# Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» ММФ БГУ

## Тема 13. Расширение numpy для численных вычислений

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

май, 2024

## 13.1 Расширение numpy

In [2]:
```
import numpy as np
```

`numpy` является сокращением от **NUMerical PYthon**. Расширение `numpy` предназначено для эффективной обработки числовых данных.

Первая версия расширения `numpy` создана в 2006 году, создатель Трэвис Олифант (*Travis Oliphant*). Текущая версия 1.26.4.

In [2]:
```
np.__version__
```

Out[2]:
```
'1.24.3'
```

Расширение `numpy` не входит в стандартную библиотеку Python.

**Преимущества `numpy`**

Расширение `numpy` реализовано как предварительно скомпилированный код на языке `C`, при импортировании `numpy` не транслируется в байт-код. Все функции расширения `numpy` выполняются в интерпретаторе со скоростью языка `C`, обеспечивая высокое быстродействие при выполнении.

Расширение `numpy` является основой для работы с другими пакетами и модулями.

- В частности, пакет `matplotlib` использует массивы `numpy` для хранения и работы с данными при построении изображений.
- Расширение `scipy` использует массивы `numpy` для вычислений и расширяет функциональность `numpy` для решения прикладных задач математики (оптимизация, математическая статистика и др.).
- Пакет `pandas` для анализа данных и работы с данными активно использует концепции расширения `numpy` .

## 13.2 Свойства массива и действия с массивом

Тип данных **массив** ( `ndarray` ) является базовой структурой данных расширения `numpy` . Основные вычисления в `numpy` выполняются с использованием массивов.

Экземпляр класса `numpy.ndarray` ($n$-**d**imensional **array**) является *многомерным* массивом данных **одного типа и одинакового размера**. Тип данных соответствует формату языка `C` , например `int32` , `float64` . **Размер всего массива является фиксированным, определяется при создании массива.**

**Преимущество `numpy`** : элементы массива хранятся в *последовательных* ячейках памяти и *эффективно* экономят используемую оперативную память.

Атрибуты экземпляра класса `numpy.ndarray` хранят свойства созданного массива.

- `shape` хранит структуру массива в виде кортежа с количеством элементов массива по каждому измерению ( `axis` ). Например, для представления матрицы из $n$ строк и $m$ столбцов используется массив со структурой `(n, m)`
- `ndim` хранит количество измерений массива
- `size` хранит количество элементов массива
- `dtype` хранит тип данных элементов массива
- `data` хранит адрес в памяти первого элемента массива
- `itemsize` хранит размер элемента массива в байтах

In [3]:
```
x = np.array([[1],[2],[3]])
type(x), x.shape, x.ndim, x.size, x.dtype, x.data, x.itemsize
```

Out[3]:
```
(numpy.ndarray,
 (3, 1),
 2,
 3,
 dtype('int32'),
 <memory at 0x000002127BDA0110>,
 4)
```

Для нахождения объема памяти в байтах, занимаемого массивом, можно значение атрибута `size` умножить на значение атрибута `itemsize`

In [4]:
```
x.size*x.itemsize
```

Out[4]:
```
12
```

In [5]:
```
x = np.array([])
type(x), x.shape, x.ndim, x.size, x.dtype, x.data, x.itemsize
```

Out[5]:
```
(numpy.ndarray,
 (0,),
 1,
 0,
 dtype('float64'),
 <memory at 0x000002127B8BD6C0>,
 8)
```

In [6]:
```
x = np.array([1])
type(x), x.shape, x.ndim, x.size, x.dtype, x.data, x.itemsize
```

Out[6]:
```
(numpy.ndarray, (1,), 1, 1, dtype('int32'), <memory at 0x000002127B8BD3C0>, 4)
```

Встроенная функция `len` для массива возвращает количество элементов по первому измерению

In [7]:
```
x = np.array([[[1,2]],[[3,4]],[[5,6]]])
print(x)
len(x)
```

Out[7]:
```
[[[1 2]]

 [[3 4]]

 [[5 6]]]
3
```

Массив является итерируемым объектом по элементам первого измерения

In [7]:
```
for el in x:
    print(el)
```

```
[[1 2]]
[[3 4]]
[[5 6]]
```

Атрибут `flat` для массива возвращает итератор по ВСЕМ элементам массива

In [8]:
```
for el in x.flat:
    print(el)
```

```
1
2
3
4
5
6
```

## Создание массива

Для создания массивов небольших размеров можно использовать функцию-конструктор `array`, аргументом которой является список, кортеж или итерируемый объект с элементами *одного типа*

In [9]:
```
np.array([1.,2,3]), np.array((1,2,3)), np.array(range(5))
```

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

При вызове функции-конструктора `array` можно явно указать тип для элементов создаваемого массива с помощью ключевого аргумента `dtype`

In [10]:
```
ar = np.array(range(5), dtype=complex)
ar, ar.dtype
```

Out[10]:
```
(array([0.+0.j, 1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j]), dtype('complex128'))
```

Типы элементов массива соответствуют формату языка `C`. Все типы хранятся в расширении `numpy`

In [3]:
```
?np.int32
```

```
Init signature: np.int32(self, /, *args, **kwargs)
Docstring:
Signed integer type, compatible with Python `int` and C ``long``.

:Character code: ``'l'``
:Canonical name: `numpy.int_`
:Alias on this platform (win32 AMD64): `numpy.int32`: 32-bit signed integer (``-2_1
47_483_648`` to ``2_147_483_647``).
File:           c:\users\olga\anaconda3\lib\site-packages\numpy\__init__.py
Type:           type
Subclasses:
```

In [4]:
```
?np.complex128
```

```
Init signature: np.complex128(real=0, imag=0)
Docstring:
Complex number type composed of two double-precision floating-point
numbers, compatible with Python `complex`.

:Character code: ``'D'``
:Canonical name: `numpy.cdouble`
:Alias: `numpy.cfloat`
:Alias: `numpy.complex_`
:Alias on this platform (win32 AMD64): `numpy.complex128`: Complex number type comp
osed of 2 64-bit-precision floating-point numbers.
File:           c:\users\olga\anaconda3\lib\site-packages\numpy\__init__.py
Type:           type
Subclasses:
```

In [5]:
```
?np.float64
```

```
Init signature: np.float64(x=0, /)
Docstring:
Double-precision floating-point number type, compatible with Python `float`
and C ``double``.

:Character code: ``'d'``
:Canonical name: `numpy.double`
:Alias: `numpy.float_`
:Alias on this platform (win32 AMD64): `numpy.float64`: 64-bit precision floating-p
oint number type: sign bit, 11 bits exponent, 52 bits mantissa.
File:           c:\users\olga\anaconda3\lib\site-packages\numpy\__init__.py
Type:           type
Subclasses:
```

Если при создании массива известна только его структура, но не известны значения элементов, то можно использовать функцию `empty`, аргументом которой является кортеж, задающий структуру, или число для создания одномерного массива. Значениями созданного массива будут случайные числа, содержащиеся в памяти, выделенной для этого массива

In [3]:
```
emp = np.empty((2,3,4)); emp1 = np.empty(4)
print(emp); print(emp1)
```

```
[[[6.23042070e-307 4.67296746e-307 1.69121096e-306 1.02360731e-306]
  [1.33511018e-306 1.33511969e-306 6.23037996e-307 6.23053954e-307]
  [9.34609790e-307 8.45593934e-307 9.34600963e-307 1.86921143e-306]]

 [[6.23061763e-307 8.90104239e-307 6.89804132e-307 1.33512376e-306]
  [6.89806849e-307 9.34609790e-307 1.69121096e-306 1.05700515e-307]
  [2.04712906e-306 7.56589622e-307 1.11258277e-307 8.90111708e-307]]]
[2.12199579e-314 1.31139340e-311 6.40309077e-321 1.31139340e-311]
```

Функции `zeros` и `ones` позволяют создавать массивы, состоящие из нулей или единиц, соответственно. Аргументом функций `zeros` и `ones` является кортеж, задающий структуру массива, или число для задания одномерного массива

In [15]:
```
np.zeros(400000), np.ones((2,2), dtype=int),
```

Out[15]:
```
(array([0., 0., 0., ..., 0., 0., 0.]),
 array([[1, 1],
        [1, 1]]))
```

Считается, что функции `zeros` и `ones` чаще всего используются для создания новых массивов.

Для создания массива по структуре уже существующего массива можно использовать функции `empty_like`, `zeros_like`, `ones_like`. Элементами созданных массивов будут случайные числа, нули или единицы, соответственно.

In [16]:
```
np.zeros_like(emp, dtype=int)
```

Out[16]:
```
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]])
```

Функция `arange([start,] stop[, step], dtype=None)` создает массив, значения элементов которого изменяются последовательно от `start` с шагом `step` до максимального значения, *меньшего* `stop`. По умолчанию `start=0`, `step=1`

In [17]:
```
np.arange(1,3,0.5)
```

Out[17]:
```
array([1. , 1.5, 2. , 2.5])
```

Функция `linspace(start, stop, num=50, endpoint=True, retstep=False)` создает массив из `num` элементов, значения которых изменяются последовательно от значения `start` до значения `stop` *включительно*, если `endpoint=True`, с одинаковым шагом. По умолчанию `num=50`, `endpoint=True`.

In [18]:
```
np.linspace(1, 3, 5)
```

Out[18]:
```
array([1. , 1.5, 2. , 2.5, 3. ])
```

При вызове функции `linspace` со значением ключевого аргумента `retstep` равным `True` возвращаемым объектом является кортеж из созданного массива и значения шага

In [19]:
```
x, dx = np.linspace(1, 3, 5, retstep=True)
x, dx
```

Out[19]:
```
(array([1. , 1.5, 2. , 2.5, 3. ]), 0.5)
```

Функция `fromfunction(func, shape)` позволяет создавать массив `x` со структурой `shape`, элементы которого `x[i,j,k,...]` являются значениями функции `func`, вычисленными по значениям индексов, соответствующих элементов: `x[i,j,k,...] = func(i,j,k,...)`

In [20]:
```
np.fromfunction(lambda i, j: i+j, (2,4))
```

Out[20]:
```
array([[0., 1., 2., 3.],
       [1., 2., 3., 4.]])
```

## Доступ к элементам

Для индексации многомерного массива используется кортеж целых чисел в квадратных скобках: `[i,j]`. Размер кортежа определяется количеством измерений массива (атрибут `ndim` массива)

In [43]:
```
matrix = np.array([[1,2],[3,4]])
print(matrix.ndim)
print(matrix[0,0], matrix[0,1], matrix[1,0], matrix[1,1])
```

```
2
1 2 3 4
```

Возможна индексация с отрицательными индексами

In [23]:
```
matrix[-1,-1]
```

Out[23]:
```
4
```

Поддерживается индексация с нарезанием для каждого измерения массива

In [45]:
```
matrix = np.array([range(5)]*3)
matrix[:, :], matrix[1:3, 2], matrix[1:,1:], matrix[:3, :1], matrix[::2,:]
```

Out[45]:
```
(array([[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]),
 array([2, 2]),
 array([[1, 2, 3, 4],
        [1, 2, 3, 4]]),
 array([[0],
        [0],
        [0]]),
 array([[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]))
```

Результатом индексации с нарезанием является не копия элементов исходного массива, а массив, содержащий ссылки на исходный массив (**представление массива**). Это означает, что при индексации с нарезанием данные НЕ копируются и любые изменения в представлении массива будут отражены в исходном массиве!

In [25]:
```
matrix1 = matrix[:, :]
matrix1[0,0] = 100
matrix
```

Out[25]:
```
array([[100,   1,   2,   3,   4],
       [  0,   1,   2,   3,   4],
       [  0,   1,   2,   3,   4]])
```

Атрибут `base` для объекта массива возвращает `None` . Атрибут `base` для представления массива возвращает исходный массив

In [26]:
```
matrix.base, matrix1.base
```

Out[26]:
```
(None,
 array([[100,   1,   2,   3,   4],
        [  0,   1,   2,   3,   4],
        [  0,   1,   2,   3,   4]]))
```

Для того, чтобы изменение результирующего массива не влияло на исходный массив, необходимо явно создавать копию результата с помощью функции `copy` или метода `copy`

In [50]:
```
matrix1 = np.copy(matrix[:, :])
# или
matrix1 = matrix[:,:].copy()
matrix1[0,0] = 200
matrix, matrix1
```

Out[50]:
```
(array([[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]),
 array([[200,   1,   2,   3,   4],
        [  0,   1,   2,   3,   4],
        [  0,   1,   2,   3,   4]]))
```

In [51]:
```
matrix.base, matrix1.base
```

Out[51]:
```
(None, None)
```

Применение функций, которые возвращают представление массива, является более эффективным, чем применение функций, которые возвращают новый объект массива.

Если при индексации количество индексов меньше количества измерений, то неуказанные индексы заменяются на `:`

In [29]:
```
matrix[0], matrix[0,:], matrix[0,...]
```

Out[29]:
```
(array([100,   1,   2,   3,   4]),
 array([100,   1,   2,   3,   4]),
 array([100,   1,   2,   3,   4]))
```

Для индексации массивов можно использовать массивы логических значений. Индекс со значением `True` означает, что соответствующий элемент индексируемого массива необходимо вернуть

In [30]:
```
x = np.array([-1, -2, 0, 1, 2])
x>0, x[x>0]
```

Out[30]:
```
(array([False, False, False,  True,  True]), array([1, 2]))
```

In [31]:
```
x[x>0] = 100
x
```

Out[31]:
```
array([ -1,  -2,   0, 100, 100])
```

## Изменение структуры массива

Для выравнивания многомерного массива в соответствии с внутренним порядком хранения элементов по строкам используются методы `flatten` и `ravel`.

In [18]:
```
matrix = np.array([[1,2],[3,4]])
f = matrix.flatten(); r = matrix.ravel()
f, r
```

Out[18]:
```
(array([1, 2, 3, 4]), array([1, 2, 3, 4]))
```

Результатом метода `flatten` является *копия* элементов исходного массива. Результатом метода `ravel` является массив, который хранит *ссылки* на элементы исходного массива (*представление массива*). Изменения в массиве-копии `f`, выравненном с помощью метода `flatten`, не изменяют исходный массив, тогда как изменения в массиве-представлении `r`, выравненном с помощью метода `ravel`, изменяют исходный массив.

In [20]:
```
f[0] = 100; r[1] = 100
matrix, f, r,
```

Out[20]:
```
(array([100,   2,   3,   4]),
 array([  1, 100,   3,   4]),
 array([[  1, 100],
        [  3,   4]]))
```

Методы `resize` изменяет структуру массива на новую структуру, указанную в аргументе, *при условии сохранения количества элементов исходного массива после замены структуры*. Метод `reshape` возвращает *представление массива* с измененной структурой исходного массива, т.е. результирующий массив хранит *ссылки* на элементы исходного массива.

In [30]:
```
matrix = np.array(range(4))
```

In [31]:
```
matrix.resize(2,2)
print(f'{matrix=}')
matrix = np.array(range(4))
m_reshape = matrix.reshape(2,2)
print(f'{m_reshape=}')
```

```
matrix=array([[0, 1],
       [2, 3]])
m_reshape=array([[0, 1],
       [2, 3]])
```

In [32]:
```
m_reshape[0,0] = 100
matrix
```

Out[32]:
```
array([100,   1,   2,   3])
```

Изменять структуру массива можно также с помощью атрибута `shape`

In [17]:
```
x = np.arange(12)
x.shape = (4,3)
print(x)
x.shape = (2,6)
print(x)

try:
    x.shape = (5,2)
except ValueError as e:
    print("ValueError:", e)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
ValueError: cannot reshape array of size 12 into shape (5,2)
```

## Объединение массивов

Функции `vstack` , `hstack` объединяют массивы вертикально (по строкам) и горизонтально (по столбцам), соответственно

In [35]:
```
row1 = np.zeros(3); row2 = np.ones(3)
np.vstack((row1,row2)), np.hstack((row1,row2))
```

Out[35]:
```
(array([[0., 0., 0.],
        [1., 1., 1.]]),
 array([0., 0., 0., 1., 1., 1.]))
```

Функция `concatenate` является аналогом `hstack`

In [42]:
```
np.concatenate((row1,row2)), np.hstack((row1,row2))
```

Out[42]:
```
(array([0., 0., 0., 1., 1., 1.]), array([0., 0., 0., 1., 1., 1.]))
```

# 13.3 Векторизация вычислений с массивами

**Векторизация вычислений** -- это выполнение арифметических операций для каждого элемента массива (поэлементные вычисления, elementwise computations) без использования циклов.

Векторизация *неявно* реализует поэлементные вычисления для массива вместо необходимости *явного* построения циклов для последовательной обработки каждого элемента массива.

In [41]:
```
x = np.ones(3)
y = 2*np.ones(3)
x*y # векторизация операции умножения
```

Out[41]:
```
array([2., 2., 2.])
```

In [42]:
```
np.array([i*j for (i,j) in zip(x,y)]) # поэлементное умножение
```

Out[42]:
```
array([2., 2., 2.])
```

Отсутствие циклических конструкций и явного индексирования элементов массива делает код более понятным, в меньшей степени подверженным ошибкам и приближает его к соответствующему математическому формату записи выражений.

При этом выполнение векторизованных операций будет максимально быстрым из-за реализации оптимизированных алгоритмов на языке `C`.

In [6]:
```
x = np.ones(10**6)
```

In [7]:
```
%timeit map(lambda el: 5*el, x)
```

```
819 ns ± 131 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

In [8]:
```
%timeit 5*x
```

```
6.66 ms ± 450 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Арифметические операции поддерживают векторизацию вычислений для массивов *одинаковой структуры*, а также в случае, когда одним из операндов является число, а вторым -- массив (**broadcasting**)

In [46]:
```
x = 3*np.ones((2,2)); y = 5*np.ones((2,2))
x + y, x*y, 2*x - 1, y**2
```

Out[46]:
```
(array([[8., 8.],
        [8., 8.]]),
 array([[15., 15.],
        [15., 15.]]),
 array([[5., 5.],
        [5., 5.]]),
 array([[25., 25.],
        [25., 25.]]))
```

In [47]:
```
x = 3*np.ones((2,2)); y = 5*np.ones((1,3))

try:
    x + y, x*y, 2*x - 1, y**2
except ValueError as e:
    print("ValueError", e)
```

```
ValueError operands could not be broadcast together with shapes (2,2) (1,3)
```

Выполнение арифметических операций также возможно, когда один из операндов имеет два измерения (матрица), а второй операнд -- одно измерение (вектор) и количество элементов по совпадающему измерению одинаковое (**broadcasting**)

In [48]:
```
x = 3*np.ones((2,3)); y = np.array([1,2,3]); z =  np.array([[4],[5]])
print(x + y)  # сложение строк
x + z  # сложение столбцов
```

Out[48]:
```
[[4. 5. 6.]
 [4. 5. 6.]]
array([[7., 7., 7.],
       [8., 8., 8.]])
```

Операторы дополненного присваивания поддерживают векторизацию вычислений для массивов. Оператор дополненного присваивания изменяет значения исходного массива, а не создает новый массив

In [49]:
```
print(x)
x += 1; x *= 2
x
```

Out[49]:
```
[[3. 3. 3.]
 [3. 3. 3.]]
array([[8., 8., 8.],
       [8., 8., 8.]])
```

Операции сравнения и логические операции поддерживают векторизацию вычислений

In [50]:
```
x = 3*np.ones((2,2)); y = 5*np.ones((2,2))
(x > y) | (x == 3)
```

Out[50]:
```
array([[ True,  True],
       [ True,  True]])
```

Для обеспечения векторизации вычислений многие математические функции модуля `math` реализованы в расширении `numpy` для *неявного* вычисления функции от каждого элемента массива: `sign`, `sqrt`, `log`, `log10`, `sin`, `arcsin`, `sinh`, `arcsinh` и т.д.

In [51]:
```
np.sin(x), np.sqrt(x), np.log(x)
```

Out[51]:
```
(array([[0.14112001, 0.14112001],
        [0.14112001, 0.14112001]]),
 array([[1.73205081, 1.73205081],
        [1.73205081, 1.73205081]]),
 array([[1.09861229, 1.09861229],
        [1.09861229, 1.09861229]]))
```

В расширении numpy определены два специальных числовых объекта nan и inf . Объект nan используется для представления неопределенного результата вычислений, объект inf -- для представления бесконечности

In [52]:
```
?np.nan
```

```
Type:        float
String form: nan
Docstring:   Convert a string or number to a floating point number, if possible.
```

In [53]:
```
?np.inf
```

```
Type:        float
String form: inf
Docstring:   Convert a string or number to a floating point number, if possible.
```

In [54]:
```
x = np.array([0,1])
x/0
```

```
C:\Users\Olga\AppData\Local\Temp/ipykernel_2900/856317653.py:2: RuntimeWarning: div
ide by zero encountered in true_divide
  x/0
C:\Users\Olga\AppData\Local\Temp/ipykernel_2900/856317653.py:2: RuntimeWarning: inv
alid value encountered in true_divide
  x/0
```

Out[54]:
```
array([nan, inf])
```

Функции isnan , isinf , isfinite позволяют осуществлять проверку результата вычислений на определенность и конечность

In [55]:
```
x, np.isnan(x/0), np.isfinite(x/0)
```

```
C:\Users\Olga\AppData\Local\Temp/ipykernel_2900/1616215112.py:1: RuntimeWarning: di
vide by zero encountered in true_divide
  x, np.isnan(x/0), np.isfinite(x/0)
C:\Users\Olga\AppData\Local\Temp/ipykernel_2900/1616215112.py:1: RuntimeWarning: in
valid value encountered in true_divide
  x, np.isnan(x/0), np.isfinite(x/0)
```

Out[55]:
```
(array([0, 1]), array([ True, False]), array([False, False]))
```

# 13.4 Операции линейной алгебры

**Вектор** с $n$ компонентами определяется в numpy как *одномерный массив* с $n$ элементами.

**Матрица** размера $k \times m$ определяется как *двумерный массив* с $k$ элементами-последовательностями, каждый из которых состоит из $m$ элементов.

Функция `eye` , функция `identity` возвращают единичную матрицу. Аргументами функции `eye` являются количество срок и столбцов матрицы, аргумент функции `identity` определяет размер квадратной матрицы

In [56]:
```
np.eye(3,4), np.identity(3)
```

Out[56]:
```
(array([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.]]),
 array([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]]))
```

Функция `diag` возвращает диагональную матрицу. Аргументом функции `diag` является список или итерируемый объект, определяющий диагональные элементы

In [57]:
```
my_m = np.diag(range(5))
my_m
```

Out[57]:
```
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4]])
```

Также функция `diag` возвращает диагональные элементы, если аргументом функции является матрица

In [58]:
```
np.diag(my_m)
```

Out[58]:
```
array([0, 1, 2, 3, 4])
```

С помощью операции умножения `*` можно умножить вектор и матрицу на скалярную величину как слева, так и справа

In [59]:
```
x = 2*np.ones(5); y = np.ones((2,2))*3
x, y
```

Out[59]:
```
(array([2., 2., 2., 2., 2.]),
 array([[3., 3.],
        [3., 3.]]))
```

Скалярное произведение векторов и матричное произведение реализовано в `numpy` с помощью операции `@` , функции `dot` и метода `dot` для объекта массива

In [60]:
```
x = np.ones(5)
x@x, np.dot(x,x), x.dot(x)
```

Out[60]:
```
(5.0, 5.0, 5.0)
```

In [61]:
```
y = 3*np.ones((2,2))
y@y, np.dot(y,y), y.dot(y)
```

Out[61]:
```
(array([[18., 18.],
        [18., 18.]]),
 array([[18., 18.],
        [18., 18.]]),
 array([[18., 18.],
        [18., 18.]]))
```

Скалярное произведение векторов реализовано в `numpy` также с помощью функции `inner`

In [62]:
```
np.inner(x,x)
```

Out[62]:
```
5.0
```

Транспонирование реализовано в `numpy` с помощью функции `transpose`, метода `transpose` и атрибута `T` для объекта массива

In [63]:
```
y = np.array([range(5), range(5)])
np.transpose(y), y.transpose(), y.T.T
```

Out[63]:
```
(array([[0, 0],
        [1, 1],
        [2, 2],
        [3, 3],
        [4, 4]]),
 array([[0, 0],
        [1, 1],
        [2, 2],
        [3, 3],
        [4, 4]]),
 array([[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]))
```

**Модуль linalg**

Модуль `linalg` расширения `numpy` предоставляет дополнительные функциональные возможности для работы с матрицами

In [64]:
```
import numpy.linalg as lin
```

Функция `matrix_power(A,k)` модуля `linalg` возвращает результат возведения матрицы `A` в степень `k`

In [65]:
```
A = np.ones((2,2))
lin.matrix_power(A,3), A@A@A
```

Out[65]:
```
(array([[4., 4.],
        [4., 4.]]),
 array([[4., 4.],
        [4., 4.]]))
```

Функция norm модуля linalg возвращает евклидову норму для вектора и норму Фробениуса для матрицы

In [66]:
```
A, lin.norm(A[0]), lin.norm(A)
```

Out[66]:
```
(array([[1., 1.],
        [1., 1.]]),
 1.4142135623730951,
 2.0)
```

Функция det модуля linalg возвращает определитель матрицы

In [67]:
```
lin.det(A)
```

Out[67]:
```
0.0
```

Функция matrix_rank модуля linalg возвращает ранг матрицы

In [68]:
```
lin.matrix_rank(A)
```

Out[68]:
```
1
```

Функция inv модуля linalg возвращает обратную матрицы, если она существует

In [69]:
```
try:
    lin.inv(A)
except lin.LinAlgError as e:
    print(e)
```

```
Singular matrix
```

In [70]:
```
mat = np.array(range(4)).reshape(2,2)
mat.dot(lin.inv(mat))
```

Out[70]:
```
array([[1., 0.],
       [0., 1.]])
```

Функция solve(A,b) модуля linalg возвращает решение системы линейных алгебраических уравнений, заданной в матричном виде $Ax = b$, при условии единственности решения

In [71]:
```
A = np.eye(3,3); b = np.array([1,2,3])
lin.solve(A,b), lin.inv(A) @ b
```

Out[71]:
```
(array([1., 2., 3.]), array([1., 2., 3.]))
```

In [72]:
```
A = np.ones((3,3)); b = np.array([1,2,3])

try:
    lin.solve(A,b)
except lin.LinAlgError as e:
    print(e)
```

```
Singular matrix
```

см. материалы для обновления https://www.dmitrymakarov.ru/python/numpy-09/