

# NumPy

## Виртуальное окружение

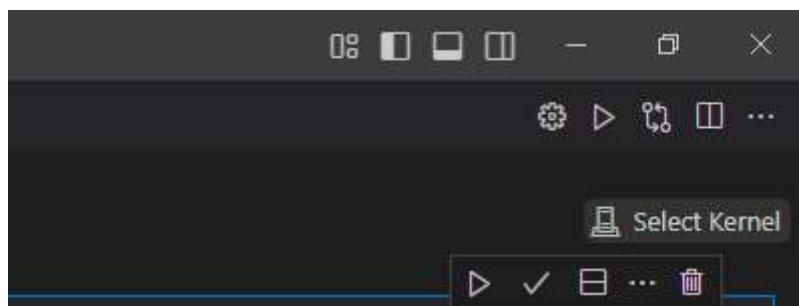
Но прежде, чем перейдём к основной теме --- есть ещё вещь, которую надо обсудить. Возможно, в этом курсе она не такая уж и важная, но в приличном обществе так делать принято. Я говорю о виртуальном окружении.

Виртуальное окружение --- это среда, изолирующая набор установленных пакетов и версий Python от системной среды и/или от других проектов. Зачем?

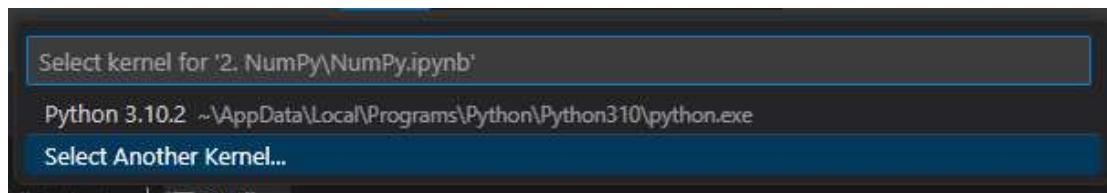
1. Разные проекты могут требовать разные версии одних и тех же библиотек. Если устанавливать их без использования виртуальных окружений (т.е. глобально), то вы неизбежно наткнётесь на несовместимость. Как можете догадаться каждый раз удалять и ставить необходимую версию библиотеки для работы с тем или иным проектом --- не самое увлекательное занятие, поэтому используйте виртуальные окружения.
2. При переходе на другую рабочую машину можно быстро воссоздать используемое [на изначальной машине] окружение (с помощью фиксации списков пакетов в `requirements.txt`, например). Опять же, ничего не сломав при этом новой.
3. Большое количество пакетов, скачанных однажды для какой-то мелкой задачки, может захламить систему и привести к конфликтам.
4. Возможность использовать несколько разных версий Python (например, для поддержки legacy проектов).

Более "традиционный" способ создавать виртуальные окружения с помощью терминала (и в этом абсолютно нет ничего сложного), но ввиду демонстративных целей данного курса и отсюда максимального упрощения --- мы сделаем это через VSCode.

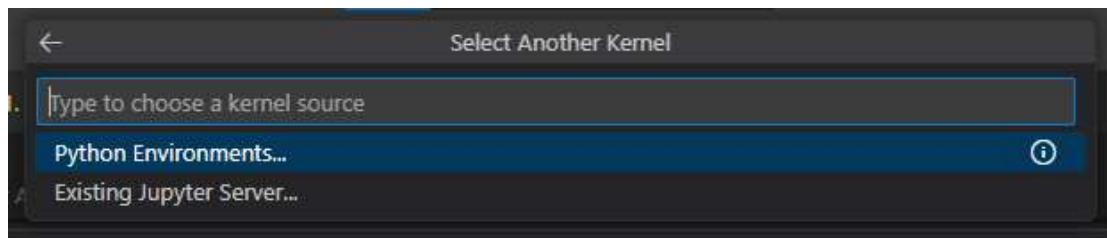
1. В правом верхнем углу нажимаем кнопку "Select Kernel" (или "Python 3.x.y", где x, у версия Python).



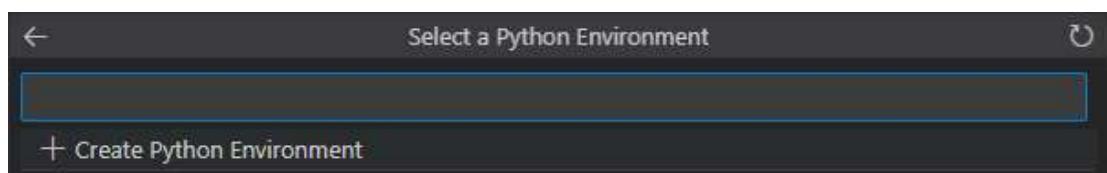
2. В появившемся меню нажимаем "Select Another Kernel..."



3. Нажимаем "Python Environments..."

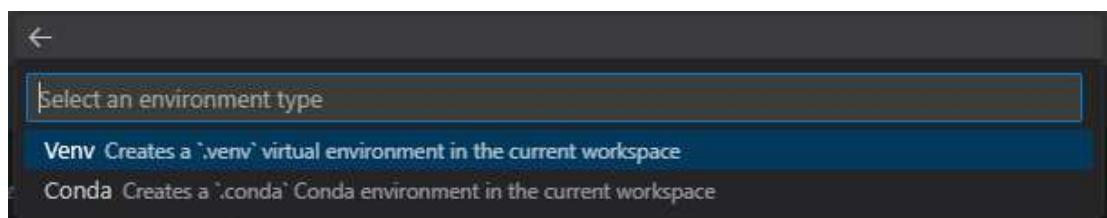


4. "Create New Environment..."

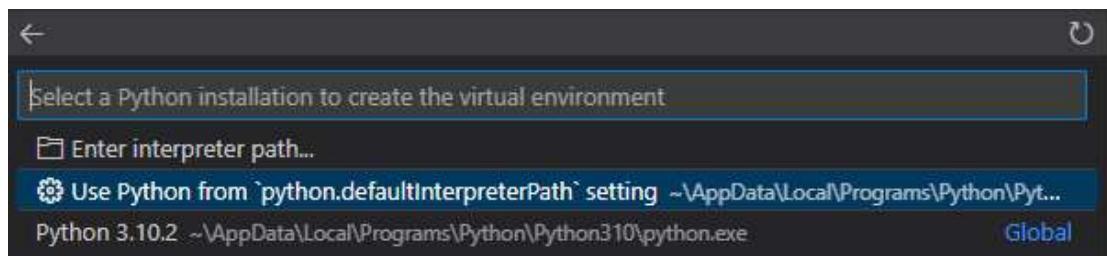


5. "Venv"

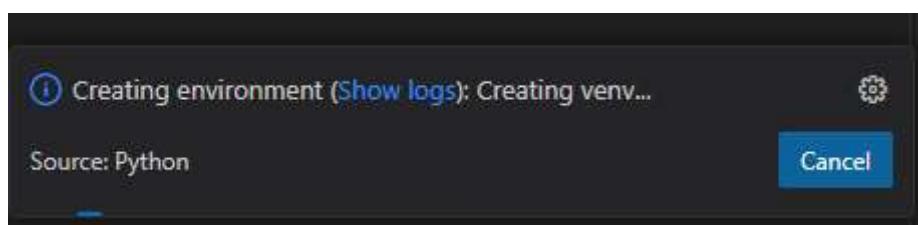
NB: Если вы используете конду, то вместо `.venv` выберете опцию конды. В шаге 8 изменения должны быть аналогичные.



6. "Use Python from `python.defaultInterpreterPath` setting



7. Ждём пока окошко в правом нижнем углу не пропадёт.



8. Если на месте кнопки "Select Kernel" или что у вас там было в шаге №1 теперь написано ".venv (Python.3.x.y)", то у вас всё удалось. Слева, в обозревателе (напоминание: если он закрыт, то его можно открыть нажав на пиктограмму двух файлов в левом верхнем углу или по комбинации клавиш **Ctrl+B**) должна появиться папка `.venv`. Если решите использовать виртуальное окружение, то не забывайте путь к этой папке (пусть она нужна будет для наших учебных целей).

Если вы не хотите использовать виртуальное окружение (зря), но всё равно выполнили все действия, то прежде, чем перейти дальше -- не забудьте вернуться к глобальной версии (нажав на кнопку выбора ядра и выбрав там то, что вам надо). С другой стороны, если вы уже создали виртуальное окружение, то зачем отказываться им пользоваться, если вещь нужная и полезная? Определитесь сами, а мы продолжим.

## O NumPy

NumPy --- библиотека для научных вычислений. Плюшки:

- Ndarray --- многомерный массив, оптимизированный для быстрых вычислений (скажите спасибо за это C (языку))
- Замена явных циклов на операции над массивами (векторизация), ускоряет вычисления
- Операции линейной алгебры, генерация псевдослучайных чисел, статистические функции и многое другое.

Зачем? Помимо вышеописанного многие библиотеки для МО и ИИ основаны на NumPy, поэтому нужно знать хотя бы его азы.

## Установка

В терминале необходимо выполнить команду: `pip install numpy` (в случае конды: `conda install numpy`). Если вы используете виртуальное окружение, которое только что создали, то убейте терминал (нажав на пиктограмму мусорного ведра, расположенной на одном уровне с нижними вкладками (среди которых есть вкладка терминала) в правой части) и откройте новый. Ну или закройте/откройте VS Code, как вам удобнее будет.

Если следующая ячейка выполнится корректно, то значит установка прошла успешно.

In [1]: `import numpy as np`

Обратим внимание на часть `as np`. `np` --- общепринятое сокращение для NumPy. Если убрать `as np`, то вместо `np` придётся писать `numpy`, поэтому рекомендуется сокращать и рационально использовать сэкономленное время.

## База

### Инициализация векторов и матриц

```
In [51]: vec = np.array([1, 2, 3, 4]) # передаём список
matrix = np.array([
    [1, 2, 3],           # передаём список списков
    [4, 5, 6]
])

print(f'Вектор: {vec}')
print(f'Матрица:\n{matrix}\n')

print(f'Размер вектора: {vec.shape}')
print(f'Размер матрицы: {matrix.shape} (строк, столбцов)')
print(f'Длина вектора: {len(vec)}')
print(f'Длина матрицы: {len(matrix)} (sic!)')
```

Вектор: [1 2 3 4]

Матрица:

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

Размер вектора: (4,)

Размер матрицы: (2, 3) (строк, столбцов)

Длина вектора: 4

Длина матрицы: 2 (sic!)

Обратите внимание на последнюю строку. *Длина* матрицы (и другого многомерного объекта) --- это длина *только первого измерения*, а не общее количество элементов.

Также отметим, что все элементы должны быть одного и того же типа.

```
In [52]: vec2 = np.array([1, 2.3, 3, 4])
vec3 = np.array([1, True, False, 4])
vec4 = np.array([1, True, False, 4.])
vec5 = np.array([1, 'x', 2, 3.])

template = '{:4} {:20} {:<21}'
print(template.format('vec', str(vec), str(vec.dtype)))
print(template.format('vec2', str(vec2), str(vec2.dtype)))
print(template.format('vec3', str(vec3), str(vec3.dtype)))
print(template.format('vec4', str(vec4), str(vec4.dtype)))
print(template.format('vec5', str(vec5), str(vec5.dtype)))
```

```
vec   [1 2 3 4]          int64
vec2  [1. 2.3 3. 4.]    float64
vec3  [1 1 0 4]          int64
vec4  [1. 1. 0. 4.]     float64
vec5  ['1' 'x' '2' '3.0'] <U32
```

Для любознательных: разобраться в коде выше, что именно и как он делает.

Для всех остальных --- это способ красиво оформить вывод, по сути тоже самое, что и `print(vec, type(vec[0]))`.

Как видим, при передаче разнородных элементов происходит конвертация:

- Дробное среди целых --- все целые становятся дробным
- Булевая среди целых --- `True` становится `1`, а `False` --- `0`.
- Случай с `vec4` разобрать самостоятельно.
- Стока --- конвертирует всё в строку.

## Функции инициализации

Функция `zeros` --- инициализирует нулевую матрицу заданного размера (по умолчанию типа `float64`). Обратите внимание на индексацию матрицы (`[x, y]` можно использовать также, как и `[x][y]`).

```
In [54]: zeros_matrix = np.zeros((2, 3)) # передаём кортеж длин измерений:
                                                # (строк, столбцов)
print(zeros_matrix, type([0,0]))
```

`[[0. 0. 0.]  
 [0. 0. 0.]] <class 'numpy.float64'>`

Тип можно указывать:

```
In [53]: int_zeros_matrix = np.zeros((2, 3), dtype=int)
print(int_zeros_matrix, int_zeros_matrix.dtype)
```

`[[0 0 0]  
 [0 0 0]] int64`

Тип данных может поменять.

```
In [54]: float_zeros_matrix = int_zeros_matrix.astype(np.float64)
print(float_zeros_matrix, float_zeros_matrix.dtype)
```

`[[0. 0. 0.]  
 [0. 0. 0.]] float64`

`ones` --- матрица, заполненная единицами (не единичная матрица).

```
In [51]: print(np.ones((2, 3)))
```

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

`full` --- создать и заполнить матрицу значением

```
In [53]: print(np.full((2, 3), 6)) # второй (не третий, его тут нет) аргумент --
                                            # желаемое значение
```

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

`identity` и `eye` для создания единичных матриц. `identity` --- прямой как палка: даёшь размер (одним числом, т.к. единичная матрица по определению квадратная), получаешь матрицу.

```
In [55]: print(np.identity(3))
```

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

Функция `eye` интереснее. Она может создавать прямоугольные "единичные" матрицы, но и смещать диагональ вверх или вниз. За это отвечают аргументы `M` (количество столбцов) и `k` соответственно.

```
In [9]: print('np.eye(3):')
print(np.eye(3))
print('\nnp.eye(3, M=5):')
print(np.eye(3, M=5))
print('\nnp.eye(3, M=2):')
print(np.eye(3, M=2))
print('\nnp.eye(3, k=1):')
print(np.eye(3, k=1))
print('\nnp.eye(3, k=-1):')
print(np.eye(3, k=-1))
```

```
np.eye(3):
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]

np.eye(3, M=5):
[[1.  0.  0.  0.  0.]
 [0.  1.  0.  0.  0.]
 [0.  0.  1.  0.  0.]]

np.eye(3, M=2):
[[1.  0.]
 [0.  1.]
 [0.  0.]]

np.eye(3, k=1):
[[0.  1.  0.]
 [0.  0.  1.]
 [0.  0.  0.]]

np.eye(3, k=-1):
[[0.  0.  0.]
 [1.  0.  0.]
 [0.  1.  0.]]
```

```
In [11]: print('np.eye(5, M=2, k=-1):')
print(np.eye(5, M=2, k=-1))
```

```
np.eye(5, M=2, k=-1):
[[0.  0.]
 [1.  0.]
 [0.  1.]
 [0.  0.]
 [0.  0.]]
```

## Создание последовательностей

`np.arange`

Вспоминаем про `range` из прошлого файла. Если по какой-то причине у вас не было его, то можете обновить с моего [githuba](#). Та же самая идея, только возвращается `ndarray`, а также шаг может быть вещественным.

```
In [55]: aranged = np.arange(3)
print(aranged)
print(aranged.dtype)
```

```
[0 1 2]
int64
```

```
In [56]: aranged = np.arange(-2, 3)
print(aranged)
print(aranged.dtype)
```

```
[-2 -1  0  1  2]
int64
```

```
In [57]: aranged = np.arange(-2, 2, 0.4) # здесь range бы сломался
print(aranged)
print(aranged.dtype)
```

```
[-2.000000e+00 -1.600000e+00 -1.200000e+00 -8.000000e-01
 -4.000000e-01 -4.4408921e-16  4.000000e-01  8.000000e-01
  1.200000e+00  1.600000e+00]
float64
```

```
In [58]: aranged = np.arange(1.1, -1, -0.4) # здесь range бы сломался
print(aranged)
print(aranged.dtype)
```

```
[ 1.1  0.7  0.3 -0.1 -0.5 -0.9]
float64
```

`np.linspace`

Идея тоже простая. Указываете начальную точку, указываете конечную (здесь **включительно**) и указываете желаемое количество точек, а функция генерирует массив равномерно расположенных точек.

```
In [21]: np.linspace(-2, 2, 10)
```

```
Out[21]: array([-2.          , -1.55555556, -1.11111111, -0.66666667, -0.22222222,
   0.22222222,  0.66666667,  1.11111111,  1.55555556,  2.        ])
```

```
In [23]: np.linspace(3, 2, 10) # в сторону уменьшения тоже можно
```

```
Out[23]: array([3.          , 2.88888889, 2.77777778, 2.66666667, 2.55555556,
   2.44444444, 2.33333333, 2.22222222, 2.11111111, 2.        ])
```

Для любознательных: разобраться с функцией `logspace`.

## Генерация псевдослучайных чисел

Из курса теории вероятности вы знаете, что "случайность бывает разная". Есть такая штука как распределение и от него зависит как "должны вести себя случайные величины".

Для генерации равномерно распределённой псевдослучайной величины на полуинтервале  $[0; 1)$  используется функция `rand` из `numpy.random`. Можно писать так:

```
In [24]: np.random.rand()
```

```
Out[24]: 0.9547204837595107
```

Можно ввести синоним (как и в случае с `np`, но в там это чуть ли не "обязательная" вещь, то здесь это вкусыщина):

```
In [27]: import numpy.random as rnd
```

```
rnd.rand()
```

```
Out[27]: 0.4162096808724933
```

Можно создать матрицу из псевдослучайных (р.р) чисел, для этого передаём желаемую форму в функцию.

```
In [28]: rnd.rand(3, 2)
```

```
Out[28]: array([[0.68702838, 0.53629233],  
                 [0.24807523, 0.76044603],  
                 [0.88669286, 0.10913238]])
```

Нормальное распределение, как известно, параметрическое, а значит определяется ими (а точнее мат., ожиданием и дисперсией). Следовательно для того, чтобы сгенерировать псевдослучайную величину, распределённую нормально, нужно передать эти параметры, если этого не сделать, то мы получим псевдослучайную, стандартно распределённую величину ( $N(\mu = 0, \sigma = 1)$ ).

```
In [36]: rnd.normal(3, 4) # \mu = 3, \sigma = 4
```

```
Out[36]: 1.3920549337803085
```

```
In [37]: rnd.normal(3, 4, (2, 3)) # 2 строки, 3 столбца
```

```
Out[37]: array([[10.54400637, 3.97854519, -0.02990886],  
                 [-2.70807928, -1.08869334, 1.81491874]])
```

Для любознательных: понять, что делает вызов кода ниже:

```
In [47]: rnd.normal((2, 3, 2), (3, 4, 5))
```

```
Out[47]: array([ 4.36351287, 2.6259662 , -2.86410163])
```

Если нам нужно просто стандартное распределение, то можем воспользоваться функцией `random.randn`, которая работает по тому же принципу, как и `random.rand`.

```
In [49]: rnd.randn(3,2)
```

```
Out[49]: array([[ 0.78520433, -1.12349732],
   [-0.45039312, -0.75701072],
   [-0.48099677,  0.3476676 ]])
```

Для любознательных: найти как генерировать другие распределения.

### Фиксирование зерна

Для повторяемости исполнения кода нужно фиксировать зерно. Это важно, поскольку в машинном обучении часто встречаются алгоритмы со случайной генерацией чего-либо.

```
In [26]: np.random.seed(42) # часто можно увидеть фиксирование зерна числом 42, но это пр
```

## Атрибуты и операции

Мы уже разобрали некоторые атрибуты, как `shape` и `dtype` (тип данных элементов, если вдруг), давайте рассмотрим остальные. Для примера создадим небольшую матрицу.

```
In [60]: matrix = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

print(f'Число измерений {matrix.ndim}\n\
      f'Общее число элементов: {matrix.size}')
```

Число измерений 2  
Общее число элементов: 6

### `reshape` (важно)

Давайте изменим вид нашей матрицы.

```
In [77]: matrix_reshaped = matrix.reshape(3, 2)

print('Оригинал:')
print(matrix)
print('\nИзменённая:')
print(matrix_reshaped)
```

Оригинал:  
[[0 2 3]  
 [4 5 6]]

Изменённая:  
[[0 2]  
 [3 4]  
 [5 6]]

**Важно**, данные не копируются, они передаются по ссылке, если изменится новая матрица, старая также изменится.

```
In [80]: matrix_reshaped[0, 0] = 0
print('Изменённая:')
```

```
print(matrix_reshaped)
print('\nОригинальная:')
print(matrix)
```

Изменённая:

```
[[0 2]
 [3 4]
 [5 6]]
```

Оригинальная:

```
[[0 2 3]
 [4 5 6]]
```

Сделать "плоским". `ravel` тоже работает со ссылками.

```
In [82]: array = matrix.ravel()
array
```

```
Out[82]: array([0, 2, 3, 4, 5, 6])
```

Если нужно скопировать используйте `flatten`.

```
In [85]: flattened = matrix.flatten()
flattened
```

```
Out[85]: array([0, 2, 3, 4, 5, 6])
```

Изменения во `flattened` не изменят оригинальный `matrix`.

```
In [86]: flattened[0] = 1
print(flattened, '\nИсходная матрица:')
print(matrix)
```

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

Исходная матрица:

```
[[0 2 3]
 [4 5 6]]
```

Продемонстрирую, что изменения в `array` (который был создан с помощью `ravel`) изменит оригинальную матрицу.

```
In [87]: array[0] = 1
print(array, '\nИсходная матрица:')
print(matrix)
```

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

Исходная матрица:

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

## Индексация

Индексация одномерных массивов не отличается от индексации в "простом" Python, в том числе и слайсы/срезы, а базовую индексацию многомерных вы видели в [этом разделе](#), так что без повторений двигаемся дальше.

Во-первых, отрицательная индексация многомерных массивов работает.

```
In [92]: print(matrix[-1, -1])
```

6

Во-вторых, для кого это было не очевидно, "одномерным индексом" в матрице индексируются строки, например ниже мы получаем первую строку из нашей матрицы.

```
In [94]: print(matrix[0])
```

[1 2 3]

## Многомерные срезы

С многомерными срезами дела обстоят интереснее. **Важно:** все срезы это **представление, а не копия!** Измените значение в срезе --- изменится значение в исходном массиве.

Для любознательных: найти чем представление (view) отличается от ссылки.

```
In [99]: matrix = (rnd.rand(3,3)*20).astype(int) # создаём случайную матрицу размером 3x3
# умножаем каждый её элемент (из
# полуинтервала [0;1)) на 20 и
# конвертируем в целочисленные для
# простоты и красоты
print(matrix)
```

[[ 6 8 3]  
[ 6 13 9]  
[ 5 5 4]]

```
In [108...]: print(matrix[:2]) # первые две строки, все столбцы
```

[[ 6 8 3]  
[ 6 13 9]]

```
In [113...]: print(matrix[-1, :]) # последняя строка
```

[5 5 4]

```
In [114...]: print(matrix[:, -1]) # последний столбец
```

[3 9 4]

```
In [111...]: print(matrix[:, :2]) # все строки, первые два столбца
```

[[ 6 8]  
[ 6 13]  
[ 5 5]]

```
In [109...]: print(matrix[1:]) # последние две строки, все столбцы
```

[[ 6 13 9]  
[ 5 5 4]]

```
In [103...]: print(matrix[:, ::2]) # все строки, каждый второй столбец
```

[[6 3]  
[6 9]  
[5 4]]

```
In [110...]: print(matrix[:, 1:]) # все строки, последние два столбца
```

```
[[ 8  3]
 [13  9]
 [ 5  4]]
```

```
In [105...]: print(matrix[:, ::2]) # каждая вторая строка, все столбцы
```

```
[[6 8 3]
 [5 5 4]]
```

```
In [106...]: print(matrix[:, ::2, ::2]) # каждая вторая строка, каждый второй столбец
```

```
[[6 3]
 [5 4]]
```

```
In [107...]: print(matrix[::2, 1:]) # первые две строки, последние два столбца
```

```
[[ 8  3]
 [13  9]]
```

Для любознательных: рассмотреть какие варианты я упустил (также потренироваться на матрицах большего размера).

## Булева индексация

Идея проста. Есть массив, есть массив булевых значений того же размера.

Применяем одно к другому, забираем только те элементы первого массива, которые по порядку совпали с `True`-элементами булевого массива.

Рассмотрим на примере. Во время зарождения black metal были споры является та или иная группа истинным (true) black metal или нет. Один из критериев --- количество фанатов, если группа была слишком известной, то она не была настоящим black metal. Ниже представлен массив количества фанатов каких-то групп. Будем считать, что если о группе знает больше двух человек (самого исполнителя и его матери), то группа не может считаться настоящим black metal. Найдите количество true black metal групп.

```
In [116...]: bm_bands_fans = np.array([3, 1, 2, 5, 6, 1, 6, 7, 4, 9]) # количество фанатов
mask = bm_bands_fans <= 2 # выбираем только те, где количество фанатов не больше
print(len(bm_bands_fans), len(mask)) # демонстрация того, что количество элементов
```

```
selected = bm_bands_fans[mask] # выбираем только подходящие
print(len(selected)) # находим количество подходящих
```

```
10 10
3
```

## "Прихотливая" индексация (фанси / фэнси / fancy indexing)

Всё снова довольно просто: массивы могут быть использованы для индексации.

```
In [ ]: arr = np.arange(11)
print(arr[[1, 6, 7]]) # передаём список индексов
print(arr[[1, 9, 3]]) # "неправильный" порядок допустим

inds = np.array([0, 2, 4])
```

```
print(arr[inds])
inds = np.array([0, 8, 4])
print(arr[inds])
```

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

Для многомерных массивов та же история:

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

print(matrix[[0, 2], :]) # первая, третья строка
print(matrix[:, [1, 2]]) # второй, третий столбец
```

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

## Изменение части массива

Можно изменять сразу несколько элементов в массиве (мне же не стоит уточнять, что изменить только один элемент также возможно?).

```
In [7]: arr[1:4] = [100, 50, 75] # напоминаю, что последний индекс не включён
print(arr)
```

```
[ 0 100  50  75  4  5  6  7  8  9 10]
```

## Операции над массивами и broadcasting (броадкастинг / транслирование)

### Арифметические операции

Операции сложения, вычитания, умножения и деления выполняются поэлементно.

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

print(f'Сложение: {a + b}')
print(f'Вычитание: {a - b}')
print(f'Умножение: {a * b}')
print(f'Деление: {a / b}')
```

```
Сложение: [5 7 9]
Вычитание: [-3 -3 -3]
Умножение: [ 4 10 18]
Деление: [0.25 0.4 0.5 ]
```

### Операции со скалярами

```
In [3]: print(f'a + 10: {a + 10}')
print(f'a * 2: {a * 2}')
```

```
a + 10: [11 12 13]
a * 2: [2 4 6]
```

## Поэлементные математические функции

Можно применить одну функцию ко всем элементам массива, без использования функций высшего порядка (см. `map` в прошлом занятии).

```
In [4]: print(np.sin(a))
print(np.log(a))
print(np.exp(a))
```

```
[0.84147098 0.90929743 0.14112001]
[0.          0.69314718 1.09861229]
[ 2.71828183  7.3890561  20.08553692]
```

## Broadcasting

В NumPy можно выполнять арифметические операции над массивами разных размеров, если они удовлетворяют правилам броадкастинга. Один из случаев был показан выше, взаимодействие массива и скаляра.

### Правила

Если размеры массивов не совпадают, NumPy добавляет недостающие оси слева.

Здесь `b` из `(3,)` дополнены до `(3, 3)`

```
In [13]: A = np.random.randint(0, 11, (3, 3))
print('A:')
print(A, A.shape)

b = np.array([10, 20, 30])
print('b: ', b, b.shape)

print(A + b)
```

```
A:
[[0 0 1]
 [6 1 0]
 [7 1 8]] (3, 3)
b: [10 20 30] (3,)
[[10 20 31]
 [16 21 30]
 [17 21 38]]
```

Если размеры осей различаются, но одна из них равна 1, NumPy расширяет её.

Случай: вектор-столбец `A` + вектор-строка `B`. В данной ситуации `A` из `(3, 1)` дополняется до `(3, 3)`, а `b` из `(3, )` --- до `(3, 3)`.

```
In [14]: A = np.array([[1],
                  [2],
                  [3]])
print('A:')
print(A, A.shape)

b = np.array([10, 20, 30])
```

```
print('b: ', b, b.shape)

print(A + b)
```

A:  
[[1]  
[2]  
[3]] (3, 1)  
b: [10 20 30] (3,)  
[[11 21 31]  
[12 22 32]  
[13 23 33]]

```
In [18]: A = np.random.randint(0, 11, (1, 3, 1))
B = np.random.randint(0, 11, (3, 1, 2))

S = A + B
print('A: ')
print(A, A.shape)
print('\nB: ')
print(B, B.shape)
print('\nA+B: ')

print(S, S.shape)
```

A:  
[[[9]  
[8]  
[7]]] (1, 3, 1)  
  
B:  
[[[10 6]]  
  
[[ 5 9]]  
  
[[ 3 4]]] (3, 1, 2)

A+B:  
[[[19 15]  
[18 14]  
[17 13]]  
  
[[14 18]  
[13 17]  
[12 16]]  
  
[[12 13]  
[11 12]  
[10 11]]] (3, 3, 2)

Для любознательных: разобраться в сложении выше.

Если две оси имеют разные размеры и ни одна из них не равна 1, то транслирование невозможно.

```
In [ ]: A = np.random.randint(0, 11, (3, 4))
B = np.random.randint(0, 11, (2, 4))
```

```
print(A + B) # получите ошибку
```

## Статистические и агрегирующие функции

```
In [4]: data = np.random.rand(10)*40 - 20
print(data, '\n')

print(f'Сумма: {np.sum(data)}')
print(f'Среднее значение: {np.mean(data)}')
print(f'Минимум: {np.min(data)}')
print(f'Максимум: {np.max(data)}')
print(f'Медиана: {np.median(data)}')
print(f'Стандартное отклонение: {np.std(data)}')
print(f'Дисперсия: {np.var(data)}')

[ -8.68839584 -18.01410588  17.60733643 -11.92741934   0.22396282
 14.3263759   -9.72140157 -11.68658634  12.32018497   5.08951664]
```

Сумма: -10.470532226879673  
 Среднее значение: -1.0470532226879672  
 Минимум: -18.014105880296935  
 Максимум: 17.60733642524906  
 Медиана: -4.23221650981284  
 Стандартное отклонение: 12.063921204641266  
 Дисперсия: 145.53819483179316

## Оси для многомерных массивов

Можно вычислять статистики по разным осям с помощью аргумента `axis`.

```
In [2]: matrix = np.array([[ 1,  2,  3],
                        [ 4,  5,  6],
                        [ 7,  8,  9],
                        [10, 11, 12]])

print(f"Сумма всей матрицы: {np.sum(matrix)}")
print(f"Сумма по столбцам: {np.sum(matrix, axis=0)}")
print(f"Сумма по строкам: {np.sum(matrix, axis=1)}")
```

Сумма всей матрицы: 78  
 Сумма по столбцам: [22 26 30]  
 Сумма по строкам: [ 6 15 24 33]

Т.е. при `axis=0` операции выполняются по столбцам (и длина такая сколько в матрице столбцов), при `axis=1` операции выполняются по строкам (длина результата равна количеству строк).

## Кумулятивные функции

Кумулятивная сумма часто встречается в анализе временных рядов.

```
In [2]: data = np.array([1, 2, 3, 4])

print(f'Кумулятивная сумма: {np.cumsum(data)}')
print(f'Кумулятивное произведение: {np.cumprod(data)}')
```

Кумулятивная сумма: [ 1 3 6 10]  
 Кумулятивное произведение: [ 1 2 6 24]

## Перцентили

Напомню, что нулевой процентиль --- минимум, процентиль больше нуля, кратный 25 --- это quartile, 50-процентиль --- медиана (и второй quartile, как было сказано ранее), 100-процентиль --- максимум (и четвёртый quartile).

```
In [4]: data = np.arange(1000)/10

print(f'1-процентиль {np.percentile(data, 1)}')
print(f'25-процентиль (первый quartile) {np.percentile(data, 25)}')
print(f'50-процентиль (медиана) {np.percentile(data, 50)}')
print(f'90-процентиль {np.percentile(data, 90)}')

1-процентиль 0.999
25-процентиль (первый quartile) 24.975
50-процентиль (медиана) 49.95
90-процентиль 89.91000000000001
```

Пример повеселее, где показана разница между медианой и средним.

```
In [6]: data = np.arange(70) * np.random.random(70)

print(data, '\n')
print(f'1-процентиль {np.percentile(data, 1)}')
print(f'25-процентиль (первый quartile) {np.percentile(data, 25)}')
print(f'50-процентиль (медиана) {np.percentile(data, 50)}')
print(f'Среднее {np.mean(data)}')
print(f'90-процентиль {np.percentile(data, 90)}')

[ 0.          0.90240344  1.45800022  0.27630902  2.45479283  4.53236199
 4.76035066  6.08668048  6.24310833  3.07854606  1.09721717  8.02802214
 7.78671296  2.77472397  3.49308003  7.15357399  7.53382074  6.2716958
15.37760631  8.97698995  10.84120226  4.62974171  21.80422507  14.14941728
 8.2736367  17.44793406  16.00307578  17.02434333  12.30460855  26.9649619
10.65394754  13.22335023  24.20212244  9.11680323  11.19105833  0.45537069
 3.95939517  30.90864483  3.28880964  28.93492207  20.04184068  0.90121968
 1.21737773  3.57577042  13.65478081  33.34732945  30.34325202  17.51824827
26.66517452  26.40509528  13.24023738  0.49492286  46.20234841  0.20573366
37.24059624  49.45484384  11.23379797  9.73480527  5.78573963  46.33901959
31.5338908  21.40555045  7.53791919  59.56995659  53.1698122  56.80984754
50.01141362  66.15301377  47.47233173  14.22637746]
```

1-процентиль 0.14195622717554618  
 25-процентиль (первый quartile) 4.556706921508723  
 50-процентиль (медиана) 11.016130293379126  
 Среднее 16.930797342524045  
 90-процентиль 46.45235080353714

## NaN

В статистике часто попадаются недостающие данные (NaN, N/A, null и другие способы их записи), из-за которых ломаются обычные вычисления. Вообще тактика работы с пропущенными данными зависит от конкретной задачи и объёма доступных данных, иногда их можно просто проигнорировать, а иногда [для того чтобы проигнорировать] придётся выкинуть большую часть данных, что

совершенно недопустимо (но опять же всё зависит от конкретного случая). По тактике устранения пропусков эмпирические правила, но нет конкретных правил, что делать, разве что **не нужно** делать. Ниже пример данных с пропуском.

```
In [5]: data = np.array([1, 2, np.nan, 4, 5])
print(data)
```

```
[ 1.  2. nan  4.  5.]
```

Если мы их просто просуммируем, то получим NaN (т.е. потеряем пользу от всех данных).

```
In [6]: print(np.sum(data))
```

```
nan
```

В NumPy есть функции, которые игнорируют отсутствующие данные (но опять же, это не всегда правильный метод).

```
In [7]: np.nansum(data)
```

```
Out[7]: np.float64(12.0)
```

Для любознательных: найти и прочитать другие "nan"-функции.

## Коэффициенты корреляции и ковариации

```
In [11]: x = np.arange(5)
y = np.arange(0, 10, 2)

print(f'x: {x}')
print(f'y: {y}')
print()
print('Коэффициент корреляции:\n{np.corrcoef(x, y)}')
print()
print('Ковариация:\n{np.cov(x, y)}')
```

```
x: [0 1 2 3 4]
y: [0 2 4 6 8]
```

Коэффициент корреляции:

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

Ковариация:

```
[[ 2.5  5. ]
 [ 5.   10. ]]
```

## Объединение и разделение массивов

### Объединение

`concatenate`

Функция `concatenate()` объединяет массивы вдоль указанной оси.

```
In [7]: a = np.arange(4)
b = np.arange(4, 7)

c = np.concatenate((a, b)) # кортеж, один параметр, а не несколько
print(c)

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

```
In [20]: A = np.array([[1, 2],
                  [3, 4]])
B = np.array([[5, 6]])

C = np.concatenate((A, B)) # axis=0 по умолчанию
print(f'C:\n{C}')
D = np.concatenate((A, B.T), axis=1) # матрица B транспонированная (2x1)
print(f'D:\n{D}')

C:
[[1 2]
 [3 4]
 [5 6]]
D:
[[1 2 5]
 [3 4 6]]
```

Т.е. для матрицы в первом случае добавили строку, а во втором --- столбец.  
Правила похожи на правила broadcasting'a.

### Вертикальное и горизонтальное объединение.

```
In [24]: C = np.hstack((A, B.T)) # горизонтальное
print(f'C:\n{C}')

D = np.vstack((A, B)) # по вертикали
print(f'D:\n{D}')

C:
[[1 2 5]
 [3 4 6]]
D:
[[1 2]
 [3 4]
 [5 6]]
```

### Создание нового измерения

```
In [27]: a = np.arange(3)
b = np.arange(4, 7)

C = np.stack((a, b)) # добавили новую строку (увеличение по оси 0)
print(f'C:\n{C}')

D = np.stack((a, b), axis=1) # теперь добавили по оси 1, обратите
                             # внимание как теперь расположены исходные
                             # векторы
print(f'D:\n{D}')
```

```
C:  
[[0 1 2]  
 [4 5 6]]  
D:  
[[0 4]  
 [1 5]  
 [2 6]]
```

## Разделение

`split`

Функция `split` разделяет массив на несколько частей по указанным индексам.

```
In [28]: a = np.arange(7)  
b, c, d = np.split(a, [2, 4]) # список индексов с которых начинается  
# новый массив  
print(f'b: {b}\nc: {c}\nd: {d}'')
```

```
b: [0 1]  
c: [2 3]  
d: [4 5 6]
```

## Вертикальное/горизонтальное разделение

```
In [29]: A = np.array([[1, 2, 3],  
 [4, 5, 6]])  
left, right = np.hsplit(A, [2]) # разрез по третьему столбцу  
print(f'left:\n{left}')  
print(f'right:\n{right}'')
```

```
left:  
[[1 2]  
 [4 5]]  
right:  
[[3]  
 [6]]
```

```
In [34]: B = np.array([[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]])  
top, bottom = np.vsplit(B, [2]) # разрез по третьему столбцу  
print(f'top:\n{top}')  
print(f'bottom:\n{bottom}'')
```

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

Для любознательных: понять, что происходит в следующей ячейке.

```
In [32]: A = np.array([[[1, 2, 3],  
 [4, 5, 6]],  
 [[7, 8, 9],  
 [10, 11, 12]]])  
left, right = np.hsplit(A, [1])  
print(f'A:\n{A}')  
print()
```

```
print(f'left:\n{left}')
print()
print(f'right:\n{right}')
```

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

```
[[ 7  8  9]
 [10 11 12]]]
```

```
left:
[[[1 2 3]]]
```

```
[[7 8 9]]]
```

```
right:
[[[ 4  5  6]]]
```

```
[[10 11 12]]]
```

## Линейная алгебра

Помимо теорвера со статистикой и доли функционального анализа, в машинном обучении ещё очень много линейной алгебры.

### Умножение матриц

Матрицы можно перемножить аж тремя способами, давайте разберём сразу все

Особенность	<code>np.dot(A, B)</code>	<code>A @ B (оператор)</code>	<code>np.matmul(A, B)</code>
<b>Поддержка 1D × 1D</b>	Скалярное произведение (вернёт число).	Скалярное произведение	Скалярное произведение
<b>Поддержка 2D × 2D</b>	Обычное матричное умножение.	Обычное матричное умножение.	Обычное матричное умножение
<b>Broadcasting</b>	Нет	Да	Да

А теперь детальнее.

```
np.dot
```

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

# 1D x 1D
print(np.dot(a, b)) # 1*4 + 2*5 + 3*6 = 32 (скалярное произведение)

# 2D x 2D
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(np.dot(A, B)) # Матричное произведение
```

```
32
[[19 22]
 [43 50]]
```

```
In [ ]: # 1D x 2D
a = np.array([1, 2, 3])
B = np.array([[4, 5],
              [6, 7],
              [8, 9]])

C = np.dot(a, B)
print(C)
```

`@` и `np.matmul`

`@` введён для удобства записи. Можно использовать вместо него `np.matmul`, но не нужно.

```
In [10]: print(A @ B)
print('\n', a @ b)

print()

print(np.matmul(A, B))
print('\n', np.matmul(a, b))
```

```
[[19 22]
 [43 50]]
```

32

```
[[19 22]
 [43 50]]
```

32

Для любознательных: понять произведение многомерных (>2 измерений) массивов.

Для крайне любознательных: понять произведение многомерных массивов с разным количеством измерений.

Для `np.dot` и `np.matmul` **логика разная!** Во всяком случае, должно быть так, но редко когда нужно думать больше чем о двух измерениях в контексте перемножения матриц, поэтому я не стал.

## Транспонирование

Тут просто. `.T`

```
In [12]: print(A)
print(A.T)
```

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

На самом деле чуть-чуть не просто.

```
In [28]: print(a, a.shape)
```

```
print(a.T) # ?!
```

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

```
[1 2 3]
```

Почему вектор-строка не стала вектор столбцом?

Во-первых, у нас не вектор-строка, а вектор столбец. Да, выводит как строка, но нужно держать в памяти, что это столбец. Т.е. правильный вопрос: почему вектор-столбец не стал вектор-строкой?

Во-вторых, пр считает, что тут только одна ось (что как бы правда), а значит транспонировать нечего (что как бы не очень правда).

Решение: добавить измерение.

```
In [33]: a = a.reshape(3, 1) # теперь видно, что это вектор-столбец
```

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

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

```
[[1]
```

```
[2]
```

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

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

## Определитель

```
In [14]: det_A = np.linalg.det(A)
```

```
det_A
```

```
Out[14]: -2.0000000000000004
```

## Обратная матрица

Тоже довольно очевидная вещь.

```
In [15]: A_inv = np.linalg.inv(A)
```

```
print(A_inv)
```

```
[[ -2.    1. ]
```

```
 [ 1.5 -0.5]]
```

Но будьте осторожны с вырожденными матрицами, если определитель 0, то выкинет ошибку.

```
In [18]: B = np.matrix([[1, 1],
```

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

```
# B_inv = np.linalg.inv(B) # выдаст ошибку
```

```
# print(B_inv)
```

## Псевдо-обратная

Часто встречается (например, в линейной регрессии) псевдо-обратная матрица Мура-Пенроуза, которую можно найти даже если определитель 0.

```
In [25]: B_pinv = np.linalg.pinv(B)
print(B_pinv)
```

```
[[0.25 0.25]
 [0.25 0.25]]
```

## Собственные вектора и собственные значения

Очень интересная тема, но придётся оставить вас без весёлой математики.

```
In [22]: eigenvalues, eigenvectors = np.linalg.eig(A)
print(f'Собственные значения: {eigenvalues}')
print(f'Собственные вектора: {eigenvectors}') # два собственных вектора, но пред
```

```
Собственные значения: [-0.37228132  5.37228132]
Собственные вектора: [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

## Для любознательных

(напоминание, ссылки почему-то работают только в html и pdf версии, VS Code не хочет)

1. [Здесь](#)
2. [Здесь](#)
3. [Здесь](#)
4. [Здесь](#)
5. [Здесь](#)
6. [Здесь](#)
7. [Здесь](#)
8. [Здесь](#)
9. [Здесь](#)
10. [Здесь](#)
11. [Здесь](#)