Вычисления с помощью NumPy

Многомерные массивы в NumPy

Библиотека `numpy` поддерживает работу с многомерными массивами, в том числе, с матрицами, и очень хороша для научных расчётов. Библиотека написана не только на `Python`, но и на языке `C`, который является более низкоуровневым и работает значительно быстрее, поэтому расчёты в `numpy` производятся во много раз быстрее, чем если бы мы использовали для этого стандартные структуры данных из `Python`.

Установить библиотеку `numpy` можно следующим образом:

- Если вы используете `Python` в составе дистрибутива `Anaconda`, то достаточно в командной строке ввести: `conda install numpy`
- Если вы используете `Python` отдельно, то же самое можно сделать с помощью пакетного менеджера `pip`: `pip install numpy`

In [1]:
```
import numpy as np
```

Чтобы задать `numpy`-массив, достаточно задать обычный питоновский список `list`, а затем поместить его внутрь функции `np.array`:

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

print(a)
```

```
[1 2 3]
```

Внутрь этой функции можно подавать также питоновский кортеж `tuple`.

Проверим, к какому типу относится массив `a`:

In [3]:
```
type(a)
```

Out[3]:
```
numpy.ndarray
```

`ndarray` - сокращение от n-dimensional array (n-мерный массив).

В отличие от стандартных питоновских структур данных, в `numpy` массивы предпочитают данные одного типа. Например, если функция `np.array` вызывается от списка, содержащего как целые (`int`), так и дробные (`float`) значения, то в результирующем массиве все значения будут приведены к типу `float`. Аналогично, если в подаваемом списке есть хотя бы одна строка `str`, то в соответствующем массиве все значения будут приведены к типу `str`. Если мы хотим задать свой тип, к которому нужно привести данные, это можно сделать с помощью аргумента `dtype`:

In [4]:
```
a = np.array([1, 2, 3.6], dtype=str)

print(a)
```

```
['1' '2' '3.6']
```

Получить конкретный элемент массива можно теми же способами, что и в стандартных питоновских структурах данных - с помощью квадратных скобок. В `numpy`, как и во всём питоне, индексация начинается с нуля. Например, получить второй элемент из массива `a` (т.е. элемент с индексом 1) можно так:

In [5]:
```
a[1]
```

Out[5]:
```
'2'
```

Также в `numpy` массивах можно использовать отрицательную индексацию и делать срезы, как и в стандартных

списках из питона:

In [6]:
```
a[-1]
```

Out[6]:
```
'3.6'
```

In [7]:
```
a[1:3]
```

Out[7]:
```
array(['2', '3.6'], dtype='<U3')
```

Двумерные массивы

Пока что мы работали лишь с одномерными массивами. Также в `numpy` можно задать и многомерные массивы. Например, двумерный массив - это массив, каждый элемент из которого - это снова массив.

Для `numpy` -массива `a` можно проверить его размерность с помощью атрибута `ndim` и форму с помощью атрибута `shape` :

In [8]:
```
print("Размерность a: {}".format(a.ndim))

print("Форма a: {}".format(a.shape))
```

```
Размерность a: 1
Форма a: (3,)
```

В этом случае размерность равна 1, а `shape` возвращает кортеж из одного элемента. Зададим теперь двумерный массив:

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

print(A)

print("Размерность A: {}".format(A.ndim))

print("Форма A: {}".format(A.shape))
```

```
[[1 2 3 1]
 [4 5 6 4]
 [7 8 9 7]]
Размерность A: 2
Форма A: (3, 4)
```

Атрибут `shape` - это всегда кортеж, размер которого равен размерности массива. Каждый элемент этого кортежа - это размер в каждом измерении. Например, у нашей матрицы A, судя по этому атрибуту, 3 строки и 4 столбца.

С помощью атрибута `size` можно увидеть общее количество элементов массива:

In [10]:
```
A.size
```

Out[10]:
```
12
```

В случае вложенных друг в друга стандартных питоновских списков `list` , чтобы получить конкретный элемент массива, нужно использовать несколько пар квадратных скобок: `A[0][0]` . В `numpy` массивы также поддерживают такую запись, однако, здесь есть и более удобный вариант - просто писать индексы через запятую:

In [11]:
```
A[0, 0]
```

Out[11]:
```
1
```

Это же работает и в случае отрицательной индексации и в случае срезов:

In [12]:
```
A[-1, -2]
```

Out[12]:
```
9
```

In [13]:
```
A[1:, :3]
```

Out[13]:
```
array([[4, 5, 6],
       [7, 8, 9]])
```

В случае срезов для `numpy`-массивов важно отметить, что, записывая срез `numpy`-массива, мы ничего нового не создаём, мы лишь получаем *представление* (*view*) - ссылку на какие-то отдельные элементы оригинального массива. Это означает, что если мы "создали" срез из `numpy`-массива, а затем поменяли в нём что-то - эти изменения коснутся и оригинального массива:

In [14]:
```
print(A)
```

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

In [15]:
```
B = A[1:, :3]

print(B)
```

```
[[4 5 6]
 [7 8 9]]
```

In [16]:
```
B[0, 0] = -4

print(A)
```

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

Наоборот, если мы меняем значения в оригинальном массиве, они коснутся и всех его представлений, в которых используются эти значения:

In [17]:
```
A[2, 0] = -7

print(B)
```

```
[[-4  5  6]
 [-7  8  9]]
```

In [18]:
```
b = np.array([[3],
     [1],
     [2]])
```

Если мы хотим всего этого избежать и создать действительно новый массив, нужно использовать метод `copy`:

In [19]:
```
C = A[1:3, 2:4].copy()

print(C)
```

```
[[6 4]
 [9 7]]
```

In [20]:
```
C[0, 0] = -6

print(A)
```

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

Типы данных в NumPy

Самыми распространёнными типами в `numpy` являются два целочисленных типа: `np.int32` и `np.int64` и два дробных типа: `np.float32` и `np.float64`. Они применяются для, соответственно, 32-битных и 64-битных чисел. Последние требуют вдвое больше памяти, чем первые, однако, если вы знаете, что в вашем массиве, например, используются целые числа, которые по модулю больше, чем $2 \cdot 10^9$, то стоит использовать `np.int64`.

Применение NumPy в линейной алгебре

Векторы

Для начала разберёмся с тем, как с помощью `numpy` работать с векторами. Зададим несколько векторов:

In [21]:
```
a = np.array([0, 1, 2, 3, 4])
b = np.array([5, 6, 7, 8, 9])
```

Сейчас мы ограничимся случаем, когда все векторы имеют одинаковый размер.

Сложение векторов можно выполнять, просто складывая массивы. Отметим, что это поведение отличается от сложения обычных питоновских списков `list`: списки в таком случае просто склеиваются в один.

In [22]:
```
a_ = [0, 1, 2, 3, 4]
b_ = [5, 6, 7, 8, 9]

c_ = a_ + b_

print(c_)
```

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

В `numpy` же массивы в результате такой операции складываются поэлементно:

In [23]:
```
c = a + b

print(c)
```

```
[ 5  7  9 11 13]
```

Также массивы `numpy` можно складывать с помощью функции `np.add`:

In [24]:
```
c = np.add(a, b)

print(c)
```

```
[ 5  7  9 11 13]
```

Аналогично, есть два способа вычитать векторы друг из друга:

- `d = a - b`
- `d = np.subtract(a, b)`

Для умножения вектора на скаляр также можно пользоваться достаточно естественной записью:

In [25]:
```
e = a * 3

print(e)
```

```
[ 0  3  6  9 12]
```

При этом скаляр может быть каким угодно. При умножении на скаляр, каждая координата вектора умножается на этот скаляр.

Это поведение тоже отличается от поведения питоновских списков `list` при умножении на число. Последние при этом дублируются заданное количество раз. Скаляр здесь, кроме того, может быть только положительным целым.

In [26]:
```
[1, 2, 3] * 3
```

Out[26]:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Вот ещё несколько способов умножить вектор на скаляр в `numpy`:

- с помощью функции `np.multiply`: `e = np.multiply(a, 3)`
- с помощью функции `np.dot`: `e = np.dot(a, 3)`
- с помощью метода `a.dot`, который есть у любого `numpy`-массива: `e = a.dot(3)`

Функция `np.dot` (так же, как и метод `a.dot`), кроме того, может вычислять скалярное произведение векторов, а также произведение матриц (об этом чуть позже).

Посчитаем скалярное произведение векторов `a` и `b`. Напомним, что чтобы вычислить скалярное произведение двух векторов, нужно попарно перемножить их координаты (первую с первой, вторую со второй и т.д.), а затем сложить результаты.

In [27]:
```
sp = a.dot(b)

print(sp)
```

```
80
```

Также скалярное произведение векторов можно вычислять с помощью оператора `@`:

In [28]:
```
sp = a @ b

print(sp)
```

```
80
```

Матрицы

Разберёмся теперь, как в `numpy` работать с матрицами. Зададим пару матриц:

In [29]:
```
A = np.array([[0, 1],
              [2, 3],
              [4, 5]])

B = np.array([[6, 7],
              [8, 9],
              [10, 11]])
```

Матрицы одинакового размера можно складывать и вычитать. Как и с векторами, это можно делать с помощью операторов `+` и `-`, а также с помощью функций `np.add` и `np.subtract`.

In [30]:
```
C = A + B

print(C)
```

```
[[ 6  8]
 [10 12]
 [14 16]]
```

In [31]:
```
D = A - B

print(D)
```

```
[[-6 -6]
 [-6 -6]
 [-6 -6]]
```

Матрицу любого размера можно умножать на скаляр. Делается это так же, как и в случае векторов:

In [32]:
```
E = A * 3

print(E)
```

```
[[ 0  3]
 [ 6  9]
 [12 15]]
```

Умножение матриц

Матрицы A и B можно умножить друг на друга, если *число столбцов* первой матрицы равняется *числу строк* второй матрицы. То есть если A - матрица размера $n \times k$, то матрица B должна иметь размер $k \times m$ для некоторого m.

В таком случае результатом умножения будет матрица C размера $n \times m$ (т.е. у неё будет строк как у первой матрицы, а столбцов - как у второй).

Рассмотрим простейший случай: умножение строки (матрицы размера $1 \times k$) на столбец (матрицу размера $k \times 1$). Как мы уже выяснили, в результате получится матрица размера 1×1, т.е. число. Что это за число?

Чтобы посчитать это число, нужно элементы из строки и столбца попарно перемножить (первый с первым, второй со вторым и т.д.), а затем сложить результаты. Это очень похоже на скалярное произведение векторов.

Например,

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} = 1 \cdot 1 + 2 \cdot 0 + 3 \cdot (-1) = -2.$$

Вернёмся к общему случаю - умножению матрицы размера $n \times k$ на матрицу размера $k \times m$. Мы уже поняли, что это будет матрица размера $n \times m$. Как "заполнить" эту матрицу?

Чтобы получить число, стоящее в этой матрице на позиции (i, j), нужно умножить i-ю строку первой матрицы на j-й столбец второй матрицы (так, как мы это делали выше).

Например,

$$\begin{pmatrix} 1 & 0 & -1 \\ 3 & 5 & -4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + 0 \cdot 4 + (-1) \cdot 7 & 1 \cdot 2 + 0 \cdot 5 + (-1) \cdot 8 & 1 \cdot 3 + 0 \cdot 6 + (-1) \cdot 9 \\ 3 \cdot 1 + 5 \cdot 4 + (-4) \cdot 7 & 3 \cdot 2 + 5 \cdot 5 + (-4) \cdot 8 & 3 \cdot 3 + 5 \cdot 6 + (-4) \cdot 9 \end{pmatrix} = ($$

Чтобы выполнять умножение матриц в библиотеке `numpy`, будем пользоваться уже знакомой функцией `np.dot`, либо методом `A.dot`:

In [33]:
```
A = np.array([[1, 0, -1],
              [3, 5, -4]])

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

C = A.dot(B)

print(C)
```

```
[[-6 -6 -6]
 [-5 -1  3]]
```

Если перемножаемые матрицы являются квадратными, то результат их умножения будет снова квадратной матрицей, причём, того же размера. Это означает, что квадратную матрицу можно возводить в степень. В `numpy` это можно делать с помощью функции `matrix_power` из модуля `numpy.linalg`:

In [34]:
```
D = np.linalg.matrix_power(B, 3)

print(D)
```

```
[[ 468  576  684]
 [1062 1305 1548]
 [1656 2034 2412]]
```

Единичная и транспонированная матрица

Единичной матрицей называется квадратная матрица, у которого на главной диагонали стоят 1, а в остальных местах - 0. (Под *главной диагональю* мы понимаем диагональ матрицы, которая начинается в левом верхнем углу и заканчивается в правом нижнем.) Единичную матрицу можно задать с помощью функции `np.eye`:

In [35]:
```
I = np.eye(3)

print(I)
```

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

При умножении любой матрицы на единичную матрицу подходящего размера, результатом будет та же самая матрица:

In [36]:
```
print(A)
```

```
[[ 1  0 -1]
 [ 3  5 -4]]
```

In [37]:
```
E = A.dot(I)

print(E)
```

```
[[ 1.  0. -1.]
 [ 3.  5. -4.]]
```

Пусть дана матрица A. *Транспонированной матрицей* называется матрица $A^\top$, полученная "отражением" матрицы A относительно её главной диагонали. Другими словами, столбцы матрицы A становятся строками матрицы $A^\top$, а строки матрицы A - столбцами матрицы $A^\top$.

Вот несколько способов посчитать транспонированную матрицу в `numpy`:

- с помощью функции `np.transpose`: `A_t = np.transpose(A)`
- с помощью метода `A.transpose`: `A_t = A.transpose()`
- с помощью атрибута `A.T`: `A_t = A.T`

In [38]:
```
A_t = A.T

print(A_t)
```

```
[[ 1  3]
 [ 0  5]
 [-1 -4]]
```

Определитель и ранг матрицы

Если матрица квадратная, то мы можем посчитать её *определитель*. Определитель матрицы - это число, которое в каком-то смысле "определяет" её свойства. Например, обратную матрицу можно посчитать только для матрицы, определитель которой не равен 0 (по аналогии с тем, что делить можно только на числа, не равные 0).

Посчитать определитель можно с помощью функции `det` из модуля `numpy.linalg`:

In [39]:
```
d = np.linalg.det(B)

print(d)
```

```
0.0
```

Также с помощью функции `matrix_rank` из модуля `numpy.linalg` можно посчитать *ранг* матрицы. Ранг матрицы - это число линейно независимых строк данной матрицы.

In [40]:
```
r = np.linalg.matrix_rank(B)

print(r)
```

```
2
```

Если матрица квадратная, то её ранг и определитель связаны следующим образом: определитель матрицы отличен от 0 тогда и только тогда, когда все её строки являются линейно независимыми. Это, в свою очередь, означает, что её ранг равен её размеру.

Например, ранг матрицы B из примера выше равен 2, при этом её размер равен 3. Это значит, что не все её строки являются линейно независимыми, поэтому её определитель равен 0.

В отличие от определителя, ранг можно считать и для матрицы, которая не является квадратной. Посчитаем ранг матрицы A размера 2×3, определённой выше:

In [41]:
```
r1 = np.linalg.matrix_rank(A)

print(r1)
```

```
2
```

Итак, если определитель квадратной матрицы не равен 0, то мы можем посчитать для неё *обратную матрицу*. Это матрица, которая при умножении на исходную матрицу даёт единичную матрицу:

$$A \cdot A^{-1} = I$$

Обратную матрицу можно посчитать с помощью функции `inv` из модуля `numpy.linalg` :

In [42]:
```
F = np.array([[7, 4, 5],
              [8, 3, 2],
              [6, 10, 12]])

print(np.linalg.det(F))
```

```
85.99999999999989
```

In [43]:
```
F_inv = np.linalg.inv(F)

print(F_inv)
```

```
[[ 0.18604651  0.02325581 -0.08139535]
 [-0.97674419  0.62790698  0.30232558]
 [ 0.72093023 -0.53488372 -0.12790698]]
```

Проверим, что условие действительно выполняется:

In [44]:
```
print(F.dot(F_inv))
```

```
[[ 1.00000000e+00 -5.55111512e-16 -2.77555756e-17]
 [ 2.22044605e-16  1.00000000e+00  0.00000000e+00]
 [ 8.88178420e-16 -4.44089210e-16  1.00000000e+00]]
```

Если определитель матрицы A равен d, то определитель обратной матрицы всегда будет равен $1/d$. Именно поэтому матрицы с определителем, равным 0, обращать нельзя.

In [45]:
```
F_d = np.linalg.det(F)

F_inv_d = np.linalg.det(F_inv)

print(F_d * F_inv_d)
```

```
0.9999999999999991
```

Генерирование массивов с заданными свойствами

Здесь мы рассмотрим способы задавать массивы различных размеров.

Функция `np.zeros` позволяет создать массив любой формы, состоящий из нулей:

In [46]:
```
a = np.zeros((3, 4))

print(a)
```

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

Аналогично, функция `np.ones` вернёт массив заданной формы, состоящий из единиц:

In [47]:
```
b = np.ones((3, 4))

print(b)
```

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

Последовательности чисел можно создавать с помощью функции `np.arange` . Вот три способа использовать эту функцию:

- Если задать только один аргумент, то вернётся последовательность чисел от 0 до этого аргумента невключительно:

In [48]:
```
ar1 = np.arange(10)

print(ar1)
```

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

- Если подать два аргумента, то вернётся последовательность чисел от первого аргумента до второго (включая первый, не включая второй):

In [49]:
```
ar2 = np.arange(2, 13)

print(ar2)
```

```
[ 2  3  4  5  6  7  8  9 10 11 12]
```

- Если подать три аргумента, то третий аргумент будет обозначать шаг, с которым берутся числа в последовательности:

In [50]:
```
ar3 = np.arange(2, 13, 2)

print(ar3)
```

```
[ 2  4  6  8 10 12]
```

Отметим, что шаг в функции `np.arange` может быть дробным:

In [51]:
```
ar4 = np.arange(2, 3, 0.1)

print(ar4)
```

```
[2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9]
```

Если шаг отрицательный, то последовательность будет возвращена в обратном порядке:

In [52]:
```
ar5 = np.arange(3, 2, -0.1)

print(ar5)
```

```
[3.  2.9 2.8 2.7 2.6 2.5 2.4 2.3 2.2 2.1]
```

Ещё одна полезная функция здесь - это функция `np.linspace` . Она позволяет вернуть заданное количество значений, равномерно расставленных между заданными началом и концом отрезка. Отметим, что здесь и левый, и првый концы отрезка включаются в массив:

In [53]:
```
c = np.linspace(2, 3, 10)

print(c)
```

```
[2.         2.11111111 2.22222222 2.33333333 2.44444444 2.55555556
 2.66666667 2.77777778 2.88888889 3.        ]
```

Функция `np.logspace` имеет похожий эффект, отличие лишь в том, что в качестве начала и конца отрезка мы подаём не сами числа, а степени числа 10. Например, в ячейке ниже мы задаём массив, содержащий 4 значения, расставленных равномерно в пределах от $10^0 = 1$ до $10^3 = 1000$.

In [54]:
```
d = np.logspace(0, 3, 4)

print(d)
```

```
[   1.   10.  100. 1000.]
```

Массивы случайных значений

Функция `sample` из модуля `numpy.random` возвращает массив заданной формы, состоящий из чисел, взятых из равномерного распределения на отрезке $[0, 1)$.

In [55]:
```
a = np.random.sample((3, 4))

print(a)
```

```
[[0.93997522 0.3649222  0.1745136  0.57145376]
 [0.18031654 0.25954888 0.29623503 0.85737173]
 [0.86543438 0.62623889 0.6032358  0.14402853]]
```

Отметим, что в эту и другие представленные ниже функции можно подавать также не кортеж, а какое-то одно целое число. В этом случае вернётся одномерный массив заданного размера. Также в эти функции можно не подавать аргументы вовсе - в этом случае вернётся лишь одно число.

In [56]:
```
print("Одно значение: {}".format(np.random.sample()))

print("Три значения: {}".format(np.random.sample(3)))
```

```
Одно значение: 0.8883357156837364
Три значения: [0.99517333 0.51290899 0.83561873]
```

Функция `randn` из модуля `numpy.random` возвращает аналогичный массив, но уже взятый из нормального распределения (со средним 0 и среднеквадратическим отклонением 1):

In [57]:
```
b = np.random.randn(3, 4)

print(b)
```

```
[[-0.14194228  0.64165498  0.04953172 -0.41440644]
 [ 1.0850997  -0.87161821  0.54785693  0.18092246]
 [-0.03356476 -1.00275702 -0.62416943  1.55027964]]
```

Обратите внимание, что эта функция получает на вход не кортеж `tuple`, а просто последовательность размеров по каждому измерению.

Функция `randint` возвращает массив из целых чисел в указанном диапазоне:

In [58]:
```
c = np.random.randint(0, 100, (3, 4))

print(c)
```

```
[[78 38 56 21]
 [18 12 80 22]
 [33 71 35 22]]
```

Функция `choice` возвращает случайно выбранные элементы из заранее заданного массива:

In [59]:
```
A = np.arange(-10, 0)

d = np.random.choice(A, (3, 4))

print(d)
```

```
[[ -1  -9  -9  -3]
 [ -9  -5  -9  -5]
 [ -6  -6  -7 -10]]
```

Изменение размеров массива

Библиотека `numpy` предоставляет функционал для удобного изменения размера массивов.

Например, рассмотрим одномерный массив с 12 элементами:

In [60]:
```
ar = np.arange(12)

print(ar)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Если нам нужно расположить эти значения в двумерном массиве, мы можем сделать это с помощью функции `np.reshape` или метода `ar.reshape`:

In [61]:
```
a = ar.reshape(3, 4)

print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Ясно, что при этом число элементов получаемого массива должно совпадать с числом элементов в оригинальном массиве. Например, следующая попытка посчитать функцию закончится ошибкой:

In [62]:
```
b = ar.reshape(3, 5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-62-92cdbc10f6ee> in <module>
----> 1 b = ar. reshape(3, 5)

ValueError: cannot reshape array of size 12 into shape (3,5)
```

Если мы знаем количество строк, которое хотим получить, но не знаем количество столбцов, в качестве второго аргумента можно передать число -1. Если наоборот мы знаем лишь количество столбцов, можно передать -1 в качестве первого аргумента.

In [63]:
```
b = ar.reshape(3, -1)

print(b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Метод `ar.reshape` не меняет сам массив `ar`, он лишь возвращает новый. Есть также метод `ar.resize`, который делает то же самое, что и `ar.reshape`, но не возвращает ничего и меняет исходный массив:

In [64]:
```
ar.resize(3, 4)

print(ar)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Обратно, чтобы получить из многомерного массива одномерный, можно воспользоваться методом `ar.flatten`:

In [65]:
```
c = ar.flatten()

print(c)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Соединение массивов

Рассмотрим два массива и разберёмся с тем, как их можно соединить в один.

In [66]:
```
a = np.zeros((2, 3))

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

Мы можем соединить эти массивы вертикально (т.е. дописать один под другим). Вот несколько способов это сделать:

- с помощью функции `np.vstack`: `c = np.vstack((a, b))` (получает на вход кортеж из массивов)
- с помощью функции `np.concatenate`: `c = np.concatenate((a, b), axis=0)` (тоже получает на вход кортеж, также нужно указать, вдоль какой оси производится конкатенация)

In [67]:
```
c = np.vstack((a, b))

print(c)
```

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

Также несколько способов это соединить массивы горизонтально (т.е. дописать один правее другого):

- с помощью функции `np.hstack`: `c = np.hstack((a, b))`
- с помощью функции `np.concatenate`: `c = np.concatenate((a, b), axis=1)` (производится теперь вдоль оси 1)

In [68]:
```
d = np.concatenate((a, b), axis=1)

print(d)
```

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

Наконец, два двумерных массива можно соединить *в глубину* (т.е. вдоль новой третьей оси) с помощью функции `np.dstack`:

In [69]:
```
e = np.dstack((a, b))

print(e)
```

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

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

Функции для работы с данными

Библиотека `numpy` предлагает удобный функционал для выбора данных из массива. Рассмотрим массив из 10 случайных целых значений от 0 до 19:

In [70]:
```
a = np.random.randint(0, 20, 10)

print(a)
```

```
[19  9 11  3 14 14 15 16  8  5]
```

Допустим, мы хотим выбрать все значения этого массива, которые больше 10. Вот как это можно сделать:

In [71]:
```
b = a[a > 10]

print(b)
```

```
[19 11 14 14 15 16]
```

Свойства можно комбинировать, используя логические операторы "и" (обозначается символом $\&$), "или" (символ $|$) и оператор отрицания "не" (символ $\sim$). При этом каждое условие необходимо поставить в круглые скобки:

In [72]:
```
c = a[(a > 0) & (a % 2 == 0)]

print(c)
```

```
[14 14 16  8]
```

Такая конструкция в `numpy` называется *булевой индексацией*. Разберёмся с ней поподробнее. Что из себя представляет объект `a > 0`?

In [73]:
```
print(a > 10)
```

```
[ True False  True False  True  True  True  True False False]
```

Как мы видим, это просто `numpy`-массив из булевых значений `True` и `False`. Когда мы подставляем такой массив в качестве *индекса* массива `a`, нам возвращаются все элементы, на позиции которых в этом массиве стоит значение `True`.

Можно просто создать такой массив вручную и передать его в качестве индекса:

In [74]:
```
ind = np.array([True, False, True, True, False, False, False, True, True, False])

print(a[ind])
```

```
[19 11  3 16  8]
```

Другой способ выбрать значения из массива - с помощью функции `np.where`. Она берёт массив из булевых значений и возвращает *индексы* истинных значений:

In [75]:
```
ind1 = np.where(a > 10)

print(ind1)
```

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

Такой список индексов можно также передать в массив `a` чтобы получить конкретные значения:

In [76]:
```
d = a[ind1]

print(d)
```

```
[19 11 14 14 15 16]
```

То же самое можно сделать и вручную: передать в квадратные скобки массива `a` какой-нибудь список из индексов:

In [77]:
```
e = a[[0, 4, 7]]

print(e)
```

```
[19 14 16]
```

Отметим также, что если массив `a` является многомерным, то чтобы выбрать таким образом из него значения, нужно указать внутри квадратных скобок через запятую столько списков, сколько имеется у массива измерений:

In [78]:
```
a.resize((5, 2))

print(a)
```

```
[[19  9]
 [11  3]
 [14 14]
 [15 16]
 [ 8  5]]
```

In [79]:
```
f = a[[1, 4], :]

print(f)
```

```
[[11  3]
 [ 8  5]]
```

Сортировка

Рассмотрим двумерный массив:

In [80]:
```
a = np.random.randint(0, 6, (3, 4))

print(a)
```

```
[[2 4 3 3]
 [2 2 2 4]
 [5 2 0 3]]
```

Допустим, мы хотим отсортировать строки этого массива по второму столбцу. Мы можем сделать это вручную, задав

индексы строк в нужном нам порядке:

In [81]:
```
b = a[[1, 2, 0], :]

print(b)
```

```
[[2 2 2 4]
 [5 2 0 3]
 [2 4 3 3]]
```

Этот процесс можно автоматизировать с помощью метода `a.argsort`. Данный метод возвращает массив из индексов массива `a` в порядке их возрастания по заданной оси:

In [82]:
```
ind = a.argsort(axis=0)

print(ind)
```

```
[[0 1 2 0]
 [1 2 1 2]
 [2 0 0 1]]
```

В каждом столбце этого массива стоят индексы строк массива `a`, расположенные в том порядке, в котором они бы отсортировали данный столбец по возрастанию. Автоматизируем процесс сортировки массива `a` по второму столбцу. Для этого нужно получить второй столбец из массива, полученного с помощью метода `a.argsort`:

In [83]:
```
ind1 = a[:, 1].argsort()

print(ind1)
```

```
[1 2 0]
```

Итоговая конструкция будет выглядеть так:

In [84]:
```
c = a[a[:, 1].argsort(), :]

print(c)
```

```
[[2 2 2 4]
 [5 2 0 3]
 [2 4 3 3]]
```

Перемешивание

Иногда оказывается нужно перемешать значения массива. Это можно сделать с помощью функции `shuffle` из модуля `numpy.random`. Эта функция ничего не возвращает, лишь перемешивает случайным образом элементы данного массива. Отметим, что она перемешивает массив только в первом измерении. Другими словами, если массив двумерный, она лишь переставит его строки местами. Содержимое самих строк при этом не изменится:

In [85]:
```
np.random.shuffle(c)

print(c)
```

```
[[2 4 3 3]
 [5 2 0 3]
 [2 2 2 4]]
```

Математические операции над массивами

Некоторые математические операции можно выполнять с массивами целиком. Например, мы уже знаем, что массивы можно умножать на число и что массивы одинаковой формы можно складывать.

In [86]:
```
a = np.arange(0, 6).reshape(2, 3)
b = np.arange(6, 12).reshape(2, 3)

print(a)
print(b)
```

```
[[0 1 2]
 [3 4 5]]
[[ 6  7  8]
 [ 9 10 11]]
```

In [87]:
```
print(a + b)
```

```
[[ 6  8 10]
 [12 14 16]]
```

In [88]:
```
print(a * 2)
```

```
[[ 0  2  4]
 [ 6  8 10]]
```

К массивам можно также прибавлять числа - в этом случае к каждому элементу массива прибавляется число:

In [89]:
```
print(a + 1)
```

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

Массивы одинакового размера можно поэлементно умножать. (Важно не путать с матричным умножением.)

In [90]:
```
print(a * b)
```

```
[[ 0  7 16]
 [27 40 55]]
```

С помощью метода `a.sum` можно посчитать сумму всех значений массива. Если указать в этом методе ось `axis`, сумма будет посчитана только вдоль этой оси:

In [91]:
```
print("Сумма всех элементов: {}".format(a.sum()))

print('Сумма по столбцам ("вдоль" строк): {}'.format(a.sum(axis=0)))

print('Сумма по строкам ("вдоль" столбцов): {}'.format(a.sum(axis=1)))
```

```
Сумма всех элементов: 15
Сумма по столбцам ("вдоль" строк): [3 5 7]
Сумма по строкам ("вдоль" столбцов): [ 3 12]
```

Broadcasting

В определённых случаях мы можем выполнять операции сложения и умножения над матрицами разных размеров. Концепция *broadcasting* заключается в том, что в некоторых случаях интерпретатор "догадывается", что одну массив надо в каком-то измерении "растянуть" до соответствия со вторым массивом. Рассмотрим массив размера 3×2:

In [92]:
```
a = np.array([[2, 5],
              [3, 4],
              [6, 1]])
```

а также одномерный массив размера 2:

In [93]:
```
b = np.array([1, 2])
```

Если мы попытаемся их сложить, интерпретатор заметит, что у них совпадает одно из измерений: у них обоих 2 столбца. Поэтому интерпретатор как бы "растянет" массив `b` до размера 2×3 и прибавит его к массиву `a`:

In [94]:
```
c = a + b

print(c)
```

```
[[3 7]
 [4 6]
 [7 3]]
```

На самом деле здесь строка `b` просто прибавится к каждой строке массива `a`.

Аналогично можно поступить и со столбцами:

In [95]:
```
d = np.array([[0],
              [1],
              [-1]])

e = a + d

print(e)
```

```
[[2 5]
 [4 5]
 [5 0]]
```

Интерпретатор заметит, что у этих массивов совпадает число строк, поэтому "растянет" массив `d` до размера массива `a`. Попросту говоря, столбец `b` прибавится к каждому из столбцов массива `a`.

Имеются и более сложные конструкции broadcasting, о них можно почитать здесь (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html).

В случае, когда размеры массивов согласовать не удаётся, выпадает ошибка:

In [96]:
```
f = np.array([0, 1, -1])

a + f
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-96-bbb4a55e5786> in <module>
      1 f = np.array([0, 1, -1])
      2
----> 3 a + f

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

Статистические функции

Вот несколько методов, позволяющих вычислить различные статистики массива `a`:

- `a.min` - минимальное значение
- `a.max` - максимальное значение
- `a.mean` - среднее значение
- `a.std` - среднее квадратическое отклонение

Все эти значения считаются по всему массиву, либо вдоль определённой оси, если задан параметр `axis`.

In [97]:
```
print(a)
```

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

In [98]:
```
print("Минимальное значение: {}".format(a.min()))

print("Средние значения строк: {}".format(a.mean(axis=1)))

print("Средние квадратические отклонения столбцов: {}".format(a.mean(axis=0)))
```

```
Минимальное значение: 1
Средние значения строк: [3.5 3.5 3.5]
Средние квадратические отклонения столбцов: [3.66666667 3.33333333]
```

Запись и чтение массивов из файла

Массивы `numpy` можно сохранять в файлы с расширением `.npy` и читать из таких файлов.

Для записи массива в файл используется функция `np.save`:

In [99]:
```
np.save("a.npy", a)
```

Для чтения из файла используется функция `np.load`:

In [100]:
```
a = np.load("a.npy")
```