In [1]:

```
import numpy as np
```

Библиотека numpy

Пакет `numpy` предоставляет n-мерные однородные массивы (все элементы одного типа); в них нельзя вставить или удалить элемент в произвольном месте. В `numpy` реализовано много операций над массивами в целом. Если задачу можно решить, произведя некоторую последовательность операций над массивами, то это будет столь же эффективно, как в `C` или `matlab` - львиная доля времени тратится в библиотечных функциях, написанных на `C` .

Замечание. Модуль `numpy.random` не рассматривается целенаправленно. Вместо него на следующем занятии мы будем изучать модуль `scipy.stats` , который больше подходит под наши задачи.

1. Одномерные массивы

1.1 Типы массивов, атрибуты

Можно преобразовать список в массив.

In [2]:

```
a = np.array([0, 2, 1])
a, type(a)
```

Out[2]:

```
(array([0, 2, 1]), numpy.ndarray)
```

`print` печатает массивы в удобной форме.

In [3]:

```
print(a)
```

```
[0 2 1]
```

Класс `ndarray` имеет много методов.

In [4]:

```
set(dir(a)) - set(dir(object))
```

Out[4]:

```
{'T',
 '__abs__',
 '__add__',
 '__and__',
 '__array__',
 '__array_finalize__',
 '__array_function__',
 '__array_interface__',
 '__array_prepare__',
 '__array_priority__',
 '__array_struct__',
 '__array_ufunc__',
 '__array_wrap__',
 '__bool__',
 '__complex__',
 '__contains__',
 '__copy__',
 '__deepcopy__',
 '__delitem__',
 '__divmod__',
 '__float__',
 '__floordiv__',
 '__getitem__',
 '__iadd__',
 '__iand__',
 '__ifloordiv__',
 '__ilshift__',
 '__imatmul__',
 '__imod__',
 '__imul__',
 '__index__',
 '__int__',
 '__invert__',
 '__ior__',
 '__ipow__',
 '__irshift__',
 '__isub__',
 '__iter__',
 '__itruediv__',
 '__ixor__',
 '__len__',
 '__lshift__',
 '__matmul__',
 '__mod__',
 '__mul__',
 '__neg__',
 '__or__',
 '__pos__',
 '__pow__',
 '__radd__',
 '__rand__',
 '__rdivmod__',
 '__rfloordiv__',
 '__rlshift__',
 '__rmatmul__',
 '__rmod__',
 '__rmul__',
 '__ror__',
 '__rpow__',
```

```
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setitem__',
'__setstate__',
'__sub__',
'__truediv__',
'__xor__',
'all',
'any',
'argmax',
'argmin',
'argpartition',
'argsort',
'astype',
'base',
'byteswap',
'choose',
'clip',
'compress',
'conj',
'conjugate',
'copy',
'ctypes',
'cumprod',
'cumsum',
'data',
'diagonal',
'dot',
'dtype',
'dump',
'dumps',
'fill',
'flags',
'flat',
'flatten',
'getfield',
'imag',
'item',
'itemset',
'itemsize',
'max',
'mean',
'min',
'nbytes',
'ndim',
'newbyteorder',
'nonzero',
'partition',
'prod',
'ptp',
'put',
'ravel',
'real',
'repeat',
'reshape',
'resize',
'round',
'searchsorted',
```

```
 'setfield',
 'setflags',
 'shape',
 'size',
 'sort',
 'squeeze',
 'std',
 'strides',
 'sum',
 'swapaxes',
 'take',
 'tobytes',
 'tofile',
 'tolist',
 'tostring',
 'trace',
 'transpose',
 'var',
 'view'}
```

Наш массив одномерный.

In [5]:

```
a.ndim
```

Out[5]:

```
1
```

В n-мерном случае возвращается кортеж размеров по каждой координате.

In [6]:

```
a.shape
```

Out[6]:

```
(3,)
```

`size` - это полное число элементов в массиве; `len` - размер по первой координате (в 1-мерном случае это то же самое).

In [7]:

```
len(a), a.size
```

Out[7]:

```
(3, 3)
```

`numpy` предоставляет несколько типов для целых (`int16` , `int32` , `int64`) и чисел с плавающей точкой (`float32` , `float64`).

In [8]:

```
a.dtype, a.dtype.name, a.itemsize
```

Out[8]:

```
(dtype('int64'), 'int64', 8)
```

Массив чисел с плавающей точкой.

In [9]:

```
b = np.array([0., 2, 1])
b.dtype
```

Out[9]:

```
dtype('float64')
```

Точно такой же массив.

In [10]:

```
c = np.array([0, 2, 1], dtype=np.float64)
print(c)
```

```
[0. 2. 1.]
```

Преобразование данных

In [11]:

```
print(c.dtype)
print(c.astype(int))
print(c.astype(str))
```

```
float64
[0 2 1]
['0.0' '2.0' '1.0']
```

1.2 Индексация

Индексировать массив можно обычным образом.

In [12]:

```
a[1]
```

Out[12]:

```
2
```

Массивы - изменяемые объекты.

In [13]:

```
a[1] = 3
print(a)
```

```
[0 3 1]
```

Массивы, разумеется, можно использовать в `for` циклах. Но при этом теряется главное преимущество `numpy` - быстродействие. Всегда, когда это возможно, лучше использовать операции над массивами как едиными целыми.

In [14]:

```
for i in a:
    print(i)
```

```
0
3
1
```

Упражнение: создайте numpy-массив, состоящий из первых пяти простых чисел, выведите его тип и размер:

In [15]:

```
# <...>
```

In [16]:

```
# решение

arr = np.array([2, 3, 5, 7, 11])
print(arr)
print(arr.shape)
print(arr.dtype)
```

```
[ 2  3  5  7 11]
(5,)
int64
```

1.3 Создание массивов

Массивы, заполненные нулями или единицами. Часто лучше сначала создать такой массив, а потом присваивать значения его элементам.

In [17]:

```
a = np.zeros(3)
b = np.ones(3, dtype=np.int64)
print(a)
print(b)
```

```
[0. 0. 0.]
[1 1 1]
```

Если нужно создать массив, заполненный нулями, длины и типа другого массива, то можно использовать конструкцию

In [18]:

```
np.zeros_like(b)
```

Out[18]:

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

Функция `arange` подобна `range` . Аргументы могут быть с плавающей точкой. Следует избегать ситуаций, когда *(конец-начало)/шаг* - целое число, потому что в этом случае включение последнего элемента зависит от ошибок округления. Лучше, чтобы конец диапазона был где-то посредине шага.

In [19]:

```
a = np.arange(0, 9, 2)
print(a)
```

```
[0 2 4 6 8]
```

In [20]:

```
b = np.arange(0., 9, 2)
print(b)
```

```
[0. 2. 4. 6. 8.]
```

Последовательности чисел с постоянным шагом можно также создавать функцией `linspace` . Начало и конец диапазона включаются; последний аргумент - число точек.

In [21]:

```
a = np.linspace(0, 8, 5)
print(a)
```

```
[0. 2. 4. 6. 8.]
```

Упражнение: создайте и выведите последовательность чисел от 10 до 20 с постоянным шагом, длина последовательности - 21.

In [22]:

```
# <...>
```

In [23]:

```
# решение
arr = np.linspace(10, 20, 21)
print(arr)
```

```
[10.  10.5 11.  11.5 12.  12.5 13.  13.5 14.  14.5 15.  15.5 16.  1
6.5
 17.  17.5 18.  18.5 19.  19.5 20. ]
```

Последовательность чисел с постоянным шагом по логарифмической шкале от 10^0 до 10^1.

In [24]:

```
b = np.logspace(0, 1, 5)
print(b)
```

```
[ 1.          1.77827941  3.16227766  5.62341325 10.        ]
```

2. Операции над одномерными массивами

2.1 Математические операции

Арифметические операции проводятся поэлементно.

In [25]:

```
a
```

Out[25]:

```
array([0., 2., 4., 6., 8.])
```

In [26]:

```
b
```

Out[26]:

```
array([ 1.        ,  1.77827941,  3.16227766,  5.62341325, 10.
])
```

In [27]:

```
print(a + b)
```

```
[ 1.          3.77827941  7.16227766 11.62341325 18.        ]
```

In [28]:

```
print(a - b)
```

```
[-1.          0.22172059  0.83772234  0.37658675 -2.        ]
```

In [29]:

```
print(a * b)
```

```
[ 0.          3.55655882 12.64911064 33.74047951 80.        ]
```

In [30]:

```
print(a / b)
```

```
[0.         1.12468265 1.26491106 1.06696765 0.8       ]
```

In [31]:

```
print(a ** 2)
```

```
[ 0.  4. 16. 36. 64.]
```

Когда операнды разных типов, они пиводятся к большему типу.

In [32]:

```
i = np.ones(5, dtype=np.int64)
print(a + i)
```

```
[1. 3. 5. 7. 9.]
```

`numpy` содержит элементарные функции, которые тоже применяются к массивам поэлементно. Они называются универсальными функциями (`ufunc`).

In [33]:

```
np.sin, type(np.sin)
```

Out[33]:

```
(<ufunc 'sin'>, numpy.ufunc)
```

In [34]:

```
print(np.sin(a))
```

```
[ 0.          0.90929743 -0.7568025  -0.2794155   0.98935825]
```

Один из операндов может быть скаляром, а не массивом.

In [35]:

```
print(a + 1)
```

```
[1. 3. 5. 7. 9.]
```

In [36]:

```
print(2 * a)
```

```
[ 0.  4.  8. 12. 16.]
```

Сравнения дают булевы массивы.

In [37]:

```
print(a > b)
```

```
[False  True  True  True False]
```

In [38]:

```
print(a == b)
```

```
[False False False False False]
```

In [39]:

```
c = a > 5
print(c)
```

```
[False False False  True  True]
```

Кванторы "существует" и "для всех".

In [40]:

```
np.any(c), np.all(c)
```

Out[40]:

```
(True, False)
```

Модификация на месте.

In [41]:

```
a
```

Out[41]:

```
array([0., 2., 4., 6., 8.])
```

In [42]:

```
a += 1
print(a)
```

```
[1. 3. 5. 7. 9.]
```

In [43]:

```
b
```

Out[43]:

```
array([ 1.        ,  1.77827941,  3.16227766,  5.62341325, 10.
])
```

In [44]:

```
b *= 2
print(b)
```

```
[ 2.          3.55655882  6.32455532 11.2468265  20.        ]
```

In [45]:

```
b /= a
print(b)
```

```
[2.         1.18551961 1.26491106 1.6066895  2.22222222]
```

При выполнении операций над массивами деление на 0 не возбуждает исключения, а даёт значения `np.nan` или `np.inf`.

In [46]:

```
print(np.array([0.0, 0.0, 1.0, -1.0]) / np.array([1.0, 0.0, 0.0, 0.0]))
```

```
[  0.  nan  inf -inf]

/Users/eugene.ivanin/opt/anaconda3/lib/python3.7/site-packages/ipyke
rnel_launcher.py:1: RuntimeWarning: divide by zero encountered in tr
ue_divide
  """Entry point for launching an IPython kernel.
/Users/eugene.ivanin/opt/anaconda3/lib/python3.7/site-packages/ipyke
rnel_launcher.py:1: RuntimeWarning: invalid value encountered in tru
e_divide
  """Entry point for launching an IPython kernel.
```

In [47]:

```
np.nan + 1, np.inf + 1, np.inf * 0, 1. / np.inf
```

Out[47]:

```
(nan, inf, nan, 0.0)
```

Сумма и произведение всех элементов массива; максимальный и минимальный элемент; среднее и среднеквадратичное отклонение.

In [48]:

```
b
```

Out[48]:

```
array([2.        , 1.18551961, 1.26491106, 1.6066895 , 2.22222222])
```

In [49]:

```
b.sum(), b.prod(), b.max(), b.min(), b.mean(), b.std()
```

Out[49]:

```
(8.279342393526044,
 10.708241812210389,
 2.2222222222222223,
 1.1855196066926152,
 1.6558684787052087,
 0.4039003342660745)
```

Имеются встроенные функции

In [50]:

```
print(np.sqrt(b))
print(np.exp(b))
print(np.log(b))
print(np.sin(b))
print(np.e, np.pi)
```

```
[1.41421356 1.08881569 1.12468265 1.26755256 1.49071198]
[7.3890561  3.27238673 3.54277764 4.98627681 9.22781435]
[0.69314718 0.17018117 0.23500181 0.47417585 0.7985077 ]
[0.90929743 0.92669447 0.95358074 0.99935591 0.79522006]
2.718281828459045 3.141592653589793
```

Иногда бывает нужно использовать частичные (кумулятивные) суммы. В нашем курсе такое пригодится.

In [51]:

```
print(b.cumsum())
```

```
[2.         3.18551961 4.45043067 6.05712017 8.27934239]
```

2.2 Сортировка, изменение массивов

Функция `sort` возвращает отсортированную копию, метод `sort` сортирует на месте.

In [52]:

```
b
```

Out[52]:

```
array([2.        , 1.18551961, 1.26491106, 1.6066895 , 2.22222222])
```

In [53]:

```
print(np.sort(b))
print(b)
```

```
[1.18551961 1.26491106 1.6066895  2.         2.22222222]
[2.         1.18551961 1.26491106 1.6066895  2.22222222]
```

In [54]:

```
b.sort()
print(b)
```

```
[1.18551961 1.26491106 1.6066895  2.         2.22222222]
```

Объединение массивов.

In [55]:

```
a
```

Out[55]:

```
array([1., 3., 5., 7., 9.])
```

In [56]:

```
b
```

Out[56]:

```
array([1.18551961, 1.26491106, 1.6066895 , 2.        , 2.22222222])
```

In [57]:

```
a = np.hstack((a, b))
print(a)
```

```
[1.         3.         5.         7.         9.         1.18551961
 1.26491106 1.6066895  2.         2.22222222]
```

Расщепление массива в позициях 3 и 6.

In [58]:

```
np.hsplit(a, [3, 6])
```

Out[58]:

```
[array([1., 3., 5.]),
 array([7.        , 9.        , 1.18551961]),
 array([1.26491106, 1.6066895 , 2.        , 2.22222222])]
```

Функции `delete`, `insert` и `append` не меняют массив на месте, а возвращают новый массив, в котором удалены, вставлены в середину или добавлены в конец какие-то элементы.

In [59]:

```
a = np.delete(a, [5, 7])
print(a)
```

```
[1.         3.         5.         7.         9.         1.26491106
 2.         2.22222222]
```

In [60]:

```
a = np.insert(a, 2, [0, 0])
print(a)
```

```
[1.         3.         0.         0.         5.         7.
 9.         1.26491106 2.         2.22222222]
```

In [61]:

```
a = np.append(a, [1, 2, 3])
print(a)
```

```
[1.         3.         0.         0.         5.         7.
 9.         1.26491106 2.         2.22222222 1.         2.
 3.        ]
```

2.3 Способы индексации массивов

Есть несколько способов индексации массива. Вот обычный индекс.

In [62]:

```
a = np.linspace(0, 1, 11)
print(a)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

In [63]:

```
b = a[2]
print(b)
```

```
0.2
```

Диапазон индексов. Создаётся новый заголовок массива, указывающий на те же данные. Изменения, сделанные через такой массив, видны и в исходном массиве.

In [64]:

```
b = a[2:6]
print(b)
```

```
[0.2 0.3 0.4 0.5]
```

In [65]:

```
b[0] = -0.2
print(b)
```

```
[-0.2  0.3  0.4  0.5]
```

In [66]:

```
print(a)
```

```
[ 0.   0.1 -0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

Диапазон с шагом 2.

In [67]:

```
b = a[1:10:2]
print(b)
```

```
[0.1 0.3 0.5 0.7 0.9]
```

In [68]:

```
b[0] = -0.1
print(a)
```

```
[ 0.  -0.1 -0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

Массив в обратном порядке.

In [69]:

```
b = a[::-1]
print(b)
```

```
[ 1.   0.9  0.8  0.7  0.6  0.5  0.4  0.3 -0.2 -0.1  0. ]
```

Подмассиву можно присвоить значение - массив правильного размера или скаляр.

In [70]:

```
a[1:10:3] = 0
print(a)
```

```
[ 0.   0.  -0.2  0.3  0.   0.5  0.6  0.   0.8  0.9  1. ]
```

Тут опять создаётся только новый заголовок, указывающий на те же данные.

In [71]:

```
b = a[:]
b[1] = 0.1
print(a)
```

```
[ 0.   0.1 -0.2  0.3  0.   0.5  0.6  0.   0.8  0.9  1. ]
```

Чтобы скопировать и данные массива, нужно использовать метод `copy` .

In [72]:

```
b = a.copy()
b[2] = 0
print(b)
print(a)
```

```
[0.  0.1 0.  0.3 0.  0.5 0.6 0.  0.8 0.9 1. ]
[ 0.   0.1 -0.2  0.3  0.   0.5  0.6  0.   0.8  0.9  1. ]
```

Можно задать список индексов.

In [73]:

```
print(a[[2, 3, 5]])
```

```
[-0.2  0.3  0.5]
```

Можно задать булев массив той же величины.

In [74]:

```
b = a > 0
print(b)
```

```
[False  True False  True False  True  True False  True  True  True]
```

In [75]:

```
print(a[b])
```

```
[0.1 0.3 0.5 0.6 0.8 0.9 1. ]
```

In [76]:

```
a
```

Out[76]:

```
array([ 0. ,  0.1, -0.2,  0.3,  0. ,  0.5,  0.6,  0. ,  0.8,  0.9,
1. ])
```

In [77]:

```
b
```

Out[77]:

```
array([False,  True, False,  True, False,  True,  True, False,  Tru
e,
        True,  True])
```

Упражнение:
1)Создайте массив чисел от -2π до 2π
2)Посчитайте сумму поэлементных квадратов синуса и косинуса для данного массива
3)С помощью `np.all` проверьте, что в ответе только единицы

In [78]:

```
# решение

x = np.linspace(-2 * np.pi, 2 * np.pi, 20)
np.all((np.sin(x)**2 + np.cos(x)**2).round() == 1)
```

Out[78]:

```
True
```

3. Двумерные массивы

3.1 Создание, простые операции

In [79]:

```
a = np.array([[0.0, 1.0], [-1.0, 0.0]])
print(a)
```

```
[[ 0.  1.]
 [-1.  0.]]
```

In [80]:

```
a.ndim
```

Out[80]:

```
2
```

In [81]:

```
a.shape
```

Out[81]:

```
(2, 2)
```

In [82]:

```
len(a), a.size
```

Out[82]:

```
(2, 4)
```

In [83]:

```
a[1, 0]
```

Out[83]:

```
-1.0
```

Атрибуту `shape` можно присвоить новое значение - кортеж размеров по всем координатам. Получится новый заголовок массива; его данные не изменятся.

In [84]:

```
b = np.linspace(0, 3, 4)
print(b)
```

```
[0. 1. 2. 3.]
```

In [85]:

```
b.shape
```

Out[85]:

```
(4,)
```

In [86]:

```
b.shape = 2, 2
print(b)
```

```
[[0. 1.]
 [2. 3.]]
```

Можно растянуть в одномерный массив

In [87]:

```
print(b.ravel())
```

```
[0. 1. 2. 3.]
```

Арифметические операции поэлементные

In [88]:

```
print(a + 1)
print(a * 2)
print(a + [0, 1])  # второе слагаемое дополняется до матрицы копированием строк
print(a + np.array([[0, 2]]).T)  # .T - транспонирование
print(a + b)
```

```
[[1. 2.]
 [0. 1.]]
[[ 0.  2.]
 [-2.  0.]]
[[ 0.  2.]
 [-1.  1.]]
[[0. 1.]
 [1. 2.]]
[[0. 2.]
 [1. 3.]]
```

3.2 Работа с матрицами

Поэлементное и матричное (только в Python >=3.5) умножение.

In [89]:

```
print(a * b)
```

```
[[ 0.  1.]
 [-2.  0.]]
```

In [90]:

```
print(a @ b)
```

```
[[ 2.  3.]
 [ 0. -1.]]
```

In [91]:

```
print(b @ a)
```

```
[[-1.  0.]
 [-3.  2.]]
```

Упражнение: создайте две матрицы $\left(\begin{pmatrix} -3 & 4 \\ 4 & 3 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\right)$. Посчитайте их поэлементное и матричное произведения.

In [92]:

```
# решение
a = np.array([[-3, 4], [4, 3]])
b = np.array([[2, 1], [1, 2]])
print(a * b)
print(b * a)
print(a @ b)
print(b @ a)
```

```
[[-6  4]
 [ 4  6]]
[[-6  4]
 [ 4  6]]
[[-2  5]
 [11 10]]
[[-2 11]
 [ 5 10]]
```

Умножение матрицы на вектор.

In [93]:

```
v = np.array([1, -1], dtype=np.float64)
print(b @ v)
```

```
[ 1. -1.]
```

In [94]:

```
print(v @ b)
```

```
[ 1. -1.]
```

Если у вас Питон более ранней версии, то для работы с матрицами можно использовать класс `np.matrix`, в котором операция умножения реализуется как матричное умножение.

In [99]:

```
np.matrix(a) * np.matrix(b)
```

Out[99]:

```
matrix([[-2,  5],
        [11, 10]])
```

Внешнее произведение $a_{ij} = u_i v_j$

In [100]:

```
u = np.linspace(1, 2, 2)
v = np.linspace(2, 4, 3)
print(u)
print(v)
```

```
[1. 2.]
[2. 3. 4.]
```

In [101]:

```
a = np.outer(u, v)
print(a)
```

```
[[2. 3. 4.]
 [4. 6. 8.]]
```

Двумерные массивы, зависящие только от одного индекса: $x_{ij} = u_j$, $y_{ij} = v_i$

In [102]:

```
x, y = np.meshgrid(u, v)
print(x)
print(y)
```

```
[[1. 2.]
 [1. 2.]
 [1. 2.]]
[[2. 2.]
 [3. 3.]
 [4. 4.]]
```

Единичная матрица.

In [103]:

```
I = np.eye(4)
print(I)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Метод `reshape` делает то же самое, что присваивание атрибуту `shape`.

In [104]:

```
print(I.reshape(16))
```

```
[1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1.]
```

In [105]:

```
print(I.reshape(2, 8))
```

```
[[1. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 1.]]
```

Строка.

In [106]:

```
print(I[1])
```

```
[0. 1. 0. 0.]
```

Цикл по строкам.

In [107]:

```
for row in I:
    print(row)
```

```
[1. 0. 0. 0.]
[0. 1. 0. 0.]
[0. 0. 1. 0.]
[0. 0. 0. 1.]
```

Столбец.

In [108]:

```
print(I[:, 2])
```

```
[0. 0. 1. 0.]
```

Подматрица.

In [109]:

```
print(I[0:2, 1:3])
```

```
[[0. 0.]
 [1. 0.]]
```

Можно построить двумерный массив из функции.

In [110]:

```
def f(i, j):
    print(i)
    print(j)
    return 10 * i + j

print(np.fromfunction(f, (4, 4), dtype=np.int64))
```

```
[[0 0 0 0]
 [1 1 1 1]
 [2 2 2 2]
 [3 3 3 3]]
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]]
```

Транспонированная матрица.

In [111]:

```
print(b.T)
```

```
[[2 1]
 [1 2]]
```

Соединение матриц по горизонтали и по вертикали.

In [112]:

```
a = np.array([[0, 1], [2, 3]])
b = np.array([[4, 5, 6], [7, 8, 9]])
c = np.array([[4, 5], [6, 7], [8, 9]])
print(a)
print(b)
print(c)
```

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

In [113]:

```
print(np.hstack((a, b)))
```

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

In [114]:

```
print(np.vstack((a, c)))
```

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

Сумма всех элементов; суммы столбцов; суммы строк.

In [115]:

```
print(b.sum())
print(b.sum(axis=0))
print(b.sum(axis=1))
```

```
39
[11 13 15]
[15 24]
```

Аналогично работают `prod` , `max` , `min` и т.д.

In [116]:

```
print(b.max())
print(b.max(axis=0))
print(b.min(axis=1))
```

```
9
[7 8 9]
[4 7]
```

След - сумма диагональных элементов.

In [117]:

```
np.trace(a)
```

Out[117]:

```
3
```

Упражнение: в статистике и машинном обучении часто приходится иметь с функцией RSS, которая вычисляется по формуле $\sum_{i=1}^{n}(y_i - a_i)^2$, где y_i - координаты одномерного вектора y, a_i - координаты одномерного вектора a. Посчитайте RSS для $y = (1, 2, 3, 4, 5), a = (3, 2, 1, 0, -1)$

In [118]:

```
# решение

y = np.arange(1, 6)
a = np.arange(3, -2, -1)
rss = np.sum((y - a)**2)
```

4. Тензоры (многомерные массивы)

In [119]:

```
X = np.arange(24).reshape(2, 3, 4)
print(X)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Суммирование (аналогично остальные операции)

In [120]:

```
# суммируем только по нулевой оси, то есть для фиксированных j и k
# суммируем только элементы с индексами (*, j, k)
print(X.sum(axis=0))
# суммируем сразу по двум осям, то есть для фиксированной i
# суммируем только элементы с индексами (i, *, *)
print(X.sum(axis=(1, 2)))
```

```
[[12 14 16 18]
 [20 22 24 26]
 [28 30 32 34]]
[ 66 210]
```

5. Линейная алгебра

In [121]:

```
a = np.array([[0, 1], [2, 3]])
```

In [122]:

```
np.linalg.det(a)
```

Out[122]:

```
-2.0
```

Обратная матрица.

In [123]:

```
a1 = np.linalg.inv(a)
print(a1)
```

```
[[-1.5  0.5]
 [ 1.   0. ]]
```

In [124]:

```
print(a @ a1)
print(a1 @ a)
```

```
[[1. 0.]
 [0. 1.]]
[[1. 0.]
 [0. 1.]]
```

Решение линейной системы $au = v$.

In [125]:

```
v = np.array([0, 1], dtype=np.float64)
print(a1 @ v)
```

```
[0.5 0. ]
```

In [126]:

```
u = np.linalg.solve(a, v)
print(u)
```

```
[0.5 0. ]
```

Проверим.

In [127]:

```
print(a @ u - v)
```

```
[0. 0.]
```

Собственные значения и собственные векторы: $au_i = \lambda_i u_i$. `l` - одномерный массив собственных значений λ_i, столбцы матрицы u - собственные векторы u_i.

In [128]:

```
l, u = np.linalg.eig(a)
print(l)
```

```
[-0.56155281  3.56155281]
```

In [129]:

```
print(u)
```

```
[[-0.87192821 -0.27032301]
 [ 0.48963374 -0.96276969]]
```

Проверим.

In [130]:

```
for i in range(2):
    print(a @ u[:, i] - l[i] * u[:, i])
```

```
[0.00000000e+00 1.66533454e-16]
[ 0.0000000e+00 -4.4408921e-16]
```

Функция `diag` от одномерного массива строит диагональную матрицу; от квадратной матрицы - возвращает одномерный массив её диагональных элементов.

In [131]:

```
L = np.diag(l)
print(L)
print(np.diag(L))
```

```
[[-0.56155281  0.        ]
 [ 0.          3.56155281]]
[-0.56155281  3.56155281]
```

Все уравнения $au_i = \lambda_i u_i$ можно собрать в одно матричное уравнение $au = u\Lambda$, где Λ - диагональная матрица с собственными значениями λ_i по диагонали.

In [132]:

```
print(a @ u - u @ L)
```

```
[[ 0.00000000e+00  0.00000000e+00]
 [ 1.66533454e-16 -4.44089210e-16]]
```

Поэтому $u^{-1}au = \Lambda$.

In [133]:

```
print(np.linalg.inv(u) @ a @ u)
```

```
[[-5.61552813e-01  2.77555756e-17]
 [-2.22044605e-16  3.56155281e+00]]
```

Найдём теперь левые собственные векторы $v_i a = \lambda_i v_i$ (собственные значения λ_i те же самые).

In [134]:

```
l, v = np.linalg.eig(a.T)
print(l)
print(v)
```

```
[-0.56155281  3.56155281]
[[-0.96276969 -0.48963374]
 [ 0.27032301 -0.87192821]]
```

Собственные векторы нормированы на 1.

In [135]:

```
print(u.T @ u)
print(v.T @ v)
```

```
[[ 1.         -0.23570226]
 [-0.23570226  1.        ]]
[[1.         0.23570226]
 [0.23570226 1.        ]]
```

Левые и правые собственные векторы, соответствующие разным собственным значениям, ортогональны, потому что $v_i a u_j = \lambda_i v_i u_j = \lambda_j v_i u_j$.

In [136]:

```
print(v.T @ u)
```

```
[[ 9.71825316e-01  0.00000000e+00]
 [-5.55111512e-17  9.71825316e-01]]
```

Упражнение: в машинном обучении есть модель линейной регрессии, для которой "хорошее" решение считается по следующей формуле: $\widehat{\theta} = (X^T \cdot X + \lambda \cdot I_n)^{-1} \cdot X^T y$. Вычислите $\widehat{\theta}$ для $X = \begin{pmatrix} -3 & 4 & 1 \\ 4 & 3 & 1 \end{pmatrix}$, $y = \begin{pmatrix} 10 \\ 12 \end{pmatrix}$, I_n - единичная матрица размерности 3, $\lambda = 0.1$

In [137]:

```
# решение

X = np.array([[-3, 4, 1], [4, 3, 1]])
y = np.array([10, 12])
I = np.eye(3)
lambd = 0.1
theta = np.linalg.inv(X.T @ X + lambd * I) @ X.T @ y
```

6. Интегрирование

In [138]:

```
from scipy.integrate import quad, odeint
from scipy.special import erf
```

In [139]:

```
def f(x):
    return np.exp(-x ** 2)
```

Адаптивное численное интегрирование (может быть до бесконечности). `err` - оценка ошибки.

In [140]:

```
res, err = quad(f, 0, np.inf)
print(np.sqrt(np.pi) / 2, res, err)
```

```
0.8862269254527579 0.8862269254527579 7.101318390472462e-09
```

In [141]:

```
res, err =  quad(f, 0, 1)
print(np.sqrt(np.pi) / 2 * erf(1), res, err)
```

```
0.7468241328124269 0.7468241328124271 8.291413475940725e-15
```

7. Сохранение в файл и чтение из файла

In [142]:

```
x = np.arange(0, 25, 0.5).reshape((5, 10))

# Сохраняем в файл example.txt данные x в формате с двумя точками после запятой и раздели
телем ';'
np.savetxt('example.txt', x, fmt='%.2f', delimiter=';')
```

Получится такой файл

In [143]:

```
! cat example.txt
```

```
0.00;0.50;1.00;1.50;2.00;2.50;3.00;3.50;4.00;4.50
5.00;5.50;6.00;6.50;7.00;7.50;8.00;8.50;9.00;9.50
10.00;10.50;11.00;11.50;12.00;12.50;13.00;13.50;14.00;14.50
15.00;15.50;16.00;16.50;17.00;17.50;18.00;18.50;19.00;19.50
20.00;20.50;21.00;21.50;22.00;22.50;23.00;23.50;24.00;24.50
```

Теперь его можно прочитать

In [144]:

```
x = np.loadtxt('example.txt', delimiter=';')
print(x)
```

```
[[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5]
 [ 5.   5.5  6.   6.5  7.   7.5  8.   8.5  9.   9.5]
 [10.  10.5 11.  11.5 12.  12.5 13.  13.5 14.  14.5]
 [15.  15.5 16.  16.5 17.  17.5 18.  18.5 19.  19.5]
 [20.  20.5 21.  21.5 22.  22.5 23.  23.5 24.  24.5]]
```

8. Производительность numpy

Посмотрим на простой пример --- сумма первых 10^8 чисел.

In [145]:

```
%%time

sum_value = 0
for i in range(10 ** 8):
    sum_value += i
print(sum_value)
```

```
4999999950000000
CPU times: user 9.29 s, sys: 8.9 ms, total: 9.3 s
Wall time: 9.3 s
```

Немного улучшеный код

In [146]:

```
%%time

sum_value = sum(range(10 ** 8))
print(sum_value)
```

```
4999999950000000
CPU times: user 1.62 s, sys: 6.77 ms, total: 1.63 s
Wall time: 1.62 s
```

Код с использованием функций библиотеки numpy

In [147]:

```
%%time

sum_value = np.arange(10 ** 8).sum()
print(sum_value)
```

```
4999999950000000
CPU times: user 254 ms, sys: 242 ms, total: 496 ms
Wall time: 495 ms
```

Простой и понятный код работает в 30 раз быстрее!

Посмотрим на другой пример. Сгенерируем матрицу размера 500×1000, и вычислим средний минимум по колонкам.

Простой код, но при этом даже использующий некоторые питон-функции

Замечание. Далее с помощью `scipy.stats` происходит генерация случайных чисел из равномерного распределения на отрезке $[0, 1]$. Этот модуль будем изучать на следующем занятии.

In [148]:

```
import scipy.stats as sps
```

In [149]:

```
%%time

N, M = 500, 1000
matrix = []
for i in range(N):
    matrix.append([sps.uniform.rvs() for j in range(M)])

min_col = [min([matrix[i][j] for i in range(N)]) for j in range(M)]
mean_min = sum(min_col) / N
print(mean_min)
```

```
0.00400095222563674
CPU times: user 16.7 s, sys: 237 ms, total: 16.9 s
Wall time: 16.9 s
```

Понятный код с использованием функций библиотеки numpy

In [150]:

```
%%time

N, M = 500, 1000
matrix = sps.uniform.rvs(size=(N, M))
mean_min = matrix.min(axis=1).mean()
print(mean_min)
```

```
0.0009739207148634659
CPU times: user 184 ms, sys: 10.3 ms, total: 194 ms
Wall time: 41.1 ms
```

Простой и понятный код работает в 1500 раз быстрее!

Введение в анализ данных, 2020

Никита Волков

https://mipt-stats.gitlab.io/ (https://mipt-stats.gitlab.io/)

На основе http://www.inp.nsk.su/~grozin/python/ (http://www.inp.nsk.su/~grozin/python/)