TOMSK POLYTECHNIC UNIVERSITY

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Углубленный курс информатики

Лекция 6

Массивы NumPy

Долганов Игорь Михайлович
к.т.н., доцент ОХИ ИШПР

15 марта 2023 г.

TOMSK POLYTECHNIC UNIVERSITY
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

СОДЕРЖАНИЕ

1. Преобразование списков Python в массивы NumPy
2. Создание массивов
3. Доступ к элементам массива
   - Срезы массивов: доступ к подмассивам
     - Одномерные подмассивы
     - Многомерные подмассивы
     - Доступ к строкам и столбцам массива
   - Создание копий массивов
4. Вычисления с массивами NumPy
   - Арифметические функции с массивами
   - Получение абсолютного значения
   - Тригонометрические функции
   - Показательные функции и логарифмы
5. Функции агрегирования
   - Суммирование значений массива
   - Минимум и максимум

Введение

- Библиотека NumPy (**Num**erical **Py**thon – «числовой Python») предоставляет набор эффективных инструментов для хранения и работы с данными.
- Массивы библиотеки NumPy отдаленно напоминают списки Python, однако обеспечивают намного более эффективное хранение и выполнение операций с данными при росте размера массивов.
- Массивы библиотеки NumPy формируют ядро практически всей экосистемы утилит для исследования данных Python.

По установившейся традиции, большинство пользователей импортируют пакет NumPy, используя сокращение np:

```
>>> import numpy as np
```

Преобразование списков Python в массивы NumPy

- Для того, чтобы создать объект массива NumPy из объекта списка Python, можно использовать функцию `np.array`:

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

[2]:
```
np.array([1, 3, 5, 4, 2])  # Массив целочисленных значений
```

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

- В отличие от стандартных списков Python, массивы NumPy могут содержать элементы только одного типа. Если типы элементов не совпадают, NumPy сделает попытку повышающего приведения типов:

[3]:
```
np.array([3.14, 4, 2, 3, 2.71])
```

[3]:
```
array([3.14, 4.  , 2.  , 3.  , 2.71])
```

Преобразование списков Python в массивы NumPy

- В тех случаях, когда требуется явно задать тип результирующего массива, необходимо воспользоваться ключевым аргументом dtype:

```
[4]: np.array([1, 3, 5, 4, 2], dtype='float32')

[4]: array([1., 3., 5., 4., 2.], dtype=float32)
```

- В отличие от списков, массивы NumPy можно явным образом описать как многомерные:

```
[5]: np.array([range(i, i + 3) for i in [2, 4, 6]])

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

Создание массивов

Массивы больших размеров эффективнее генерировать с помощью встроенных методов.

- Создаем массив целых чисел длины 10, заполненный нулями:

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

```
[2]: np.zeros(10, dtype=int)
```

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

- Создадим массив размером 3×5 значений с плавающей точкой, заполненный единицами:

```
[3]: np.ones((3, 5), dtype=float)
```

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

Создание массивов

- Создадим массив размером 3×5, заполненный значением 2.98:

```
[4]: np.full((3, 5), 2.98)
```

```
[4]: array([[2.98, 2.98, 2.98, 2.98, 2.98],
            [2.98, 2.98, 2.98, 2.98, 2.98],
            [2.98, 2.98, 2.98, 2.98, 2.98]])
```

- Создадим массив, заполненный линейной последовательностью, начинающейся с 0 и заканчивающейся 20, с шагом 2 (аналогично встроенной функции `range()`):

```
[5]: np.arange(0, 20, 2)
```

```
[5]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

- Создадим массив из пяти значений, равномерно располагающихся между 0 и 1:

```
[6]: np.linspace(0, 1, 5)
```

```
[6]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Создание массивов

- Создадим массив размером 3×3 равномерно распределенных случайных значений от 0 до 1

```
[7]: np.random.random((3, 3))
```

```
[7]: array([[0.28303209, 0.54071726, 0.93183376],
            [0.02403954, 0.92295936, 0.62619599],
            [0.06875703, 0.61762719, 0.47795471]])
```

- Создадим массив размером 3×3 случайных целых чисел в интервале $[0, 10]$

```
[8]: np.random.randint(0, 10, (3, 3))
```

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

Доступ к элементам массива

В одномерном массиве обратиться к i-му (считая с 0) значению можно по требуемому индексу в квадратных скобках, по аналогии со стандартными списками:

[2]:
```
x1 = np.array([5, 0, 3, 3, 7, 9])
```

[3]:
```
x1[0]
```

[3]: 5

[4]:
```
x1[4]
```

[4]: 7

[5]:
```
x1[-1]
```

[5]: 9

[6]:
```
x1[-2]
```

[6]: 7

Доступ к элементам массива

Для обращения к элементам матрицы нужно указать кортеж индексов, разделенных запятыми:

[7]:
```
x2 = np.array([[3, 5, 2, 4], [7, 6, 8, 8], [1, 6, 7, 7]])
```

[8]:
```
x2
```

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

[9]:
```
x2[0, 0]
```

[9]:
```
3
```

[10]:
```
x2[2, 0]
```

[10]:
```
1
```

[11]:
```
x2[2, -1]
```

[11]:
```
7
```

Доступ к элементам массива

При помощи любой из указанных выше нотаций можно изменять значения элементов массива:

[12]:
```
x2[0, 0] = 24
```

[13]:
```
x2
```

[13]:
```
array([[24,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

Следует помнить, что, в отличие от списков, массивы NumPy имеют фиксированный тип данных. Если вставить в массив целых чисел значение с плавающей точкой, оно будет неявно усечено:

[14]:
```
x1[0] = 2.71828  # Это значение будет усечено!
```

[15]:
```
x1
```

[15]:
```
array([2, 0, 3, 3, 7, 9])
```

Срезы массивов: доступ к подмассивам

- Синтаксически срезы массивов NumPy соответствуют срезам стандартных списков Python:

  `x[начало:конец:шаг]`

  при этом любое из значений можно не указывать, тогда по умолчанию будут приняты следующие значения: `начало = 0`, `конец = размер соответствующего измерения`, `шаг = 1`.

Одномерные подмассивы

[3]:
```
x = np.arange(10)
x
```

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

[4]:
```
x[:5]
```

[4]:
```
array([0, 1, 2, 3, 4])
```

[5]:
```
x[4:7]
```

[5]:
```
array([4, 5, 6])
```

Срезы массивов: доступ к подмассивам


```
[6]: x[::2]  # каждый второй элемент
```

```
[6]: array([0, 2, 4, 6, 8])
```

```
[7]: x[1::2]  # каждый второй элемент, начиная с индекса 1
```

```
[7]: array([1, 3, 5, 7, 9])
```

- Также сохранена возможность использования отрицательного значения параметра **шаг**. Тогда значения по умолчанию для параметров **начало** и **конец** будут поменяны местами. Это быстрый способ перевернуть массив:

```
[8]: x[::-1]  # все элементы в обратном порядке
```

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

```
[9]: x[5::-1]  # все элементы в обратном порядке, начиная с индекса 5
```

```
[9]: array([5, 4, 3, 2, 1, 0])
```

Срезы массивов: доступ к подмассивам

Многомерные подмассивы

[2]:
```
x = np.random.randint(1, 10, (3, 4))
x
```

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

[3]:
```
x[:2, :3]  # две строки, три столбца
```

[3]:
```
array([[3, 3, 4],
       [8, 6, 5]])
```

Срезы массивов: доступ к подмассивам

Многомерные подмассивы

[4]:
```
x[:3, ::2]  # все строки, каждый второй столбец
```

[4]:
```
array([[3, 4],
       [8, 5],
       [1, 9]])
```

[5]:
```
x[::-1, ::-1]  # обратный порядок строк и столбцов
```

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

Доступ к строкам и столбцам массива

Распространенной задачей является доступ к отдельным строкам или столбцам массива. Получить такой доступ можно при помощи комбинации операций индексации и среза:

```
>>> print(x2[:, 0])   # первый столбец массива x2
[24  7  1]
>>> print(x2[0, :])   # первая строка массива x2
[24  5  2  4]
```

При необходимости получения доступа к строке, операция взятия среза может быть опущена для более лаконичной записи:

```
>>> print(x2[0])  # эквивалентно x2[0, :]
[24  5  2  4]
```

Срезы массивов создают разделяемые ссылки

- Срезы массивов возвращают **разделяемые ссылки** (синонимы), а не **копии** данных массива.
- Этим срезы массивов библиотеки NumPy отличаются от срезов списков языка Python (срезы списков создают новые объекты):

```
>>> print(x2)
[[24  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Получим из него матрицу 2×2:

```
>>> x2_sub = x2[:2, :2]
>>> print(x2_sub)
[[24  5]
 [ 7  6]]
```

Срезы массивов создают разделяемые ссылки

Теперь, если изменить значения этой матрицы, исходный массив также поменялся:

```
>>> x2_sub[0, 0] = 100
>>> print(x2_sub)
[[100   5]
 [  7   6]]
>>> print(x2)
[[100   5   2   4]
 [  7   6   8   8]
 [  1   6   7   7]]
```

Создание копий массивов

В ряде случаев требуется явно скопировать содержимое массива или его части. Для решения данной задачи существует метод `copy()`:

```
>>> x2_sub_copy = x2[:2, :2].copy()
>>> print(x2_sub_copy)
[[100   5]
 [  7   6]]
```

Теперь, если изменить значения этого подмассива, то исходный массив не изменится:

```
>>> x2_sub_copy[0, 0] = 24
>>> print(x2_sub_copy)
[[24  5]
 [ 7  6]]
>>> print(x2)
[[100   5   2   4]
 [  7   6   8   8]
 [  1   6   7   7]]
```

TOMSK POLYTECHNIC UNIVERSITY

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Вычисления с массивами NumPy

Арифметические функции с массивами

Универсальные функции NumPy могут быть легко использованы, т.к. основаны на нативных арифметических операторах Python. Доступны обычные операторы сложения, вычитания, умножения и деления:

```
>>> x = np.arange(4)
>>> print('x      =', x)
x      = [0 1 2 3]
>>> print('x + 5  =', x + 5)
x + 5  = [5 6 7 8]
>>> print('x - 5  =', x - 5)
x - 5  = [-5 -4 -3 -2]
>>> print('x * 2  =', x * 2)
x * 2  = [0 2 4 6]
>>> print('x / 2  =', x / 2)
x / 2  = [0.  0.5 1.  1.5]
>>> print('x // 2 =', x // 2)
x // 2 = [0 0 1 1]
```

Арифметические функции с массивами

Определены также унарная универсальная функция изменения знака, оператор ** для возведения в степень и оператор % для деления по модулю:

```
>>> print('-x     =', -x)
-x     = [ 0 -1 -2 -3]
>>> print('x ** 2 =', x ** 2)
x ** 2 = [0 1 4 9]
>>> print('x % 2  =', x % 2)
x % 2  = [0 1 0 1]
```

Арифметические функции с массивами

Данные операции могут быть использованы в выражениях любыми способами с соблюдением стандартных приоритетов:

```
20 >>> -(0.5 * x + 1) ** 2
21 array([-1.  , -2.25, -4.  , -6.25])
```

Все арифметические операторы – удобные аналоги для встроенных функций библиотеки NumPy. Например, оператор `+` является аналогом функции `add()`:

Оператор	Эквивалентная функция	Описание
`+`	`np.add`	Сложение: `1 + 1 = 2`
`-`	`np.subtract`	Вычитание: `3 - 2 = 1`
`-`	`np.negative`	Унарная операция изменения знака: `-2`
`*`	`np.multiply`	Умножение: `2 * 3 = 6`
`/`	`np.divide`	Деление: `3 / 2 = 1.5`
`//`	`np.floor_divide`	Деление с округлением в меньшую сторону: `3 // 2 = 1`
`**`	`np.power`	Возведение в степень: `3 ** 2 = 9`
`%`	`np.mod`	Модуль/остаток: `5 % 2 = 1`

Получение абсолютного значения

Наряду со встроенными арифметическими операторами, с массивами NumPy можно использовать стандартную функцию `abs()` языка Python для получения абсолютного значения:

```
>>> x = np.array([-2, -1, 0, 1, 2])
>>> abs(x)
array([2, 1, 0, 1, 2])
```

Аналогичная универсальная функция NumPy – `np.absolute()`, доступна также под псевдонимом `np.abs()`:

```
>>> np.absolute(x)
array([2, 1, 0, 1, 2])
>>> np.abs(x)
array([2, 1, 0, 1, 2])
```

Тригонометрические функции

Библиотека NumPy предоставляет набор тригонометрических функций:

```
>>> alpha = np.linspace(0, np.pi, 3)
>>> print('alpha      = ', alpha)
alpha      =  [0.         1.57079633 3.14159265]
>>> print('sin(alpha) = ', np.sin(alpha))
sin(alpha) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
>>> print('cos(alpha) = ', np.cos(alpha))
cos(alpha) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
>>> print('tan(alpha) = ', np.tan(alpha))
tan(alpha) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

Значения вычисляются в пределах точности конкретной вычислительной машины, вследствие чего некоторые из них не всегда точно равны нулю, хотя должны.

Тригонометрические функции

Определены также и обратные тригонометрические функции:

```
>>> x = [-1, 0, 1]
>>> print('x = ', x)
x =  [-1, 0, 1]
>>> x = [-1, 0, 1]
>>> print('x = ', x)
x =  [-1, 0, 1]
>>> print('arcsin(x) = ', np.arcsin(x))
arcsin(x) =  [-1.57079633  0.          1.57079633]
>>> print('arccos(x) = ', np.arccos(x))
arccos(x) =  [3.14159265 1.57079633 0.        ]
>>> print('arctan(x) = ', np.arctan(x))
arctan(x) =  [-0.78539816  0.          0.78539816]
```

Показательные функции и логарифмы

Показательные функции – один из распространенных типов операций, доступных в NumPy:

```
>>> x = [1, 2, 3]
>>> print('x   = ', x)
x   =  [1, 2, 3]
>>> print('e^x = ', np.exp(x))
e^x =  [ 2.71828183  7.3890561  20.08553692]
>>> print('2^x = ', np.exp2(x))
2^x =  [2. 4. 8.]
>>> print('3^x = ', np.power(3, x))
3^x =  [ 3  9 27]
```

Показательные функции и логарифмы

Определены также и логарифмы. Простейшая функция `np.log()` возвращает натуральный логарифм числа. Если Вам требуется логарифм по основанию 2 или 10, они также доступны:

```
>>> x = [1, 2, 4, 10]
>>> print('x        =', x)
x        = [1, 2, 4, 10]
>>> print('ln(x)    =', np.log(x))
ln(x)    = [0.         0.69314718 1.38629436 2.30258509]
>>> print('log2(x)  =', np.log2(x))
log2(x)  = [0.         1.         2.         3.32192809]
>>> print('log10(x) =', np.log10(x))
log10(x) = [0.         0.30103    0.60205999 1.        ]
```

TOMSK POLYTECHNIC UNIVERSITY

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Функции агрегирования

Суммирование значений массива

В стандартном Python данная задача решается при помощи встроенной функции `sum()`. Синтаксис этой функции крайне похож на функцию `sum()` библиотеки NumPy:

```
>>> import numpy as np
>>> arr = np.random.random(100)
>>> sum(arr)
49.496408327779065
>>> np.sum(arr)
49.49640832777906
```

NumPy версия функции `sum()` работает намного быстрее:

```
>>> big_array = np.random.rand(1000000)
>>> %timeit sum(big_array)  # Магическая команда в IPython
62.8 ms ± 598 µs per loop (mean ± std. dev. of 7 runs, 10 loops each
>>> %timeit np.sum(big_array)
522 µs ± 5.53 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Важно заметить, что функции `sum()` и `np.sum()` не идентичны. Так, смысл их опциональных аргументов отличается и функция `np.sum()` может работать с многомерными массивами.

Минимум и максимум

В стандартном Python определены встроенные функции `min()` и `max()`, служащие для вычисления минимального и максимального значений любой коллекции:

```
>>> min(big_array), max(big_array)
(1.4200179176970806e-07, 0.9999998044567884)
```

Соответствующие функций NumPy имеют аналогичный синтаксис и работают быстрее:

```
>>> np.min(big_array), np.max(big_array)
(1.4200179176970806e-07, 0.9999998044567884)
>>> %timeit min(big_array)
43.4 ms ± 719 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> %timeit np.min(big_array)
304 µs ± 2.03 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Существует возможность вычисления некоторых сводных показателей путем вызова соответствующих методов объекта массива NumPy для более лаконичной записи:

```
>>> print(big_array.min(), big_array.max(), big_array.sum())
1.4200179176970806e-07 0.9999998044567884 499940.4506893933
```

TOMSK POLYTECHNIC UNIVERSITY
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Контакты

Долганов Игорь Михайлович
к.т.н., доцент ОХИ ИШПР

Учебный корпус №2, ауд. 136
dolganovim@tpu.ru
+7-960-978-43-07

Благодарю за внимание!