# Лекция 7. Python и NumPy

Сергей Лебедев

sergei.lebedev@jetbrains.com

27 марта 2015 г.

# IPython

- IPython — более лучшая интерактивная оболочка для языка Python.
- Она поддерживает:
  - интроспекцию и дополнение имён модулей, классов, методов, переменных,
  - систему “магических” команд `%%magic`,
  - а также
    - расширенную работу с историей,
    - использование стандартных команд UNIX, например, `ls`,
    - отрисовку LaTeX формул и графиков `matplotlib`
    - и многое другое, о чём мы сегодня не будем говорить.
- Установить можно с помощью менеджера пакетов `pip`:

```
$ pip install "ipython[all]"
```

```
In [1]: from collections import de<TAB>
defaultdict  deque

In [2]: d = {"foo": "bar"}

In [3]: d.c<TAB>
d.clear  d.copy

In [4]: def f():
   ...:     "I do nothing and return 42."
   ...:     return 42
   ...:

In [5]: f?  # вывод ?? также содержит исходный код.
Type:           function
String form: <function f at 0x103d68ae8>
File:           ...
Definition:  f()
Docstring:   I do nothing and return 42.
```

- “Магические” команды отвечают за магию.
- Однострочные “магические” команды начинаются с символа %, многострочные — с %%.
- Пример однострочной “магической” команды, которая позволяет отредактировать и вычислить содержимое ячейки с указанным номером:

```
In [1]: def f():
   ...:     pass
   ...:

In [2]: %edit 1
IPython will make a temporary file named: [...]
done. Executing edited code...
Out[2]: 'def f():\n    return 42\n'

In [3]: f()
Out[3]: 42
```

- По умолчанию IPython работает в режиме `%automagic`, то есть префикс % у однострочных и %% у многострочных команд можно не писать.
- Пример многострочной “магической” команды, которая сохраняет содержимое ячейки в указанный файл:

```
In [1]: %%writefile /tmp/example.py
   ...: def f():
   ...:     return 42
   ...:
Writing /tmp/example.py

In [2]: %load /tmp/example.py

In [4]: f()
Out[4]: 42

In [5]: %lsmagic  # DIY.
```

- IPython — удобная и полезная альтернатива стандартной интерактивной оболочке Python.
- Мы поговорили только про основы её использования. Больше информации можно найти на сайте: `http://ipython.org`.
- У IPython также есть веб-интерфейс, именуемый IPython Notebook. Если вы никогда о нём не слышали, обязательно попробуйте `ipython notebook` или посмотрите мотивирующий пример по ссылке `http://bit.ly/ipnb-example`.

# Python и производительность

```python
import math

def euclidean(xs, ys):
    n = len(xs)  # == len(ys)
    acc = 0.
    for i in range(n):
        acc += (xs[i] - ys[i]) ** 2

    return math.sqrt(acc)
```

- В IPython есть “магическая” команда `timeit`, которая позволяет измерять время выполнения выражения:

```
%timeit euclidean([1, 2, 3], [4, 5, 6])
100000 loops, best of 3: 2.56 µs per loop
```

- Команду `timeit` также можно использовать в многострочном варианте:

```
In [2]: import random

In [3]: def setup(size):
   ...:     xs = [random.random() for _ in range(size)]
   ...:     ys = [random.random() for _ in range(size)]
   ...:     return xs, ys

In [4]: %%timeit xs, ys = setup(8192)
   ...: euclidean(xs, ys)
   ...:
100 loops, best of 3: 2.72 ms per loop
```

# Модуль `line_profiler` и “магическая” команда `lprun`

- Модуль `line_profiler` анализирует время работы кода на Python с точностью до строки в исходном коде.
  `$ pip install line_profiler`
- Он расширяет систему “магических” команд IPython командой `lprun`. Чтобы воспользоваться ей, нужно загрузить модуль расширения:

```
In [1]: %load_ext line_profiler

In [2]: %lprun -f euclidean euclidean(*setup(8192))
#                ^                    ^
#                |                    |
#           имя функции          выражение
```

# Модуль line_profiler и расстояние Евклида

```
In [1]: %lprun -f euclidean euclidean(*setup(8192))
Timer unit: 1e-06 s

Total time: 0.011904 s

Time  Per Hit   % Time  Line Contents
==============================================
                                    def euclidean(xs, ys):
   3      3.0      0.0                  n = len(xs) # == len(ys)
   1      1.0      0.0                  acc = 0.
4716      0.6     39.6                  for i in range(n):
7177      0.9     60.3                      acc += (xs[i] - ys[i]) ** 2
   7      7.0      0.1                  return math.sqrt(acc)
```

Python — интерпретируемый язык $\Rightarrow$ любая операция с объектами соответствует одной или нескольким инструкциям интерпретатора[1].

```
acc += (xs[i] - ys[i]) ** 2
```

```
LOAD_FAST                3 (acc)
LOAD_FAST                0 (xs)
LOAD_FAST                4 (i)
BINARY_SUBSCR
LOAD_FAST                1 (ys)
LOAD_FAST                4 (i)
BINARY_SUBSCR
BINARY_SUBTRACT
LOAD_CONST               2 (2)
BINARY_POWER
INPLACE_ADD
STORE_FAST               3 (acc)
```

[1] Получить байт-код любой функции, метода, класса или модуля можно с помощью функции `dis` из одноимённого модуля стандартной библиотеки.

- В отличие, например, от Java или C/C++ в Python нет “примитивных” типов, все сущности языка — объекты.
- Таким образом, при сложении двух целых чисел `42 + 24` фактически исполняется что-то похожее на:

```
static PyObject *
long_add(PyLongObject *a, PyLongObject *b)
{
    PyObject *result = PyLong_FromLong(
        a->ob_value + b->ob_value)
    return result;
}
```

- Ещё одно досадное следствие: `42` и `24` — это совсем не 4 и даже не 8 байт. Объекту нужно хранить, как минимум, своё значение, указатель на свой класс и счётчик ссылок для сборщика мусора.

- Может показаться, что писать численные методы на Python — дело неблагодарное, потому что:
  - интерпретатор повсюду,
  - циклы медленные,
  - численные типы — это объекты.
- Но выход есть!
  - Вместо циклов можно использовать встроенные функции `map`, `filter`, `zip` или выражения-генераторы:

    ```
    def euclidean(xs, ys):
        acc = sum((x - y) ** 2 for x, y in zip(xs, ys))
        return math.sqrt(acc)
    ```

  - Вместо списков объектов численных типов — типизированные массивы из пакета NumPy.
- Бонус: можно ли переписать функцию `euclidean` без использования `zip` и выражения-генератора?

```
In [1]: %paste
def euclidean(xs, ys):
    for i in range(len(xs)):
        acc += (xs[i] - ys[i]) ** 2
    return math.sqrt(acc)

In [2]: %%timeit xs, ys = setup(8192)
   ...: euclidean(xs, ys)
   ...:
100 loops, best of 3: 2.74 ms per loop

In [3]: %paste
def euclidean(xs, ys):
    acc = sum(map(lambda x, y: (x - y) ** 2, xs, ys))
    return math.sqrt(acc)

In [4]: %%timeit xs, ys = setup(8192)
   ...: euclidean(xs, ys)
   ...:
100 loops, best of 3: 1.95 ms per loop
```

# NumPy

- NumPy — библиотека для научных вычислений на Python.
- В основе библиотеки многомерный типизированный массив фиксированного размера.
- Сравним результаты наших стараний с NumPy:

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

In [2]: def setup(size):
   ...:     xs = np.random.random(size)
   ...:     ys = np.random.random(size)
   ...:     return xs, ys

In [6]: %%timeit x, y = setup(8192)
   ...: np.sqrt(np.square(x - y).sum())
   ...:
10000 loops, best of 3: 28.2 µs per loop  # :-(
```

- Массивы можно создавать из любых коллекций Python, например, из списков или кортежей:

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

- Конструктор массива пытается угадать тип по элементам переданной коллекции. Это поведение можно изменить, передав тип явно:

```
In [2]: np.array([1, 2, 3]).dtype
Out[2]: dtype('int64')

In [3]: np.array([1, 2, 3], dtype=float).dtype
Out[3]: dtype('float64')
```

- Приведение типов при этом происходит автоматически:

```
In [4]: np.array([1, 2, 3], dtype=bytes)
Out[4]:
array([b'1', b'2', b'3'],
      dtype='|S1')
```

- Теоретически конструктора массива достаточно, чтобы покрыть все сценарии использования, например:

```
In [1]: np.array([0] * 8)
Out[1]: array([0, 0, 0, 0, 0, 0, 0, 0])
```

- Проблема такого подхода — промежуточный список. Чтобы этого избежать можно использовать специализированные функции-конструкторы:

```
In [2]: np.zeros(8, dtype=int)
Out[2]: array([0, 0, 0, 0, 0, 0, 0, 0])

In [3]: np.ones(8, dtype=int)
Out[3]: array([1, 1, 1, 1, 1, 1, 1, 1])

In [4]: np.full(8, 42, dtype=int)
Out[4]: array([42, 42, 42, 42, 42, 42, 42, 42])
```

# Операции с массивами: арифметические

```
In [1]: x = np.arange(4)  # [0, 1, 2, 3]

In [2]: -x
Out[2]: array([ 0, -1, -2, -3])

In [3]: y = np.array([0., 0.25, 0.5, 0.75])

In [4]: x + y   # автоматическое приведение типов
Out[4]: array([ 0., 1.25, 2.5, 3.75])

In [5]: x ** 2  # автоматическое приведение размерности
Out[5]: array([0, 1, 4, 9])

In [6]: y /= x + 1

In [6]: y
Out[6]: array([ 0., 0.125, 0.16666667, 0.1875])
```

# Операции с массивами: булевы

```
In [1]: x = np.arange(2)  # [0, 1]

In [2]: x < 5
Out[2]: array([ True, True], dtype=bool)

In [3]: x == 0
Out[3]: array([ True, False], dtype=bool)

In [4]: (x < 5) & (x == 0)  # или np.logical_and
Out[4]: array([ True, False], dtype=bool)

In [5]: (x < 5) | (x == 0)  # или np.logical_or
Out[5]: array([ True, True], dtype=bool)

In [6]: x == np.array([2, 1])
Out[6]: array([False,  True], dtype=bool)
```

- Арифметические и булевы операции — частные случаи универсальных функций (**u**niversal **func**tions), то есть функций, которые работают с массивом или массивами поэлементно.
- Универсальные функции могут использовать векторные (SIMD, single instruction-multiple data) инструкции, если ваш процессор их поддерживает.
- Среди универсальных функций, реализуемых NumPy, есть и математические функции[2]:

```
In [1]: np.exp(x)
Out[1]: array([ 1., 2.71828183])

In [2]: np.log1p(x)
Out[2]: array([ 0. ,  0.69314718])
```

[2] Полный список можно посмотреть в документации NumPy по ссылке: http://docs.scipy.org/doc/numpy/reference/ufuncs.html.

- Синтаксис срезов работает для массивов NumPy практически так же, как и для встроенных коллекций:

```
In [1]: x = np.arange(4)  # [0, 1, 2, 3]

In [2]: x[:2]
Out[2]: array([0, 1])

In [3]: x[1:]
Out[3]: array([1, 2, 3])
```

- Но если срез списка или кортежа — это копия данных, то срез массива NumPy разделяет данные со своим родителем.

```
In [4]: y = x[1:]

In [5]: y[0] = 42

In [6]: x
Out[6]: array([ 0, 42,  2,  3])
```

# Операции с массивами: маски и целочисленные индексы

- Массивы NumPy можно индексировать булевыми или целочисленными массивами:

```
In [1]: x = np.arange(4)  # [0, 1, 2, 3]

In [2]: x[x >= 2]
Out[2]: array([2, 3])
```

- Размер индексирующего массива может отличаться по размеру от индексируемого:

```
In [3]: x[[-1, -1]]
Out[3]: array([3, 3])
```

- Срезы, использующие маски или целочисленные индексы, копируют данные:

```
In [4]: y = x[[-1, -1]]

In [5]: y[0] = 42

In [6]: x
Out[6]: array([0, 1, 2, 3])
```

# Пример: построение обучающей и тестовой выборок

```
In [1]: %paste
def train_test_split(X, y, *, ratio):
    mask = np.random.uniform(size=len(y)) < ratio
    return X[mask], y[mask], X[~mask], y[~mask]
## -- End pasted text --

In [2]: X = np.arange(6).reshape((6, 1))

In [3]: X
Out[3]:
array([[0], [1], [2], [3], [4], [5]])

In [4]: y = np.ones(6)

In [5]: train_test_split(X, y, ratio=.5)
Out[5]:
(array([[0], [4], [5]]),  # X_train
 array([ 1.,  1.,  1.]),  # y_train
 array([[1], [2], [3]]),  # X_test
 array([ 1.,  1.,  1.]))  # y_test
```

Метод `np.array.astype` позволяет привести массив к другому типу, например:

```
In [1]: x = np.random.normal(42, size=4)

In [2]: x
Out[2]: array([ 42.09062774,  41.86181938,
   ...:         41.63114822,  42.76527928])

In [3]: x.astype(np.uint)
Out[3]: array([42, 41, 41, 42], dtype=uint64)

In [4]: x.astype(np.bool)
Out[4]: array([ True,  True,  True,  True], dtype=bool)
```

- Массивы NumPy могут иметь произвольную размерность[3]:

```
In [1]: X = np.zeros((2, 3))

In [2]: X
Out[2]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

In [3]: X.ndim
Out[3]: 2

In [4]: X.shape
Out[4]: (2, 3)
```

- Практически всё, сказанное про работу с одномерными массивами, справедливо и для многомерных массивов.

[3] N. B. Атрибут `shape` — это всегда кортеж, для одномерного массива он равен `(len(...), )`.

# Операции с многомерными массивами: транспонирование

Многомерные массивы можно транспонировать. На примере двумерного:

```
In [1]: X = np.arange(6).reshape((2, 3))

In [2]: X
Out[2]:
array([[0, 1, 2],
       [3, 4, 5]])

In [3]: X.T  # работает за O(1)!
Out[3]:
array([[0, 3],
       [1, 4],
       [2, 5]])
```

# Операции с многомерными массивами: умножение

- Вопреки интуиции двумерный массив NumPy — это не матрица в математическом смысле: умножение и возведение в степень работает **поэлементно**.

```
In [4]: X * X
Out[4]:
array([[ 0,  1,  4],
       [ 9, 16, 25]])
```

- Для матричного умножения следует использовать метод `np.array.dot` или одноимённую функцию:

```
In [5]: X.dot(X.T)
Out[5]:
array([[ 5, 14],
       [14, 50]])

In [6]: np.dot(X.T, X)
Out[6]:
array([[ 9, 12, 15],
       [12, 17, 22],
       [15, 22, 29]])
```

В отличие от одномерных массивов многомерные можно индексировать кортежем размерности `ndims`:

```
In [1]: X = np.arange(6).reshape((2, 3))

In [2]: X
Out[2]:
array([[0, 1, 2],
       [3, 4, 5]])

In [3]: X[0, 1]
Out[3]: 1

In [4]: X[:1, 1:]  # подматрица
Out[4]: array([[1, 2]])

In [5]: X[0, :]    # строчка; эквивалентно X[0]
Out[5]: array([0, 1, 2])

In [6]: X[:, 1]    # столбец; эквивалентно X.T[1]
Out[6]: array([1, 4])
```

# Автоматическое приведение размерности[4]

Массив `a` приводим к массиву `b` если:

- `a.ndim == b.ndim` и длины размерностей либо попарно совпадают, либо в паре одна из размерностей равняется 1.
- `a.ndim != b.ndim`, но в **начало** `a.shape` и `b.shape` можно добавить несколько размерностей длины 1 так, чтобы выполнялось предыдущее условие.

## Вопрос

Какое из условий работает в каждом из примеров?

```
In [1]: np.arange(6).reshape((2, 3)) + [[1, 2, 3]]
Out[1]:
array([[1, 3, 5],
       [4, 6, 8]])

In [2]: np.arange(6).reshape((2, 3)) + [1, 2, 3]
Out[2]:
array([[1, 3, 5],
       [4, 6, 8]])
```

[4] http://bit.ly/numpy-broadcasting

# Специальное значение np.newaxis

- Мотивирующий пример:

```
In [1]: X = np.arange(6).reshape((3, 2))

In [2]: X
Out[2]:
array([[0, 1],
       [2, 3],
       [4, 5]])

In [3]: X + np.array([1, 2, 3])

ValueError: operands [...] with shapes (3,2) (3,)
```

- С помощью специального значения `np.newaxis` можно добавить к существующему массиву размерность длины 1.

```
In [4]: X + np.array([1, 2, 3])[:, np.newaxis]
Out[4]: # работает за O(1)!
array([[1, 2],
       [4, 5],
       [7, 8]])
```

# Примеры использования np.newaxis

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

In [2]: x.shape
Out[2]: (3,)

In [3]: x[:, np.newaxis]  # x.shape?
Out[3]:
array([[1],
       [2],
       [3]])

In [4]: x[np.newaxis, :]  # x.shape?
Out[4]: array([[1, 2, 3]])

In [5]: x[np.newaxis, np.newaxis, :]
Out[5]: ???
```

## Вопрос

`x[:, np.newaxis]` — это копия данных `x`?

- Массивы NumPy реализуют некоторое количество стандартных свёрток:

```
In [1]: X = np.arange(6).reshape((2, 3))

In [2]: X.sum()
Out[2]: 15

In [3]: X.max()
Out[3]: 5

In [4]: np.unravel_index(np.argmin(X), X.shape)
Out[4]: (0, 0)
```

- Можно вычислить свёртку **вдоль** оси многомерного массива, например:

```
In [5]: X.mean(axis=0)  # вдоль строчек
Out[5]: array([ 1.5,  2.5,  3.5])

In [6]: X.mean(axis=1)  # вдоль столбцов
Out[6]: array([ 1.,  4.])
```

Массив NumPy — это блок памяти, который знает про тип элементов и способ их индексирования.

```
typedef struct PyArrayObject {
        /* Блок памяти */
        char *data;

        /* Дескриптор данных:
         *     тип (uint32, float64)
         *     размер в байтах (4, 8)
         *     порядок байт (BE, LE)
         */
        PyArray_Descr *descr;

        /* Способ индексирования данных */
        int nd;
        npy_intp *dimensions;
        npy_intp *strides;

        /* ... */
} PyArrayObject;
```

Шаги используются для преобразования многомерного индекса в индекс элемента в блоке памяти.

```
In [1]: X = np.arange(6, dtype=np.int8).reshape((2, 3))

In [2]: X
Out[2]:
array([[0, 1, 2],
       [3, 4, 5]], dtype=int8)

In [3]: X.strides  # Зачем так сложно?
Out[3]: (3, 1)

In [4]: bytes(X.data)
Out[4]: b'\x00\x01\x02\x03\x04\x05'

In [4]: pos = np.dot((1, 2), x.strides)
   ...: # ^^^ np.ravel_multi_index

In [5]: bytes(X.data)[pos]
Out[5]: 5
```

Многие операции над многомерными массивами выражаются в терминах манипуляций с `strides`, `shapes` и `data`:

```
In [1]: X = np.arange(6, dtype=np.int8).reshape((2, 3))

In [2]: X.strides
Out[2]: (3, 1)

In [3]: X.T.shape, X.T.strides
Out[3]: ((3, 2), (1, 3))

In [4]: X.ravel()
Out[4]: array([0, 1, 2, 3, 4, 5], dtype=int8)

In [5]: X.ravel().shape, X.ravel().strides
Out[5]: ((6, ), (1, ))

In [6]: X[0, :].shape, X[0, :].strides
Out[6]: (???, ???)
```

- Пакет NumPy реализует базовую функциональность для научных вычислений на Python — многомерный типизированный массив фиксированного размера.
- Использовать массив просто и приятно благодаря
  - перегрузке операторов,
  - автоматическому приведению типов,
  - автоматическому приведению размерности,
  - большому количеству универсальных функций, которые работают над массивами поэлементно.
- Инфраструктура для научных вычислений на Python переживает активный рост, в частности, мы **не** обсудили:
  - JIT и AOT компиляторы, поддерживающие NumPy, например, Numba или Cython,
  - SciPy и специализированные пакеты scikits,
  - средства визуализации `matplotlib`, `ggplot`, `Bokeh` и др.

# Хорошо и плохо

# Хорошо и плохо: инициализация

```
In [1]: np.array([x for x in range(6)])
Out[1]: array([0, 1, 2, 3, 4, 5])

In [2]: np.arange(6)                    # хорошо
Out[2]: array([0, 1, 2, 3, 4, 5])

In [3]: np.full(6, 42, dtype=int)   # хорошо
Out[4]: array([42, 42, 42, 42, 42, 42])

In [4]: np.array([42] * 6)
Out[4]: array([42, 42, 42, 42, 42, 42])

In [5]: [np.array([0.] * 3) for i in range(2)]
Out[5]: [array([ 0.,  0.,  0.]), array([ 0.,  0.,  0.])]

In [6]: np.zeros((2, 3))                # хорошо
Out[6]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

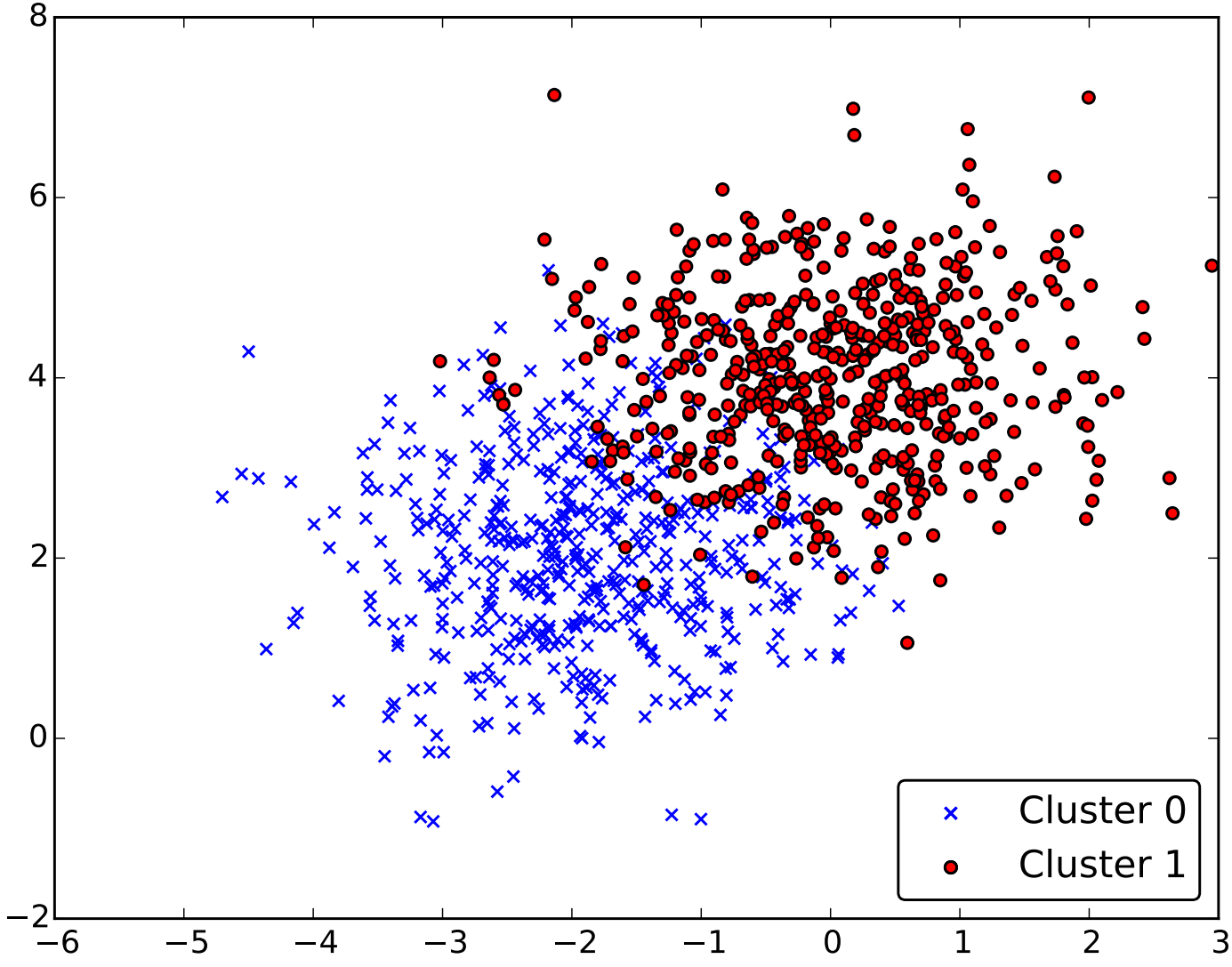
# Хорошо и плохо: индексирование

```
In [1]: X = np.arange(6).reshape((2, 3))

In [2]: X[0][1]
Out[2]: 1

In [3]: X[0, 1]  # хорошо
Out[3]: 1
```

```
In [1]: X = np.arange(6, dtype=np.int8).reshape((2, 3))

In [2]: X.sum(axis=1)  # хорошо
Out[2]: array([ 3, 12])

In [3]: np.array([row.sum() for row in X])
Out[3]: array([ 3, 12])

In [4]: acc = []

In [5]: for row in X:
   ....:     acc.append([])
   ....:     for col in row:
   ....:         acc[-1].append(np.exp(col))
   ....:

In [6]: np.array(acc)
Out[6]:
array([[   1.       ,    2.71875  ,    7.390625],
       [  20.078125,   54.59375  ,  148.375   ]])
```

# k-средних

```
import numpy as np
from numpy import random


def sample(size, ratio=.5):
    y = np.random.random(size) <= ratio
    n1 = np.count_nonzero(y)
    n0 = size - n1

    covar = np.diag([1, 1])
    X = np.empty((size, 2))
    X[y == 0, :] = random.multivariate_normal(
        [-2, 2], covar, n0)
    X[y == 1, :] = random.multivariate_normal(
        [0, 4], covar, n1)
    return X, y
```


8
6
4
2
0
−2
−6
−5
−4
−3
−2
−1
0
1
2
3
Cluster 0
Cluster 1

```
def ceuclidean(A, B):
    assert A.ndim == B.ndim == 2
    D = np.empty((len(A), len(B)), dtype=np.float64)
    for i, Ai in enumerate(A):
        for j, Bj in enumerate(B):
            D[i, j] = np.sqrt(np.square(Ai - Bi).sum())
    return D
```

**Вопрос**

Можно ли ускорить функцию `ceuclidean` с помощью рассмотренных нами возможностей NumPy?

```
def ceuclidean(A, B):
    assert A.ndim == B.ndim == 2
    D = np.empty((len(A), len(B)), dtype=np.float64)
    for i, Ai in enumerate(A):
        D[i, :] = np.sqrt(np.square(Ai - B).sum(axis=1))
    return D
```

**Замечание**

Функция `ceuclidean` интересна нам исключительно в учебных целях, на практике лучше использовать функцию `cdist` из пакета SciPy.

```
def init_centers(X, n_clusters):
    n_samples, n_features = X.shape
    centers = np.empty((n_clusters, n_features))
    centers[0] = X[random.choice(n_samples)]

    for i in range(1, n_clusters):
        [D] = np.square(ceuclidean(centers[i - 1:i], X))
        D /= D.sum()

        centers[i] = X[random.choice(n_samples, p=D)]

    return centers
```

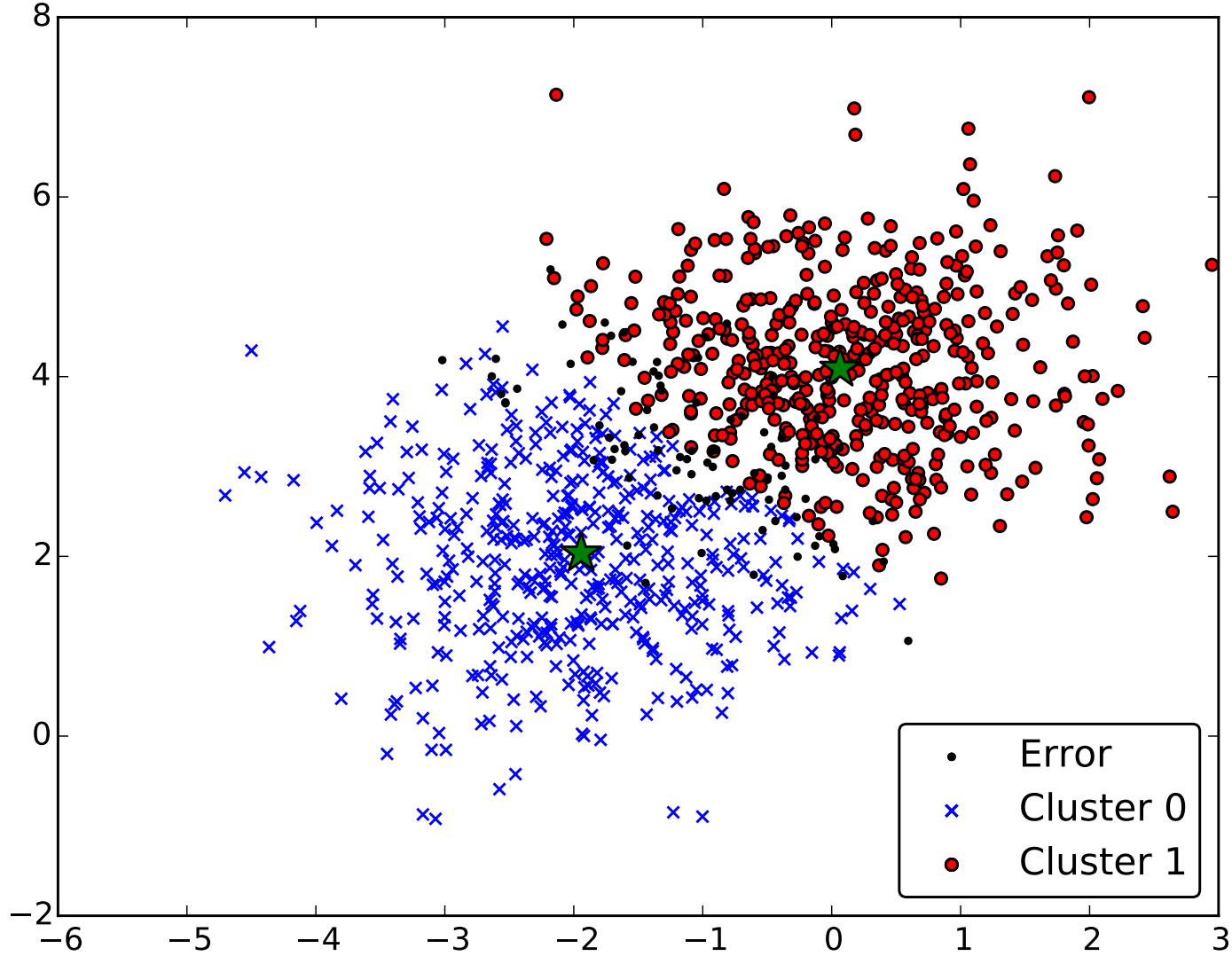
**Вопрос**

Ничего не напоминает?

```
def kmeans(X, n_clusters):
    centers = init_centers(X, n_clusters)
    y = None
    while True:
        D = ceuclidean(centers, X)  # *
        new_y = D.argmin(axis=0)
        if np.array_equal(y, new_y):
            break

        y = new_y
        for i in range(n_clusters):
            centers[i] = X[y == i].mean(axis=0)
    return centers, y
```

## Вопрос

Есть ли разница, в каком порядке передавать аргументы в функцию `ceuclidean` в строчке, помеченной `*`?


8
6
4
2
0
−2
−6
−5
−4
−3
−2
−1
0
1
2
3
Error
Cluster 0
Cluster 1