Библиотеки. Часть 3

План урока

1. Вычислительные возможности Python. Numpy
2. Как замерять скорость?
3. Массивы в Numpy
4. Размерность массива
5. Индексация в массивах
6. Операции с массивами
    A. Массовые операции
    B. Немного о матрицах
    C. Сортировки
    D. Вспомним PIL
7. *Игра «Жизнь»

Аннотация

Как мы уже говорили, Python — язык для быстрой разработки. Однако чистый Python не предназначен для написания быстрых программ. Это интерпретируемый язык, поэтому программы на Python выполняются медленнее аналогов на C, C++ или Fortran. С другой стороны математики, физики, биологи и инженеры часто применяют Python для решения вычислительных задач, именно вычислительным мощностям Python и библиотеки Numpy посвящен этот урок.

Нет ли тут противоречия? Как интерпретируемый язык может быть эффективен в вычислительной математике?

Оказывается, всё дело в библиотеках. Python отлично подходит на роль промежуточной среды, оболочки, «клея» между библиотеками, написанными на разных языках.

В этом уроке мы поговорим про наиболее фундаментальную библиотеку для работы с вычислительной математикой — **numpy**. Мы затронем работу с многомерными массивами и изучим (или вспомним) немного линейной алгебры.

Многие другие пакеты для работы с данными и вычислениями используют базовые объекты этой библиотеки. В числе таких пакетов **OpenCV** — открытая библиотека для работы с распознаванием образов — и **Pandas** — библиотека, ориентированная на анализ данных.

Как замерять скорость?

В Python для замера времени работы кода служит библиотека `timeit`.

Например, мы можем замерить три разных способа заполнить список из миллиона квадратных корней.

```
import timeit

print(timeit.timeit("[sqrt(x) for x in range(1000000)]",
                    "from math import sqrt", number=1))

print(timeit.timeit("for i in range(1000000): a.append(sqrt(i))",
                    "from math import sqrt; a=[]", number=1))

print(timeit.timeit("list(map(sqrt, range(1000000)))",
                    "from math import sqrt; a=[]", number=1))
```

```
0.18135334600083297
0.21764946899929782
0.1530561779945856
```

Как видим, в этой версии интерпретатора (3.6) предпочтительно использовать `map`. Самый медленный способ — это, конечно же, динамическое расширение существующего списка (**append**). Причем, чем больше список — тем медленнее он меняет свой размер. Это вызвано необходимостью иногда переносить данные из одного места списка в другое.

Несмотря на относительную быстроту (0,13 секунды на извлечение 1000000 квадратных корней), скорость можно увеличить ещё примерно в 10 раз. Давайте посмотрим как.

Массивы в Numpy

Основной объект в **Numpy** — это многомерный массив. Массивы — одна из базовых структур данных, которая позволяет моделировать многие объекты, относящиеся как к науке, так и к обычной жизни: список покупок, результаты наблюдения температуры, матрицы и вектора, изображения, видео и т. д.

Напомним, что в чистом Python нет типа данных с именем **массив**, и нам приходится моделировать его с помощью списков. Другое дело **numpy**. За тип массива здесь отвечает объект `array`.

Как же создать массив?

Ну, во-первых, массив можно сделать из обычного списка:

```
import numpy as np

np.array([1, 4, 10, 34])
```

```
array([ 1,  4, 10, 34])
```

или из диапазона:

```
np.array(range(10))
```

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

А можно сделать и из итератора с помощью функции `fromiter()`:

```
np.fromiter(map(int, ["1", "2", "3", "4"]), dtype=np.int8)
```

```
array([1, 2, 3, 4], dtype=int8)
```

В **Numpy** элементы одного массива должны быть однородны (одного типа). Это — самое важное идеологическое отличие массивов от списков, в которых можно хранить объекты разной природы.

```
np.array([11, 234.5, "hello"]) # => array(['11', '234', 'hello'], dtype='<U11')
```

```
array(['11', '234.5', 'hello'],
      dtype='<U32')
```

Numpy создаст массив из юникод-строк длиной 11. За тип элементов в большинстве случаев отвечает параметр `dtype(data type)`. Обратите внимание на тип данных элементов массива. Посмотрите так же на использование параметра `dtype`:

```
a = np.array([1, 3, 8])
a
```

```
array([1, 3, 8])
```

```
a.dtype
```

```
dtype('int64')
```

```
type(a[0])
```

```
numpy.int64
```

```
a = np.array([1, 3, 8], dtype=np.float64)
type(a[0])
```

```
numpy.float64
```

Указание типов и работа с ними нужны, поскольку языки, на которых написана эта библиотека, строго типизованы. Вдобавок это увеличивает скорость обработки данных.

Размерность массива

Размерность массива можно в любой момент изменить операцией `reshape`. Узнать размерность можно атрибутом `shape`.

```
a = np.arange(100)
a.shape
```

```
(100,)
```

```
a.reshape(10, 10)
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

```
a.reshape(5, 20)
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 1
6,
        17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 3
6,
        37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 5
6,
        57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 7
6,
        77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 9
6,
        97, 98, 99]])
```

```
a.reshape(5, 5, 4)
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]],

       [[20, 21, 22, 23],
        [24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35],
        [36, 37, 38, 39]],

       [[40, 41, 42, 43],
        [44, 45, 46, 47],
        [48, 49, 50, 51],
        [52, 53, 54, 55],
        [56, 57, 58, 59]],

       [[60, 61, 62, 63],
        [64, 65, 66, 67],
        [68, 69, 70, 71],
        [72, 73, 74, 75],
        [76, 77, 78, 79]],

       [[80, 81, 82, 83],
        [84, 85, 86, 87],
        [88, 89, 90, 91],
        [92, 93, 94, 95],
        [96, 97, 98, 99]]])
```

Например, фильм можно представить в виде 4-мерного массива кадров. Кадр — это картинка, то есть трёхмерный массив. Его также можно представить как двумерный массив пикселей, где каждый пиксель — одномерный массив из трёх элементов: R, G, B.

Самое главное — при использовании функции `reshape()` произведение ее параметров должно быть равно количеству элементов в массиве. Иначе мы получим ошибку:

```
a.reshape(2,3,4)
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
<ipython-input-31-a907d0800243> in <module>()
----> 1 a.reshape(2,3,4)

ValueError: cannot reshape array of size 100 into shape (2,3,4)
```

Индексация в массивах

Давайте рассмотрим массив 10x10, созданный ранее.

```
a = a.reshape(10,10)
a
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

В нём работает привычная индексация.

```
a[1][2]
```

```
12
```

```
a[2][1]
```

```
21
```

```
a[5]
```

```
array([50, 51, 52, 53, 54, 55, 56, 57, 58, 59])
```

По аналогии со списками мы можем применять срезы:

```
a[3:5, 1:6]
```

```
array([[31, 32, 33, 34, 35],
       [41, 42, 43, 44, 45]])
```

```
a[:, 2:4]
```

```
array([[ 2,  3],
       [12, 13],
       [22, 23],
       [32, 33],
       [42, 43],
       [52, 53],
       [62, 63],
       [72, 73],
       [82, 83],
       [92, 93]])
```

Кроме того, доступ можно организовать через списки с индексами:

```
a[[1], [4, 4, 7, 8]]
```

```
array([14, 14, 17, 18])
```

Но самой удобной альтернативой «обычному способу» является тот, в котором в качестве "адреса" элемента используется кортеж координат:

```
a[(7, 9)]
```

```
79
```

Скобки, конечно, же можно опустить

```
a[7,9]
```

```
79
```

Массовые операции

Инициализация:

```
np.ones(10) # заполняем единицами
```

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

```
np.ones(10, dtype=np.int32) # заполняем единицами целого типа
```

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

```
np.zeros(30).reshape(5, 6) # заполняем нулями и сразу указываем форму
```

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

```
np.random.randint(1, 10, (5, 5)) # заполняем случайными целыми из диапазона [1..10] и с
разу указать форму
```

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

```
np.random.random(10) # заполняем случайными вещественными числами из диапазона [0..1)
```

```
array([ 0.96373978,  0.73252097,  0.32213768,  0.81560531,  0.31843467,
        0.51289493,  0.99791928,  0.5999641 ,  0.15257882,  0.16074567])
```

```
np.fromstring("1, 3, 4, 5, 120", sep=",") # формируем массив из строки чисел, указывая
 разделитель
```

```
array([   1.,    3.,    4.,    5.,  120.])
```

```
np.fromfunction(lambda x, y: x*5+y, (3, 5), dtype=np.int8) # каждый элемент массива выч
исляется по функции
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]], dtype=int8)
```

Работают основные операции арифметики, сравнения, причем на всем массиве целиком. И это здорово!

```
a = np.random.randint(1, 5, 10)
b = np.random.randint(1, 5, 10)
print(a)
print(b)
```

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

```
print(a>b)
print(a+b)
print(a*b)
print(a**2)
print(a[a>b])
print(a.sum())
print(np.sqrt(a))
```

```
[False  True False False  True False False False  True  True]
[5 4 2 4 4 8 6 7 4 4]
[ 4  3  1  3  3 16  8 12  3  3]
[ 1  9  1  1  9 16  4  9  9  9]
[3 3 3 3]
24
[ 1.          1.73205081  1.          1.          1.73205081  2.
  1.41421356  1.73205081  1.73205081  1.73205081]
```

Тригонометрические операции тоже работают, только надо использовать их версии из библиотеки **numpy**, а не из **math**:

```
print(np.sin(a))
```

```
[ 0.84147098  0.14112001  0.84147098  0.84147098  0.14112001 -0.7568025
  0.90929743  0.14112001  0.14112001  0.14112001]
```

Вспомним начало этого урока, когда мы говорили о скорости работы. Решим ту же задачу по вычислению 1 миллиона корней с помощью **numpy** и посмотрим, какого ускорения мы добились:

```
timeit.timeit("np.sqrt(np.arange(1000000))", "import numpy as np", number=1)
```

```
0.014551434993336443
```

Немного о матрицах

Numpy позволяет очень эффективно работать с двумерными и вообще n-мерными массивами.

```
# Заполним матрицу "последовательно" по строкам
a = np.arange(1,21).reshape(4,5)
a
```

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20]])
```

```
# Транспонируем матрицу (строки станут столбцами, а столбцы строками)
b = a.transpose()
b
```

```
array([[ 1,  6, 11, 16],
       [ 2,  7, 12, 17],
       [ 3,  8, 13, 18],
       [ 4,  9, 14, 19],
       [ 5, 10, 15, 20]])
```

```
# Повернем матрицу вправо и влево
b = np.rot90(a)
c = np.rot90(a, -1)
print(b)
print(c)
```

```
[[ 5 10 15 20]
 [ 4  9 14 19]
 [ 3  8 13 18]
 [ 2  7 12 17]
 [ 1  6 11 16]]
[[16 11  6  1]
 [17 12  7  2]
 [18 13  8  3]
 [19 14  9  4]
 [20 15 10  5]]
```

```
x = np.array([[1,2,3], [4,5,6]])
y = np.array([[0, 1, -1, 2], [2, -1, 2, 0], [0, 1, 1, 0]])
print(np.dot(x,y))
```

```
[[ 4  2  6  2]
 [10  5 12  8]]
```

```
d = np.arange(1, 82). reshape(9, 9)
np.linalg.det(d)
```

```
0.0
```

Сортировки

Библиотека **numpy** предлагает свои функции по сортировке. Давайте посмотрим, как они работают:

```
# заполним матрицу случайными целыми числами
a = np.random.randint(20, size=(5,4))
a
```

```
array([[ 2, 12,  0,  7],
       [12,  1, 11, 18],
       [ 6,  1,  4, 10],
       [ 0,  9,  5,  3],
       [18, 12, 12,  5]])
```

```
# отсортируем матрицу по умолчанию
print(np.sort(a))
```

```
[[ 0  2  7 12]
 [ 1 11 12 18]
 [ 1  4  6 10]
 [ 0  3  5  9]
 [ 5 12 12 18]]
```

В этом случае сортировка происходит по **последнему** измерению. Обходя матрицу, мы сначала выбираем строку, а потом идем по этой строке, поэтому последнее измерение в данном случае — это строка. В результате мы отсортировали независимо каждую строку.

```
# а теперь укажем конкретное измерение
print(np.sort(a, axis=0))
```

```
[[ 0  1  0  3]
 [ 2  1  4  5]
 [ 6  9  5  7]
 [12 12 11 10]
 [18 12 12 18]]
```

Матрица отсортирована по столбцам.

Но самое интересное в том, что если в качестве значения параметра **axis** указать **None**, то матрица перед сортировкой будет линеаризована, то есть превращена в одномерный массив.

```
print(np.sort(a, axis=None))
```

```
[ 0  0  1  1  2  3  4  5  5  6  7  9 10 11 12 12 12 18 18]
```

Обратите внимание, что подобное поведение характерно не только для функции **sort()**, но и для многих других функций: **min()**, **sum()** и т.д.

Но об этом вы можете почитать самостоятельно на странице с документацией (https://docs.scipy.org/doc/numpy/genindex.html).

Вспомним PIL

Работая с библиотекой PIL тоже можно использовать средства **numpy**. Например, если мы хотим сделать изображение темнее оригинала, то можем просто поделить его составляющие, например, на 10:

```
from PIL import Image
import numpy as np

# получим массив питру из картинки, которую откроем из файла.
image = np.asarray(Image.open('images/Риана.jpg'))
# поделим все элементы массива на 10, приведем к типу uint8 (один байт без знака)
# преобразуем в изображение и сохраним в файл
Image.fromarray(np.uint8(image // 10)).save('r2.jpg')
```

Игра «Жизнь»

Несколько десятилетий назад Джон Конуэй придумал один из самых известных клеточных автоматов, который назвал игрой «Жизнь». Простота правил сочетается в ней с богатством результатов. Многие компьютерные инженеры хоть раз обращались к программированию и исследованию этой игры, которая послужила интересной моделью для многих отраслей науки.

Клеточный автомат — это модель однородного пространства с некоторыми клетками. Каждая клетка может находиться в одном из нескольких состояний и иметь некоторое количество соседей. Задаются правила перехода из одного состояния в другое в зависимости от текущего состояния клетки и её соседей.

Пространство «Жизни» — бесконечное поле клеток. Каждая клетка имеет 8 соседей (сверху, снизу, справа, слева и по диагонали). Клетка может иметь два состояния: живая (на клетке стоит фишка) и мёртвая (фишки нет). Правила изменения следующие:

- Если клетка была живой, то она выживет, если у неё 2 или 3 соседа. Если соседей 4, 5, 6, 7 или 8, то она умирает от перенаселённости, а если 0 или 1 — то от одиночества.
- Новая клетка рождается в поле, у которого есть ровно 3 соседа.

Время в этой игре дискретно и считается поколениями. Всё начинается с начальной расстановки фишек (0 поколение), в дальнейшем рассматривается эволюция клеточного пространства в 1, 2, 3 и т. д. поколении. Процессы смерти и рождения происходят одновременно, после чего строится следующее поколение.

Давайте попробуем написать игру «Жизнь», используя библиотеку **numpy**. Пусть у нас будет поле 10 x 10, в центр которого поместим конструкцию, известную как «глайдер». Мы скоро выясним, почему она так называется.

```
import numpy as np

population = np.array(
    [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
     [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=np.uint8)
```

Поле имеет тип `uint8`, чтобы оно занимало меньше памяти. Каждый его элемент занимает ровно 1 байт (8 бит) и является целым беззнаковым (`unsigned`) числом в диапазоне от 0 до 255.

Живые клетки обозначаются единицей, а мёртвые — нулём. Нужно решить, что делать на границах поля. Мы не можем обеспечить бесконечность в обоих направлениях, поэтому замкнём поле само на себя. ЕСли выйти за нижнюю границу, мы окажемся наверху, а если за правую — появимся слева, и наоборот. Получается что-то вроде глобуса.

Для начала познакомимся с операцией `roll`, доступной для массивов. Она сдвигает исходный массив вдоль одного из измерений (в данном случае — строки или столбца).

```
population
```

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

```
np.roll(population, 2, 0)
```

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

```
np.roll(population, 2, 1)
```

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

Мы можем посчитать количество соседей у каждой клетки, просто сделав 8 копий со сдвигом массива и просуммировав их.

```
neighbors = sum([
    np.roll(np.roll(population, -1, 1), 1, 0),
    np.roll(np.roll(population, 1, 1), -1, 0),
    np.roll(np.roll(population, 1, 1), 1, 0),
    np.roll(np.roll(population, -1, 1), -1, 0),
    np.roll(population, 1, 1),
    np.roll(population, -1, 1),
    np.roll(population, 1, 0),
    np.roll(population, -1, 0)
])
```

Таки образом, матрица количества соседей выглядит так:

```
neighbors
```

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 1, 2, 1, 0, 0, 0],
       [0, 0, 1, 3, 5, 3, 2, 0, 0, 0],
       [0, 0, 1, 1, 3, 2, 2, 0, 0, 0],
       [0, 0, 1, 2, 3, 2, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

Теперь нужно получить новую популяцию. Выполним на матрице следующую операцию: «если у клетки 3 соседа, то в следующем поколении на этом месте будет клетка; а если 2 соседа, то клетка будет при условии, что она была "жива" в текущем поколении». Для этого воспользуемся операторами | (или) и & (и).

```
# выделим клетки, у которых ровно три соседа
neighbors == 3
```

```
array([[False, False, False, False, False, False, False, False, False,
        False],
       [False, False, False, False, False, False, False, False, False,
        False],
       [False, False, False, False, False, False, False, False, False,
        False],
       [False, False, False, False, False, False, False, False, False,
        False],
       [False, False, False,  True, False,  True, False, False, False,
        False],
       [False, False, False, False,  True, False, False, False, False,
        False],
       [False, False, False, False,  True, False, False, False, False,
        False],
       [False, False, False, False, False, False, False, False, False,
        False],
       [False, False, False, False, False, False, False, False, False,
        False],
       [False, False, False, False, False, False, False, False, False,
        False]], dtype=bool)
```

```
# а теперь те, в которых была жизнь и имеется ровно два соседа
population & (neighbors == 2)
```

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

```
# и объединим их
population = (neighbors == 3) | (population & (neighbors == 2))
population
```

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

Объединить матрицы с логическими и целочисленными элементами можно, поскольку они в данном случае могут быть сведены друг к другу: 0 — False, 1 — True.

Проследим эволюцию глайдера на протяжении 4 поколений. Для этого создадим функцию `next_population()`.

```
def next_population(population):
    neighbors = sum([
        np.roll(np.roll(population, -1, 1), 1, 0),
        np.roll(np.roll(population, 1, 1), -1, 0),
        np.roll(np.roll(population, 1, 1), 1, 0),
        np.roll(np.roll(population, -1, 1), -1, 0),
        np.roll(population, 1, 1),
        np.roll(population, -1, 1),
        np.roll(population, 1, 0),
        np.roll(population, -1, 0)
    ])
    return (neighbors == 3) | (population & (neighbors == 2))

population = np.array(
    [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
     [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=np.uint8)

for _ in range(4):
    print(population, '\n')
    population = next_population(population)
```

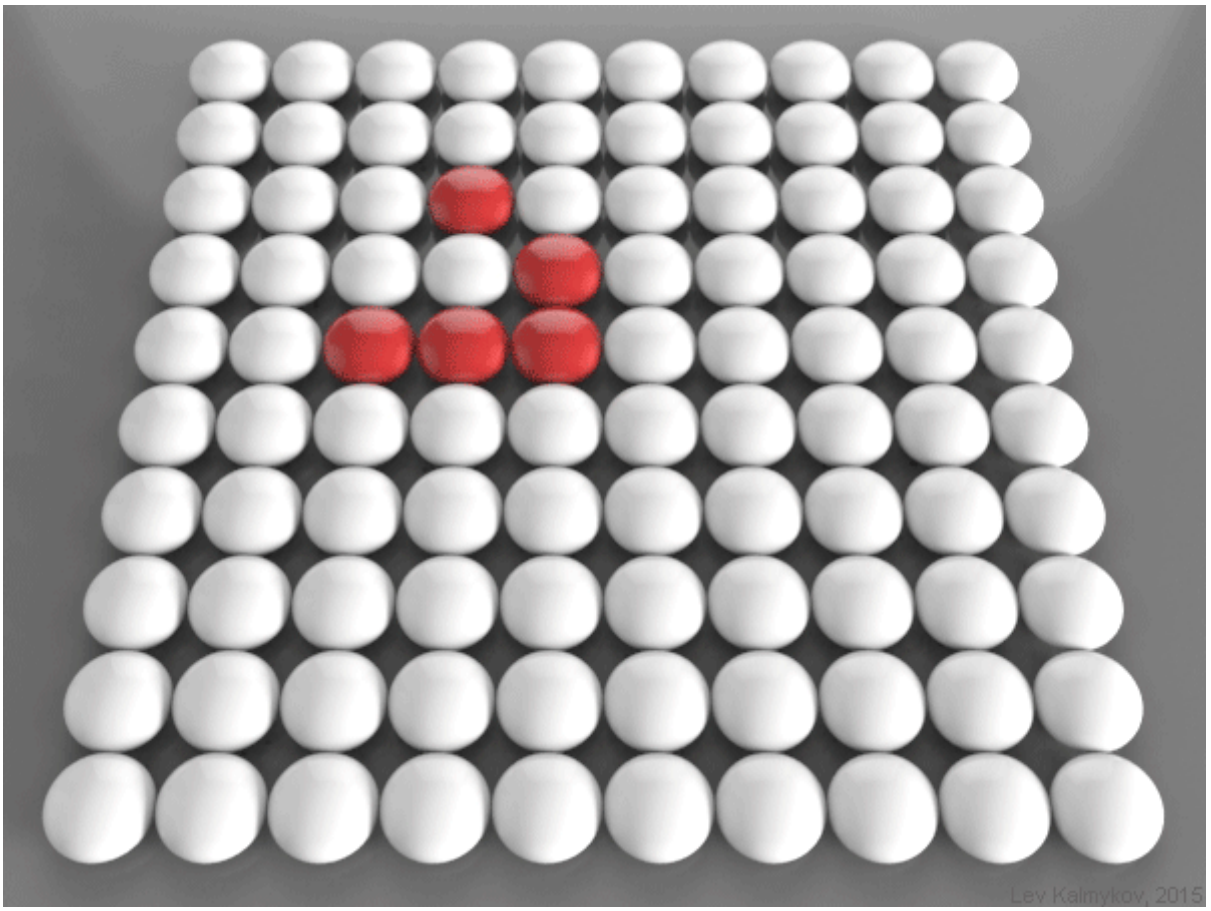
```
[[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0]
 [0 0 0 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]]

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

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

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

С визуализацией у нас не очень здорово, но видно, что глайдер «летит»: каждые четыре поколения он сдвигается вниз и вправо. Иными словами, он движется в правый нижний угол, что демонстрирует красивая анимация (источник (https://ru.wikipedia.org/wiki/Игра_«Жизнь»)).



За время поисков были найдены разнообразные движущиеся комбинации, периодические комбинации, порождающие глайдеры («глайдерные ружья»). Была даже доказана возможность построить в игре «Жизнь» универсальную вычислительную машину.

Итоги

Мы увидели, что значительную часть вычислений можно реализовывать в библиотеках, избавляясь от циклов в Python и ускоряя вычисления. Массивы **numpy** — одна из самых востребованных структур в вычислительной математике.