Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

Тема 11. Расширение numpy для численных вычислений

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

май, 2025

11.1 Расширение numpy

In [3]:
```
import numpy as np
```

`numpy` является сокращением от **NUMerical PYthon**. Расширение `numpy` предназначено для *эффективной* обработки числовых данных и (научных) вычислений на основе типа данных **многомерный массив** `ndarray`.

Первая версия расширения `numpy` создана в 2006 году, создатель Трэвис Олифант (*Travis Oliphant*). Текущая версия 2.2.5.

In [7]:
```
np.__version__
```

Out[7]:
```
'1.26.4'
```

Расширение `numpy` не входит в стандартную библиотеку Python.

Преимущество `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`.
- `scikit-learn`, `scikit-image` используют библиотеку `numpy` для высокопроизводительных операций линейной алгебры и работы с массивами.

11.2 Атрибуты 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. Например, для представления матрицы из n строк и m столбцов используется массив со структурой `(n, m)`
- `ndim` хранит количество измерений массива
- `size` хранит количество всех элементов массива
- `dtype` хранит тип данных элементов массива
- `data` хранит адрес в памяти первого элемента массива
- `itemsize` хранит размер элемента массива в байтах
- `nbytes` хранит общий размер массива в байтах

In [17]:
```
# пример двумерного массива
x = np.array([[1],[2],[3]])
(type(x), x.shape, x.ndim, x.size,
x.dtype, x.data, x.itemsize, x.nbytes)
```

Out[17]:
```
(numpy.ndarray,
 (3, 1),
 2,
 3,
 dtype('int32'),
 <memory at 0x000001CD1C30CBA0>,
 4,
 12)
```

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

In [20]:
```
x.size*x.itemsize, x.nbytes
```

Out[20]: (12, 12)

In [21]:
```
# числа имеют нулевую размерность
x = np.array(1)
(type(x), x.shape, x.ndim, x.size,
x.dtype, x.data, x.itemsize, x.nbytes)
```

Out[21]: (numpy.ndarray, (), 0, 1, dtype('int32'), <memory at 0x000001CD1C2EC540>, 4, 4)

In [22]:
```
# пример одномерного массива
x = np.array(['1','2000','3'])
(type(x), x.shape, x.ndim, x.size,
x.dtype, x.data, x.itemsize, x.nbytes)
```

Out[22]:
```
(numpy.ndarray,
 (3,),
 1,
 3,
 dtype('<U4'),
 <memory at 0x000001CD1C073DC0>,
 16,
 48)
```

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

In [25]:
```
# пример трехмерного массива
x = np.array([[[1,2]],[[3,4]],[[5,6]]])
print(x)
x.shape, len(x)
```

```
[[[1 2]]

 [[3 4]]

 [[5 6]]]
```

Out[25]: ((3, 1, 2), 3)

Массив является итерируемым объектом по элементам первого измерения

In [29]:
```
for el in x:
    print(el)
```

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

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

In [31]:
```
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 [35]:
```
np.array([1.,2,3]), np.array((1,2,3)), np.array(range(5))
```

Out[35]:
```
(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 [40]:
```
ar = np.array(range(5), dtype=complex)
ar, ar.dtype
```

Out[40]:
```
(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 [43]:
```
?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
_147_483_648`` to ``2_147_483_647``).
File:           c:\users\user\anaconda3\lib\site-packages\numpy\__init__.py
Type:           type
Subclasses:
```

In [44]:
```
?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 co
mposed of 2 64-bit-precision floating-point numbers.
File:           c:\users\user\anaconda3\lib\site-packages\numpy\__init__.py
Type:           type
Subclasses:
```

In [45]:
```
?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
-point number type: sign bit, 11 bits exponent, 52 bits mantissa.
File:           c:\users\user\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 [47]:
```
emp = np.empty((2,3,4)); emp1 = np.empty(4)
print(emp); print(emp1)
```

```
[[[6.23042070e-307 4.67296746e-307 1.69121096e-306 9.34609111e-307]
  [1.33511018e-306 1.33511969e-306 6.23037996e-307 6.23053954e-307]
  [9.34609790e-307 8.45593934e-307 9.34600963e-307 7.56593017e-307]]

 [[2.04719290e-306 9.34600963e-307 6.89804133e-307 8.45610231e-307]
  [9.34601642e-307 2.04719289e-306 9.34600963e-307 8.45559303e-307]
  [1.06811422e-306 1.05694828e-307 1.42410974e-306 1.42404727e-306]]]
[1. 2. 3. 4.]
```

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 [49]:
```
np.zeros(400000), np.ones((2,2), dtype=int),
```

Out[49]:
```
(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 [53]:
```
np.zeros_like(emp, dtype=int)
```

Out[53]:
```
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 `full` создает массив, заполненный заданным значением. Метод `fill` заменяет все элементы массива на заданное число

In [57]:
```
m=np.full((2, 3), 1)
print(m); m.fill(9); print(m)
```

```
[[1 1 1]
 [1 1 1]]
[[9 9 9]
 [9 9 9]]
```

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

In [60]:
```
np.arange(1,3,0.5)
```

Out[60]:
```
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 [63]:
```
np.linspace(1, 3, 5)
```

Out[63]:
```
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 [66]:
```
x, dx = np.linspace(1, 3, 5, retstep=True)
x, dx
```

Out[66]:
```
(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 [68]:
```
np.fromfunction(lambda i, j: i+j, (2,4))
```

Out[68]:
```
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 [76]:
```
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 [79]:
```
matrix[-1,-1]
```

Out[79]:
```
4
```

Поддерживается индексация с нарезанием для каждого измерения массива

In [81]:
```
matrix = np.array([range(5)]*3)
matrix[:, :], matrix[1:3, 2], matrix[1:,1:], matrix[:3, :1], matrix[::2,:]
```

Out[81]:
```
(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 (view)**). Это означает, что при индексации с нарезанием данные НЕ копируются и любые изменения в представлении массива будут отражены в исходном массиве!

In [83]:
```
matrix1 = matrix[:, :]
matrix1[0,0] = 100
matrix
```

Out[83]:
```
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 [87]:
```
matrix.base, matrix1.base
```

Out[87]:
```
(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 [90]:
```
matrix1 = np.copy(matrix[:, :])
# или
matrix1 = matrix[:,:].copy()
matrix1[0,0] = 200
matrix, matrix1
```

Out[90]:
```
(array([[100,   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 [91]:
```
matrix.base, matrix1.base
```

Out[91]:
```
(None, None)
```

Применение функций, которые возвращают представление массива (view), является более эффективным по времени и памяти, чем применение функций, которые возвращают новый объект массива.

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

In [94]:
```
matrix[0], matrix[0,:], matrix[0,...]
```

Out[94]:
```
(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 [97]:
```
x = np.array([-1, -2, 0, 1, 2])
x>0, x[x>0]
```

Out[97]:
```
(array([False, False, False,  True,  True]), array([1, 2]))
```

In [98]:
```
x[x>0] = 100
x
```

Out[98]:
```
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 [106…
```
matrix = np.array([[1,2],[3,4]])
f = matrix.flatten(); r = matrix.ravel()
f, r
```

Out[106…
```
(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 (view)*). 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 [109…
```
f[0] = 100; r[1] = 100
matrix, f, r,
```

Out[109…
```
(array([[  1, 100],
        [  3,   4]]),
 array([100,   2,   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 [112…
```
matrix = np.array(range(4))
```

In [113…
```
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 [114…
```
m_reshape[0,0] = 100
matrix
```

Out[114…
```
array([100,   1,   2,   3])
```

Изменять структуру массива можно также с помощью атрибута `shape`

In [118…
```
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 [122…
```
row1 = np.zeros(3); row2 = np.ones(3)
np.vstack((row1,row2)), np.hstack((row1,row2))
```

Out[122…
```
(array([[0., 0., 0.],
        [1., 1., 1.]]),
 array([0., 0., 0., 1., 1., 1.]))
```

Функция `concatenate` является аналогом `hstack`

In [124…
```
np.concatenate((row1,row2)), np.hstack((row1,row2))
```

Out[124…
```
(array([0., 0., 0., 1., 1., 1.]), array([0., 0., 0., 1., 1., 1.]))
```

11.3 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 [131…
```
x = np.ones(3)
y = 2*np.ones(3)
x*y # векторизация операции умножения
```

Out[131…
```
array([2., 2., 2.])
```

In [133…
```
np.array([i*j for (i,j) in zip(x,y)]) # поэлементное умножение
```

Out[133… 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 [137…
```
x = np.ones(10**6)
```

In [138…
```
%timeit map(lambda el: 5*el, x)
```

```
367 ns ± 22.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

In [139…
```
%timeit 5*x
```

```
4.67 ms ± 297 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 [141…
```
x = 3*np.ones((2,2)); y = 5*np.ones((2,2))
x + y, x*y, 2*x - 1, y**2
```

Out[141…
```
(array([[8., 8.],
        [8., 8.]]),
 array([[15., 15.],
        [15., 15.]]),
 array([[5., 5.],
        [5., 5.]]),
 array([[25., 25.],
        [25., 25.]]))
```

In [143…
```
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**). Broadcasting позволяет производить арифметические операции над массивами разных, но согласованных измерений.

In order to broadcast, the size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one

In [145…
```
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
```

```
[[4. 5. 6.]
 [4. 5. 6.]]
```

Out[145…
```
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 [147…
```
print(x)
x += 1; x *= 2
x
```

```
[[3. 3. 3.]
 [3. 3. 3.]]
```

Out[147…
```
array([[8., 8., 8.],
       [8., 8., 8.]])
```

Операции сравнения и логические операции поддерживают векторизацию вычислений

In [149…
```
x = 3*np.ones((2,2)); y = 5*np.ones((2,2))
(x > y) | (x == 3)
```

Out[149…
```
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.

Также реализованы в расширении `numpy` функции `min`, `max`, `mean`, `sum` и др. для обеспечения векторизации вычислений.

In [151…
```
np.sin(x), np.sqrt(x), np.log(x)
```

Out[151…
```
(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 [153…
```
?np.nan
```

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

In [154…
```
?np.inf
```

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

In [155…
```
x = np.array([0,1])
x/0
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_11720\856317653.py:2: RuntimeWarning:
divide by zero encountered in divide
  x/0
C:\Users\user\AppData\Local\Temp\ipykernel_11720\856317653.py:2: RuntimeWarning:
invalid value encountered in divide
  x/0
```

Out[155…
```
array([nan, inf])
```

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

In [157…
```
x, np.isnan(x/0), np.isfinite(x/0)
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_11720\1616215112.py:1: RuntimeWarning:
divide by zero encountered in divide
  x, np.isnan(x/0), np.isfinite(x/0)
C:\Users\user\AppData\Local\Temp\ipykernel_11720\1616215112.py:1: RuntimeWarning:
invalid value encountered in divide
  x, np.isnan(x/0), np.isfinite(x/0)
```

Out[157…
```
(array([0, 1]), array([ True, False]), array([False, False]))
```

11.4 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 [162…
```
np.eye(3,4), np.identity(3)
```

Out[162…
```
(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 [177…
```
my_m = np.diag(range(5))
my_m
```

Out[177…
```
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 [180…
```
np.diag(my_m)
```

Out[180…
```
array([0, 1, 2, 3, 4])
```

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

In [183…
```
x = 2*np.ones(5); y = np.ones((2,2))*3
x, y
```

Out[183…
```
(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 [186…
```
x = np.ones(5)
x@x, np.dot(x,x), x.dot(x)
```

Out[186…
```
(5.0, 5.0, 5.0)
```

In [188…
```
y = 3*np.ones((2,2))
y@y, np.dot(y,y), y.dot(y)
```

Out[188…
```
(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 [191…
```
np.inner(x,x)
```

Out[191…
```
5.0
```

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

In [194…
```
y = np.array([range(5), range(5)])
np.transpose(y), y.transpose(), y.T.T
```

Out[194…
```
(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 (более эффективным считается модуль `linalg` расширения `scipy`)

In [198…
```
import numpy.linalg as lin
```

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

In [201…
```
A = np.ones((2,2))
lin.matrix_power(A,3), A@A@A
```

Out[201…
```
(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 [204…
```
A, lin.norm(A[0]), lin.norm(A)
```

Out[204…
```
(array([[1., 1.],
        [1., 1.]]),
 1.4142135623730951,
 2.0)
```

Функция `det` модуля `linalg` возвращает определитель матрицы

In [207…
```
lin.det(A)
```

Out[207…
```
0.0
```

Функция `matrix_rank` модуля `linalg` возвращает ранг матрицы

In [210…
```
lin.matrix_rank(A)
```

Out[210… 1

Функция `inv` модуля `linalg` возвращает обратную матрицы, если она существует

In [213…
```
try:
    lin.inv(A)
except lin.LinAlgError as e:
    print(e)
```

```
Singular matrix
```

In [215…
```
mat = np.array(range(4)).reshape(2,2)
mat.dot(lin.inv(mat))
```

Out[215…
```
array([[1., 0.],
       [0., 1.]])
```

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

In [218…
```
A = np.eye(3,3); b = np.array([1,2,3])
lin.solve(A,b), lin.inv(A) @ b
```

Out[218… (array([1., 2., 3.]), array([1., 2., 3.]))

In [220…
```
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
```