Массивы в Numpy

```
In [3]: import numpy as np # from numpy import *
```

Значения в ячейках имеют одинаковый тип.

```
In [122]: a = np.array([(1, 2.15, 3),(1, 2, 45)])
          print a, type(a), type(a[0])
```

```
[[  1.    2.15  3.  ]
 [  1.    2.   45.  ]] <type 'numpy.ndarray'> <type 'numpy.ndarray'>
```

```
In [21]: b = np.array([[1, 2], [1, 2.15, 3]])
         print b, type(b), type(b[0])
```

```
[[1, 2] [1, 2.15, 3]] <type 'numpy.ndarray'> <type 'list'>
```

Можно обращаться к ячейке и менять значение. В стиле C или Python - как больше нравится

```
In [35]: a[0][0] = 2
         a[0, 2] = 25.5
         print a
```

```
[[  2.    2.15  25.5 ]
 [  1.    2.   45.  ]]
```

Есть куча полезных функций - размерность, размерность, ... размер :-)
Тип данных в ячейках, размер элемента в байтах

```
In [27]: print a.ndim, a.shape, a.size
         print a.dtype, a.itemsize
```

```
2 (2, 3) 6
float64 8
```

Numpy при возможности всегда возвращает объект своего типа

```
In [123]: a, b = np.arange(3), range(3)
          print a, type(a)
          print b, type(b)
```

```
[0 1 2] <type 'numpy.ndarray'>
[0, 1, 2] <type 'list'>
```

Можно менять форму матрицы, но ступенчатую не получить никак

```
In [124]: a = np.array([(1, 2.15, 3),(1, 2, 45)])
          print a
```

```
[[  1.    2.15  3.  ]
 [  1.    2.   45.  ]]
```

```
In [125]: a.reshape(3, 2)

Out[125]: array([[  1.  ,   2.15],
                 [  3.  ,   1.  ],
                 [  2.  ,  45.  ]])
```

Можно задать нужный тип

```
In [39]: np.array([[1, 2], [3, 4]], dtype=complex)

Out[39]: array([[ 1.+0.j,  2.+0.j],
                [ 3.+0.j,  4.+0.j]])
```

Есть всякие генераторы массивов.

empty вернет массив с мусором! И все три функции вернут массивы с объектами типа float, если не задано иное.

```
In [43]: np.empty((2,3))

Out[43]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.]])

In [54]: np.zeros((3, 4), dtype=int)

Out[54]: array([[0, 0, 0, 0],
                [0, 0, 0, 0],
                [0, 0, 0, 0]])

In [45]: np.ones((2, 4))

Out[45]: array([[ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.]])
```

Numpy позаботится и выведет красиво.

```
In [60]: np.arange(10000).reshape(100,100)

Out[60]: array([[   0,    1,    2, ...,   97,   98,   99],
                [ 100,  101,  102, ...,  197,  198,  199],
                [ 200,  201,  202, ...,  297,  298,  299],
                ...,
                [9700, 9701, 9702, ..., 9797, 9798, 9799],
                [9800, 9801, 9802, ..., 9897, 9898, 9899],
                [9900, 9901, 9902, ..., 9997, 9998, 9999]])
```

0.1 Арифметические операции

```
In [85]: a = np.arange(10).reshape(2, 5)
         print a

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

In [67]: b = np.arange(0, 30, 3).reshape(2, 5)
         print b

[[ 0  3  6  9 12]
 [15 18 21 24 27]]
```

Есть функции вида np.subtract(a, b), но обычные операторы питона тоже работают.

```
In [68]: print a - b
```

```
[[  0  -2  -4  -6  -8]
 [-10 -12 -14 -16 -18]]
```

```
In [69]: print a * b
```

```
[[  0   3  12  27  48]
 [ 75 108 147 192 243]]
```

Ну да, логично... Стоп. Что-то пошло не так. Умножать мартицы все же стоит иначе.

```
In [131]: c = np.linspace(0, 4.5, 10).reshape(5, 2)
          print c
```

```
[[ 0.   0.5]
 [ 1.   1.5]
 [ 2.   2.5]
 [ 3.   3.5]
 [ 4.   4.5]]
```

Вот так надо умножать матрицы :-)

```
In [86]: print np.dot(a, c)
```

```
[[ 30.   35. ]
 [ 80.   97.5]]
```

Есть все стандартные функции. Они также применяются поэлементно.

```
In [74]: np.sin(a)
```

```
Out[74]: array([[ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ],
                [-0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849]])
```

Ну и есть набор для статистики. В изображениях тоже может пригодиться

```
In [132]: c.sum(), c.min(), c.max()
```

```
Out[132]: (22.5, 0.0, 4.5)
```

0.1.1 Приведение типов

```
In [5]: a = np.arange(10).reshape(5, 2)
        print a, a.dtype
```

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]] int64
```

```
In [4]: c = np.linspace(0, 4.5, 10).reshape(5, 2)
        print c, c.dtype
```

```
[[ 0.   0.5]
 [ 1.   1.5]
 [ 2.   2.5]
 [ 3.   3.5]
 [ 4.   4.5]] float64
```

```
In [129]: a += c
          print a, a.dtype
```

```
[[ 0  1]
 [ 3  4]
 [ 6  7]
 [ 9 10]
 [12 13]] int64
```

```
In [6]: c += a
        print c, c.dtype
```

```
[[  0.    1.5]
 [  3.    4.5]
 [  6.    7.5]
 [  9.   10.5]
 [ 12.   13.5]] float64
```

0.1.2 Как итерироваться, если очень хочется?

```
In [96]: for row in c:
             print row
```

```
[ 0.   0.5]
[ 1.   1.5]
[ 2.   2.5]
[ 3.   3.5]
[ 4.   4.5]
```

```
In [7]: for i in c.flat:
            print i
```

```
0.0
1.5
3.0
4.5
6.0
7.5
9.0
10.5
12.0
13.5
```

0.1.3 Когда будет копирование?

Ответ - почти никогда.

```
In [103]: b = c
          print b is c
```

```
True
```

```
In [106]: b = np.arange(10)
          print c
```

```
[[ 0.   0.5]
 [ 1.   1.5]
 [ 2.   2.5]
 [ 3.   3.5]
 [ 4.   4.5]]
```

```
In [108]: b = c
          b.shape = 10, 1
          print c.shape
```

```
(10, 1)
```

Пареметры у вызова функций тоже не создают копий.

0.1.4 Представления

Возможность посмотреть на те же данные.

Копирования нет, менять ячейки можно, атрибуты - нельзя.

```
In [109]: c = a.view()
          c is a
```

```
Out[109]: False
```

```
In [110]: a.shape
```

```
Out[110]: (5, 2)
```

```
In [114]: c.shape = 1, 10
          c.shape
```

```
Out[114]: (1, 10)
```

```
In [115]: a.shape
```

```
Out[115]: (5, 2)
```

```
In [116]: print c
```

```
[[ 0  1  3  4  6  7  9 10 12 13]]
```

```
In [117]: c[0, 0] = 42
          print c
```

```
[[42  1  3  4  6  7  9 10 12 13]]
```

```
In [118]: print a
```

```
[[42  1]
 [ 3  4]
 [ 6  7]
 [ 9 10]
 [12 13]]
```

Можно ли скопировать по-честному? Да.

```
In [120]: f = a.copy()
          f[0, 0] = -200
          print f
```

```
[[-200    1]
 [   3    4]
 [   6    7]
 [   9   10]
 [  12   13]]
```

```
In [121]: print a
```

```
[[42  1]
 [ 3  4]
 [ 6  7]
 [ 9 10]
 [12 13]]
```

0.1.5 Что такое item() и зачем он нужен?

```
In [137]: print a
```

```
[[ 0  1]
 [ 3  4]
 [ 6  7]
 [ 9 10]
 [12 13]]
```

```
In [134]: a.item(2)
```

```
Out[134]: 3
```

```
In [136]: a.item(2, 1)
```

```
Out[136]: 7
```

Вернет в любом случае копию одного числа. Итерироваться с помощью item() будет быстрее, но поможет это лишь для чтения данных. Чтобы записать, придется все равно обратиться по индексу.

Нужно для того, чтобы делать сложные математические вычисления, которые Python смог бы оптимизировать.

Не советую использовать в рамках курса. На мой взгляд, это не то, что нам нужно.

```
In [ ]:
```