Вычисления с помощью NumPy

1 Многомерные массивы в NumPy

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

Установить библиотеку numpy можно как показано на рисунке 1:

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

Рисунок 1 – Импорт numpy

Чтобы задать **numpy**-массив, достаточно задать обычный питоновский список **list**, а затем поместить его внутрь функции **np.array** (рис.2):

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

        print(a)

        [1 2 3]
```

Риснунок 2 – Создание массива

В отличие от стандартных питоновских структур данных, в **numpy** массивы предпочитают данные одного типа. Например, если функция **np.array** вызывается от списка, содержащего как целые (**int**), так и дробные (**float**) значения, то в результирующем массиве все значения будут приведены к типу **float**. Аналогично, если в подаваемом списке есть хотя бы одна строка **str**, то в соответствующем массиве все значения будут приведены к типу **str**. Если мы хотим задать свой тип (рис.3), к которому нужно привести данные, это можно сделать с помощью аргумента **dtype**:

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

        print(a)

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

Рисунок 3 – Создание массива заданного типа данных

Зададим теперь двумерный массив (рис.4):

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

        print(A)

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

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

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

Рисунок 4 – Создание двумерного массива

В случае срезов для **numpy**-массивов важно отметить, что, записывая срез numpy-массива, мы ничего нового не создаём, мы лишь получаем *представление* (*view*) - ссылку на какие-то отдельные элементы оригинального массива. Это означает, что если мы "создали" срез из **numpy**-массива, а затем поменяли в нём что-то - эти изменения коснутся и оригинального массива (рис.5):

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

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

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

        print(B)
```

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

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

        print(A)
```

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

Рисунок 5 – Срезы

Если мы хотим всего этого избежать и создать действительно новый массив, нужно использовать метод **copy** (рис.6):

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

        print(C)

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

Рисунок 6 – Создание нового массива

2 Работа с матрицами

Зададим матрицы, как на рисунке 7.

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

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

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

Рисунок 7 – Новые исходные данные для работы матрицами

В таблице 1 представлены основные операции над матрицами и результаты вычислений.

Таблица 1 – Основные операции над матрицами

Операция	Команда/ пример	Результат – print(массив)
Сложение	np.add C = np.add(A, B)	[[6 8] [10 12] [14 16]]
Вычитание	np.subtract C = np.subtract (A, B)	[[-6 -6] [-6 -6] [-6 -6]]
Умножение на скаляр	E = A * 3	[[0 3] [6 9] [12 15]]
Умножение матриц	C = A.dot(B)	[[-6 -6 -6] [-5 -1 3]]
Возведение матрицы в степень	Функция `matrix_power` из модуля numpy.linalg C = np.linalg.matrix_power(B, 3)	[[468 576 684] [1062 1305 1548] [1656 2034 2412]]
Задать единичную матрицу	I = np.eye(3)	[[1. 0. 0.] [0. 1. 0.] [0. 0. 1.]]

Транспонирование	np.transpose A_t = np.transpose(A) A.transpose A_t = A.transpose() A.T A_t = A.T	[[1 3] [0 5] [-1 -4]]
Вычислить определительл матрицы	функция det из модуля numpy.linalg d = np.linalg.det(B)	0.0
Вычислить ранг матрицы	функция matrix_rank из модуля numpy.linalg r = np.linalg.matrix_rank(B)	2
Найти обратную матрицу	Функция inv из модуля numpy.linalg F_inv = np.linalg.inv(F)	[[0.186 0.023 -0.081] [-0.976 0.627 0.302] [0.720 -0.534 -0.127]]

3 Генерирование массивов с заданными свойствами

В таблице 2 представлены разные способы задания массивов

Таблица 2 – Способы генерации массивов

Операция	Команда/ пример	Результат - print(массив)
Массив, состоящий из 0	np.zeros a = np.zeros((3, 4))	[[0. 0. 0. 0.] [0. 0. 0. 0.] [0. 0. 0. 0.]]
Массив, состоящий из 1	np.ones b = np.ones((3, 4))	[[1. 1. 1. 1.] [1. 1. 1. 1.] [1. 1. 1. 1.]]
Последовательность чисел от 0 до этого аргумента невключительно	np.arange ar1 = np.arange(10)	[0 1 2 3 4 5 6 7 8 9]
Последовательность чисел от первого аргумента до второго (включая первый, не включая второй)	np.arange ar2 = np.arange(2, 13)	[2 3 4 5 6 7 8 9 10 11 12]
Последовательность с установленным шагом (шаг может быть дробным или отрицательным)	np.arange ar3 = np.arange(2, 13, 2)	[2 4 6 8 10 12]
Вернуть заданное количество значений, равномерно расставленных между заданными началом и концом отрезка (левый, и правый концы отрезка включаются в массив)	np.arange c = np.linspace(2, 3, 10)	[2. 2.11111111 2.22222222 2.33333333 2.44444444 2.55555556 2.66666667 2.77777778 2.88888889 3.]

Массив заданной формы, состоящий из чисел, взятых из равномерного распределения на отрезке [0,1)	Функция sample из модуля numpy.random a = np.random.sample((3, 3))	[[0.939 0.364 0.174] [0.180 0.259 0.296] [0.867 0.626 0.603]]
Массив, но уже взятый из нормального распределения (со средним 0 и среднеквадратическим отклонением 1)	Функция randn из модуля numpy.random b = np.random.randn(3, 3)	[[-0.1419 0.641 0.049] [1.085 -0.8716 0.545] [-0.033 -1.002 -0.624]]
Массив из целых чисел в указанном диапазоне	Randint c = np.random.randint(0, 100, (3, 4))	[[78 38 56 21] [18 12 80 22] [33 71 35 22]]
Случайно выбранные элементы из заранее заданного массива	Choice A = np.arange(-10, 0) d = np.random.choice(A, (3, 4))	[[-1 -9 -9 -3] [-9 -5 -9 -5] [-6 -6 -7 -10]]

В numpy предусмотрено несколько способов изменения размерностей массивов (табл.3).

Таблица 3 – Способы изменения размеров массивов

Операция	Команда/ пример	Результат - print(массив)
Сделать из одномерного массива двумерный	reshape ar = np.arange(12) a = ar.reshape(3, 4)	Исходный массив [0 1 2 3 4 5 6 7 8 9 10 11] После преобразования [[0 1 2 3] [4 5 6 7] [8 9 10 11]]
Получить из многомерного массива одномерный	ar.flatten c = ar.flatten()	[0 1 2 3 4 5 6 7 8 9 10 11]

4 Соединение массивов

Рассмотрим два массива и разберёмся с тем, как их можно соединить в один. Зададим два массива (рис.8).

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

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

Рисунок 8 – Инициализация массивов

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

– с помощью функции **np.vstack**: c = np.vstack((a, b)) (получает на вход кортеж из массивов)

– с помощью функции **np.concatenate**: c = np.concatenate((a, b), axis=0) (тоже получает на вход кортеж, также нужно указать, вдоль какой оси производится конкатенация), что показано на рис.9.

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

        print(c)
```

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

Рисунок 9 – Вертикальное соединение саммивов

Также несколько способов это соединить массивы горизонтально (т.е. дописать один правее другого) (рис.10):

– с помощью функции **np.hstack**: c = np.hstack((a, b))

– с помощью функции **np.concatenate**: c = np.concatenate((a, b), axis=1) (производится теперь вдоль оси 11).

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

        print(d)
```

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

Рисунок 10 – Горизонтальное соединение массивов

Наконец, два двумерных массива можно соединить *в глубину* (т.е. вдоль новой третьей оси) с помощью функции **np.dstack** (рис.11):

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

        print(e)
```

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

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

Рисунок 11 – Соединение вдоль новой оси

5 Функции для работы с данными

Библиотека **numpy** предлагает удобный функционал для выбора данных из массива. Рассмотрим массив из 10 случайных целых значений от 0 до 19 (рис.12):

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

        print(a)

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

Рисунок 12 – Генерация массива из целых значений

Допустим, мы хотим выбрать все значения этого массива, которые больше 10. Вот как это можно сделать (рис. 13):

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

        print(b)

        [19 11 14 14 15 16]
```

Рисунок 13 – Выбор значений по условию

Свойства можно комбинировать, используя логические операторы "и" (обозначается символом &), "или" (символ |) и оператор отрицания "не" (символ ~). При этом каждое условие необходимо поставить в круглые скобки (рис.14):

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

        print(c)

        [14 14 16  8]
```

Рисунок 14 – Построение сложного условия

Такая конструкция в **numpy** называется *булевой индексацией*.

6 Сортировка

Рассмотрим двумерный массив (рис.15):

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

        print(a)

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

Рисунок 15 – Генерация массива для сортировки

Допустим, мы хотим отсортировать строки этого массива по второму столбцу. Мы можем сделать это вручную, задав индексы строк в нужном нам порядке (рис. 16):

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

        print(b)
```

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

Рисунок 16 – Сортировка через индесы

Этот процесс можно автоматизировать с помощью метода **a.argsort**. Данный метод возвращает массив из индексов массива **a** в порядке их возрастания по заданной оси (рис.17):

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

        print(ind)
```

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

Рисунок 17 – Сортировка при помощи метода **argsort**

В каждом столбце этого массива стоят индексы строк массива **a**, расположенные в том порядке, в котором они бы отсортировали данный столбец по возрастанию.

7 Перемешивание

Иногда оказывается нужно перемешать значения массива. Это можно сделать с помощью функции **shuffle** из модуля **numpy.random**. Эта функция ничего не возвращает, лишь перемешивает случайным образом элементы данного массива. Отметим, что она перемешивает массив только в первом измерении. Другими словами, если массив двумерный, она лишь переставит его строки местами. Содержимое самих строк при этом не изменится (рис.18):

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

        print(c)
```

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

Рисунок 18 – Переремешивание

8 Математические операции над массивами

Некоторые математические операции можно выполнять с массивами целиком. Например, мы уже знаем, что массивы можно умножать на число и что массивы одинаковой формы можно складывать. Зададим новые массивы (рис.19).

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

        print(a)
        print(b)
```

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

Рисунок 19 – Новые массивы для математических операций

На рисунке 20 представлен результат сложения и умнодения массивов

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

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

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

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

Рисунок 20 – Сложение и умножение массивов

К массивам можно также прибавлять числа - в этом случае к каждому элементу массива прибавляется число. Массивы одинакового размера можно поэлементно умножать. (Важно не путать с матричным умножением.)

С помощью метода `a.sum` можно посчитать сумму всех значений массива. Если указать в этом методе ось `axis`, сумма будет посчитана только вдоль этой оси (рис.21):

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

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

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

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

Рисунок 21 – Математические операции наз массивами

К массивам можно применять статистические функции, представленные в таблице 3.

Таблица 3 – Статистические функции

Фукнция	Значение
`a.min`	Минимальное значение
`a.max`	Максимальное значение
`a.mean`	Среднее значение
`a.std`	Среднее квадратическое отклонение