Тема 2. Пакет NumPy

NumPy — это расширение языка Python, добавляющее поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых математических функций для операций с этими массивами.

NumPy начало работы

NumPy — это библиотека языка Python, добавляющая поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.

Установка NumPy

Linux:

```
sudo pip3 install numpy
```

Windows:

```
pip3 install numpy
```

Интересная ссылка
https://pyprog.pro/reference_manual.html

Наиболее важные атрибуты

Основным объектом NumPy является однородный многомерный массив (в **numpy** называется **numpy.ndarray**). Это многомерный массив элементов (обычно чисел), одного типа.

Наиболее важные атрибуты объектов **ndarray**:

ndarray.ndim - число измерений (чаще их называют "оси") массива.

ndarray.shape - размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси. Для матрицы из n строк и m столбов, shape будет (n,m). Число элементов кортежа shape равно ndim.

ndarray.size - количество элементов массива. Очевидно, равно произведению всех элементов атрибута shape.

ndarray.dtype - объект, описывающий тип элементов массива. Можно определить **dtype**, используя стандартные типы данных Python. NumPy здесь предоставляет целый букет возможностей, как встроенных, например: bool_, character, int8, int16, int32, int64, **float**8, **float**16, **float**32, **float**64, complex64, object_, так и возможность определить собственные типы данных, в том числе и составные.

ndarray.itemsize - размер каждого элемента массива в байтах.

ndarray.data - буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

Создание массивов

В NumPy существует много способов создать массив.

Один из наиболее простых - создать массив из обычных списков или кортежей Python. Для этого используетя функция **numpy**.**array**() (**array** - функция, создающая объект типа **ndarray**):

```
>>>
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>
```

Функция **array**() трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности (но можно и переопределить его в момент создания).

```
>>>
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Можно также переопределить тип в момент создания:

```
>>>
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]],
                                        dtype=np.complex)
>>> b
array([[ 1.5+0.j,  2.0+0.j,  3.0+0.j],
       [ 4.0+0.j,  5.0+0.j,  6.0+0.j]])
```

Функция **array()** не единственная функция для создания массивов. Обычно элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип создаваемого массива — **float64**).

Функция

zeros() создает массив из нулей,

а функция

ones() — массив из единиц.

Обе функции принимают кортеж с размерами, и аргумент **dtype**:

```
>>>
>>> np.zeros((3, 5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> np.ones((2, 2, 2))
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]])
```

Функция **eye()** создаёт единичную матрицу (двумерный массив)

```
>>>
>>> np.eye(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

Функция

empty() создает массив без его заполнения. Исходное содержимое случайно и зависит от состояния памяти на момент создания массива (то есть от того мусора, что в ней хранится):

```
>>>
>>> np.empty((3, 3))
array([[  6.93920488e-310,   6.93920488e-310,   6.93920149e-310],
       [  6.93920058e-310,   6.93920058e-310,   6.93920058e-310],
       [  6.93920359e-310,   0.00000000e+000,   6.93920501e-310]])
>>> np.empty((3, 3))
array([[  6.93920488e-310,   6.93920488e-310,   6.93920147e-310],
       [  6.93920149e-310,   6.93920146e-310,   6.93920359e-310],
       [  6.93920359e-310,   0.00000000e+000,   3.95252517e-322]])
```

Полный формат вызова функций:

```
numpy.zeros(shape, dtype = float, order = 'C')
numpy.ones(shape, dtype = float, order = 'C')
numpy.empty(shape, dtype = float, order = 'C')
```

где:

shape - обязательный аргумент, **кортеж** требуемой размерности массива;

dtype - опциональный аргумент, тип элементов массива, по умолчанию **float**;

order - опциональный аргумент, строка, способ представления данных массива в памяти, два возможных значения 'C' и 'F' – «как в C» или «как в фортране».

Для создания последовательностей чисел, в NumPy имеется функция

arange(), аналогичная встроенной в Python **range(),** только вместо списков она возвращает массивы, и принимает не только целые значения:

```
>>>
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 1, 0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,
0.8,  0.9])
```

Полный формат вызова функции:

numpy.arange([start,] stop[, step,], dtype = None),

где:

start - опциональный аргумент, начальное значение интервала, по умолчанию равен 0;

stop - обязательный аргумент, конечное значение интервала, не входящее в сам интервал, интервал замыкает значение stop – step;

step - опциональный аргумент, шаг итерации, разность между каждым последующим и предыдущим значениями интервала, по умолчанию равен 1;

dtype - тип элементов массива, по умолчанию None, в этом случае тип элементов определяется по типу переданных функции аргуметов start, stop.

Массив может быть создан так же с помощью функции **numpy.linspace()**.

Вообще, при использовании **arange()** с аргументами типа **float**, сложно быть уверенным в том, сколько элементов будет получено (из-за ограничения точности чисел с плавающей запятой). Поэтому, в таких случаях обычно лучше использовать функцию:

linspace(), которая, *вместо шага в качестве одного из аргументов принимает число, равное количеству нужных элементов*:

```
>>>
>>> np.linspace(0, 2, 9) #9 чисел от 0 до 2 включительно
array([  0. ,  0.25,  0.5 ,  0.75, 1. , 1.25, 1.5 , 1.75, 2. ])
```

Полный формат вызова функции:

numpy.**linspace**(start, stop, num = 50,
endpoint = **True**, retstep = **False**),

где:

start - обязательный аргумент, первый член последовательности элементов массива;

stop - обязательный аргумент, последний член последовательности элементов массива;

num - опциональный аргумент, количество элементов массива, по умолчанию равен 50;

endpoint - опциональный аргумент, логическое значение, по умолчанию **True**. Если передано **True**, то stop последний элемент массива. Если установлено в **False**, то последовательность элементов формируется от start до stop для num + 1 элементов, *при этом в возвращаемый массив последний элемент не входит*;

```
>>> d = np.linspace(1.0, 6.0, 5, endpoint = False)
>>> d
array([ 1.,  2.,  3.,  4.,  5.])
```

retstep - опциональный аргумент, логическое значение, по умолчанию **False**. Если передано **True** то, функция возвращает кортеж из двух членов, первый - массив, последовательность элементов, второй - число, приращение между элементами последовательности.

```
>>> f = np.linspace(.5, -.5, 5, retstep = True)
>>> f
(array([ 0.5 ,  0.25,  0.  , -0.25, -0.5 ]), -0.25)
```

fromfunction(): применяет функцию ко всем комбинациям индексов

```
>>>
>>> def f1(i, j):
...     return 3 * i + j
...
>>> np.fromfunction(f1, (3, 4))
array([[ 0.,  1.,  2.,  3.],
       [ 3.,  4.,  5.,  6.],
       [ 6.,  7.,  8.,  9.]])

>>> np.fromfunction(f1, (3, 3))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

Ну и последний из рассматриваемых способов с помощью функций **numpy.zeros_like()**, **numpy.ones_like()**, **numpy.empty_like()**.

Обязательный аргумент, принимаемый функциями - массив, функции возвращают массив такой же структуры, содержащий, соответственно, нули, единицы и то, что оказалось в памяти на момент создания массива.

```
>>> d
array([ 1.,  2.,  3.,  4.,  5.])

>>> dz = np.zeros_like(d)
>>> dz
```

```
array([ 0.,  0.,  0.,  0.,  0.])

>>> do = np.ones_like(d)
>>> do
array([ 1.,  1.,  1.,  1.,  1.])

>>> de = np.empty_like(d)
>>> de
array([  2.24122267e+201,   6.32526950e-317,
                                  0.00000000e+000,
         2.24686637e+201,   6.34874355e-321])
```

Полный формат вызова функций:

numpy.zeros_like(a)
numpy.ones_like(a)
numpy.empty_like(a)

где:

a - обязательный аргумент, массив, структуру которого необходимо повторить;

Массивы можно использовать в различных итерациях:

```
>>> for i in a:
...     print( i )
...
0.1
0.2
0.3
0.4
0.5

>>> for i in a3:
...     for j in i:
...             print( j )
...
[1 2]
[3 4]
[5 6]
[7 8]
[ 9 10]
[11 12]
```

Печать массивов

Если массив слишком большой, чтобы его печатать, NumPy автоматически скрывает центральную часть массива и выводит только его уголки.

```
>>>
>>> print(np.arange(0, 3000, 1))
[   0    1    2 ..., 2997 2998 2999]
```

Если вам действительно нужно увидеть весь массив, используйте функцию
numpy.set_printoptions:

```
np.set_printoptions(threshold=np.nan)
```

И вообще, с помощью этой функции можно настроить печать массивов "под себя".

Функция **numpy.set_printoptions** принимает несколько аргументов:

precision : количество отображаемых цифр после запятой (по умолчанию 8).

threshold : количество элементов в массиве, вызывающее обрезание элементов (по умолчанию 1000).

edgeitems : количество элементов в начале и в конце каждой размерности массива (по умолчанию 3).

linewidth : количество символов в строке, после которых осуществляется перенос (по умолчанию 75).

suppress : если **True**, не печатает маленькие значения в scientific notation (по умолчанию False).

nanstr : строковое представление NaN (по умолчанию 'nan').

infstr : строковое представление inf (по умолчанию 'inf').

formatter : позволяет более тонко управлять печатью массивов. Здесь не рассматриваем – смотрите
https://docs.scipy.org/doc/numpy/reference/generated/numpy.set_printoptions.html

Пользуйтесь официальной документацией по numpy - https://docs.scipy.org/doc/numpy/reference/.

NumPy, базовые операции над массивами

Математические операции над массивами выполняются поэлементно. Создается новый массив, который заполняется результатами действия оператора.

```
>>>
>>> import numpy as np
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
```

```
>>> a + b
array([20, 31, 42, 53])
>>> a - b
array([20, 29, 38, 47])
>>> a * b
array([  0,  30,  80, 150])
>>> a / b  # При делении на 0 возвращается inf (бесконечность)
array([        inf,  30.        ,  20.        ,  16.66666667])
<string>:1: RuntimeWarning: divide by zero encountered in true_divide
>>> a ** b
array([     1,     30,   1600, 125000])
>>> a % b  # При взятии остатка от деления на 0 возвращается 0
<string>:1: RuntimeWarning: divide by zero encountered in remainder
array([0, 0, 0, 2])
```

Для этого, естественно, массивы должны быть одинаковых размеров.

```
>>>
>>> c = np.array([[1, 2, 3], [4, 5, 6]])
>>> d = np.array([[1, 2], [3, 4], [5, 6]])
>>> c + d
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: operands could not be broadcast together
with shapes (2,3) (3,2)
```

Также можно производить математические операции между массивом и числом. В этом случае к каждому элементу прибавляется (или что вы там делаете) это число.

```
>>>
>>> a + 1
array([21, 31, 41, 51])
>>> a ** 3
array([  8000,  27000,  64000, 125000])
>>> a < 35  # И фильтрацию можно проводить
array([ True,  True, False, False], dtype=bool)
```

NumPy также предоставляет множество математических операций для обработки массивов:

```
>>>
>>> np.cos(a)
array([ 0.40808206,  0.15425145, -0.66693806,
0.96496603])
>>> np.arctan(a)
array([ 1.52083793,  1.53747533,  1.54580153,
1.55079899])
>>> np.sinh(a)
array([  2.42582598e+08,   5.34323729e+12,
1.17692633e+17,
```

```
        2.59235276e+21])
```

Полный список можно посмотреть
https://docs.scipy.org/doc/numpy/reference/routines.math.html

Список полезных математических функций пакета NumPy

Во всех определениях ниже **a** массив или скаляр.

numpy.**abs**(a) - абсолютное значение a;

numpy.**around**(a, **decimals** = 0, **out** = None) - округляет a до заданного количества десятичных разрядов, по умолчанию до целого. Придерживается следующего правила округления. Если значение a находиться точно по середине между двумя целыми, округление производиться до ближайшего **четного** целого. Так если a равно 1.5 или 2.5 будет возвращено 2, если a равно -0.5 или 0.5 будет возвращено 0.0. Аргумент **decimals** - целое, десятичный разряд после запятой, до которого производиться округление, если **decimals** отрицательное, разряд отсчитывается влево от запятой.

```
Print(np.around(np.array([0.5, 1.8, 2.5, 3.5])))
[ 0.  2.  2.  4.]

print(np.around(np.array([1, 5, 15, 45]), decimals= 1))
[ 1  5 15 45]

Print(np.around(np.array([1, 5, 15, 45]),decimals= -1))
[ 0  0 20 40]
```

Аргумент **out** - массив, в который будет передан результат, структура массива out, должна совпадать со структурой возвращаемого массива. Если None (по умолчанию) будет создан новый массив.

numpy.**fix**(a, **out** = None) - отбрасывает дробную часть a;

numpy.**ceil**(a, **out** = None) - округляет a до меньшего из целых больших или равных a;

```
>>>  print (np.ceil(np.array([-2.7,
                          -1.2, -0.5, 1.8, 2.4, 3.6])))
[-2. -1. -0.  2.  3.  4.]
```

numpy.**floor**(a, **out** = None) - округляет a до большего из целых меньших или равных a;

```
>>> print np.floor(np.array([-2.7, -1.2, -0.5, 1.8,
2.4, 3.6]))
```

```
[-3. -2. -1.  1.  2.  3.]
```

numpy.**sign**(a) - возвращает -1 если a < 0, 0 если a == 0, 1 если a > 0;

numpy.**degrees**(a) - конвертирует a из радиан в градусы;

numpy.**radians**(a) - конвертирует a из градусов в радианы;

numpy.**cos**(a), **numpy**.**sin**(a), **numpy**.**tan**(a) - возвращает косинус, синус, тангенс a. a в радианах;

numpy.**cosh**(a), **numpy**.**sinh**(a), **numpy**.**tanh**(a) - возвращает гиперболические косинус, синус, тангенс a. a в радианах;

numpy.**arccos**(a), **numpy**.**arcsin**(a), **numpy**.**arctan**(a) - возвращает арккосинус, арксинус, арктангенс a в радианах. Для арккосинуса в диапазоне [0, **nump**.**pi**], для арксинуса [-**numpy**.pi/2, **numpy**.pi/2], для арктангенса [-**numpy**.pi/2, **numpy**.pi/2];

numpy.**arccosh**(a), **numpy**.**arcsinh**(a), **numpy**.**arctanh**(a) - возвращает гиперболические косинус, синус, тангенс a. a в радианах;

numpy.**exp**(a) - возвращает основание натурального логарифма (число e) в степени a;

numpy.**log**(a), **numpy**.**log10**(a), **numpy**.**log2**(a) - возвращает натуральный логарифм a, логарифм a по основанию 10, логарифм a по основанию 2;

numpy.**log1p(a**) - возвращает натуральный логарифм a + 1;

numpy.**sqrt(a)** - возвращает положительный квадратный корень a;

numpy.**square**(a) - возвращает квадрат a.

В выражения можно включать несколько массивов. В случае если атрибуты **ndarray**.shape этих массивов совпадают, результат очевиден - действия над массивами будут производиться поэлементно:

```
>>> a1 = np.array([ [1.0, 2.0], [3.0, 4.0] ])
>>> a2 = np.array([ [5.0, 6.0], [7.0, 8.0] ])
>>> a3 = np.array([ [9.0, 10.0], [11.0, 12.0] ])

>>> print (a1 + a2 + a3)
[[ 15.  18.]
 [ 21.  24.]]

>>> print (a1 * a2)
[[  5.  12.]
 [ 21.  32.]]
```

```
>>> print a3 / a1
[[ 9.          5.        ]
 [ 3.66666667  3.        ]]

>>> print (a3 - a1) * a2
[[ 40.  48.]
 [ 56.  64.]]
```

Если атрибуты **ndarray.shape** не совпадают - действия над массивами производятся в соответствии с концепцией транслирования (***broadcasting***).

Операция транслирования - расширение одного или обоих массивов операндов до массивов с равной размерностью.

Для начала несколько примеров:

```
>>> a2 = np.array([1, 2])
>>> print a2
[1 2]
>>> print a2.shape
(2,)

>>> a22 = np.array([ [1, 2], [3, 4] ])
>>> print a22
[[1 2]
 [3 4]]
>>> print a22.shape
(2, 2)

>>> a32 = np.array([ [1, 2], [3, 4], [5, 6] ])
>>> print a32
[[1 2]
 [3 4]
 [5 6]]
>>> print a32.shape
(3, 2)

>>> a222 = np.array([ [ [1, 2], [3, 4] ], [ [5, 6], [7,
8] ] ])
>>> print a222
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
>>> print a222.shape
(2, 2, 2)
```

```
>>> print a22 + a2
[[2 4]
 [4 6]]

>>> print a32 + a2
[[2 4]
 [4 6]
 [6 8]]

>>> print a222 + a2
[[[ 2  4]
  [ 4  6]]

 [[ 6  8]
  [ 8 10]]]

>>> print a32 + a22
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast
to a single shape

>>> print a222 + a22
[[[ 2  4]
  [ 6  8]]

 [[ 6  8]
  [10 12]]]

>>> print a222 + a32
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast
to a single shape
```

И так **правило**. Если длины осей массивов, начиная с замыкающей, попарно равны или в каждой из сравниваемых пар длин хотя бы одна равна единице, то к таким массивам может быть применена операция транслирования.

Длина замыкающей оси - длина массива вложенного наиболее глубоко.

```
>>> a = np.ones((7, 3, 4, 9, 8))
>>> b = np.ones((4, 9, 8))
>>> c = np.ones((4, 3, 8))
```

```
>>> print (a + b).shape
(7, 3, 4, 9, 8)

>>> print (a + c).shape
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast
to a single shape

>>> d = np.ones((9, 7, 1, 6, 1))
>>> e = np.ones((7, 2, 1, 4))
>>> print (d + e).shape
(9, 7, 2, 6, 4)
```

В случае если в массивах присутствуют оси единичной длины, расширяться могут оба массива.

```
>>> f = np.array([1, 2])
>>> print f
[1 2]
>>> print f.shape
(2,)

>>> g = np.array([ [3], [4], [5] ])
>>> print g
[[3]
 [4]
 [5]]
>>> print g.shape
(3, 1)

>>> h = f + g
>>> print h
[[4 5]
 [5 6]
 [6 7]]
>>> print h.shape
(3, 2)
```

Трансляция массивов происходит и при вызове некоторых функций.

Например, функция **numpy**.power(a1, a2), где a1, a2 массив или скаляр, возвращает a1 в степени a2:

```
>>> print np.power(np.array([-2.0, -1.0, 0.0, 2.0, 3.0,
4.0]), 4)
[  16.    1.    0.   16.   81.  256.]
```

```
>>> print np.power(np.array([-2.0, -1.0, 0.0, 2.0, 3.0,
4.0]), -4)
[ 0.0625      1.                  Inf  0.0625
0.01234568  0.00390625]
```

в случае если у переданных массивов не совпадает структура, проводит трансляцию.

```
>>> print np.power(np.array([3, 7]), np.array([ [1],
[2], [3] ]))
[[  3   7]
 [  9  49]
 [ 27 343]]
```

Многие унарные операции, такие как, например, вычисление суммы всех элементов массива, представлены также и в виде методов класса **ndarray**.

```
>>>
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.sum(a)
21
>>> a.sum()
21
>>> a.min()
1
>>> a.max()
6
```

По умолчанию, эти операции применяются к массиву, как если бы он был списком чисел, независимо от его формы. Однако, указав параметр **axis,** можно применить операцию для указанной оси массива:

```
>>>
>>> a.min(axis=0)
                    # Наименьшее число в каждом столбце
array([1, 2, 3])
>>> a.min(axis=1)
                    # Наименьшее число в каждой строке
array([1, 4])
```

Индексы, срезы, итерации

Одномерные массивы осуществляют операции индексирования, срезов и итераций очень схожим образом с обычными списками и другими последовательностями Python (разве что удалять с помощью срезов нельзя).

```
>>>
>>> a = np.arange(10) ** 3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[1]
1
```

```
>>> a[3:7]
array([ 27,  64, 125, 216])
>>> a[3:7] = 8
>>> a
array([  0,   1,   8,   8,   8,   8,   8, 343, 512, 729])
>>> a[::-1]
array([729, 512, 343,   8,   8,   8,   8,   8,   1,   0])
>>> del a[4:6]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: cannot delete array elements
>>> for i in a:
...     print(i ** (1/3))
...
0.0
1.0
2.0
2.0
2.0
2.0
2.0
7.0
8.0
9.0
```

У многомерных массивов на каждую ось приходится один индекс. Индексы передаются в виде последовательности чисел, разделенных запятыми (то есть, **кортежами**):

```
>>>
>>> b = np.array([[ 0,  1,  2,  3],
...               [10, 11, 12, 13],
...               [20, 21, 22, 23],
...               [30, 31, 32, 33],
...               [40, 41, 42, 43]])
...
>>> b[2,3]  # Вторая строка, третий столбец
23
>>> b[(2,3)]
23
>>> b[2][3]  # Можно и так
23
>>> b[:,2]  # Третий столбец
array([ 2, 12, 22, 32, 42])
>>> b[:2]  # Первые две строки
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
>>> b[1:3, : : ]  # Вторая и третья строки
```

```
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Когда индексов меньше, чем осей, отсутствующие индексы предполагаются дополненными с помощью срезов:

```
>>>
>>> b[-1]  # Последняя строка. Эквивалентно b[-1,:]

array([40, 41, 42, 43])
```

b[i] можно читать как b[i, <столько символов ':', сколько нужно>].
В NumPy это также может быть записано *с помощью точек*, как b[i, ...].

Например, если x имеет ранг 5 (то есть у него 5 осей), тогда
x[1, 2, ...] эквивалентно x[1, 2, :, :, :],
x[... , 3] то же самое, что x[:, :, :, :, 3] и
x[4, ... , 5, :] это x[4, :, :, 5, :].

```
>>>
>>> a = np.array(([[0, 1, 2], [10, 12, 13]], [[100, 101, 102],
                                               [110, 112, 113]]))
>>> a.shape
(2, 2, 3)
>>> a[1, ...]  # то же, что a[1, : , :] или a[1]

array([[100, 101, 102],
       [110, 112, 113]])

>>> c[... ,2]  # то же, что a[: , : ,2]
array([[  2,  13],
       [102, 113]])
```

Итерирование многомерных массивов начинается с первой оси:

```
>>>
>>> for row in a:
...     print(row)
...
[[ 0  1  2]
 [10 12 13]]
[[100 101 102]
 [110 112 113]]
```

Однако, если нужно перебрать поэлементно весь массив, как если бы он был одномерным, для этого можно использовать атрибут **flat**:

```
>>>
>>> for el in a.flat:
...     print(el)
```

```
...
0
1
2
10
12
13
100
101
102
110
112
113
```

Манипуляции с формой

Как уже говорилось, у массива есть форма (**shape**), определяемая числом элементов вдоль каждой оси:

```
>>>
>>> a
array([[[  0,   1,   2],
        [ 10,  12,  13]],

       [[100, 101, 102],
        [110, 112, 113]]])
>>> a.shape
(2, 2, 3)
```

Форма массива может быть изменена с помощью различных команд:

```
>>>
>>> a.ravel()  # Делает массив плоским
array([  0,   1,   2,  10,  12,  13, 100, 101, 102, 110, 112, 113])

>>> a.shape = (6, 2)  # Изменение формы
>>> a
array([[  0,   1],
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])

>>> a.transpose()  # Транспонирование
array([[  0,   2,  12, 100, 102, 112],
       [  1,  10,  13, 101, 110, 113]])
>>> a.reshape((3, 4))  # Изменение формы
array([[  0,   1,   2,  10],
```

```
       [ 12,  13, 100, 101],
       [102, 110, 112, 113]])
```

Порядок элементов в массиве в результате функции **ravel**() соответствует обычному "C-стилю", то есть, чем правее индекс, тем он "быстрее изменяется": за элементом a[0,0] следует a[0,1].

Если одна форма массива была изменена на другую, массив переформировывается также в "C-стиле".

Функции ravel() и reshape() также могут работать (при использовании дополнительного аргумента) в FORTRAN-стиле, в котором быстрее изменяется более левый индекс.

```
>>>
>>> a
array([[  0,   1],
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])
>>> a.reshape((3, 4), order='F')
array([[  0, 100,   1, 101],
       [  2, 102,  10, 110],
       [ 12, 112,  13, 113]])
```

Метод **reshape**() возвращает ее аргумент с измененной формой, в то время как метод **resize**() изменяет сам массив:

```
>>>
>>> a.resize((2, 6))
>>> a
array([[  0,   1,   2,  10,  12,  13],
       [100, 101, 102, 110, 112, 113]])
```

Если при операции такой перестройки один из аргументов задается как -1, то он автоматически рассчитывается в соответствии с остальными заданными:

```
>>>
>>> a.reshape((3, -1))
array([[  0,   1,   2,  10],
       [ 12,  13, 100, 101],
       [102, 110, 112, 113]])
```

Объединение массивов

Несколько массивов могут быть объединены вместе вдоль разных осей с помощью функций **hstack** и **vstack**.

hstack() объединяет массивы по первым осям,
vstack() — по последним:

```
>>>
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
>>> np.vstack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.hstack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Функция **column_stack()** объединяет одномерные массивы в качестве столбцов двумерного массива:

```
>>>
>>> np.column_stack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Аналогично для строк имеется функция
row_stack().

```
>>>
>>> np.row_stack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

Разбиение массива

Используя **hsplit()** вы можете разбить массив вдоль горизонтальной оси, указав либо число возвращаемых массивов одинаковой формы, либо номера столбцов, после которых массив разрезается "ножницами":

```
>>>
>>> a = np.arange(12).reshape((2, 6))
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> np.hsplit(a, 3)  # Разбить на 3 части
[array([[0, 1], [6, 7]]),
 array([[2, 3], [8, 9]]),
 array([[ 4,  5], [10, 11]])]
>>> np.hsplit(a, (3, 4))  # Разрезать a после третьего
                                    и четвёртого столбца
```

```
[array([[0, 1, 2], [6, 7, 8]]),
 array([[3], [9]]),
 array([[ 4,  5], [10, 11]])]
```

Функция
vsplit() разбивает массив вдоль вертикальной оси, а
array_split() позволяет указать оси, вдоль которых произойдет разбиение.

Копии и представления

При работе с массивами, их данные иногда необходимо копировать в другой массив, а иногда нет. Это часто является источником путаницы. *Возможно 3 случая*:

Вообще никаких копий

Простое присваивание не создает ни копии массива, ни копии его данных:

```
>>>
>>> a = np.arange(12)
>>> b = a  # Нового объекта создано не было
>>> b is a  # a и b это два имени для одного и того же
объекта ndarray
True
>>> b.shape = (3,4)  # изменит форму a
>>> a.shape
(3, 4)
```

Python передает изменяемые объекты как ссылки, поэтому вызовы функций также не создают копий.

Представление или поверхностная копия

Разные объекты массивов могут использовать одни и те же данные. Метод **view**() создает новый объект массива, являющийся представлением тех же данных.

```
>>>
>>> c = a.view()
>>> c is a
False
>>> c.base is a  # c это представление данных,
принадлежащих a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = (2,6)  # форма a не поменяется
```

```
>>> a.shape
(3, 4)
>>> c[0,4] = 1234  # данные a изменятся
>>> a
array([[  0, 1, 2, 3],
       [1234, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

Срез массива это представление:

```
>>>
>>> s = a[:,1:3]
>>> s[:] = 10
>>> a
array([[  0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

Глубокая копия

Метод **copy()** создаст настоящую копию массива и его данных:

```
>>>
>>> d = a.copy()  # создается новый объект массива с
                                        новыми данными
>>> d is a
False
>>> d.base is a  # d не имеет ничего общего с a
False
>>> d[0, 0] = 9999
>>> a
array([[  0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

NumPy случайные числа

Рассмотрим, как создавать массивы из случайных элементов и как работать со случайными элементами в NumPy.

Списки в массивы

Создавать списки, используя встроенный модуль **random**, а затем преобразовывать их в **numpy**.**array**:

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

```
>>> import random

>>> np.array([random.random() for i in range(10)])

array([ 0.99538667,  0.16860511,  0.78952804,
                     0.09676316,  0.86110208,
        0.89674666,  0.56401347,  0.63431468,
                         0.51110935,  0.64944844])
```

Но есть способ лучше.

Модуль numpy.random

Для создания массивов со случайными элементами служит модуль **numpy.random**.

```
>>>
>>> import numpy as np  # Импортировать numpy
                                  и писать np.random
>>> np.random
<module 'numpy.random' from <module 'numpy.random' from
'C:\\Program Files\\Python36\\lib\\site-
packages\\numpy\\random\\__init__.py'>
>>> import numpy.random as rand  # Можно и присвоить
                               отдельное имя. Вопрос вкуса
>>> rand
<module 'numpy.random' from 'C:\\Program Files\\Python36\\lib\\site-
packages\\numpy\\random\\__init__.py'>
```

Создание массивов со случайными элементами

Самый простой способ задать массив со случайными элементами - использовать функцию **sample** (или **random**, или **random_sample**, или **ranf** - это всё одна и та же функция).

```
>>>
>>> np.random.sample()
0.6336371838734877
>>> np.random.sample(3)
array([ 0.53478558,  0.1441317 ,  0.15711313])
>>> np.random.sample((2, 3))
array([[ 0.12915769,  0.09448946,  0.58778985],
       [ 0.45488207,  0.19335243,  0.22129977]])
```

Без аргументов возвращает просто число в промежутке [0, 1),

- с одним целым числом - одномерный массив,

- с кортежем - массив с размерами, указанными в кортеже (все числа - из промежутка [0, 1)).

С помощью функции **randint** или **random_integers** можно создать массив из целых чисел.

Аргументы: **low, high, size:** от какого, до какого числа (**randint** не включает в себя это число, а **random_integers** включает), и **size** - размеры массива.

```
>>>
>>> np.random.randint(0, 3, 10)
array([0, 2, 0, 1, 1, 0, 2, 2, 2, 0])
>>> np.random.random_integers(0, 3, 10)
array([2, 2, 3, 3, 1, 1, 0, 2, 3, 2])
>>> np.random.randint(0, 3, (2, 10))
array([[0, 1, 2, 0, 0, 0, 1, 1, 1, 2],
       [0, 0, 2, 2, 2, 0, 1, 2, 2, 1]])
```

Также можно генерировать числа согласно различным распределениям (Гаусса, Парето и другие). Чаще всего нужно равномерное распределение, которое можно получить с помощь функции **uniform**.

```
>>>
>>> np.random.uniform(2, 8, (2, 10))
array([[ 3.1517914 ,  3.10313483,  2.84007134,
                      3.21556436,  4.64531786,
         2.99232714,  7.03064897,  4.38691765,
                      5.27488548,  2.63472454],
       [ 6.39470358,  5.63084131,  4.69996748,
                      7.07260546,  7.44340813,
         4.10722203,  7.52956646,  4.8596943 ,
                      3.97923973,  5.64505363]])
```

Выбор и перемешивание

Перемешать NumPy массив можно с помощью функции **shuffle**:

```
>>>
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.random.shuffle(a)
>>> a
array([2, 8, 7, 3, 5, 0, 4, 9, 1, 6])
```

Также можно перемешать массив с помощью функции **permutation** (она, в отличие от **shuffle**, возвращает перемешанный массив). Также она, вызванная с одним аргументом (целым числом), возвращает перемешанную последовательность от 0 до N.

```
>>>
>>> np.random.permutation(10)
array([1, 2, 3, 8, 7, 9, 4, 6, 5, 0])
```

Сделать случайную выборку из массива можно с помощью функции **choice**.

numpy.random.choice(a, **size**=None, **replace**=**True**, **p**=None)

- a : одномерный массив или число. Если массив, будет производиться выборка из него. Если число, то выборка будет производиться из **np.arange**(a).
- size : размерности массива. Если None, возвращается одно значение.
- replace : если **True**, то одно значение может выбираться более одного раза.
- p : вероятности. Это означает, что элементы можно выбирать с неравными вероятностями. Если не заданы, используется равномерное распределение.

```
>>>
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.random.choice(a, 10, p=[0.5, 0.25, 0.25, 0, 0,
                               0, 0, 0, 0, 0])
array([0, 0, 0, 0, 1, 2, 0, 0, 1, 1])
```

Инициализация генератора случайных чисел

seed(число) - инициализация генератора.

```
>>>
>>> np.random.seed(1000)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,
                    0.4821914 ,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,
                             0.2331322 ,  0.84174072])
>>> np.random.seed(1000)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,
                              0.4821914 ,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,
                             0.2331322 ,  0.84174072])
```

get_state и **set_state** - возвращают и устанавливают состояние генератора.

```
>>>
>>> np.random.seed(1000)
>>> state = np.random.get_state()
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,
                                0.4821914 ,  0.87247454,
```

```
        0.21233268,  0.04070962,  0.39719446,
                          0.2331322 ,  0.84174072])
>>> np.random.set_state(state)
>>> np.random.random(10)
array([ 0.65358959,  0.11500694,  0.95028286,
                         0.4821914 ,  0.87247454,
        0.21233268,  0.04070962,  0.39719446,
                       0.2331322 ,  0.84174072])
```

Некоторые полезные функции

numpy.min(a, axis = None, **out** = None),
numpy.max(a, axis = None, **out** = None)

возвращает минимальное, максимальное значение элементов массива соответственно:

```
>>> import numpy as np
>>> print np.min(np.array([ [1.0, -0.5, 3.0], [4.0,
3.0, -0.5] ]))
-0.5
```

axis - опциональный аргумент, индекс оси (измерения) массива по которому проводится поиск минимального, максимального значения. Под индексом оси понимается индекс в кортеже **ndarray.shape.**

```
>>> a = np.array([ [ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0]
], [ [7.0, 8.0, 9.0], [11, 12, 13] ] ])
>>> print a
[[[  1.   2.   3.]
  [  4.   5.   6.]]

 [[  7.   8.   9.]
  [ 11.  12.  13.]]]
>>> print a.shape
(2, 2, 3)

>>> print np.min(a, axis = 0)
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

>>> print np.min(a, axis = 1)
[[ 1.  2.  3.]
 [ 7.  8.  9.]]

>>> print np.min(a, axis = 2)
[[  1.   4.]
```

```
 [  7.  11.]]
```

out - опциональный аргумент, массив, в который будет помещен результат. Структура массива **out** должна соответствовать структуре массива результата. Если **out** = None (по умолчанию) будет создан новый массив.

numpy.argmin(a, axis = None),
numpy.argmax(a, axis = None) -
возвращает индекс минимального, максимального значения элементов массива соответственно:

```
>>> print np.argmin(np.array([ [1.0, -0.5, 3.0], [4.0,
3.0, -0.5] ]))
1

>>> print np.argmin(a, axis = 0)
[[0 0 0]
 [0 0 0]]

>>> print np.argmin(a, axis = 1)
[[0 0 0]
 [0 0 0]]

>>> print np.argmin(a, axis = 2)
[[0 0]
 [0 0]]
```

numpy.sum(a, axis = None, **dtype** = None, **out** = None),
numpy.prod(a, axis = None, **dtype** = None, **out** = None)

возвращает сумму, произведение элементов массива соответственно:

```
>>> print np.sum(np.array([ [1.0, -0.5, 3.0], [4.0,
3.0, -0.5] ]))
10.0
>>> print np.sum(a, axis = 0)
[[  8.  10.  12.]
 [ 15.  17.  19.]]
>>> print np.sum(a, axis = 1)
[[  5.   7.   9.]
 [ 18.  20.  22.]]
>>> print np.sum(a, axis = 2)
[[  6.  15.]
 [ 24.  36.]]
```

NumPy linalg некоторые операции линейной алгебры

Рассмотрим модуль **numpy**.**linalg**, позволяющий делать многие операции из линейной алгебры.

Возведение в степень

linalg.matrix_power(M, n) - возводит матрицу в степень n.

Разложения

linalg.cholesky(a) - разложение Холецкого.
linalg.qr(a[, mode]) - QR разложение.
linalg.svd(a[, full_matrices, compute_uv]) - сингулярное разложение.

Некоторые характеристики матриц

linalg.eig(a) - собственные значения и собственные векторы.
linalg.norm(x[, ord, axis]) - норма вектора или оператора.
linalg.cond(x[, p]) - число обусловленности.
linalg.det(a) - определитель.
linalg.slogdet(a) - знак и логарифм определителя (для избежания переполнения, если сам определитель очень маленький).

Системы уравнений

linalg.solve(a, b) - решает систему линейных уравнений Ax = b.
linalg.tensorsolve(a, b[, axes]) - решает тензорную систему линейных уравнений Ax = b.
linalg.lstsq(a, b[, rcond]) - метод наименьших квадратов.
linalg.inv(a) - обратная матрица.

Замечания:

- **linalg.LinAlgError** - исключение, вызываемое данными функциями в случае неудачи (например, при попытке взять обратную матрицу от вырожденной).
- Подробная документация: https://docs.scipy.org/doc/**numpy**/reference/routines.linalg.html
- Массивы большей размерности в большинстве функций **linalg** интерпретируются как набор из нескольких массивов нужной размерности. Таким образом, можно одним вызовом функции проделывать операции над несколькими объектами.

```
>>>
>>> a = np.arange(18).reshape((2,3,3))
>>> a
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
```

```
        [12, 13, 14],
        [15, 16, 17]]])

>>> np.linalg.det(a)
array([ 0.,  0.])
```

Исключения numpy.linalg.LinAlgError

numpy.linalg.LinAlgError
exception numpy.linalg.LinAlgError

Объект linalg.LinAlgError генерирует исключения Python, вызванные функциями модуля linalg.

Данный объект образован классом исключений общего назначения Python и является производным от него. Данный класс вызывается, только в том случае если дальнейшая работа какой-либо функции модуля *linalg* невозможна.

Примеры

```
>>> import numpy as np
>>> from numpy import linalg as LA
>>>
>>> a = [[0, 0], [0, 0]]
>>>
>>> try:
...     LA.inv(a)
... except LA.LinAlgError:
...     print('Матрица "a" является либо вырожденной либо прямоугольной')
...
```

Матрица "a" является либо вырожденной либо прямоугольной

Примеры

Произведение одномерных массивов представляет собой скалярное произведение векторов:

```
>>> a = np.array([1,2])
>>> b = np.array([3,4])
>>>
>>> np.dot(a,b)
11
```

Произведение двумерных массивов по правилам линейной алгебры так же возможно:

```
>>> a = np.arange(2,6).reshape(2,2)
>>> a
array([[2, 3],
       [4, 5]])
>>>
>>> b = np.arange(6,10).reshape(2,2)
>>> b
array([[6, 7],
       [8, 9]])
>>>
>>> np.dot(a,b)
```

```
array([[36, 41],
       [64, 73]])
```

При этом размеры матриц (массивов) должны быть либо равны, а сами матрицы квадратными, либо быть согласованными, т.е. если размеры матрицы `A` равны [m,k], то размеры матрицы `B` должны быть равны [k,n]:

```
>>> a = np.arange(2,8).reshape(2,3)
>>> a
array([[2, 3, 4],
       [5, 6, 7]])
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4, 5],
       [6, 7],
       [8, 9]])
>>>
>>> np.dot(a,b)
array([[ 58,  67],
       [112, 130]])
```

Так же по правилам умножения матриц, мы можем умножить матрицу на вектор (одномерный массив). При этом в таком умножении вектор столбец должен находиться справа, а вектор строка слева:

```
>>> a = np.array([1,2,3])
>>> a
array([1, 2, 3])
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4, 5],
       [6, 7],
       [8, 9]])
>>>
>>> np.dot(a,b)
array([40, 46])
>>>
>>> a = np.arange(1,3).reshape(2,1)
>>> a
array([[1],
       [2]])
>>>
>>> b = np.arange(4,10).reshape(3,2)
>>> b
array([[4, 5],
       [6, 7],
       [8, 9]])
>>>
>>> np.dot(b,a)
array([[14],
       [20],
       [26]])
```

Квадратные матрицы можно возводить в степень `n` т.е. умнажать сами на себя `n` раз:

```
>>> a = np.arange(1,5).reshape(2,2)
>>> a
array([[1, 2],
       [3, 4]])
>>>
>>> np.dot(a,a)     #  Равносильно a**2
array([[ 7, 10],
       [15, 22]])
```

```
>>>
>>> np.linalg.matrix_power(a,2)
array([[ 7, 10],
       [15, 22]])
>>>
>>> np.linalg.matrix_power(a,5)
array([[1069, 1558],
       [2337, 3406]])
>>>
>>> np.linalg.matrix_power(a,0)
array([[1, 0],
       [0, 1]])
```

Довольно часто приходится вычислять ранг матриц:

```
>>> a = np.arange(1,10).reshape(3,3)
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>>
>>> np.linalg.matrix_rank(a)
2
>>>
>>> b = np.arange(1,24,2).reshape(3,4)
>>> b
array([[ 1,  3,  5,  7],
       [ 9, 11, 13, 15],
       [17, 19, 21, 23]])
>>>
>>> np.linalg.matrix_rank(b)
2
```

Еще чаще приходится вычислять определитель матриц ,хотя результат вас может немного удивить:

```
>>> a = np.array([[1,3],[4,3]])
>>> a
array([[1, 3],
       [4, 3]])
>>>
>>> np.linalg.det(a)
-8.9999999999999982
>>>
>>> 1*3 - 3*4    #  Результат должен быть целым числом
-9
```

В данном случае, из-за двоичной арифметики, результат не целое число и округлять до ближайшего целого прийдется вручную. Это связано с тем, что алгоритм вычисления определителя использует LU-разложение - это намного быстрее чем обычный алгоритм, но за скорость все же приходится немного заплатить ручным округлением (конечно, если таковое требуется):

```
>>> np.linalg.det(a)
-8.9999999999999982
>>> round(np.linalg.det(a))
-9.0
>>>
>>> b = np.arange(1,48,3).reshape(4,4)
>>> np.linalg.det(b)
-1.0223637331664275e-27
>>> round(np.linalg.det(b))
-0.0
```

Транспонирование матриц:

```
>>> a
matrix([[1, 3],
```

```
       [4, 3]])
>>>
>>> a.T
matrix([[1, 4],
        [3, 3]])
>>>
>>> b
array([[ 1,  4,  7, 10],
       [13, 16, 19, 22],
       [25, 28, 31, 34],
       [37, 40, 43, 46]])
>>>
>>> b.T
array([[ 1, 13, 25, 37],
       [ 4, 16, 28, 40],
       [ 7, 19, 31, 43],
       [10, 22, 34, 46]])
```

Вычисление обратных матриц:

```
>>> a = np.array([[1,3],[4,3]])
>>> a
array([[1, 3],
       [4, 3]])
>>>
>>> b = np.linalg.inv(a)
>>> b
array([[-0.33333333,  0.33333333],
       [ 0.44444444, -0.11111111]])
>>>
>>> np.dot(a,b)
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Решение систем линейных уравнений:

```
... #  система из двух линейных уравнений:
...
... #  1*x1 + 5*x2 = 11
... #  2*x1 + 3*x2 = 8
...
>>> a = np.array([[1,5],[2,3]])
>>> b = np.array([11,8])
>>>
>>> x = np.linalg.solve(a,b)
>>> x
array([ 1.,  2.])
>>>
>>> np.dot(a,x)
array([ 11.,   8.])
```

NumPy Преобразование фурье

По-простому, **преобразование Фурье** — разложение некоторого сигнала на гармонические (синусы или косинусы) колебания (спектр) (по материалам http://old.pynsk.ru/posts/2015/Nov/09/matematika-v-python-preobrazovanie-fure/).

Преобразование Фурье является **интегральным преобразованием**. Если речь идёт о дискретном сигнале, то интеграл обращается в сумму (и становится дискретным преобразованием Фурье, ДПФ). Чтобы посчитать такую сумму `N` элементов, надо совершить N^2 операций с комплексными

числами. Но достаточно давно (Cooley, Tukey, 1965 г, а ещё сам Гаусс в 1805 г.) придумал алгоритм, вычисляющий ДПФ N элементов в N*log(N) операций (большая часть из которых над действительными числами), что существенно экономит вычислительное время. Такой алгоритм часто называют «быстрое преобразование Фурье, БПФ (Fast Fourier Trans**for**m, FFT)». Именно так реализовано ДПФ в современных компьютерных программах.

В библиотеке NumPy содержится все, что нужно для дискретного преобразования Фурье. Всё это лежит в модуле **`numpy.fft`**.

Вот эти функции.

Общий случай: сигнал может быть как из действительных чисел, так и из комплексных.

`fft(a, n=None, axis=-1)` — прямое одномерное ДПФ.

`ifft(a, n=None, axis=-1)` — обратное одномерное ДПФ.

Здесь:

`a` — "сигнал", входной массив (массив `numpy array` или даже питоновский `список` или `кортеж`, если в нём только числа).

Массив может быть и многомерным, тогда будет вычисляться много `ОДНОМЕРНЫХ ПФ` по строкам (по умолчанию) или столбцам, в зависимости от параметра `axis`.

Например, `a` — двухмерный, `a[n][m]`:

при `axis=1` или `-1` будет такое (под `fourier(a...)` понимается результат действия ПФ на `a`):

```
[fourier(a[0][j]), fourier(a[1][j]), ...
                                    foirier(a[n][j])]
```

При axis=0 такое:

```
[fourier(a[i][0]), fourier(a[i][1]), ...
                                      foirier(a[i][m])]
```

`n` — сколько элементов массива брать. Если меньше длины массива, то обрезать, если больше, то дополнить нулями, по умолчанию `len(a)`.

`fft2(a, s=None, axes=(-2, -1))` — прямое двухмерное ПФ.

`ifft2(a, s=None, axes=(-2, -1))` — обратное двухмерное ПФ.

`fftn(a, s=None, axes=None)` — прямое многомерное ПФ.

`ifftn(a, s=None, axes=None)` — обратное многомерное ПФ.

Всё так же, как и для одномерных, но **`s`** и **`axes`** теперь кортежи для каждой размерности. О размерности **`fftn`**, **`ifftn`** догадаются по размерности входных массивов или **`s`** и **`axes`**.

Когда сигнал действительный (`real`) (пожалуй, самый распространённый случай):

`rfft(a, n=None, axis=-1`) — прямое одномерное ДПФ (для действительных чисел).

irfft(a, n=None, axis=-1) — обратное одномерное ДПФ.

rfft2(a, s=None, axes=(-2, -1)) — прямое двухмерное ДПФ.

irfft2(a, s=None, axes=(-2, -1)) — обратное двухмерное ДПФ.

rfftn(a, s=None, axes=None) — прямое многомерное ДПФ.

irfftn(a, s=None, axes=None) — обратное многомерное ДПФ.

Всё так же, как и для общего случая.

Все эти функции возвращают массив соответствующей размерности, в котором записан результат ДПФ.

Разница такая. Если длина входного массива (или какой-либо его размерности) **N,** то в общем случае (с комплексным сигналом) длина выходного массива **N.**

Там содержатся сначала положительные частоты от нуля до частоты Котельникова (Найквиста), потом отрицательные в порядке возрастания.

В случае действительного сигнала отрицательные частоты полностью симметричны положительным, и тогда нет нужды их записывать: длина выходного массива **N/2+1,** частоты от нуля до частоты Котельникова.

Если спектр сигнала действительный (а сигнал обладает "эрмитовой симметрией": его половины симметричны относительно центра по модулю и являются комплексно сопряжёнными друг другу), то можно применить такие функции:

hfft(a, n=None, axis=-1) — прямое одномерное ДПФ.

ihfft(a, n=None, axis=-1) — обратное одномерное ДПФ.

Длина входного массива **N**, а выходного **2*N+1.**

Кроме того, есть вспомогательные функции (будет понятнее из примера):

fftfreq(n, d=1.0) — возвращает частоты для выходных массивов функций fft*.

rfftfreq(n, d=1.0) — возвращает частоты для выходных массивов функций rfft*.

Здест **-** **n** — длина входного массива, **d** — период дискретизации (обратная частота дискретизации).

fftshift(x, axes=None) — преобразует массив (с результатом ДПФ, от функций fft*) так, чтобы нулевая частота была в центре.

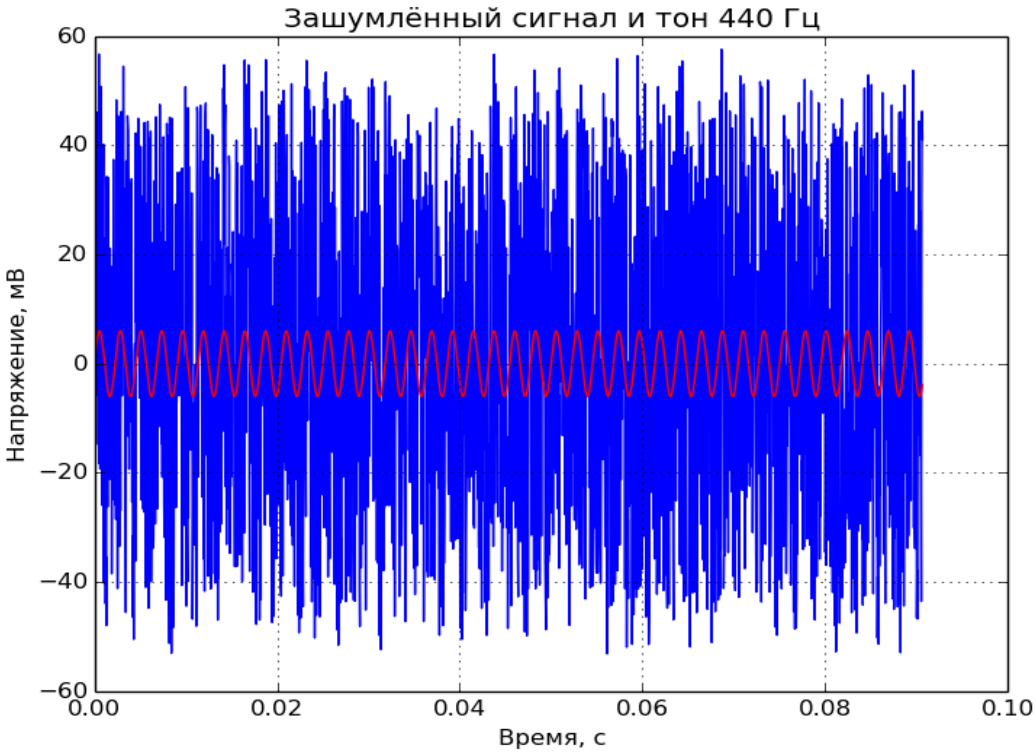
ifftshift(x, axes=None) — делает обратную операцию.

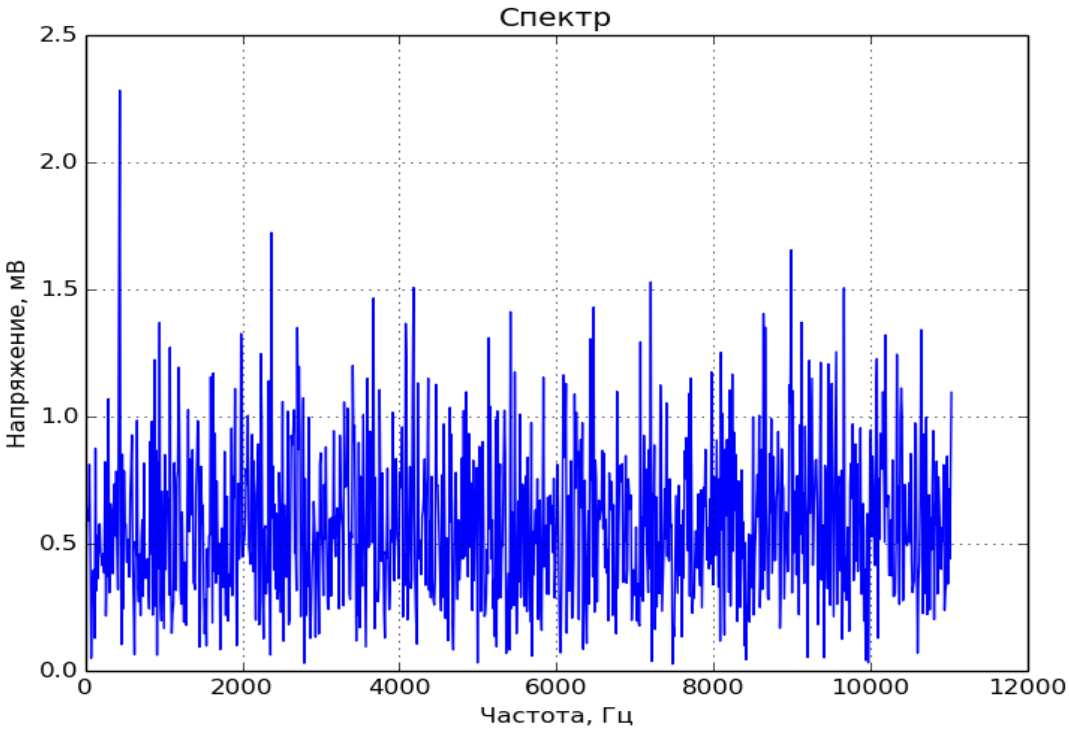
Приведём такой пример. Допустим, записали мы микрофоном какой-то шум, и надо определить, есть ли там какой-нибудь тон.

```
from numpy import array, arange, abs as np_abs
from numpy.fft import rfft, rfftfreq
from numpy.random import uniform
from math import sin, pi
```

```
import matplotlib.pyplot as plt
# а можно импортировать numpy и писать: numpy.fft.rfft
FD = 22050 # частота дискретизации, отсчётов в секунду
# а это значит, что в дискретном сигнале представлены
# частоты от нуля до 11025 Гц (это и есть теорема
# Котельникова)
N = 2000  # длина входного массива, 0.091 секунд при
          #такой частоте дискретизации
# сгенерируем сигнал с частотой 440 Гц длиной N
pure_sig =
   array([6.*sin(2.*pi*440.0*t/FD) for t in range(N)])
# сгенерируем шум, тоже длиной N (это важно!)
noise = uniform(-50.,50., N)
# суммируем их и добавим постоянную составляющую 2 мВ
#(допустим, не очень хороший микрофон попался.
# Или звуковая карта или АЦП)
sig = pure_sig + noise + 2.0
# в numpy так перегружена функция сложения
# вычисляем преобразование Фурье.
#Сигнал действительный, поэтому надо
# использовать rfft, это быстрее, чем fft
spectrum = rfft(sig)

# нарисуем всё это, используя matplotlib
# Сначала сигнал зашумлённый и тон отдельно
plt.plot(arange(N)/float(FD), sig) # по оси времени
#                                              секунды!
plt.plot(arange(N)/float(FD), pure_sig, 'r') # чистый
#                         сигнал будет нарисован красным
plt.xlabel(u'Время, с')
plt.ylabel(u'Напряжение, мВ')
plt.title(u'Зашумлённый сигнал и тон 440 Гц')
plt.grid(True)
plt.show()
# когда закроется этот график, откроется следующий
# Потом спектр
plt.plot(rfftfreq(N, 1./FD), np_abs(spectrum)/N)
# rfftfreq сделает всю работу по преобразованию
#                    номеров элементов массива в герцы
# нас интересует только спектр амплитуд,
# поэтому используем abs из numpy
#   (действует на массивы поэлементно)

# делим на число элементов, чтобы амплитуды были в
# милливольтах, а не в суммах Фурье.
```

```
# Проверить просто — постоянные составляющие должны
# совпадать в сгенерированном сигнале и в спектре

plt.xlabel(u'Частота, Гц')
plt.ylabel(u'Напряжение, мВ')
plt.title(u'Спектр')
plt.grid(True)
plt.show()
```




Спектр
Напряжение, мВ
Частота, Гц
0.0
0.5
1.0
1.5
2.0
2.5
0
2000
4000
6000
8000
10000
12000