси минимума, первого квартиля Q_1, медианы Q_2, третьего квартиля Q_3 и максимума.

Если в наборе данных есть выбросы (по английски *outliers*), они будут обозначены отдельными точками (или другими знаками) за пределами основной области графика. В таком случае линии границ «усов» окажутся не в точках минимума и максимума, а на границах, в пределах которых находятся простые наблюдения, не являющиеся выбросами.

Эти границы рассчитываются следующим образом:

- $Q_1 - 1.5IQR$ (нижний квартиль - 1,5 межквартильного размаха);
- $Q_3 + 1.5IQR$ (верхний квартиль + 1,5 межквартильного размаха).

Т.е. для норме наблюдения должны находится в пределах $[Q_1 - 1.5IQR; Q_3 + 1.5IQR]$. Максимум и минимум – это не максимальное и не минимальное значение. Выбросы не входят. Ящик с усами чистит выбросы.

Также преимущество боксплота заключается в том, что можно сравнивать на одном графике данные сразу для нескольких признаков.

Задача. Имеются данные о двух группах пользователей определенного сервиса–помощника в администрировании. Данные соответствуют общему количество времени, которое пользователи провели онлайн, используя соответствующее приложение (в часах за месяц). Постройте в Python на одном графике 2 параллельных боксплота.

Ввод [86]:

```
import pandas as pd
df = pd.DataFrame({
    'group_a': [72, 66, 10, 13, 63.75, 60.75, 60.75, 58.5, 58.5, 57, 57, 57,
                52.5, 49.5, 45.3, 45, 39, 28.95, 40.5, 31.5, 49, 90],
    'group_b': [71, 72, 60, 50, 57, 77, 81, 58, 62, 61, 73, 74, 76,
                65, 66, 67, 67, 67, 69, 63, 59, 77]
})

df.head()
```

Out[86]:

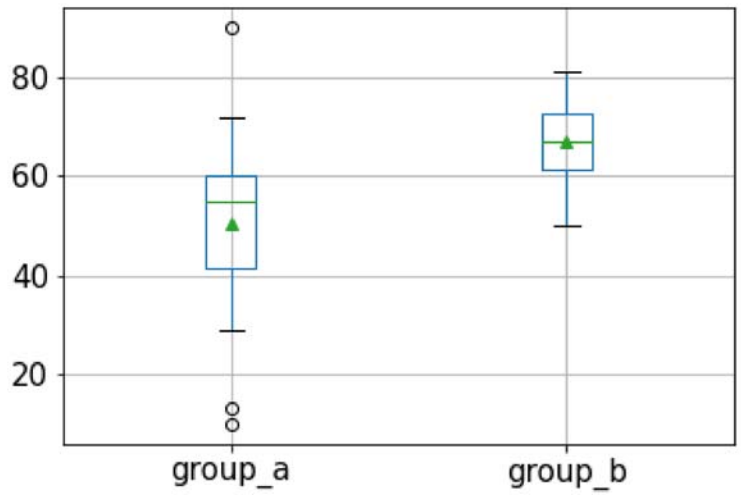
	group_a	group_b
0	72.00	71
1	66.00	72
2	10.00	60
3	13.00	50
4	63.75	57

Ввод [96]:

```
df.boxplot(column=['group_a', 'group_b'], showmeans=True, fontsize=15)
```

Out[96]:

```
<AxesSubplot:>
```



На основании этого графика можно сделать **выводы**:

1. В целом больше времени на сайте проводят пользователи группы $\mathrm{Б}$.
2. Разброс значений больше в группе A.

Ввод [88]:

```
print('Описательные статистики по group_a:')
print('   Среднее  {0:.1f}'.format(df.group_a.mean()))
print('   Медиана  {0:.1f}'.format(df.group_a.median()))
print('   Стандартное отклонение:  {0:.1f}'.format(df.group_a.std()))
print('   Размах  {0:.1f}'.format(df.group_a.max()-df.group_a.min()))
Q1 = df.group_a.quantile(q=0.25, interpolation='midpoint')
Q3 = df.group_a.quantile(q=0.75, interpolation='midpoint')
IQR = Q3 - Q1
print('   Межквартильный размах  {0:.1f}'.format(IQR))
min_bp = Q1 - 1.5*IQR
print('\n   Минимум для boxplot  {0:.1f}'.format(min_bp))
index_bot = df.group_a[df.group_a < (Q1 - 1.5*IQR)].index
print('   Нижние выбросы: ', df.group_a[df.group_a < (Q1 - 1.5*IQR)].values)
max_bp = Q3 + 1.5*IQR
print('\n   Максимум для boxplot  {0:.1f}'.format(max_bp))
index_top = df.group_a[df.group_a > (Q3 + 1.5*IQR)].index
print('   Верхние выбросы: ', df.group_a[df.group_a > (Q3 + 1.5*IQR)].values)
```

```
Описательные статистики по group_a:
   Среднее  50.2
   Медиана  54.8
   Стандартное отклонение:  18.3
   Размах  80.0
   Межквартильный размах  16.9

   Минимум для boxplot  17.4
   Нижние выбросы:  [10. 13.]

   Максимум для boxplot  84.9
   Верхние выбросы:  [90.]
```

Ввод [89]:

```
# Описательные статистики можно смотреть так:
df.describe()
```

Out[89]:

	group_a	group_b
count	22.000000	22.000000
mean	50.250000	66.909091
std	18.349134	7.782209
min	10.000000	50.000000
25%	41.625000	61.250000
50%	54.750000	67.000000
75%	60.187500	72.750000
max	90.000000	81.000000

§ 7.8 Тензоры (многомерные массивы)

Про них – не сегодня

§ 7.9 Примеры решения задач с использованием библиотеки Numpy

Пример. Дан двумерный массив. Сколько в массиве элементов, принадлежащих отрезку $[-3; 6]$.

Ввод [20]:

```
import numpy as np

ls = [[ 0,  -1,  2,  3,  4,  5,  0,  -7],
      [ 8,  9, -4, 11, 12, 13, -4, 15],
      [16, 7, 18, 9, 2, 2, 2, 23],
      [7, 25, 26, 9, 0, 0, 30, 31],
      [2, 3, 4, 5, 6, 7, 8, 2]]

a = np.array(ls)
print(len(a[(-3 <= a) & (a <= 6)]))
```

```
18
```

Пример. Дан двумерный массив. Найти минимальный нечетный элемент.

Ввод [37]:

```
import numpy as np

ls = [[ 0,  -1,  2,  3,  4,  5,  0,  -7],
      [ 8,  9, -4, 11, 12, 13, -4, 15],
      [16, 7, 18, 9, 2, 2, 2, 23],
      [7, 25, -26, 9, 0, 0, 30, -31],
      [2, 3, 4, 5, 6, 7, 8, 2]]

ls = [[ 0,  12,  2,  30], [ 80,  74, -4, 10],  [16, 18, 90, 20]]

a = np.array(ls)
try:
    print(min(a[a % 2 != 0]))
except:
    print('Нет нечетных элементов')
```

```
Нет нечетных элементов
```

Пример. Отсортировать элементы двумерного массива: 1) по строкам; 2) по столбцам.

Примечание. Сортировка выполняется с помощью функции `sort`, в качестве параметров функция получает сам массив, а также номер оси (для столбцов – 0, для строк – 1), элементы которой необходимо отсортировать.

Ввод [25]:

```
import numpy as np

ls = [[ 0,  -1,  2,  3,  4,  5,  0,  -7],
      [ 8,  9, -4, 11, 12, 13, -4, 15],
      [16, 7, 18, 9, 2, 2, 2, 23],
      [7, 25, 26, 9, 0, 0, 30, 31],
      [2, 3, 4, 5, 6, 7, 8, 2]]

a = np.array(ls)
print('Исходный массив:\n', a)
print('Сортировка по строкам:\n', np.sort(a, axis=1))
print('Сортировка по столбцам:\n', np.sort(a, axis=0))
```

```
Исходный массив:
 [[ 0 -1  2  3  4  5  0 -7]
 [ 8  9 -4 11 12 13 -4 15]
 [16  7 18  9  2  2  2 23]
 [ 7 25 26  9  0  0 30 31]
 [ 2  3  4  5  6  7  8  2]]
Сортировка по строкам:
 [[-7 -1  0  0  2  3  4  5]
 [-4 -4  8  9 11 12 13 15]
 [ 2  2  2  7  9 16 18 23]
 [ 0  0  7  9 25 26 30 31]
 [ 2  2  3  4  5  6  7  8]]
Сортировка по столбцам:
 [[ 0 -1 -4  3  0  0 -4 -7]
 [ 2  3  2  5  2  2  0  2]
 [ 7  7  4  9  4  5  2 15]
 [ 8  9 18  9  6  7  8 23]
 [16 25 26 11 12 13 30 31]]
```

Пример. Обнулить двумерный массив в шахматном порядке.

Ввод [10]:

```
import numpy as np

def f(i, j):
    return (i + j) % 2

ls = [[ 1,  -1,  2,  3,  4,  5,  5,  -7],
      [ 8,  9, -4, 11, 12, 13, -4, 15],
      [16, 7, 18, 9, 2, 2, 2, 23],
      [7, 25, 26, 9, 4, 5, 30, 31],
      [2, 3, 4, 5, 6, 7, 8, 2]]

a = np.array(ls)
mask = np.array(np.fromfunction(f, a.shape), dtype=bool)
a[mask]=0
print(a)
```

```
[[ 1  0  2  0  4  0  5  0]
 [ 0  9  0 11  0 13  0 15]
 [16  0 18  0  2  0  2  0]
 [ 0 25  0  9  0  5  0 31]
 [ 2  0  4  0  6  0  8  0]]
```

Пример. В квадратной матрице найти максимальный элемент, расположенный под побочной диагональю.

Ввод [17]:

```
import numpy as np

n = 5
def f(i, j):
    return (i + j) >= n

a = np.random.randint(-100, 100, (n, n))

mask = np.array(np.fromfunction(f, a.shape), dtype=bool)
print(a)
print(max(a[mask]))
```

```
[[-13 -56 -72 -23  92]
 [-62 -16  -3 -18  85]
 [ 39  98 -47 -38 -70]
 [ 65  61   0 -94 -93]
 [ 60  81  -8  45  75]]
85
```

Пример. В матрице найти сумму элементов по строкам и столбцам.

Ввод [26]:

```
import numpy as np

ls = [[ 0,  1,  0,  0,  0,  0,  0,  0],
      [ 1,  0, 0, 0, 0, 0, 1, 0],
      [0, 0, 1, 1, 1, 0, 0, 0],
      [1, 0, 1, 1, 0, 0, 1, 0],
      [1, 0, 0, 1, 1, 1, 1, 0]]

a = np.array(ls)
print('Исходный массив:\n', a)
print('Суммы по строкам:\n', np.sum(a, axis=1))

# Конечно, проходить по строкам никто не запрещал, но ЗАЧЕМ?
for row in a:
    print(sum(row), end=' ')

print('\nСуммы по столбцам:\n', np.sum(a, axis=0))

# И по стольбцам проходить никто не запрещал, но ЗАЧЕМ?
for col in range(a.shape[1]):
    print(sum(a[:, col]), end=' ')
```

```
Исходный массив:
 [[0 1 0 0 0 0 0 0]
 [1 0 0 0 0 0 1 0]
 [0 0 1 1 1 0 0 0]
 [1 0 1 1 0 0 1 0]
 [1 0 0 1 1 1 1 0]]
Суммы по строкам:
 [1 2 3 4 5]
1 2 3 4 5
Суммы по столбцам:
 [3 1 2 3 2 1 3 0]
3 1 2 3 2 1 3 0
```

Пример. Удалить строку и столбец, на пересечении которых стоит (первый) максимальный элемент.

Ввод [3]:

```
import numpy as np

ls = [[ 0,  0,  2,  3,  4,  5,  5, -7],
      [ 89,  9, -4, 11, 12, 13, -4, 15],
      [16, 7, 7, 93, 2, 2, 2, 23],
      [7, 25, 26, 9, 4, 5, 30, 31],
      [2, 3, 4, 5, 6, 7, 8, 9]]

a = np.array(ls)
print('Исходный массив:\n', a)
positon = a.argmax()
row = positon // a.shape[1] + 1
col = (positon - row * a.shape[1]) % a.shape[1] + 1

print(positon, row, col)
a = np.vstack((np.hstack((a[:(row-1), :(col-1)], a[:(row-1), col:])), np.hstack((a[row:,
print(a)
```

```
Исходный массив:
 [[ 0  0  2  3  4  5  5 -7]
 [89  9 -4 11 12 13 -4 15]
 [16  7  7 93  2  2  2 23]
 [ 7 25 26  9  4  5 30 31]
 [ 2  3  4  5  6  7  8  9]]
19 3 4
[[ 0  0  2  4  5  5 -7]
 [89  9 -4 12 13 -4 15]
 [ 7 25 26  4  5 30 31]
 [ 2  3  4  6  7  8  9]]
```

Пример. Поменять 2 строки в матрице

Ввод [14]:

```
a =  np.array([[ 0,  -1,  2,  3,  4,  5,  0,  -7],
               [ 8,  9, -4, 11, 12, 13, -4, 15],
               [16, 7, 18, 9, 2, 2, 2, 23],
               [7, 25, 26, 9, 0, 0, 30, 31],
               [2, 3, 4, 5, 6, 7, 8, 2]])
print(a)

a[[3, 1], :] = a[[1, 3], :]
print(a)
```

```
[[ 0 -1  2  3  4  5  0 -7]
 [ 8  9 -4 11 12 13 -4 15]
 [16  7 18  9  2  2  2 23]
 [ 7 25 26  9  0  0 30 31]
 [ 2  3  4  5  6  7  8  2]]
[[ 0 -1  2  3  4  5  0 -7]
 [ 7 25 26  9  0  0 30 31]
 [16  7 18  9  2  2  2 23]
 [ 8  9 -4 11 12 13 -4 15]
 [ 2  3  4  5  6  7  8  2]]
```

Пример. Поменять 2 столбца в матрице

Ввод [7]:

```
a = np.array([[1, 25, 4, 16],
              [36, 100, 64, 5],
              [2, -5, 3, -9],
              [-20, -52, -9, 1]])
print(a)

a[:, [0,3]] = a[:, [3,0]]
print(a)
```

```
[[  1  25   4  16]
 [ 36 100  64   5]
 [  2  -5   3  -9]
 [-20 -52  -9   1]]
[[ 16  25   4   1]
 [  5 100  64  36]
 [ -9  -5   3   2]
 [  1 -52  -9 -20]]
```

Пример. Найти наиболее частое значение в массиве

Ввод [62]:

```
a = np.random.randint(0, 10, 50)
print(a)
print(np.bincount(a).argmax())
```

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

Пример. Найти n наибольших значений в массиве

Ввод [68]:

```
a = np.array([23, 67, 89, 64, 45, 34, 2, -90, 56, -8, 34])
print(a)
n = 5
print (np.sort(a)[-n:])
```

```
[ 23  67  89  64  45  34   2 -90  56  -8  34]
[45 56 64 67 89]
```

§ 7.10 Задачи для самостоятельного выполнения

В программном решении вашей задачи требуется использовать функции библиотеки `Numpy` для обработки двумерного массива (введенного или тестового, заранее введенного). При выполнении заданий **НЕ использовать циклы** для перебора элементов.

Задача 1. Выполните над массивом предложенные действия: 1) найти минимальный элемент в массиве; 2) определить в массиве сумму элементов каждой строки; 3) найти наиболее частое значение в массиве (моду); 4) определить в массиве количество элементов, равных максимальному; 5) транспонируйте массив.

Задача 2. Выполните над массивом предложенные действия: 1) найти среднее значение из всех элементов массива; 2) определить в массиве максимальный элемент каждого столбца; 3) отсортировать матрицу по k-ому столбцу; 4) определить в массиве количество элементов, принадлежащих отрезку $[-5; 5]$; 5) обнулите крайние строки в массиве.

Задача 3. Выполните над массивом предложенные действия: 1) найти максимальный элемент в массиве; 2) определить в массиве среднее значение элементов каждого столбца; 3) поменять 2 строки в массиве; 4) увеличить в 2 раза значения элементов, которые равны минимальному; 5) обнулите крайние столбцы в массиве.

Задача 4. Выполните над массивом предложенные действия: 1) найти произведение всех элементов в массиве; 2) определить в массиве минимальный элемент каждого столбца; 3) найти n наибольших значений в массиве; 4) поменять знак у элементов, принадлежащих отрезку $[-5; 5]$; 5) замените абсолютными значениями отрицательные элементы в массиве.

Задача 5. Выполните над массивом предложенные действия: 1) найти сумму всех элементов в массиве; 2) определить в массиве максимальный элемент каждой строки; 3) поменять 2 столбца в массиве; 4) определить количество элементов массива, превышающих среднее значение; 5) обнулите четные столбцы в массиве.

Задача 6. Выполните над квадратным массивом предложенные действия: 1) найти максимальный элемент в массиве; 2) определить в массиве минимальный элемент каждой строки; 3) найти значение определителя; 4) определить количество элементов массива, ниже среднего значения; 5) обнулите четные строки в массиве.

Задача 7. Выполните над массивом предложенные действия: 1) найти максимальный элемент в массиве; 2) определить в массиве произведение элементов каждой строки; 3) посчитать ранг матрицы; 4) определить в массиве количество элементов, равных минимальному; 5) замените абсолютными значениями отрицательные элементы в массиве.

Задача 8. Выполните над массивом предложенные действия: 1) найти среднее значение из всех элементов массива; 2) определить в массиве максимальный элемент каждой строки; 3) найти n наименьших значений в массиве; 4) определить в массиве количество элементов, равных максимальному; 5) обнулите элементы, расположенные по контуру.

Задача 9. Выполните над квадратным массивом предложенные действия: 1) найти сумму четных элементов в массиве; 2) определить в массиве среднее арифметическое значение в каждой строке; 3) найти n наибольших значений в массиве; 4) определить в массиве количество элементов, равных нулю; 5) обнулите элементы под главной диагональю.

Задача 10. Выполните над квадратным массивом предложенные действия: 1) найти максимальный элемент в массиве; 2) определить в массиве минимальный элемент каждой строки; 3) найти значение определителя; 4) определить количество элементов массива, ниже среднего значения; 5) увеличить в 2 раза элементы над главной диагональю.

§ 7.11 Дополнительные материалы

Серия статей:

- NumPy в Python. Часть 1: https://habr.com/ru/post/352678/ (https://habr.com/ru/post/352678/)
- NumPy в Python. Часть 2: https://habr.com/ru/post/353416/ (https://habr.com/ru/post/353416/)
- NumPy в Python. Часть 3: https://habr.com/ru/post/413381/ (https://habr.com/ru/post/413381/)
- NumPy в Python. Часть 4: https://habr.com/ru/post/415373/ (https://habr.com/ru/post/415373/)

- Оригинальный текст на английском языке: https://sites.engineering.ucsb.edu/~shell/che210d/numpy.pdf (https://sites.engineering.ucsb.edu/~shell/che210d/numpy.pdf)
- Подборка ресурсов на русском языке: https://pythonworld.ru/numpy (https://pythonworld.ru/numpy)
- Подборка из 100 задач, с помощью которых можно закрепить навыки работы с библиотекой: https://pythonworld.ru/numpy/100-exercises.html (https://pythonworld.ru/numpy/100-exercises.html)

Официальная документация (на английском языке):

- Ресурс библиотеки NumPy: https://www.numpy.org/ (https://www.numpy.org/)
- Руководство для быстрого старта: https://www.numpy.org/devdocs/user/quickstart.html (https://www.numpy.org/devdocs/user/quickstart.html)

Еще ссылки:

- оригинальный tutorial: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html (https://docs.scipy.org/doc/numpy-dev/user/quickstart.html);
- полный список команд: https://docs.scipy.org/doc/numpy/genindex.html (https://docs.scipy.org/doc/numpy/genindex.html);
- перевод некоторых частей: https://pythonworld.ru/numpy/1.html (https://pythonworld.ru/numpy/1.html)
- https://pyprog.pro/reference_manual.html (https://pyprog.pro/reference_manual.html)

Класс `ndarray` имеет много методов. Перечень всех методов можно посмотреть командой

```
set(dir(a)) - set(dir(object))
```

Ввод [67]:

```
set(dir(a)) - set(dir(object))
```

Out[67]:

```
{'T',
 '__abs__',
 '__add__',
 '__and__',
 '__array__',
 '__array_finalize__',
 '__array_function__',
 '__array_interface__',
 '__array_prepare__',
 '__array_priority__',
 '__array_struct__',
 '__array_ufunc__',
 '__array_wrap__',
 '__bool__',
 '__complex__',
 '__contains__',
 '__copy__',
 '__deepcopy__',
```