Перед началом работы

В Colab, среде которой мы работаем, все файлы, загруженные нами, будут удалены после завершения работы. Чтобы сохранить все файлы мы сделали GitHub репозиторий и будем его каждый раз подключать заново. Для этого выполняем команду

```
!git clone https://github.com/neuralcomputer/ML_School.git
```

Все данные скачаются, загрузятся в среду в папку ML_School/

Всегда будем писать такой путь к файлам.

Если вы работаете на локальном компьютере, то необходимо писать путь для вашего компьютера.

```
!git clone https://github.com/neuralcomputer/ML_School.git
```

```
Cloning into 'ML_School'...
remote: Enumerating objects: 94, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 94 (delta 5), reused 0 (delta 0), pack-reused 79 (from 1)
Receiving objects: 100% (94/94), 33.83 MiB | 16.97 MiB/s, done.
Resolving deltas: 100% (29/29), done.
```

Математика в Python

Приветствую! Мы начинаем изучение замечательной области машинного обучения, которая позволяет обучать компьютеры решать многие задачи образом, похожим на то, как обучается человек. В курсе мы будем пользоваться многими библиотеками замечательного языка Python, с которым вы немного знакомы. И прежде чем начинать изучать машинное обучение мы познакомимся со способами работы и отображения данных.

Сегодня мы посмотрим, как можно выполнять математические операции в Python, посмотрим на разные типы *объектов* и переменных.

Объект - это то, что действительно хранится в определенном месте памяти компьютера. Конечно, там хранятся только последовательности из 0 и 1, но мы можем по-разному *представлять* себе к чему такие последовательности относятся. В нашем представлении это могут быть числа, это могут быть строки, изображения, и все что угодно. Важно правильно хранить и правильно обращаться (или как говорят *читать*) к этим данным. Чтобы человеку-программисту было понятно с каким объектом он работает, объект имеет название или, по-другому, *имя*.

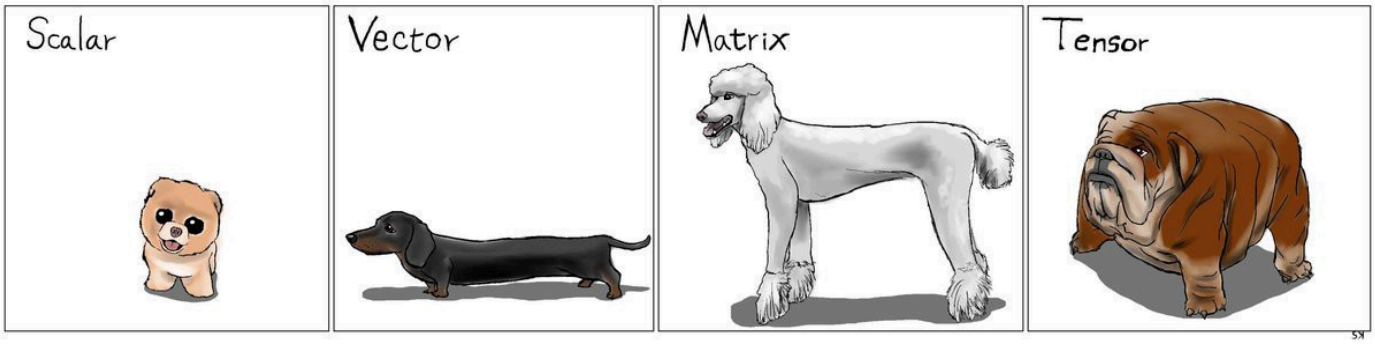
По своей сути *имя* объекта для человека это название, но для компьютера это *адрес* (ссылка, номер) ячейки памяти, где хранится объект. Если объект хранится в нескольких ячейках, то имя это адрес первой ячейки.

Надо различать имя переменной и ее содержимое, т.е. тот объект, на который она ссылается. Обычно одному объекту соответствует одно имя, но может быть и по-другому. Один и тот же объект может иметь два и больше имен, это бывает удобно для программиста, если надо, а не хочется переименовать свой объект.

Один из типов данных, и самый важный, это *числа*.

Одно число называют *скаляр* и с ним можно выполнять разные математические операции.

Набор (говорят *массив* или *тензор*) из нескольких чисел можно представить по-разному, в одномерном, двумерном, трехмерном варианте или даже больше.



Одномерный массив называется *вектор* (или строка), двумерный - *матрица*. Многомерный называют просто массивом указывая его размерность. Часто трехмерный массив называют *тензор*, хотя это неправильно, тензор может быть любой размерности.

В массивах (а мы помним, что массив это набор чисел) важно не только количество элементов, но и как именно они расположены, массивы имеют *форму*. 30 чисел можно по-разному расположить:

- друг за другом в одной строке - получим одномерный массив - строку.
- друг под другом в одном столбце - получим тоже одномерный массив, но уже вектор.
- 6 строк, в каждой по 5 элементов - получили матрицу размером 6 * 5
- 5 строк, в каждой по 6 элементов - получили матрицу размером 5 * 6
- 3 строки по 5 элементов - получим матрицу 3 * 5 и сделаем еще одну матрицу 3 * 5 - получили две матрицы одинакового размера, можем их поставить друг за другом и представить как параллелепипед - получили трехмерный массив.

  И так далее, придумайте свои варианты размещения элементов и определите форму (размерность и размеры) массива.

1. Математические операции со скалярами.

Со скалярами можно выполнять разные математические операции. Многие библиотеки (модули) имеют уже готовые решения для этого, базовые операции встроены в сам Python.

```
a=5     # зададим число a
b=0.3   # зададим число b
```

```
a
```

```
5
```

```
c=a+7 # сложение чисел
```

```
c
```

```
12
```

```
a-b # вычитание чисел
```

```
4.7
```

```
a*b # умножение чисел
```

```
1.5
```

```
a/b # деление чисел
```

```
16.666666666666668
```

```
# Раскомментируйте две строчки ниже и попробуйте поделить на ноль.
b=0 # а если поделим на 0?
a/b # возникнет ошибка, на 0 делить нельзя.
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-8-1a1f650e9fd4> in <cell line: 3>()
      1 # Раскомментируйте две строчки ниже и попробуйте поделить на ноль.
      2 b=0 # а если поделим на 0?
----> 3 a/b # возникнет ошибка, на 0 делить нельзя.

ZeroDivisionError: division by zero
```

2. Модуль `math`

Для более сложных операций рассмотрим модуль `math`.

Сначала его нужно подключить с помощью команды `import`.

```
import math
```

```
#math
```

Округление

Попробуйте округлять разные числа *a* и ответьте, чем отличаются эти два способа `ceil` и `floor`.

```
# Округление
a=3.2
b=math.ceil(a) # округление к ближайшему большему целому.
c=math.floor(a) #округление вниз
b, c
```

```
(4, 3)
```

Модуль числа

Попробуйте для положительных и отрицательных чисел.

```
a=-7
d=math.fabs(a)
```

Остаток от деления $c = mod_b(a)$

Попробуйте для разных целых чисел **a** и **b** и скажите, что же это за функция `fmod`.

Могут ли **a** и **b** быть отрицательными?

Может ли **a=0**? Может ли **b=0**? Может ли **b=1**, какой будет результат в этом случае?

> В отличии от того, чему нас учат в школе, в этой функции **a** и **b** могут быть дробными. Но мы таким пользоваться не будем.

```
a=7
b=2
math.fmod(a,b)
```

```
1.0
```

Константы

Посмотрите на результат и скажите, что за константы мы тут получили? Что вы про них знаете?

```
math.pi
```

```
3.141592653589793
```

```
math.e
```

```
2.718281828459045
```

Возведение в степень, экспонента, логарифм.

Возведение в степень: $c = a^b$

Попробуйте для разных чисел a и b.

Что если a=0, а если b = 0? Может ли a или b быть отрицательным? Нецелым?

Что мы получим если b=0.5? Как такую операцию назвать по-другому?

А если b=0.5 и a - отрицательное?

```
# Возведение в степень.
a=3
b=4
math.pow(a, b)
```

```
81.0
```

Корень квадратный: $c = \sqrt{a}$

Функция `sqrt()`.

Попробуйте для разных чисел **a**

Что если **a=0**? Может ли **a** быть отрицательным? Нецелым?

А как извлечь корень третьей степени?

```
# Корень квадратный.
a=16
math.sqrt(a)
```

```
# Корень кубический.
a=8
math.pow(a,1/3)
```

Экспонента c $= e^b$

Очень часто в качестве основания **a** используют экспоненту. Для этой операции сделали отдельную функцию.

Попробуйте сделать тоже самое с помощью `pow()`.

```
b=10
math.exp(b)
```

```
math.pow(math.e,b)
```

Логарифм $b = log_a(c)$

Если мы знаем результат возведения в степень **c**, знаем основание **a**, а хотим узнать, какая же степень **b** должна при этом быть, то нам на помощь придет *логарифм*, функция `log()`.

Указываем сначала **c** от которого мы ищем логарифм, а потом, необязательно, основание **a** по которому мы ищем логарифм. Если основание не указано - используется *натуральный логарифм*.

Попробуйте с разными числами.

Может ли **c=0**? А меньше нуля может быть?

А если **c=1**, какой будет результат? Зависит ли он от **a**?

Может ли **a=0** , **a=1**? А отрицательным **a** может быть? А дробным?

```
c=81
a=3
#math.log(x[, base])
math.log(c, a)
```

Наиболее часто используемые основания логарифмов сделаны в отдельных функциях, так быстрей и точней считать.

Двоичный логарифм, по основанию **a=2**: `log2`

Десятичный логарифм, по основанию **a=10**: `log10`

А как сделать *натуральный логарифм*? Какое у него основание?

```
c=4
math.log2(c)
```

```
c=100
math.log10(c)
```

```
c=math.e*math.e
math.log(c)
```

Тригонометрические функции

- синус `sin`
- косинус `cos`
- тангенс `tan`
- арктангенс `atan` (обратная функция для тангенса) - часто используется в нейронных сетях.
- гиперболический тангенс `tanh` - тоже часто используется в нейронных сетях.
- и другие...

Для известных углов (например, 0, pi/4, pi/2 ,2 * pi и др.) посчитайте сами и сравните с расчетом в Питоне.

Все углы указываются в радианах!

```
x = math.pi/4 # зададим угол
```

```
math.cos(x)
```

```
math.sin(x)
```

```
#x = math.pi/2 #
math.tan(x) # а если x=math.pi/2?
```

Арктангенс `atan(x)` - обратная функция для тангенса.

Позволяет узнать для какого угла тангенс равен x.

Поделим на Pi чтобы узнать этот угол.

Узнали?

```
# арктангенс - обратная функция для тангенса
x=1
math.atan(x)/math.pi
```

Гиперболический тангенс `tanh(x)` - довольно сложная функция, но по форме похожа на арктангенс.

> Формула для сведения, можете не запоминать сейчас:
>
> **гиперболический тангенс**:
>
> $$\operatorname{th} x = \frac{\operatorname{sh} x}{\operatorname{ch} x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

```
# гипертангенс
x=1000000000000
math.tanh(x)
```

Факториал

Функция `factorial()`.

Находит факториал целого числа, произведение всех целых чисел начиная с единицы до заданного **a**.

Обозначается как "!"

Например 5! = 1 * 2 * 3 * 4 * 5 =120

Часто возникает в задачах с комбинациями различных событий.

Попробуйте для разных чисел **a**.

Может ли **a** быть отрицательным? Дробным?

Чему равен факториал нуля?

```
a=2
math.factorial(a)
```

Домашнее задание

Изучите документацию на модуль `math`.

Посмотрите какие еще в нем есть функции.

Ответьте, как сконвертировать градусы в радианы и наоборот. Сделайте это.

Что делать, если надо реализовать какую-то математическую функцию, которую вы не знаете, но она нужна? Внимательно читать документацию. Искать ответ в Интернет. Так часто поступают даже профессиональные разработчики - нельзя все помнить и знать. Но нужно уметь **искать** информацию.

Если какой-то функции нет в этом модуле - поищите в другом.

3. Модуль `numpy`

Библиотека `numpy` предназначена для работы с массивами. Для нас это самая любимая и часто используемая библиотека. Ее документацию можно посмотреть здесь: https://numpy.org/

Numpy создает и обрабатывает массивы как объекты своего собственного класса (типа): `ndarray`

Основные его атрибуты это `shape` - форма массива и `dtype` - тип данных.

Подключим библиотеку с помощью `import` и назовем коротко: `np`

```
import numpy as np
```

Создание массивов

Для создания массивов используем функцию `numpy.array()`

В ней мы указываем какой массив хотим создать.

Создадим массив, отобразим его и посмотрим, как называется его тип. Тип можно узнать с помощью команды `type()` .

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

```
type(a)
```

```
numpy.ndarray
```

```
a.shape
```

```
(3,)
```

Видим, что тип - `numpy.ndarray` .

Это одномерный массив.

Теперь создадим двумерный массив (матрицу)

```
b = np.array([[1.5, 2, 3], [4, 5, 6]])
```

```
b
```

```
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

```
b.shape
```

```
(2, 3)
```

Это двумерный массив. Атрибут `shape` показывает нам его форму, он имеет размер 2 на 3: 2 строки, 3 столбца.

Чтобы создавать массивы, заполненные какими-то числами, можно воспользоваться специальными функциями.

Например

`numpy.zeros(...)` делает массив заданной формы, заполненный нулями

`numpy.ones(...)` делает массив заданной формы, заполненный единицами

Мы указываем число элементов по каждой оси массива - разумеется это натуральные числа.

```
np.zeros((3, 5)) # двумерный массив из нулей, размером 3 на 5
```

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

```
np.ones((2, 3, 4)) # трехмерный массив из единиц размером 2 на 3 на 4
```

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

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

Создадим любимую для линейной алгебры единичную матрицу с помощью команды `eye()`. Она квадратная, поэтому только один аргумент указываем - число строк. Если очень надо сделать по-другому, например не главную диагональ заполнить, то смотри документацию.

```
np.eye(5) # создаем единичную матрицу
```

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

Создать массив, но ничем его не заполнять можно с помощью функции `numpy.empty()`.

Заполнен он будет мусором, который остался в памяти, где создается такой массив. Это гораздо быстрее, чем создать и заполнять массив нулями. Но нужно быть осторожным и потом все-таки заполнить массив, потому что мусор может быть любой.

```
np.empty((3, 3))
```

```
array([[5.01961143e-310, 0.00000000e+000, 0.00000000e+000],
       [0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       [0.00000000e+000, 0.00000000e+000, 0.00000000e+000]])
```

```
np.empty((3, 2))
```

Для создания последовательностей чисел, в NumPy имеется функция `numpy.arange(начало, конец, шаг)`, аналогичная встроенной в Python `range()`, только вместо списков она возвращает массивы, и принимает не только целые значения. Мы указываем **начало** (необязательно, по умолчанию 0), **конец** и **шаг** (тоже не обязательно, по умолчанию 1) нашей последовательности. Конец не входит в последовательность. Шаг может быть отрицательным (но в этом случае конец должен быть меньше начала, иначе вернется пустой массив).

Попробуйте сами создать разные последовательности. Что получится если начало и конец совпадают? А если шаг=0?

```
a=np.arange(20, 30, 5)
```

```
np.arange(1, 0, -0.1)
```

```
array([1. , 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1])
```

Вывести на экран массив можно командой `print()`.

```
a=np.arange(1,10,0.001)
print(a)
```

```
[1.    1.001 1.002 ... 9.997 9.998 9.999]
```

Видим, что выводится не весь массив. Это сделано для экономии места.

Если хотим задать сколько знаков после запятой выводить в числах, сколько чисел выводить на экран и др. надо задать опции вывода `set_printoptions`:

- параметр `treshold`: сколько чисел на экран выводить
- параметр `precision`: сколько знаков после запятой выводить в числах

Это влияет только на вывод, не на сами значения. Эти свойства будут применяться для всех выводов, не только для текущего, поэтому осторожно.

```
np.set_printoptions(threshold=10000, precision=2) # будем выводить до 10 десяти тысяч чисел, два знака посл
```

```
print(a)
```

```
np.set_printoptions(threshold=1000, precision=8) # вернем параметры по умолчанию
print(a)
```

Базовые операции

Основные математические операции над массивами выполняются поэлементно. Создается новый массив, который заполняется результатами.

Для выполнения этих операций массивы должны быть одинаковой формы. Если будут разной - получим ошибку.

```
a = np.array([20, 30, 40, 50]) # создадим один массив
a
```

```
array([20, 30, 40, 50])
```

```
b = np.arange(4) # создадим другой массив
b
```

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

```
c=a+b # сложим их
c
```

```
array([20, 31, 42, 53])
```

```
d=np.array([2,5])
```

```
a+d # размер не подходит для расширения
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-0368ff28784a> in <cell line: 1>()
----> 1 a+d # размер не подходит для расширения

ValueError: operands could not be broadcast together with shapes (4,) (2,)
```

```
c=a - b # вычтем
c
```

```
array([20, 29, 38, 47])
```

```
c=a*b # умножим поэлементно
c
```

```
array([  0,  30,  80, 150])
```

```
c=a/b # поделим поэлементно
c
```

```
<ipython-input-36-e43254edbba7>:1: RuntimeWarning: divide by zero encountered in divide
  c=a/b # поделим поэлементно
array([        inf, 30.        , 20.        , 16.66666667])
```

Ой-ой, что это, мы поделили на 0! Но NumPy справился с этим и вернул нам для такого случая специальное число - бесконечность `inf`. Бесконечности тоже числа, хоть и ненастоящие. Их можно складывать, вычитать, умножать ...

Попробуйте сами и определите, что получается.

```
np.inf + np.inf
```

```
inf
```

```
-1*np.inf
```

```
-inf
```

```
np.inf * np.inf
```

```
inf
```

```
np.inf - np.inf
```

```
nan
```

При попытке вычесть одну бесконечность `inf` из другой мы тоже что-то получили. А вы знаете сколько будет бесконечность минус бесконечность? А NumPy знает: будет `nan` - не число (not a number).

`nan` тоже ненастоящее число, и с ним можно делать математические операции.

Попробуйте сами и определите, что получается.

Все эти ненастоящие числа нужны чтобы облегчить программирование, но на самом деле они смысла не имеют, и если у вас такое число получилось - что-то не так. Поэтому иногда надо проверять результат, чтобы там не было бесконечностей или "нечисел".

```
np.nan*(-1)
```

```
nan
```

```
np.nan+np.nan
```

```
nan
```

```
np.nan+5
```

```
nan
```

```
a ** b # возведение в степень, первый аргумент - основание, второй - степень.
```

```
array([     1,     30,   1600, 125000])
```

```
a % b  # Взятие остатка от деления (при взятии остатка от деления на 0 возвращается 0)
```

```
<ipython-input-45-90e17d18130c>:1: RuntimeWarning: divide by zero encountered in remainder
  a % b  # Взятие остатка от деления (при взятии остатка от деления на 0 возвращается 0)
array([0, 0, 0, 2])
```

Как мы уже отметили, формы массивов должны быть одинаковые (попробуйте что будет если нет).

Но есть исключение - когда один из аргументов - скаляр. Тогда он автоматически будет применяться к каждому элементу массива.

```
a + 1
```

```
array([21, 31, 41, 51])
```

```
1 + b
```

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

```
a ** 3
```

```
array([  8000,  27000,  64000, 125000])
```

```
2 ** b
```

```
array([1, 2, 4, 8])
```

```
a=np.ones((3,1))
```

```
b=2*np.transpose(a)
```

```
print(a.shape, b.shape)
```

```
(3, 1) (1, 3)
```

```
a*b # ?????
```

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

```
a=3*np.ones((4,4))
b=-1*np.ones((4,1))
c=a+b
c
```

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

Тригонометрия, округления

В `numpy` как и в `math` реализованы многочисленные тригонометрические операции.

Они работают с массивами поэлементно.

Округлений гораздо больше, самых разных типов: `ceil()`, `floor()`, но и другие `round()` - округление до заданного числа знаков после запятой, `fix()` - округление до ближайшего целого со стороны нуля.

Поупражняйтесь с разными способами округления, сравните.

```
np.cos(a) # косинус
```

```
array([[-0.9899925, -0.9899925, -0.9899925, -0.9899925],
       [-0.9899925, -0.9899925, -0.9899925, -0.9899925],
       [-0.9899925, -0.9899925, -0.9899925, -0.9899925],
       [-0.9899925, -0.9899925, -0.9899925, -0.9899925]])
```

```
np.arctan(a) # арктангенс, название отличается от модуля math
```

```
array([[1.24904577, 1.24904577, 1.24904577, 1.24904577],
       [1.24904577, 1.24904577, 1.24904577, 1.24904577],
       [1.24904577, 1.24904577, 1.24904577, 1.24904577],
       [1.24904577, 1.24904577, 1.24904577, 1.24904577]])
```

```
np.tanh(a) # гипертангенс
```

```
array([[0.99505475, 0.99505475, 0.99505475, 0.99505475],
       [0.99505475, 0.99505475, 0.99505475, 0.99505475],
       [0.99505475, 0.99505475, 0.99505475, 0.99505475],
       [0.99505475, 0.99505475, 0.99505475, 0.99505475]])
```

```
c=5*np.cos(a) #
c
```

```
array([[-4.94996248, -4.94996248, -4.94996248, -4.94996248],
       [-4.94996248, -4.94996248, -4.94996248, -4.94996248],
       [-4.94996248, -4.94996248, -4.94996248, -4.94996248],
       [-4.94996248, -4.94996248, -4.94996248, -4.94996248]])
```

```
np.round(c,2)
```

```
array([[-4.95, -4.95, -4.95, -4.95],
       [-4.95, -4.95, -4.95, -4.95],
       [-4.95, -4.95, -4.95, -4.95],
       [-4.95, -4.95, -4.95, -4.95]])
```

```
np.ceil(c)
```

```
array([[-4., -4., -4., -4.],
       [-4., -4., -4., -4.],
```

```
           [-4., -4., -4., -4.],
           [-4., -4., -4., -4.]])
```

```
np.fix(c)
```

```
array([[-4., -4., -4., -4.],
       [-4., -4., -4., -4.],
       [-4., -4., -4., -4.],
       [-4., -4., -4., -4.]])
```

Агрегирующие функции.

Можно совместно обрабатывать все элементы массивов, а не каждый по отдельности. Такие функции называются агрегирующими.

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

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

```
np.sum(a) # суммирование всех элементов массива между собой
```

```
21
```

или

```
a.sum() # другой способ вызова той же функции суммирования
```

```
a.min() # минимальный элемент массива
```

```
1
```

```
a.max() # максимальный элемент массива
```

```
6
```

По умолчанию, эти операции применяются к массиву, как если бы он был списком чисел, независимо от его формы. Однако, указав параметр `axis`, можно применить операцию для указанной оси (размерности) массива. Оси нумеруются начиная с 0 (первая ось имеет номер ноль).

```
a.min(axis=1) # минимум по строкам для каждого столбца
```

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

```
a.min(axis=0) # минимум по столбцам для каждой строки
```

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

Вообще, указание осей (особенно в случае тензоров - многомерных массивов) очень важно для таких функций. Мы будем часто пользоваться этим, когда будем изучать нейронные сети.

Индексы, срезы, итерации

До сих пор мы обрабатывали все элементы массива, которые в нем есть. Но часто нужно работать только с некоторыми из них. Для этого надо узнать эти элементы. Например, хотим узнать какой в матрице первый элемент. Для этого нужно обратиться к этому элементу по его индексу (номеру). В Python индексы начинаются с нуля, т.е. первый элемент имеет индекс 0.

Обращение к элементам, или как говорят *индексация*, массива выполняется с помощью квадратных скобок [] внутри которых мы указываем **срез** тех индексов элементов, к которым хотим обратиться. Вспомним что такое **срез** из уроков по основам Python. Это последовательность целых чисел, которую можно задать, например, использую оператор **начало:конец:шаг**

```
a = np.arange(10) ** 3 # создадим массив. Какая у него форма?
a
```

```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

```
a[4] # Посмотрим на первый элемент этого массива, помним что первый элемент имеет индекс 0.
```

```
64
```

```
a[3:5] # Посмотрим сразу на четвертый и пятый элементы. Помним что в срезах конец не входит в диапазон, поз
```

```
array([27, 64])
```

```
a[[3,7]]
```

```
array([ 27, 343])
```

```
a[[3,3]]
```

```
array([27, 27])
```

изменим четвертый и пятый элементы. Вообще, при изменении формы объектов слева и справа от знака присвоения = должны быть одинаковые. Но для скаляра это исключение.

```
a[3:5] = 2 # изменяем
a
```

```
array([  0,   1,   8,   2,   2, 125, 216, 343, 512, 729])
```

А если вот так сделать, что получится?

```
a[0.5] # такие индексы невозможны
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-76-7a577bf306cf> in <cell line: 1>()
----> 1 a[0.5] # такие индексы невозможны

IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`) and integer or
boolean arrays are valid indices
```

```
a[::-1] # Что здесь происходит?
```

```
array([729, 512, 343, 216, 125,   2,   2,   8,   1,   0])
```

```
a
```

```
a[-1]
```

```
729
```

Вспомнив, как создаются срезы, понимаем, что здесь **начало, конец** опущены, а значит будут использованы значения по умолчанию, а **шаг = -1**. В результате получили массив, в котором элементы переставлены в обратном порядке.

У *многомерных* массивов на каждую ось приходится один индекс. Индексы передаются в виде последовательности чисел, разделенных запятыми (кортежами):

```
b = np.array([[ 0,  1,  2,  3],
...           [10, 11, 12, 13],
...           [20, 21, 22, 23],
...           [30, 31, 32, 33],
...           [40, 41, 42, 43]]) # создадим двумерный массив
b
```

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

```
b[2,3] # обратимся к элементу на третьей строке в четвертом столбце
```

```
23
```

```
b[[2,3],3]
```

```
array([23, 33])
```

тоже самое можно получить и так:

```
b[(2,3)]
```

```
23
```

или даже так

```
b[2]
```

```
array([20, 21, 22, 23])
```

```
b[2][3]
```

```
23
```

Но не следует путать

```
b[:,2] # это третий столбец целиком (все строки)
```

```
array([ 2, 12, 22, 32, 42])
```

```
b[:]
```

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

и

```
b[: 2] # это первая и вторая строки массива
```

```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
```

Тут мы запятую не поставили, и NumPy считает, что мы указываем только один индекс, самый первый - индекс строк. А для столбцов индекс (по умолчанию) - все.

Теперь разгадайте вот такую индексацию:

```
b[1:3, : : ] # что это за смайлик?
```

```
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Мы указали *два* индекса, один для строк: `1:3`, т.е. вторая и третья строки,

другой для столбцов: `: :` здесь **начало, конец, шаг** опущены и использованы значения по умолчанию. **начало = 0 , конец = "длина объекта" = 4 , шаг = 1** т.е. первый, второй, третий и четвертый столбцы.

А что будет, если обратиться к элементу номер 100? Проверьте.

```
b
```

```
b[2:3][0] #????
```

```
array([20, 21, 22, 23])
```

```
e=np.ones((2,3,4,2,3,4))
```

```
e[...,2] # даже такое работает, опустили все предыдущие индексы
```

```
array([[[[[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., 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., 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.]],

  [[1., 1., 1.],
   [1., 1., 1.]],

  [[1., 1., 1.],
   [1., 1., 1.]]],


 [[[1., 1., 1.],
   [1., 1., 1.]],

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

Еще больше информации и примеров по индексации можно найти в документации.

Итерирование массивов

Зададим массив посложней. Узнаем его форму используя атрибут `shape`. Это трехмерный массив, 2 строки, 2 столбца, 3 третьих измерения (им не придумали названия).

```
import numpy as np


a = np.array(([[0, 1, 2], [10, 12, 13]], [[100, 101, 102], [110, 112, 113]]))


a
```

```
array([[[  0,   1,   2],
        [ 10,  12,  13]],

       [[100, 101, 102],
        [110, 112, 113]]])
```

```
a.shape
```

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

Итерирование многомерных массивов ведется по первой оси (т.е. строкам).

Изменяется номер строки, а все остальные индексы берутся "все".

В результате выполнится **две** итерации (число строк равно двум).

Объект `row` будет массивом, который содержит:

- на первой итерации элементы из первой строки, всех столбцов и всех третьих измерений массива **a**.
- на второй итерации элементы из второй строки, всех столбцов и всех третьих измерений массива **a**.

```
for row in a:
   print(row)
   print(' ')
```

```
[[ 0  1  2]
 [10 12 13]]

[[100 101 102]
 [110 112 113]]
```

Если нужно перебрать поэлементно весь массив, как если бы он был одномерным, для этого можно использовать `flat`. При этом наиболее быстро изменяется последний индекс, в нашем случае третий, потом второй (столбцы) и только в конце первый (строки).

Значит первая половина результата - это элементы с первой строки. Вторая половина - элементы со второй строки.

`flat` это итератор, он привязан к самому массиву, если изменим массив, то и его итератор поменяется, он *не* делает копию массива. Если нужна копия объекта используйте `flatten`.

```
a.flat
```

```
<numpy.flatiter at 0x5958e4f1b250>
```

```
for el in a.flat:
    print(el)
```

```
0
1
2
10
12
13
100
101
102
110
112
113
```

```
a
```

Форма массива может быть изменена с помощью различных команд, результаты некоторых привязаны к самому массиву, изменив его изменится и результат, копии массива не создаются.

```
a.ravel()  # Делает массив плоским, но сам массив не изменяется
```

```
array([  0,   1,   2,  10,  12,  13, 100, 101, 102, 110, 112, 113])
```

```
a
```

```
array([[[  0,   1,   2],
        [ 10,  12,  13]],

       [[100, 101, 102],
        [110, 112, 113]]])
```

```
a.shape = (6, 2)  # Изменение формы, сам массив изменяется
a
```

```
array([[  0,   1],
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])
```

```
a.transpose()  # Транспонирование, сам массив не изменяется
```

```
array([[  0,   2,  12, 100, 102, 112],
       [  1,  10,  13, 101, 110, 113]])
```

```
a
```

```
a.reshape((3, 4))  # Изменение формы, но сам массив не изменяется
```

```
array([[  0,   1,   2,  10],
       [ 12,  13, 100, 101],
       [102, 110, 112, 113]])
```

```
a
```

```
array([[  0,   1],
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])
```

Объединение и разбиение массивов

Несколько массивов могут быть объединены вместе вдоль разных осей с помощью функций `hstack` и `vstack`.

`hstack()` — объединяет массивы по вторым осям (горизонтально),

`vstack()` — по первым (вертикально).

Есть функция `stack()` для объединения по любой оси и другие функции.

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

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

```
b = np.array([[5, 6], [7, 8]])
b
```

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

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

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

```
np.hstack((a, b))
```

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

```
np.column_stack((a, b))
```

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

```
np.row_stack((a, b))
```

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

Используя `hsplit()` можно разбить массив по горизонтальной (второй, столбцы) оси, указав либо число возвращаемых массивов одинаковой формы, либо номера столбцов, после которых массив разрезается "ножницами".

Аналогично `vsplit()` разрезает массивы по вертикальной (первой, строки) оси.

```
a = np.arange(12).reshape((2, 6))
```

```
a
```

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

```
z1,z2,z3=np.hsplit(a, 3)  # Разбить на 3 части по столбцам
```

```
z1
```

```
np.hsplit(a, (3, 4))  # Разрезать a после третьего и четвёртого столбца
```

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

```
       [9]]),
 array([[ 4,  5],
       [10, 11]])]
```

```
np.vsplit(a, 2)  # Разбить на 2 части по строкам
```

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

Копии и представления

Простое присваивание **не создает** копии массива:

```
a = np.arange(12)
```

```
a
```

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

```
b = a
```

```
b
```

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

a и **b** это два имени для одного и того же объекта ndarray, нового объекта не было создано.

Поменяв **a**, поменяется и **b**.

Чтобы проверить что это один и тот же объект можно использовать команду `is`

Она вернет *истину* (True) или *ложь* (False)

```
b is a # проверим что это один и тот же объект
```

```
True
```

```
b.shape
```

```
(12,)
```

```
b.shape = (3,4) # изменим массив b
```

```
a.shape # при этом изменился и массив a
```

```
(3, 4)
```

`view()` создаст **представление** массива **a**.

Представление - это другой объект, он находится в другом месте памяти, но он связан с исходным массивом. Если поменяем массив **a**, то и его представление **c** тоже поменяется.

Себе вы можете представить это так, мы смотрим на стол одним глазом и другим. Глаза разные, находятся в разных местах, но если стол поменяется, то обоими глазами мы увидим эти изменения.

```
c = a.view()
```

```
c
```

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

```
c is a
```

```
False
```

```
a[0,0]=100500 # изменим массив a
c # изменилось и его представление c
```

```
array([[100500,      1,      2,      3],
       [     4,      5,      6,      7],
       [     8,      9,     10,     11]])
```

```
c[1,1]=-100500 # поменяем c
c # оно конечно изменилось
```

```
array([[ 100500,       1,       2,       3],
       [      4, -100500,       6,       7],
       [      8,       9,      10,      11]])
```

```
a # и a тоже изменилось
```

```
array([[ 100500,       1,       2,       3],
       [      4, -100500,       6,       7],
       [      8,       9,      10,      11]])
```

Если нужно создать копию массива, которая будет независима от самого массива, но содержать те же данные, то используем `copy()`.

```
d = a.copy()  # создается новый объект массива
d           # данные в нем такие же
```

```
array([[ 100500,       1,       2,       3],
       [      4, -100500,       6,       7],
       [      8,       9,      10,      11]])
```

```
d is a # Это тот же объект? Нет
```

```
False
```

```
a[0,0]=-1 # изменим массив a
a
```

```
array([[     -1,       1,       2,       3],
       [      4, -100500,       6,       7],
       [      8,       9,      10,      11]])
```

```
d        # изменился ли массив d? Нет.
```

```
array([[ 100500,       1,       2,       3],
       [      4, -100500,       6,       7],
       [      8,       9,      10,      11]])
```

Случайные числа в numpy

Очень часто для работы нужны *случайные* числа. Это числа, которые каждый раз принимают разные значения. Сегодня одно, завтра другое. На случайности построено много алгоритмов, а иногда качество такой случайности очень сильно влияет на работу.

В обычных компьютерах нет настоящих случайных чисел, это слишком дорого. Поэтому все пользуются **псевдослучайными** числами, которые похожи на случайные, но на самом деле были один раз записаны и сохранены. В компьютерах есть большие таблицы псевдослучайных чисел, из которых они и выбираются (таблицы могут создаваться специальными алгоритмами-генераторами, см. например, Вихрь Мерсена).

Случайные числа описываются *распределением*, функцией, которая примерно показывает сколько разных случайных чисел каждого значения используется.

- Равномерное распределение (на отрезке [0 , 1]) означает что есть примерно одинаковое количество чисел каждого значения из этого отрезка. Например, 1000 штук со значением 0, 1005 штук со значением 0.1, 998 штук со значением 0.2 и т.д.
- Нормальное распределение - если нарисовать функцию такого распределения, то она похожа на колокольчик, который расположен в нуле (точная формула нам сейчас не интересна). Это значит, что есть очень много чисел близких к нулю, но есть мало больших чисел, и чем больше число, тем меньше таких чисел в распределении.

Случайные числа из распределений выбираются наугад.

Для работы со случайными числами нам потребуются модули `random` и `numpy.random`.

```
import random
```

```
import numpy as np
import numpy.random as rand
```

Самый простой способ задать массив со случайными элементами - использовать функцию `sample()` (или `random()`, или `random_sample()`, или `ranf()` - это всё одна и та же функция). Ей мы сообщаем размеры массива, который хотим создать. Эта функция берет случайные числа из равномерного распределения на интервале [0, 1) (это значит, что ноль может выпасть, а единичка нет)

Позапускайте код ниже несколько раз, каждый раз будут другие числа возвращаться.

Если нужны случайные числа из другого диапазона, то их нужно сделать из этих. Сместить, растянуть диапазон.

```
np.random.sample() # одно случайное число
```

```
0.7745943815566008
```

```
a=-3
b=5
a+(b-a)*np.random.sample() # одно случайное число из диапазона [a,b)
```

```
-1.5442459175280314
```

```
np.random.sample(5) # Массив из 5 случайных чисел
```

```
array([0.2723564 , 0.26920352, 0.43247283, 0.29136998, 0.28344936])
```

```
np.random.sample((1, 1, 4)) # 4 случайных числа в трехмерном массиве
```

```
array([[[1.48980624e-01, 8.82839754e-01, 7.66383055e-01, 3.39738728e-04]]])
```

С помощью функции `randint()` или `random_integers()` можно создать массив из **целых** случайных чисел. Указываем аргументы: `low`, `high`, `size`: от какого `low`, до какого `high` числа (`randint` не включает в себя это число, а `random_integers` включает), и `size` - размеры массива.

```
np.random.randint(0, 3, 10)# массив из 10 случайных целых чисел от 0 до 2 (3 не включается)
```

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

```
np.random.random_integers(0, 3, 10)# массив из 10 случайных целых чисел от 0 до 3 (3 включается)
```

```
<ipython-input-147-b8d9942d7909>:1: DeprecationWarning: This function is deprecated. Please call randin
  np.random.random_integers(0, 3, 10)# массив из 10 случайных целых чисел от 0 до 3 (3 включается)
array([0, 1, 2, 2, 2, 0, 2, 3, 3, 0])
```

```
np.random.randint(0, 3, (2, 10))# двумерный массив случайных чисел от 0 до 2.
```

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

Можно генерировать числа согласно различным распределениям (Гаусса, Парето и другие). Если понадобятся - см. документацию.

Чаще всего нужно равномерное распределение, которое можно получить с помощь функции `uniform()`, смысл аргументов такой же как у `random_integers`, начало и конец могут быть дробными.

```
np.random.uniform(2, 8, (2, 10)) # двумерный массив со случайными числами от 2 до 8
```

```
array([[7.76020346, 3.65436475, 7.02616787, 7.31054906, 5.77193106,
        6.81996706, 5.32552314, 6.75338692, 5.48786602, 4.20346796],
       [4.24200187, 6.40990475, 7.38893899, 7.146736  , 2.57360589,
        3.6022044 , 4.41106486, 3.3731881 , 7.29483037, 7.34309776]])
```

Случайно перемешать массив можно с помощью функции `shuffle()`. Изменится сам массив.

```
a = np.arange(10)
```

```
a=np.random.randint(0,5,(2,3))
```

```
a
```

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

```
np.random.shuffle(a)
```

```
a
```

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

Инициализация генератора случайных чисел

Случайные числа берутся из таблицы случайных чисел, можно принудительно заставить компьютер брать случайные числа из одной и той же части таблицы, или из разных.

Первое может понадобиться, когда хотим, чтобы вычисления были одинаковыми.

Установить, с какого места таблицы начинать брать случайные числа можно с помощью `random.seed()`

Попробуйте сделать одинаковые и разные места и сравните результаты.

Увеличивайте число `seed` в 100 раз, пока не возникнет ошибка, так примерно узнаете какой размер

```
np.random.seed(42) # начинаем с места №1000

np.random.random(10) # берем 10 случайных чисел
```

```
array([0.37454012, 0.95071431, 0.73199394, 0.59865848, 0.15601864,
       0.15599452, 0.05808361, 0.86617615, 0.60111501, 0.70807258])
```

```
np.random.seed(100) # Начинаем с другого места №100

np.random.random(10) # Берем 10 случайных чисел, они другие
```

```
array([0.54340494, 0.27836939, 0.42451759, 0.84477613, 0.00471886,
       0.12156912, 0.67074908, 0.82585276, 0.13670659, 0.57509333])
```

```
np.random.seed(1000) # Начинаем с того же места №1000

np.random.random(10) # Берем 10 случайных чисел, они такие же как в первом варианте
```

```
array([0.65358959, 0.11500694, 0.95028286, 0.4821914 , 0.87247454,
       0.21233268, 0.04070962, 0.39719446, 0.2331322 , 0.84174072])
```

Пример из будущих уроков

Мы будем часто выполнять операции с массивами, чтобы подготовить данные для наших расчетов. Ниже кусок кода, который нам еще не раз встретится, который показывает как правильно формировать данные для расчетов.

Попробуйте по шагам объяснить, что тут происходит.

Не получится - не беда, узнаем позже.

```
batchsize, maps, h, w = 1, 1, 3, 3
data = (np.arange(batchsize * maps * h * w).reshape(batchsize, maps, h, w).astype(np.float32))
```

```
data
```