Основы NumPy: массивы и векторные вычисления

С.В. Лемешевский (`sergey.lemeshevsky@gmail.com`)

Институт математики НАН Беларуси

Mar 18, 2020

Содержание

1 Объект многомерных массивов `ndarray` 3
- 1.1 Создание массива . . . 4
- 1.2 Арифметические операции с массивами NumPy . . . 6
- 1.3 Основы индексирования и срезы . . . 7
- 1.4 Логическое (Boolean) индексирование . . . 10
- 1.5 Необычное индексирование . . . 13
- 1.6 Транспонирование массивов и замена осей . . . 15

2 Универсальные функции: быстрые поэлементные функции от массивов 16

3 Программирование с использованием массивов 20
- 3.1 Выражение условной логики в операциях с массивами . . . 22
- 3.2 Математические и статистические методы . . . 23
- 3.3 Методы для булевых массивов . . . 25
- 3.4 Сортировка . . . 25
- 3.5 Основные операции над множествами для массивов . . . 27

4 Чтение и запись в/из файлы массивов 28

5 Линейная алгебра 28

6 Генерация псевдослучайных числел 31

7 Пример: Случайное блуждание 32

8 Упражнения **34**
8.1 Матрица из нулей и единиц 34
8.2 Моделирование нескольких случайных блужданий за раз 34

Рассматривается библиотека NumPy. NumPy (сокращение от *Numerical Python*) предоставляет эффективный интерфейс для хранения и работы с данными. В какой-то степени массивы NumPy аналогичны спискам Python, но массивы NumPy обеспечивают гораздо более эффективное хранение и операции с данными при увеличении массивов в размере. Если NumPy установлен его можно импортировать следующей командой:

```
import numpy
```

Большинство людей в мире научного Python импортируют NumPy, используя `np` в качестве псевдонима:

```
import numpy as np
```

О встроенной документации
Как было рассказано выше IPython дает возможность быстро просмотреть содержание пакета, а также документацию по функциям.

Например, для того чтобы отобразить содержание пространства имен `numpy`, можно выполнить следующее:

```
In [1]: np.<TAB>
```

Для отображения встроенной документации NumPy можно набрать следующее:

```
In [2]: np?
```

Более детальную документацию вместе с учебниками и другими ресурсами можно найти по адресу: `https://numpy.org/`

Одна из причин того, почему NumPy настолько важен для численных расчетов в Python, заключается в том, что он разработан, как уже упоминалось выше, для более эффективной работы с большими массивами данных, а именно:

- NumPy хранит данные в непрерывных блоках памяти, независимо от других встроенных объектов Python. Библиотека алгоритмов NumPy, написанная на языке C, может работать на этой памяти без проверки типов или других накладных расходов. Массивы NumPy также используют намного меньше памяти, чем встроенные объекты Python.

- Операции NumPy выполняют сложные вычисления над всем массивом без использования цикла `for`.

Для того чтобы показать разницу в производительности, рассмотрим массив NumPy из одного миллиона целых чисел и эквивалентный список:

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

In [2]: my_arr = np.arange(1000000)

In [3]: my_list = list(range(1000000))
```

```
In [4]: %%time for _ in range(10): my_arr2 = my_arr*2
CPU times: user 18 ms, sys: 26.4 ms, total: 44.4 ms
Wall time: 86.1 ms
```

```
In [5]: %%time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 544 ms, sys: 105 ms, total: 650 ms
Wall time: 655 ms
```

1. Объект многомерных массивов `ndarray`

Одной из ключевых особенностей NumPy является объект N-мерных массивов, или `ndarray`, который является быстрым, гибким контейнером для больших наборов данных в Python. Массивы позволяют выполнять математические операции над целыми блоками данных, используя схожий синтаксис для эквивалентных операций со скалярными элементами.

Рассмотрим пример: создадим небольшой массив случайных данных:

```
In [6]: import numpy as np

In [7]: data = np.random.randn(2, 3)

In [8]: data
Out[8]:
array([[-0.21595829, -0.8706619 ,  0.5635687 ],
       [-0.52986695,  1.04566656,  0.57054307]])

In [9]: data * 10
Out[9]:
array([[-2.15958287, -8.70661895,  5.635687  ],
       [-5.29866954, 10.45666559,  5.70543068]])

In [10]: data + data
Out[10]:
array([[-0.43191657, -1.74132379,  1.1271374 ],
       [-1.05973391,  2.09133312,  1.14108614]])
```

`ndarray` является общим контейнером для однородных многомерных данных, то есть все элементы должны быть одного типа. Каждый массив имеет атрибут `shape` — кортеж, указывающий размер каждого измерения, и атрибут `dtype`, объект, описывающий тип данных массива:

```
In [11]: data.shape
Out[11]: (2, 3)

In [12]: data.dtype
Out[12]: dtype('float64')
```

1.1. Создание массива

Простейший способ создания массива — использование функции `array`. Она принимает некоторый объект типа последовательностей (включая другие массивы) и создает новый массив NumPy, содержащий переданные данные. Например:

```
In [13]: data1 = [6, 7.5, 8, 0, 1]

In [14]: arr1 = np.array(data1)

In [15]: arr1
Out[15]: array([6. , 7.5, 8. , 0. , 1. ])
```

Вложенные последовательности, как список списков одинаковой длины, будут преобразованы в многомерный массив:

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [17]: arr2 = np.array(data2)

In [18]: arr2
Out[18]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Если явно не указано, `np.array` пытается вывести подходящий тип данных для массива, который он создает. Тип данных хранится в специальном объекте `dtype` метаданных; например, в двух предыдущих примерах мы имеем:

```
In [19]: arr1.dtype
Out[19]: dtype('float64')

In [20]: arr2.dtype
Out[20]: dtype('int64')
```

Кроме `np.array` есть несколько других функций для создания новых массивов:

```
In [22]: np.zeros(10)
Out[22]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [23]: np.zeros((3, 6))
Out[23]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])

In [24]: np.empty((2, 3, 2))
Out[24]:
array([[[4.63950122e-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, 0.00000000e+000],
        [0.00000000e+000, 0.00000000e+000]]])

In [25]: np.arange(15)
Out[25]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Таблица 1: Функции создания массивов

Функция	Описание
`array`	Преобразует входные данные (список, кортеж, массив или другая последовательность) в `ndarray`, либо прогнозируя `dtype`, либо используя заданный `dtype`; копирует данные по-умолчанию
`asarray`	Преобразует входные данные в `ndarray`, но не копирует их, если аргумент уже типа `ndarray`
`arange`	Подобна встроенной функции `range`, но возвращает `ndarray` вместо списка
`ones`	Создает массив из единиц заданной формы и `dtype`
`ones_like`	Получает на вход массив и создает массив из единиц с такими же формой и `dtype`
`zeros` и `zeros_like`	Подобны `ones` и `ones_like`, но создают массивы из нулей
`empty` и `empty_like`	Создают новые массивы, выделяя новую память, но не инициализируют их какими-либо значениями, как `ones` и `zeros`
`full`	Создает массив заданных формы и `dtype`, при этом все элементы инициализируются заданным значением `fill_value`
`full_like`	Получает на вход массив и создает массив с такими же формой и `dtype` и значениями `fill_value`
`eye` и `identity`	Создает квадратную единичную матрицу (с единицами на диагонали и нулями вне нее) размера $N \times N$

1.2. Арифметические операции с массивами NumPy

Массивы NumPy, как упоминалось выше, позволяют выполнять операции без использования циклов. Любые арифметические операции между массивами одинакового размера выполняются поэлементно:

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

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

In [3]: arr
Out[3]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [4]: arr * arr
Out[4]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

In [5]: arr - arr
Out[5]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Арифметические операции со скалярами распространяют скалярный аргумент к каждому элементу массива:

```
In [6]: 1 / arr
Out[6]:
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])

In [7]: arr ** 0.5
Out[7]:
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])

In [8]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [9]: arr2
Out[9]:
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])

In [10]: arr2 > arr
Out[10]:
array([[False,  True, False],
       [ True, False,  True]])
```

1.3. Основы индексирования и срезы

Существует много способов выбора подмножества данных или элементов массива. Одномерные массивы — это просто, на первый взгляд они аналогичны спискам Python:

```
In [1]: arr = np.arange(10)

In [2]: arr
Out[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [3]: arr[5]
Out[3]: 5

In [4]: arr[5:8]
Out[4]: array([5, 6, 7])

In [5]: arr[5:8] = 12

In [6]: arr
Out[6]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Как видно, если присвоить скалярное значение срезу, как например, `arr[5:8] = 12`, значение присваивается всем элементам среза. Первым важным отличием от списков Python заключается в том, что срезы массива являются *представлениями* исходного массива. Это означает, что данные не копируются и любые изменения в представлении будут отражены в исходном массиве.

Рассмотрим пример. Сначала создадим срез массива `arr`:

```
In [7]: arr_slice = arr[5:8]

In [8]: arr_slice
Out[8]: array([12, 12, 12])
```

Теперь, если мы изменим значения в массиве `arr_slice`, то они отразятся в исходном массиве `arr`:

```
In [9]: arr_slice[1] = 12345

In [10]: arr
Out[10]:
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

«Голый» срез `[:]` присвоит все значения в массиве:

```
In [11]: arr_slice[:] = 64

In [12]: arr
Out[12]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Поскольку NumPy был разработан для работы с очень большими массивами, вы можете представить себе проблемы с производительностью и памятью, если NumPy будет настаивать на постоянном копировании данных.

Замечание
Если вы захотите скопировать срез в массив вместо отображения, нужно явно скопировать массив, например, `arr[5:8].copy()`.

С массивами более высокой размерности существует больше вариантов. В двумерных массивах каждый элемент это уже не скаляр, а одномерный массив.

```
In [13]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [14]: arr2d[2]
Out[14]: array([7, 8, 9])
```

Таким образом, к отдельному элементу можно получить доступ рекурсивно, либо передать разделенный запятыми список индексов. Например, следующие два примера эквивалентны:

```
In [15]: arr2d[2]
Out[15]: array([7, 8, 9])

In [16]: arr2d[0][2]
Out[16]: 3
```

Если в многомерном массиве опустить последние индексы, то возвращаемый объект будет массивом меньшей размерности. Например, создадим массив размерности $2 \times 2 \times 3$:

```
In [17]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [18]: arr3d
Out[18]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

При этом `arr3d[0]` — массив размерности 2×3:

```
In [19]: arr3d[0]
Out[19]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Можно присваивать `arr3d[0]` как скаляр, так и массивы:

```
In [20]: old_values = arr3d[0].copy()

In [21]: arr3d[0] = 42

In [22]: arr3d
Out[22]:
array([[[42, 42, 42],
        [42, 42, 42]],

       [[ 7,  8,  9],
        [10, 11, 12]]])

In [23]: arr3d[0] = old_values

In [24]: arr3d
Out[24]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

Аналогично, `arr3d[1, 0]` возвращает все значения, чьи индексы начинаются с `(1, 0)`, формируя одномерный массив:

```
In [25]: arr3d[1, 0]
Out[25]: array([7, 8, 9])
```

Это выражение такое же, как если бы мы проиндексировали в два этапа:

```
In [26]: x = arr3d[1]

In [27]: x
Out[27]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [28]: x[0]
Out[28]: array([7, 8, 9])
```

Индексирование с помощью срезов. Как одномерные объекты, такие как списки, можно получать срезы массивов посредством знакомого синтаксиса:

```
In [28]: arr
Out[28]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [29]: arr[1:6]
Out[29]: array([ 1,  2,  3,  4, 64])
```

Рассмотрим введенный выше двумерный массив `arr2d`. Получение срезов этого массива немного отличается от одномерного:

```
In [30]: arr2d
Out[30]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [31]: arr2d[:2]
Out[31]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Как видно, мы получили срез вдоль оси 0, первой оси. Срез, таким образом, выбирает диапазон элементов вдоль оси. Выражение `arr2d[:2]` можно прочитать как «выбираем первые две строки массива ‘arr2d‘».

Можно передавать несколько срезов:

```
In [32]: arr2d[:2, 1:]
Out[32]:
array([[2, 3],
       [5, 6]])
```

При получении срезов мы получаем только отображения массивов того же числа размерностей. Используя целые индексы и срезы, можно получить срезы меньшей размерности:

```
In [33]: arr2d[1, :2]
Out[33]: array([4, 5])

In [34]: arr2d[:2, 2]
Out[34]: array([4, 5])
```

Смотрите рис. 1.

1.4. Логическое (Boolean) индексирование

Рассмотрим следующий пример: пусть есть массив с данными (случайными) и массив, содержащий имена с повторениями:

```
In [35]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [36]: data = np.random.randn(7, 4)

In [37]: names
Out[37]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

In [38]: data
Out[38]:
```

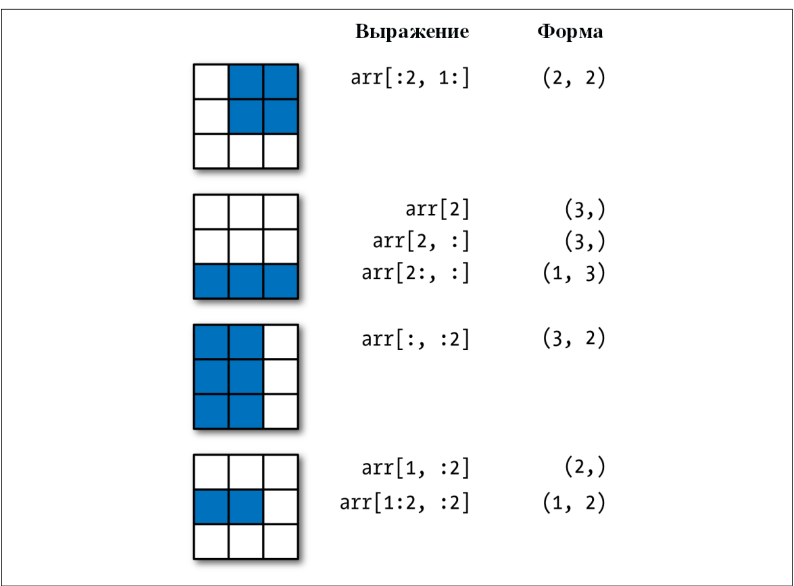

Рис. 1: Срезы двумерного массива

```
array([[-0.30602191, -0.30319088,  0.33639925,  0.67844077],
       [-0.58702763, -0.29292886,  0.0071339 , -0.72423144],
       [ 0.97107817, -0.29963124,  0.25907764,  0.47690155],
       [ 0.61053052, -0.62803392, -0.08214009,  0.05142378],
       [-0.71103802,  0.05003893,  0.41204829, -0.26279151],
       [-0.03198355,  0.82015773, -0.86561954, -0.15198867],
       [-0.88311743, -0.81266173, -0.10336611, -0.80341886]])
```

Предположим, что каждое имя соответствует строке в массиве `data`, и мы хотим выбрать все строки с соответствующим именем `'Bob'`. Как и арифметические операции, операции сравнения (такие как `==`) с массивами также векторизованы. Таким образом, сравнение массива `names` со строкой `'Bob'` возвращает булев массив:

```
In [39]: names == 'Bob'
Out[39]: array([ True, False, False,  True, False, False, False])
```

Этот булев массив может использоваться при индексировании массива:

```
In [40]: data[names == 'Bob']
Out[40]:
array([[-0.30602191, -0.30319088,  0.33639925,  0.67844077],
       [ 0.61053052, -0.62803392, -0.08214009,  0.05142378]])
```

Булев массив должен быть той же длины, что и ось массива, по которой осуществляется индексация. Вы даже можете смешивать и сопоставлять логические массивы со срезами или целыми числами (или последовательностями целых чисел).

```
In [41]: data[names == 'Bob', 2:]
Out[41]:
array([[ 0.33639925,  0.67844077],
       [-0.08214009,  0.05142378]])

In [42]: data[names == 'Bob', 3]
Out[42]: array([0.67844077, 0.05142378])
```

Чтобы выбрать все, кроме 'Bob', можно использовать != или обращение условия :

```
In [43]: names != 'Bob'
Out[43]: array([False,  True,  True, False,  True,  True,  True])

In [44]: data[~(names == 'Bob')]
Out[44]:
array([[-0.58702763, -0.29292886,  0.0071339 , -0.72423144],
       [ 0.97107817, -0.29963124,  0.25907764,  0.47690155],
       [-0.71103802,  0.05003893,  0.41204829, -0.26279151],
       [-0.03198355,  0.82015773, -0.86561954, -0.15198867],
       [-0.88311743, -0.81266173, -0.10336611, -0.80341886]])
```

Оператор может быть полезен при инвертировании общего условия:

```
In [45]: cond = names == 'Bob'

In [46]: data[~cond]
Out[46]:
array([[-0.58702763, -0.29292886,  0.0071339 , -0.72423144],
       [ 0.97107817, -0.29963124,  0.25907764,  0.47690155],
       [-0.71103802,  0.05003893,  0.41204829, -0.26279151],
       [-0.03198355,  0.82015773, -0.86561954, -0.15198867],
       [-0.88311743, -0.81266173, -0.10336611, -0.80341886]])
```

Выбрав два из трех имен для объединения нескольких логических условий, можно использовать логические арифметические операторы, такие как & (и) и | (или):

```
In [47]: mask = (names == 'Bob') | (names == 'Will')

In [48]: mask
Out[48]: array([ True, False,  True,  True,  True, False, False])

In [49]: data[mask]
Out[49]:
array([[-0.30602191, -0.30319088,  0.33639925,  0.67844077],
       [ 0.97107817, -0.29963124,  0.25907764,  0.47690155],
       [ 0.61053052, -0.62803392, -0.08214009,  0.05142378],
       [-0.71103802,  0.05003893,  0.41204829, -0.26279151]])
```

Выбор данных из массива с помощью логического индексирования *всегда* создает копию данных, даже если возвращаемый массив не изменяется.

Предупреждение
Ключевые слова Python `and` и `or` не работают с булевыми массивами. Используйте `&` (и) и `|` (или).

Присвоение значений массивам работает обычным образом. Замену всех отрицательных значений на `0` можно сделать следующим образом:

```
In [50]: data[data < 0] = 0

In [51]: data
Out[51]:
array([[0.        , 0.        , 0.33639925, 0.67844077],
       [0.        , 0.        , 0.0071339 , 0.        ],
       [0.97107817, 0.        , 0.25907764, 0.47690155],
       [0.61053052, 0.        , 0.        , 0.05142378],
       [0.        , 0.05003893, 0.41204829, 0.        ],
       [0.        , 0.82015773, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        ]])
```

Можно также легко присваивать значения целым строкам или столбцам:

```
In [52]: data[names != 'Joe'] = 7

In [53]: data
Out[53]:
array([[7.        , 7.        , 7.        , 7.        ],
       [0.        , 0.        , 0.0071339 , 0.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [0.        , 0.82015773, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        ]])
```

1.5. Необычное индексирование

Необычное индексирование (*fancy indexing*) — это термин, принятый в NumPy для описания индексации с использованием целочисленных массивов.

Предположим, у нас есть массив размера 8×4

```
In [54]: arr = np.empty((8, 4))

In [55]: for i in range(8):
    ...:     arr[i] = i

In [56]: arr
Out[56]:
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
```

```
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

Чтобы выбрать подмножество строк в определенном порядке, можно просто передать список или массив целых чисел, указывающих желаемый порядок:

```
In [57]: arr[[4, 3, 0, 6]]
Out[57]:
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

Использование отрицательных индексов выделяет строки с конца:

```
In [58]: arr[[-3, -5, -7]]
Out[58]:
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

Передача нескольких индексных массивов делает кое-что другое: выбирается одномерный массив элементов, соответствующий каждому кортежу индексов:

```
In [59]: arr = np.arange(32).reshape((8, 4))

In [60]: arr
Out[61]:
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]])

In [61]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[61]: array([ 4, 23, 29, 10])
```

Здесь выбраны элементы с индексами `(1, 0)`, `(5, 3)`, `(7, 1)` и `(2, 2)`. Независимо от того какая размерность у массива (в нашем случае двумерный массив), результат такого индексирования — всегда одномерный массив.

Поведение индексирования в этом случае немного отличается от того, что могли ожидать некоторые пользователи, а именно: пользователь мог ожидать прямоугольную область, сформированную путем выбора поднабора строк и столбцов матрицы. Ниже представлен один из способов получения таких массивов с помощью необычного индексирования:

```
In [62]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[62]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Имейте в виду, что необычное индексирование, в отличие от среза, всегда копирует данные в новый массив.

1.6. Транспонирование массивов и замена осей

Транспонирование — это особый способ изменения формы массива, который возвращает представление исходных данных без их копирования. Массивы имеют метод `transpose`, а также специальный атрибут `T`:

```
In [63]: arr = np.arange(15).reshape((3, 5))

In [64]: arr
Out[64]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [65]: arr.T
Out[65]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

При выполнении матричных вычислений эта процедура может выполняться очень часто, например, при вычислении произведения матриц с помощью функции `np.dot`:

```
In [66]: arr = np.random.randn(6, 3)

In [67]: arr
Out[67]:
array([[-0.31858673, -0.74194068, -0.5057573 ],
       [ 0.83588823, -0.53996512, -0.97953623],
       [ 0.58273205,  0.67279648, -1.10365259],
       [ 0.88643344,  0.01374888,  3.00932538],
       [ 1.01328971,  1.62965388, -1.12032883],
       [-1.23646751, -0.56660122, -1.24328081]])

In [68]: np.dot(arr.T, arr)
Out[68]:
array([[ 4.48115546,  2.54116501,  1.76883634],
       [ 2.54116501,  4.27169117, -0.91830522],
       [ 1.76883634, -0.91830522, 14.29025379]])
```

Для массивов большей размерности метод `transpose` принимает кортеж с номерами осей, задающий перестановку осей:

```
In [69]: arr = np.arange(16).reshape((2, 2, 4))

In [70]: arr
Out[70]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [71]: arr.transpose((1, 0, 2))
Out[71]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Здесь оси были переупорядочены следующим образом: вторая ось стала первой, первая ось — второй, а последняя осталась без изменений.

Простое транспонирование с помощью `.T` является частным случаем замены осей. Массивы имеют метод `swapaxes`, который получает пару номеров осей и переставляет указанные оси.

```
In [72]: arr.swapaxes(1, 2)
Out[73]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

Метод `swapaxes` возвращает представление данных без копирования.

2. Универсальные функции: быстрые поэлементные функции от массивов

Универсальные функции (или *ufunc*) — это функции, которые выполняют поэлементные операции над данными массива. Можно рассматривать их как быстрые векторизованные обертки для простых функций, которые принимают одно или несколько скалярных значений и дают один или несколько скалярных результатов.

Многие универсальные функции — это простые поэлементные преобразования, такие как `sqrt` и `exp`:

```
In [74]: arr = np.arange(10)

In [75]: arr
Out[75]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [76]: np.sqrt(arr)
Out[76]:
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])

In [77]: np.exp(arr)
Out[77]:
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

Они являются *унарными* универсальными функциями. Такие функции как `add` или `maximum` принимают два массива (таким образом, они *бинарные*) и возвращают один массив:

```
In [78]: x = np.random.randn(8)

In [79]: y = np.random.randn(8)

In [80]: x
Out[80]:
array([-0.6170843 ,  0.60137616, -1.07327493,  0.84037086, -0.34779787,
       -1.44563308, -0.08710135,  0.99717   ])

In [81]: y
Out[81]:
array([ 0.37385011, -2.77545805, -0.25609749,  1.40563358,  0.32337793,
        0.85690424,  0.26611575,  0.18535124])

In [82]: np.maximum(x, y)
Out[82]:
array([ 0.37385011,  0.60137616, -0.25609749,  1.40563358,  0.32337793,
        0.85690424,  0.26611575,  0.99717   ])
```

Функция `np.maximum` вычисляет максимумы элементов массивов `x` и `y`.

Универсальная функция может возвращать несколько массивов (хотя и не часто). Например, функция `np.modf` (векторизованная версия встроенной функции `divmod`) возвращает целую и дробную части массива чисел с плавающей запятой:

```
In [83]: arr = np.random.randn(7) * 5

In [84]: arr
Out[84]:
array([ 8.93980752, -4.39765192, -1.63135584, -4.02920646,  6.02708204,
        0.43352538,  1.00886326])

In [85]: frac_part, int_part = np.modf(arr)
```

```
In [86]: frac_part
Out[86]:
array([ 0.93980752, -0.39765192, -0.63135584, -0.02920646,  0.02708204,
        0.43352538,  0.00886326])

In [87]: int_part
Out[87]: array([ 8., -4., -1., -4.,  6.,  0.,  1.])
```

Универсальные функции принимают опциональный аргумент `out`, который позволяет выполнять операции прямо в заданном массиве.

```
In [88]: arr
Out[88]:
array([ 8.93980752, -4.39765192, -1.63135584, -4.02920646,  6.02708204,
        0.43352538,  1.00886326])

In [89]: np.sqrt(arr)
<ipython-input-68-b58949107b3d>:1: RuntimeПредупреждение: invalid value encountered in sqrt
  np.sqrt(arr)
Out[89]:
array([2.98995109,        nan,        nan,        nan, 2.45501162,
       0.65842645, 1.00442185])

In [90]: np.sqrt(arr, arr)
<ipython-input-69-164954cb2c14>:1: RuntimeПредупреждение: invalid value encountered in sqrt
  np.sqrt(arr, arr)
Out[90]:
array([2.98995109,        nan,        nan,        nan, 2.45501162,
       0.65842645, 1.00442185])

In [91]: arr
Out[91]:
array([2.98995109,        nan,        nan,        nan, 2.45501162,
       0.65842645, 1.00442185])
```

В таблицах 2 и 3 представлены доступные универсальные функции.

Таблица 2: Унарные универсальные функции

Функция	Описание
`abs`, `fabs`	Вычисляет абсолютные значения каждого элементов массива
`sqrt`	Вычисляет квадратный корень из каждого элемента массива (эквивалентно `arr ** 0.5`)
`square`	Вычисляет квадрат каждого элемента массива (эквивалентно `arr ** 2`)
`exp`	Вычисляет экспоненту (e^x) от каждого элемента массива
`log`, `log10`, `log2`, `log1p`	Вычисляет натуральный, десятичный логарифмы, логарифм по основанию 2 и $\log(1 + x)$, соответственно
`sign`	Вычисляет знак каждого элемента: 1 (положительный элемент), 0 (ноль), -1 (отрицательный элемент)

<table>
<tr><td>ceil</td><td>Вычисляет наименьшее целое число большее либо равное каждого элемента массива</td></tr>
<tr><td>floor</td><td>Вычисляет наибольшее целое число меньшее либо равное каждого элемента массива</td></tr>
<tr><td>rint</td><td>Округляет элементы к ближайшим целым сохраняя dtype</td></tr>
<tr><td>modf</td><td>Возвращает дробные и целые части каждого элемента массива</td></tr>
<tr><td>isnan</td><td>Возвращает булев массив, указывающий является каждый элемент входного массива NaN (Not A Number)</td></tr>
<tr><td>isfinite, isinf</td><td>Возвращает булев массив, указывающий является каждый элемент конечным (не inf и не NaN) или бесконечным, соотвественно</td></tr>
<tr><td>cos, cosh, sin, sinh, tan, tanh</td><td>Обычные и тригонометрические функции</td></tr>
<tr><td>arccos, arccosh, arcsin, arcsinh, arctan, arctanh</td><td>Обратные тригонометрические функции</td></tr>
<tr><td>logical_not</td><td>Вычисляет истиное значение not x для каждого элемента (эквивалентно arr)</td></tr>
</table>

Таблица 3: Бинарные универсальные функции

<table>
<tr><th>Функция</th><th>Описание</th></tr>
<tr><td>add</td><td>Складывает соответствующие элементы массивов</td></tr>
<tr><td>subtract</td><td>Вычитает соответствующие элементы второго массива из элементов первого</td></tr>
<tr><td>multiply</td><td>Перемножает элементы массивов</td></tr>
<tr><td>divide, floor_devide</td><td>Деление или деление с отбрасыванием остатка</td></tr>
<tr><td>power</td><td>Возведение элементов первого массива в степени указанные во втором массиве</td></tr>
<tr><td>maximum, fmax</td><td>Поэлементный максимум, fmax игнорирует NaN</td></tr>
<tr><td>minimum, fmin</td><td>Поэлементный минимум, fmin игнорирует NaN</td></tr>
<tr><td>mod</td><td>Поэлементный модуль (остаток от деления)</td></tr>
<tr><td>copysign</td><td>Копирует знаки элементов второго массива в элементы первого массива</td></tr>
<tr><td>greater, greater_equal, less, less_equal, equal, not_equal</td><td>Поэлементное сравнение (эквивалентны операторам >, >=, <, <= '==', !=)</td></tr>
<tr><td>logical_and, logical_or, logical_xor</td><td>Вычисляет поэлементное значение истинности логической операций (эквивалентны операторам &, ' | ', ^)</td></tr>
</table>

3. Программирование с использованием массивов

Использование массивов NumPy позволяет выражать многие виды задач обработки данных в виде кратких выражений с массивами, которые в противном случае потребовали использования циклов. Такая практика замены явных циклов на выражения с массивами обычно называется *векторизацией*. Вообще говоря, векторизованные операции с массивами часто на один-два (или более) порядка быстрее, чем их эквиваленты Python, что оказывает большое влияние на любые виды вычислений.

В качестве простого примера, предположим, что мы хотим вычислить функцию `sqrt(x^2 + y^2)` по всей регулярной сетке значений. Функция `np.meshgrid` получает два одномерных массива и возвращает две двумерные матрицы соответствующие всем парам `(x, y)` в двух массивах:

```
In [92]: points = np.arange(-5, 5, 0.01)

In [93]: xs, ys = np.meshgrid(points, points)

In [94]: xs
Out[94]:
array([[-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       ...,
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99]])

In [95]: ys
Out[95]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Теперь для вычисления значений на всей сетке нужно написать то же выражение, которое было бы написано для двух координат:

```
In [96]: z = np.sqrt(xs ** 2 + ys ** 2)

In [97]: z
Out[97]:
array([[7.07106781, 7.06400028, 7.05693985, ..., 7.04988652, 7.05693985,
        7.06400028],
       [7.06400028, 7.05692568, 7.04985815, ..., 7.04279774, 7.04985815,
        7.05692568],
       [7.05693985, 7.04985815, 7.04278354, ..., 7.03571603, 7.04278354,
        7.04985815],
       ...,
       [7.04988652, 7.04279774, 7.03571603, ..., 7.0286414 , 7.03571603,
```

```
        7.04279774],
       [7.05693985, 7.04985815, 7.04278354, ..., 7.03571603, 7.04278354,
        7.04985815],
       [7.06400028, 7.05692568, 7.04985815, ..., 7.04279774, 7.04985815,
        7.05692568]])
```

Теперь воспользуемся библиотекой `matplotlib` (ее мы рассмотрим позже) для визуализации двумерного массива:

```
In [98]: import matplotlib.pyplot as plt

In [99]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[99]: <matplotlib.colorbar.Colorbar at 0x7f99116dfeb0>

In [100]: plt.title('Визуализация функции $\sqrt{x^2 + y^2}$ на сетке')
Out[100]: Text(0.5, 1.0, 'Визуализация функции $\\sqrt{x^2 + y^2}$ на сетке')
```

Результат представлен на рисунке 2. Здесь использовалась функция `imshow` библиотеки `matplotlib` для создания изображения по двумерному массиву значений сеточной функции.

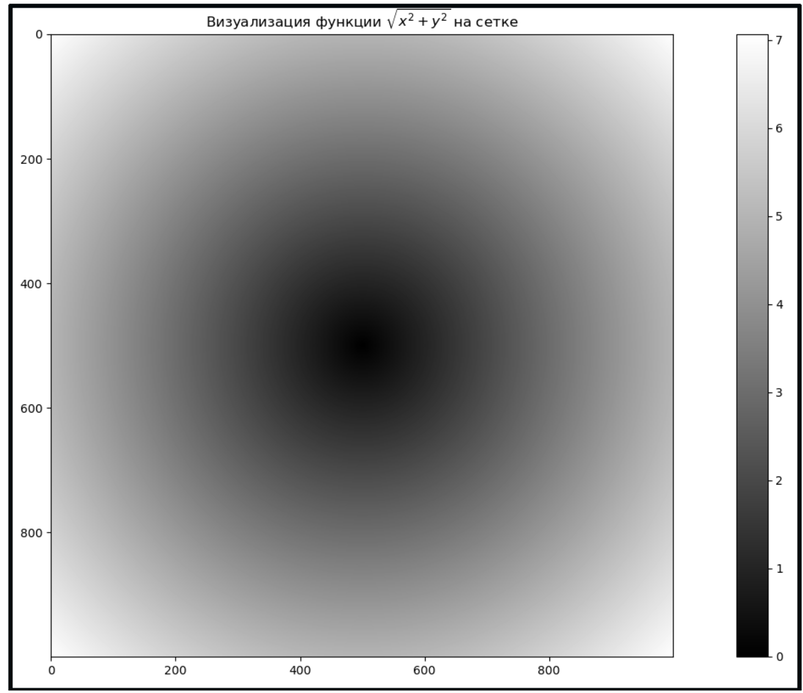

Рис. 2: Визуализация функции, вычисленной на сетке

3.1. Выражение условной логики в операциях с массивами

Функция `np.where` — векторизованная версия тернарного выражения `x if condition else y`. Предположим, у нас есть булев массив и два массива значений:

```
In [101]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [102]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [103]: cond = np.array([True, False, True, True, False])
```

Предположим, мы хотим выбрать из массива `xarr` значения в том случае, когда значение элемента массива `cond` равно `True`, иначе выбираем значение из массива `yarr`. С использованием списка это может выглядеть следующим образом:

```
In [104]: result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]

In [105]: result
Out[105]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

Такой подход имеет несколько проблем. Во-первых, это не будет быстро работать для очень больших массивов (потому что вся работа будет выполняться интерпретируемым Python-кодом). Во-вторых, это не будет работать с многомерными массивами. С помощью `np.where` все это можно записать коротко:

```
In [106]: result
Out[106]: [1.1, 2.2, 1.3, 1.4, 2.5]

In [107]: result = np.where(cond, xarr, yarr)

In [108]: result
Out[108]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Второй и третий аргументы функции `np.where` не обязательно должны быть массивами, они могут быть скалярами. Типичное использование функции `np.where` в анализе данных — это создание нового массива значений на основе другого массива. Предположим есть матрица случайно сгенерированных значений и нужно заменить все положительные значения на число 2, а отрицательные значения — на -2. Это легко сделать с помощью функции `np.where`:

```
In [109]: arr = np.random.randn(4, 4)

In [110]: arr
Out[110]:
array([[-1.1144006 , -1.26323994,  0.4345178 ,  1.32382344],
       [ 1.15196751,  0.42816243,  1.08377185, -1.28017115],
       [-0.50388302, -0.94943293, -0.49274078, -0.42359743],
       [-0.70021236, -0.5096619 ,  0.6006224 , -0.99140402]])
```

```
In [111]: arr > 0
Out[111]:
array([[False, False,  True,  True],
       [ True,  True,  True, False],
       [False, False, False, False],
       [False, False,  True, False]])

In [112]: np.where(arr > 0, 2, -2)
Out[112]:
array([[-2, -2,  2,  2],
       [ 2,  2,  2, -2],
       [-2, -2, -2, -2],
       [-2, -2,  2, -2]])
```

Можно объединять скаляры и массивы при использовании `np.where`. Например, заменим все положительные элементы массива на 2:

```
In [113]: np.where(arr > 0, 2, arr)
Out[113]:
array([[-1.1144006 , -1.26323994,  2.        ,  2.        ],
       [ 2.        ,  2.        ,  2.        , -1.28017115],
       [-0.50388302, -0.94943293, -0.49274078, -0.42359743],
       [-0.70021236, -0.5096619 ,  2.        , -0.99140402]])
```

3.2. Математические и статистические методы

Некоторые математические функции, которые вычисляют статистику по данным всего массива или по данным по какой-либо оси, доступны как методы класса. Вы можете использовать *агрегаты* (часто называемые *редукциями*), такие как `sum`, `mean` и `std` (стандартное отклонение), либо вызывая метод экземпляра массива, либо используя функцию NumPy верхнего уровня.

Ниже сгенерированы случайные нормально распределенные данные и вычислены некоторые статистические свойства:

```
In [113]: arr = np.random.randn(5, 4)

In [114]: arr
Out[114]:
array([[-1.43597861,  0.01372626,  0.01284379, -0.92143533],
       [-0.9509138 ,  3.25326976,  0.05585994,  1.03115936],
       [ 0.13655357,  1.14912738, -0.38493582, -1.5076265 ],
       [ 0.35841008, -0.5325945 ,  0.85600472,  0.31257975],
       [-0.26906625, -0.31187753,  0.24331333,  0.47455591]])

In [115]: arr.mean()
Out[115]: 0.07914877668565846

In [116]: np.mean(arr)
Out[116]: 0.07914877668565846

In [117]: arr.sum()
Out[117]: 1.5829755337131692
```

Функции типа mean и sum принимают опциональный аргумент axis, указывающий по какой оси вычислять статистику. В результате получается массив на одну размерность меньше.

```
In [118]: arr.mean(axis=1)
Out[119]: array([-0.58271097,  0.84734381, -0.15172034,  0.24860001,  0.03423137])

In [120]: arr.sum(axis=0)
Out[120]: array([-2.160995   ,  3.57165138,  0.78308596, -0.6107668 ])
```

Здесь arr.mean(axis=1) означает «вычислить средние значения по столбцам», а arr.sum(axis=0) означает «вычислить сумму по строкам».

Другие методы, такие как cumsum и cumprod, не агрегируют, а создают массив промежуточных результатов:

```
In [121]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [122]: arr.cumsum()
Out[122]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

В многомерных массивах В многомерных массивах функции накопления, такие как cumsum, возвращают массив того же размера, но с частичными агрегатами, вычисленными вдоль указанной оси в соответствии с каждым срезом меньшего размера:

```
In [123]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [124]: arr
Out[124]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [125]: arr.cumsum(axis=0)
Out[125]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [126]: arr.cumprod(axis=1)
Out[126]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

В таблице 4 представлен полный список таких функций.

Таблица 4: Основные статистические методы для массивов

Функция	Описание
sum	Сумма всех элементов массива или вдоль оси. Массив нулевой длины имеет сумму, равную 0
mean	Арифметическое среднее. Массив нулевой длины имеет среднее значение NaN
std, var	Стандартное отклонение и дисперсия, соответственно, с возможностью задания степени свободы (по умолчанию знаменатель равен n)
min, max	Минимум и максимум
argmin, argmax	Индексы минимального и максимального элементов, соответственно
cumsum	Накопленная сумма элементов, начиная с 0
cumprod	Накопленное произведение элементов, начиная с 1

3.3. Методы для булевых массивов

В рассмотренных выше методах булевы значения приводятся к 1 (`True`) и 0 (`False`). Таким образом, `sum` часто используется как средство подсчета значений `True` в логическом массиве:

```
In [127]: arr = np.random.randn(100)

In [128]: (arr > 0).sum()
Out[128]: 49
```

Есть два дополнительных метода: `any` и `all`, которые очень полезны при работе с булевыми массивами. Метод `any` проверяет, есть ли хотя бы одно значение в массиве равное `True`, а `all` проверяет, все ли значения в массиве равны `True`:

```
In [129]: arr = np.array([False, False, True, False])

In [130]: arr.any()
Out[130]: True

In [131]: arr.all()
Out[131]: False
```

Эти методы также работают с небулевыми массивами. В этом случае ненулевые элементы оцениваются как `True`.

3.4. Сортировка

Как и встроенный тип `list` массивы NumPy могут быть отсортированы с помощью метода `sort`:

```
In [132]: arr = np.random.randn(6)

In [133]: arr
Out[133]:
array([-0.12728925,  0.24554644, -1.15625417, -1.4625911 , -0.78147401,
        1.58324829])

In [134]: arr.sort()

In [134]: arr
Out[134]:
array([-1.4625911 , -1.15625417, -0.78147401, -0.12728925,  0.24554644,
        1.58324829])
```

Можно отсортировать каждый одномерный массив многомерного вдоль оси, которая задается как аргумент метода `sort`:

```
In [135]: arr = np.random.randn(5, 3)

In [136]: arr
Out[136]:
array([[-0.17959553, -0.93747164,  0.38332596],
       [-0.1051853 ,  0.90182293,  1.30222401],
       [ 0.29822932, -0.976582  , -0.01074546],
       [-0.7052856 , -0.19126606,  0.38607724],
       [ 0.996878  ,  0.94214515, -1.53962274]])

In [137]: arr.sort(1)

In [138]: arr
Out[138]:
array([[-0.93747164, -0.17959553,  0.38332596],
       [-0.1051853 ,  0.90182293,  1.30222401],
       [-0.976582  , -0.01074546,  0.29822932],
       [-0.7052856 , -0.19126606,  0.38607724],
       [-1.53962274,  0.94214515,  0.996878  ]])
```

Метод верхнего уровня `np.sort` возвращает отсортированный массив вместо изменения исходного массива. Быстрый и простой способ вычислить квантили массива — это отсортировать его и выбрать значение в определенном ранге:

```
In [139]: large_arr = np.random.randn(1000)

In [140]: large_arr.sort()

In [141]: large_arr[int(0.05 * len(large_arr))]  # 5% квантиль
Out[141]: -1.718770519734767
```

3.5. Основные операции над множествами для массивов

В NumPy имеются некоторые основные операции над множествами для одномерных массивов. Обычно используется функция `np.unique`, которая возвращает отсортированные уникальные значения в массиве:

```
In [142]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [143]: np.unique(names)
Out[143]: array(['Bob', 'Joe', 'Will'], dtype='<U4')

In [144]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [145]: np.unique(ints)
Out[145]: array([1, 2, 3, 4])
```

Сравните `np.unique` с альтернативой на чистом Python:

```
In [146]: sorted(set(names))
Out[146]: ['Bob', 'Joe', 'Will']
```

Другая функция, `np.in1d`, проверяет нахождение значений из одного массива в другом, возвращая логический массив:

```
In [147]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [148]: np.in1d(values, [2, 3, 6])
Out[148]: array([ True, False, False,  True,  True, False,  True])
```

Таблица 5: Операции над множествами из массивов

Метод	**Описание**
`unique(x)`	Возвращает отсортированные единственные элементы из `x`
`intersect1d(x, y)`	Возвращает отсортированные общие элементы массивов `x` и `y`
`union1d(x, y)`	Возвращает отсортированное объединение элементов массивов `x` и `y`
`in1d(x, y)`	Возвращает булев массив, указывающий содержится ли каждый элемент массива `x` в `y`
`setdiff1d(x, y)`	Разность множеств: элементы массива `x`, которых нет в `y`
`setxor1d(x, y)`	Симметричная разность: элементы, которые есть либо в `x`, либо в `y`, но не в обоих массивах

4. Чтение и запись в/из файлы массивов

NumPy позволяет сохранять на диск и загружать с диска в текстовые или двоичные файлы. Здесь рассмотрим только встроенные двоичные файлы NumPy, так как большинство предпочтут использовать pandas или другие инструменты для загрузки текста или таблиц.

Функции np.save и np.load — это две рабочие лошадки для эффективного сохранения и загрузки данных массива на диск. Массивы сохраняются по умолчанию в несжатый двоичный формате с расширением файла .npy:

```
In [149]: arr = np.arange(10)

In [150]: np.save('some_array', arr)
```

В пути к файлу не указывается разширение, оно добавляется по умолчанию. Массив с диска потом можно загрузить с помощью функции np.load:

```
In [151]: np.load('some_array.npy')
Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Можно сохранить несколько массивов в один несжатый файл, используя функцию np.savez и передавая массивы в качестве аргументов:

```
In [152]: np.savez('array_archive.npz', a=arr, b=arr)
```

При загрузке файла .npz возвращается объект типа словаря, который содержит отдельные массивы:

```
In [153]: arch = np.load('array_archive.npz')

In [154]: arch['b']
Out[154]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Если данные хорошо сжимаются, можно использовать функцию np.savez_compressed:

```
In [155]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

5. Линейная алгебра

Матричные вычисления, такие как умножение матриц, построение разложений матриц, вычисление определителя и др. являются важной частью любой библиотеки программ для численных расчетов. В отличие от некоторых языков программирования, таких как MATLAB, в NumPy операция * — это поэлементное умножение матриц, а не стандартное умножение матриц из линейной алгебры. В связи с этим в NumPy для умножения матриц реализована функция dot как в виде метода объекта типа ndarray, так и в виде функции из пространства имен numpy:

```
In [156]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [157]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [158]: x
Out[158]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [159]: y
Out[159]:
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])

In [160]: x.dot(y)
Out[160]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

Выражение `x.dot(y)` эквивалентно `np.dot(x, y)`:

```
In [161]: np.dot(x, y)
Out[161]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

Произведение матрицы на согласованный одномерный массив дает одномерный массив:

```
In [162]: np.dot(x, np.ones(3))
Out[162]: array([ 6., 15.])
```

Операция @ также выполняет умножение матриц:

```
In [163]: x @ np.ones(3)
Out[163]: array([ 6., 15.])
```

Модуль `numpy.linalg` стандартный набор функций для разложения матриц, а также вычисления определителя и обратной матрицы. Они реализуются с помощью тех же стандартных библиотек линейной алгебры, которые используются в других языках (например, MATLAB и R), таких как BLAS, LAPACK, или, возможно (в зависимости от вашей сборки NumPy), проприетарной Intel MKL (Math Kernel Library):

```
In [164]: from numpy.linalg import inv, qr

In [165]: X = np.random.randn(5, 5)

In [166]: mat = X.T.dot(X)
```

```
In [167]: inv(mat)
Out[167]:
array([[  1.17126787,   0.63442101,  -0.43366495,   0.12451843,
         -0.43842659],
       [  0.63442101,  20.97155103,   4.85020898,   1.20337434,
        -25.85438743],
       [ -0.43366495,   4.85020898,   1.82425292,   0.15709542,
         -6.39429734],
       [  0.12451843,   1.20337434,   0.15709542,   0.33177322,
         -1.54477735],
       [ -0.43842659, -25.85438743,  -6.39429734,  -1.54477735,
         32.3946724 ]])

In [168]: mat.dot(inv(mat))
Out[168]:
array([[ 1.00000000e+00,  4.44089210e-16,  0.00000000e+00,
        -1.38777878e-17, -4.44089210e-16],
       [ 0.00000000e+00,  1.00000000e+00,  7.10542736e-15,
         0.00000000e+00, -2.84217094e-14],
       [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00,
         8.88178420e-16, -1.42108547e-14],
       [-4.44089210e-16, -1.42108547e-14, -3.55271368e-15,
         1.00000000e+00,  1.42108547e-14],
       [ 0.00000000e+00, -1.42108547e-14, -3.55271368e-15,
         8.88178420e-16,  1.00000000e+00]])

In [169]: q, r = qr(mat)

In [170]: r
Out[170]:
array([[-1.98177521, -1.01858515, -3.90022484, -1.74739689, -1.69358168],
       [ 0.        , -8.4068366 , -5.93064791, -5.55543324, -8.16388208],
       [ 0.        ,  0.        , -1.63499732, -2.80467309, -0.45129639],
       [ 0.        ,  0.        ,  0.        , -3.17546169, -0.14914608],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.02382757]])
```

В таблице представлены наиболее часто используемые функции линейной алгебры.

Таблица 6: Часто используемые функции модуля `numpy.linalg`

Функция	Описание
`diag`	Возвращает диагональные элементы квадратной матрицы в виде одномерного массива или преобразует одномерный массив в квадратную матрицу с нулями вне диагонали
`dot`	Умножение матриц
`trace`	След матрицы — сумма диагональных элементов
`det`	Определитель матрицы
`eig`	Вычисляет собственные значения и собственные векторы квадратной матрицы
`inv`	Вычисляет обратную матрицу
`pinv`	Вычисляет псевдообратную матрицу Мура—Пенроуза

`qr`	Вычисляет QR разложение матрицы
`svd`	Вычисляет сингулярное разложение матрицы (SVD)
`solve`	Решает линейную систему $Ax = b$, где A — квадратная матрица
`lstsq`	Находит решение линейной системы $Ax = b$ методом наименьших квадратов

6. Генерация псевдослучайных чисел

Модуль `numpy.random` дополняет встроенный в Python модуль `random` функциями для эффективной для эффективной генерации целых массивов выборок, подчиненых многим видам вероятностных распределений. Например, можно получить массив размера 4×4 выборок из нормального распределения, используя функцию `normal`:

```
In [171]: samples = np.random.normal(size=(4, 4))

In [172]: samples
Out[172]:
array([[-1.34227933, -0.19399381,  0.03387371, -0.08797433],
       [ 0.93296259,  0.25866046,  0.08337615,  0.85956704],
       [-0.52511101,  0.47723459,  1.68167773, -0.11159956],
       [-0.59314399,  0.90182334, -1.36131921,  0.14753701]])
```

Встроенный модуль `random` генерирует только одно число за раз. Как видно из приведенного ниже теста, `numpy.random` на порядок быстрее генерирует очень большие выборки:

```
In [173]: from random import normalvariate

In [174]: N = 1000000

In [175]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
715 ms ± 262 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [176]: %timeit np.random.normal(size=N)
38.7 ms ± 748 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

163/5000 Мы говорим, что это *псевдослучайные* числа, потому что они генерируются алгоритмом с детерминированным поведением на основе *начального* (seed) числа генератора случайных чисел. Можно изменить начальное число генератора случайных чисел, используя функцию `np.random.seed`:

```
In [177]: np.random.seed(1234)
```

Функция, генерирующие данные, из `numpy.random` используют глобальное начальное число. Чтобы избежать глобального состояния, вы можете использовать `np.random.RandomState` для создания генератора случайных чисел, изолированного от других:

```
In [178]: rng = np.random.RandomState(1234)

In [179]: rng.randn(10)
Out[179]:
array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.72058873,
        0.88716294,  0.85958841, -0.6365235 ,  0.01569637, -2.24268495])
```

В таблице 7 представлены некоторые функции модуля `numpy.random`.

Таблица 7: Некоторые функции модуля `numpy.random`

Функция	Описание
`seed`	Начальная точка генератора случайных чисел
`permutation`	Возвращает случайную перестановку последовательности или возвращает переставленный диапазон
`shuffle`	Произвольно переставляет последовательность
`rand`	Генерирует выборку из равномерного распределения
`randint`	Генерирует случайные целые числа из заданного интервала
`randn`	Генерирует выборку из нормального распределения со средним значением 0 и стандартным отклонением 1
`binomial`	Генерирует выборку из биномиального распределения
`normal`	Генерирует выборку из нормального (гауссового) распределения
`beta`	Генерирует выборку из β-распределения
`chisquare`	Генерирует выборку из χ^2-распределения
`gamma`	Генерирует выборку из Γ-распределения
`uniform`	Генерирует выборку из равномерного распределения на $[0, 1)$

7. Пример: Случайное блуждание

Моделирование случайного блуждания предоставляет иллюстрацию применения операций с массивами. Вначале рассмотрим случайное блуждание, начинающееся с 0 с шагами 1 и -1, происходящие с равной вероятностью.

Ниже представлен код сценария на чистом Python, который реализует простое случайное блуждание в 1000 шагов и использует модуль `random`:

```
import random

position = 0
walk = [position]
steps = 1000
for i in range(100):
    step = 1 if random.randint(0, 1) else -1
```

```
    position += step
    walk.append(position)
```

На рис. 3 графически представлены 100 первых значений одного случайного блуждания.

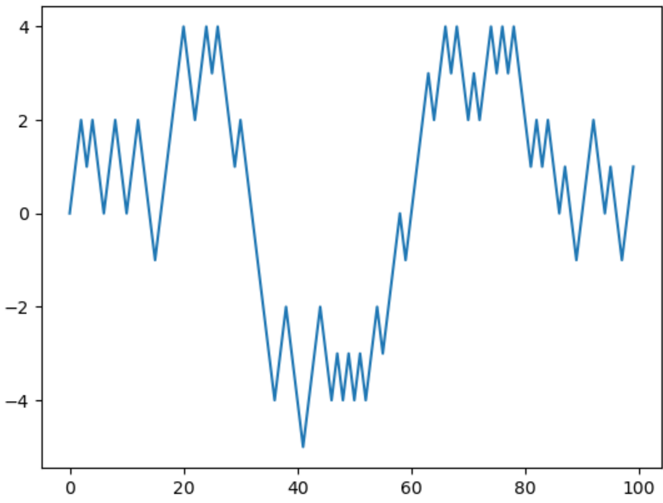

Рис. 3: Простое случайное блуждание

Заметим, что `walk` — это просто накопленная сумма случайных шагов, а она может быть вычислена с помощью метода массива. Таким образом, можно воспользоваться модулем `np.random` для генерации 1000 бросаний монеты за раз, установить соответствующие значения в 1 и -1 и вычислить накопленную сумму:

```
nsteps = 1000
draws = np.random.randint(0, 2, size=nsteps)
steps = np.where(draws > 0, 1, -1)
```

Отсюда можно получить статистические данные, такие, например, как минимум и максимум:

```
In [180]: walk.min()
Out[180]: -9

In [181]: walk.max()
Out[181]: 60
```

Более сложная статистика — *время первого «перехода»*, т.е. шаг, на котором пусть достигает некоторого значения. Например, можно узнать, сколько времени понадобилось случайному

блужданию, чтобы пройти как минимум 10 шагов от начала в любом направлении. Выражение `np.abs(walk) >= 10` даст булев массив, указывающий, где элемент `walk` достиг или превысил значение 10. Но нам нужен индекс первого элемента, равного 10 или -10. Мы можем получить это с помощью функции `argmax`, которая вернет первый индекс максимального значения в булевом массиве (`True` — максимальное значение):

```
In [182]: (np.abs(walk) >= 10).argmax()
Out[182]: 297
```

Обратите внимание, что использование argmax здесь не всегда эффективно, потому что оно всегда выполняет полное сканирование массива. В этом особом случае, когда есть значение `True`, мы знаем, что это максимальное значение.

8. Упражнения

8.1. Матрица из нулей и единиц

Напишите функцию `decorate_matrix`, которая получает на вход одно целое число больше единицы. Функция должна возвращать матрицу $N \times N$, у которой на границах стоят единицы, а на всех остальных позициях(если остались позиции не на границах) стоят нули.

8.2. Моделирование нескольких случайных блужданий за раз

Написать сценарий, в котором выполняется моделирование многих случайных блужданий (путей) (например, 5000 путей). Кроме того, вычислить максимальное и минимальное значения, полученные по всем путям. Вычислить минимальное время перехода через 30 или -30. Используйте разные генераторы случайных чисел. Имя файла: `many_random_walk_ans.py`.