**Библиотека Numpy**

**Введение**

NumPy это open-source модуль для python, который предоставляет общие математические и числовые операции в виде пре-скомпилированных, быстрых функций. Они объединяются в высокоуровневые пакеты. Они обеспечивают функционал, который можно сравнить с функционалом MatLab. NumPy (Numeric Python) предоставляет базовые методы для манипуляции с большими массивами и матрицами.

Таким образом, 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.

**Импорт модуля numpy**

Есть несколько путей импорта. Стандартный метод это — использовать простое выражение:

| **import** numpy |
| --- |

Тем не менее, для большого количества вызовов функций numpy, становится утомительно писать numpy.X снова и снова. Вместо этого намного легче сделать это так:

| **import** numpy **as** np |
| --- |

Это выражение позволяет нам получать доступ к numpy объектам используя np.X вместо numpy.X. Также можно импортировать numpy прямо в используемое пространство имен, чтобы вообще не использовать функции через точку, а вызывать их напрямую:

| **from** numpy **import** \* |
| --- |

Однако, этот вариант не приветствуется в программировании на python, так как убирает некоторые полезные структуры, которые модуль предоставляет. До конца этого урока мы будем использовать второй вариант импорта (import numpy as np).

## **Начинаем работу**

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, float8, float16, float32, float64, 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, используя функцию numpy.array() (запомните: array - функция, создающая объект типа ndarray):

| import numpy as np  a = np.array([0, 2, 1])  print(a, type(a))  b = np.array([1, 4, 5, 8], float)  print(b, type(b)) |
| --- |

Рузультат:

| [0 2 1] <class 'numpy.ndarray'>  [1. 4. 5. 8.] <class 'numpy.ndarray'> |
| --- |

Здесь на втором примере (Массив b) функция array принимает два аргумента: список для конвертации в массив и тип для каждого элемента. Ко всем элементам можно получить доступ и манипулировать ими так же, как вы бы это делали с обычными списками:

| import numpy as np  b = np.array([1, 4, 5, 8], float)  print(b, type(b))  print(b[:2])  print(b[3])  b[0]=20  print(b) |
| --- |

Рузультат:

| [1. 4. 5. 8.] <class 'numpy.ndarray'>  [1. 4.]  8.0  [20. 4. 5. 8.] |
| --- |

Массивы могут быть и многомерными. В отличии от списков можно использовать запятые в скобках. Вот пример двумерного массива (матрица):

| import numpy as np  a = np.array([[1, 2, 3], [4, 5, 6]], float)  print(a)  print(a[0,0])  print(a[0,1]) |
| --- |

Результат:

| [[1. 2. 3.]  [4. 5. 6.]]  1.0  2.0 |
| --- |

Array slicing работает с многомерными массивами аналогично, как и с одномерными, применяя каждый срез, как фильтр для установленного измерения. Используйте ":" в измерении для указывания использования всех элементов этого измерения:

| import numpy as np  a = np.array([[1, 2, 3], [4, 5, 6]], float)  print(a)  print(a[1,:])  print(a[:,2])  print(a[-1:, -2:]) |
| --- |

Результат:

| [[1. 2. 3.]  [4. 5. 6.]]  [4. 5. 6.]  [3. 6.]  [[5. 6.]] |
| --- |

Метод shape возвращает количество строк и столбцов в матрице:

| import numpy as np  a = np.array([[1, 2, 3], [4, 5, 6]], float)  print(a)  print(a.shape) |
| --- |

Результат:

| [[1. 2. 3.]  [4. 5. 6.]]  (2, 3) |
| --- |

Метод dtype возвращает тип переменных, хранящихся в массиве:

| import numpy as np  a = np.array([[1, 2, 3], [4, 5, 6]], float)  print(a)  print(a.dtype) |
| --- |

Результат:

| [[1. 2. 3.]  [4. 5. 6.]]  float64 |
| --- |

Тут float64, это числовой тип данных в numpy, который используется для хранения вещественных чисел двойной точности. Так же как float в Python.

Метод len возвращает длину первого измерения (оси):

| import numpy as np  a = np.array([[1, 2, 3], [4, 5, 6]], float)  print(a)  print(len(a))  print(len(a[0])) |
| --- |

Результат:

| [[1. 2. 3.]  [4. 5. 6.]]  2  3 |
| --- |

Метод in используется для проверки на наличие элемента в массиве:

| import numpy as np  a = np.array([[1, 2, 3], [4, 5, 6]], float)  print(a)  print(2 in a)  print(0 in a) |
| --- |

Результат:

| [[1. 2. 3.]  [4. 5. 6.]]  True  False |
| --- |

Массивы можно переформировать при помощи метода, который задает новый многомерный массив. Следуя следующему примеру, мы переформатируем одномерный массив из десяти элементов во двумерный массив, состоящий из пяти строк и двух столбцов:

| import numpy as np  a = np.array(range(10), float)  print(a)  a = a.reshape((5, 2))  print(a)  print(a.shape) |
| --- |

Результат:

| [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]  [[0. 1.]  [2. 3.]  [4. 5.]  [6. 7.]  [8. 9.]]  (5, 2) |
| --- |

Обратите внимание, метод reshape создает новый массив, а не модифицирует оригинальный.

Имейте ввиду, связывание имен в python работает и с массивами. Метод copy используется для создания копии существующего массива в памяти:

| import numpy as np  a = np.array([1, 2, 3], float)  b = a  c =  a.copy()  a[0] = 0  print(a)  print(b)  print(c)  a[0]=7  print(b)  print(b)  print(c) |
| --- |

Результат:

| [0. 2. 3.]  [0. 2. 3.]  [1. 2. 3.]  [7. 2. 3.]  [7. 2. 3.]  [1. 2. 3.] |
| --- |

Списки можно тоже создавать с массивов:

| import numpy as np  a = np.array([1, 2, 3], float)  print(a.tolist())  print(list(a)) |
| --- |

Результат:

| [1.0, 2.0, 3.0]  [1.0, 2.0, 3.0] |
| --- |

Можно также переконвертировать массив в бинарную строку (то есть, не human-readable форму). Используйте метод tostring для этого. Метод fromstring работает в для обратного преобразования. Эти операции иногда полезны для сохранения большого количества данных в файлах, которые могут быть считаны в будущем.

| import numpy as np  a = np.array([1, 2, 3], float)  s = a.tostring()  print(s)  print(np.fromstring(s)) |
| --- |

Результат:

| b'\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08@'  [1. 2. 3.] |
| --- |

Заполнение массива одинаковым значением.

| import numpy as np  a = np.array([1, 2, 3], float)  print(a)  a.fill(20)  print(a) |
| --- |

Результат:

| [1. 2. 3.]  [20. 20. 20.] |
| --- |

Транспонирование массивов также возможно, при этом создается новый массив:

| import numpy as np  a = np.array(range(6), float).reshape((2, 3))  print(a)  print("Transpose:")  print(a.transpose()) |
| --- |

Результат:

| [[0. 1. 2.]  [3. 4. 5.]]  Transpose:  [[0. 3.]  [1. 4.]  [2. 5.]] |
| --- |

Многомерный массив можно переконвертировать в одномерный при помощи метода flatten:

| import numpy as np  a = np.array([[1, 2, 3], [4, 5, 6]], float)  print(a)  print("Flatten:")  b=a.flatten()  print(b) |
| --- |

Результат:

| [[1. 2. 3.]  [4. 5. 6.]]  Flatten:  [1. 2. 3. 4. 5. 6.] |
| --- |

Два или больше массивов можно сконкатенировать при помощи метода concatenate:

| import numpy as np  a = np.array([1,2], float)  b = np.array([3,4,5,6], float)  c = np.array([7,8,9], float)  print("A:")  print(a)  print("B:")  print(b)  print("C:")  print(c)  print("Concatenate:")  d=np.concatenate((a, b, c))  print(d) |
| --- |

Результат:

| A:  [1. 2.]  B:  [3. 4. 5. 6.]  C:  [7. 8. 9.]  Concatenate:  [1. 2. 3. 4. 5. 6. 7. 8. 9.] |
| --- |

Если массив не одномерный, можно задать ось, по которой будет происходить соединение. По умолчанию (не задавая значения оси), соединение будет происходить по первому измерению:

| import numpy as np  a = np.array([[1, 2], [3, 4]], float)  b = np.array([[5, 6], [7,8]], float)  print("A:")  print(a)  print("B:")  print(b)  print("Concatenate:")  c=np.concatenate((a,b))  print(c)  print("Concatenate axis=0:")  d=np.concatenate((a,b), axis=0)  print(d)  print("Concatenate axis=1:")  e=np.concatenate((a,b), axis=1)  print(e) |
| --- |

Результат:

| A:  [[1. 2.]  [3. 4.]]  B:  [[5. 6.]  [7. 8.]]  Concatenate:  [[1. 2.]  [3. 4.]  [5. 6.]  [7. 8.]]  Concatenate axis=0:  [[1. 2.]  [3. 4.]  [5. 6.]  [7. 8.]]  Concatenate axis=1:  [[1. 2. 5. 6.]  [3. 4. 7. 8.]] |
| --- |

В заключении, размерность массива может быть увеличена при использовании константы newaxis в квадратных скобках:

| import numpy as np  a = np.array([[1, 2], [3, 4]], float)  b = np.array([[5, 6], [7,8]], float)  print("A:")  print(a)  print("B:")  print(b)  print("a[:,np.newaxis]:")  print(a[:,np.newaxis])  print("a[:,np.newaxis].shape:")  print(a[:,np.newaxis].shape)  print("[np.newaxis,:]")  print( b[np.newaxis,:])  print("b[np.newaxis,:].shape")  print(b[np.newaxis,:].shape) |
| --- |

Результат:

| A:  [[1. 2.]  [3. 4.]]  B:  [[5. 6.]  [7. 8.]]  a[:,np.newaxis]:  [[[1. 2.]]  [[3. 4.]]]  a[:,np.newaxis].shape:  (2, 1, 2)  [np.newaxis,:]  [[[5. 6.]  [7. 8.]]]  b[np.newaxis,:].shape  (1, 2, 2) |
| --- |

Заметьте, тут каждый массив двумерный; созданный при помощи newaxis имеет размерность один. Метод newaxis подходит для удобного создания надлежаще-мерных массивов в векторной и матричной математике.

Другие пути создания массивов

Функция arange аналогична функции range, но возвращает массив:

| import numpy as np  a=np.arange(5, dtype=float)  print(a)  print()  b=np.arange(1, 10, 2, dtype=int)  print(b) |
| --- |

Результат:

| [0. 1. 2. 3. 4.]  [1 3 5 7 9] |
| --- |

Функции zeros и ones создают новые массивы с установленной размерностью, заполненные этими значениями. Это, наверное, самые простые в использовании функции для создания массивов:

| import numpy as np  a=np.ones((2,3), dtype=float)  b=np.zeros(7, dtype=int)  print(a)  print(b) |
| --- |

Результат:

| [[1. 1. 1.]  [1. 1. 1.]]  [0 0 0 0 0 0 0] |
| --- |

Функции zeros\_like и ones\_like могут преобразовать уже созданный массив, заполнив его нулями и единицами соответственно:

| import numpy as np  a = np.array([[1, 2, 3], [4, 5, 6]], float)  b=np.zeros\_like(a)  c= np.ones\_like(a)  print("A:")  print(a)  print("B:")  print(b)  print("C:")  print(c) |
| --- |

Результат:

| A:  [[1. 2. 3.]  [4. 5. 6.]]  B:  [[0. 0. 0.]  [0. 0. 0.]]  C:  [[1. 1. 1.]  [1. 1. 1.]] |
| --- |

Также есть некоторое количество функций для создания специальных матриц. Для создания квадратной матрицы с главной диагональю, которая заполненная единицами, воспользуемся методом identity:

| import numpy as np  a=np.identity(4, dtype=float)  print(a) |
| --- |

Результат:

| [[1. 0. 0. 0.]  [0. 1. 0. 0.]  [0. 0. 1. 0.]  [0. 0. 0. 1.]] |
| --- |

Функция eye возвращает матрицу с единичками на к-атой диагонали:

| import numpy as np  a=np.eye(4, k=1, dtype=float)  print(a) |
| --- |

Результат:

| [[0. 1. 0. 0.]  [0. 0. 1. 0.]  [0. 0. 0. 1.]  [0. 0. 0. 0.]] |
| --- |

**Математические операции над массивами**

Когда для массивов мы используем стандартные математические операции, должен соблюдаться принцип: элемент--элемент. Это означает, что массивы должны быть одинакового размера во время сложения, вычитания и тому подобных операций:

| import numpy as np  a = np.array([1,2,3], float)  b = np.array([5,2,6], float)  print("A:")  print(a)  print("B:")  print(b)  print("A+B:")  print(a+b)  print("A-B:")  print(a-b)  print("A\*B:")  print(a\*b)  print("A/B:")  print(a/b)  print("A%B:")  print(a%b)  print("B^A:")  print(b\*\*a) |
| --- |

Результат:

| A:  [1. 2. 3.]  B:  [5. 2. 6.]  A+B:  [6. 4. 9.]  A-B:  [-4. 0. -3.]  A\*B:  [ 5. 4. 18.]  A/B:  [0.2 1. 0.5]  A%B:  [1. 0. 3.]  B^A:  [ 5. 4. 216.] |
| --- |

Для двухмерных массивов, умножение остается поэлементным и не соответствует умножению матриц. Для этого существуют специальные функции, которые мы изучим позже.

| import numpy as np  a = np.array([[1,2], [3,4]], float)  b = np.array([[2,0], [1,3]], float)  c=a \* b  print(c) |
| --- |

Результат:

| [[ 2. 0.]  [ 3. 12.]] |
| --- |

При несоответствии в размере выбрасываются ошибки:

| import numpy as np  a = np.array([1,2,3], float)  b = np.array([4,5], float)  c=a + b  print(c) |
| --- |

Результат:

| **Exception has occurred: ValueError**  operands could not be broadcast together with shapes (3,) (2,)  File "C:\Users\User\Desktop\Untitled-2.py", line 4, in <module> c=a + b |
| --- |

Однако, если размерность массивов не совпадает, они будут преобразованы для выполнения математических операций. Это зачастую означает, что меньший массив будет использован несколько раз для завершения операций. Рассмотрим такой пример:

| import numpy as np  a = np.array([[1, 2], [3, 4], [5, 6]], float)  b = np.array([-1, 3], float)  print("A:")  print(a)  print("B:")  print(b)  print("A+B")  print(a+b) |
| --- |

Результат:

| A:  [[1. 2.]  [3. 4.]  [5. 6.]]  B:  [-1. 3.]  A+B  [[0. 5.]  [2. 7.]  [4. 9.]] |
| --- |

Тут, одномерный массив b был преобразован в двухмерный, который соответствует размеру массива a. По существу, b был повторен несколько раз, для каждой «строки» a. Иначе его можно представить так:

| array([[-1., 3.],  [-1., 3.],  [-1., 3.]]) |
| --- |

Python автоматически преобразовывает массивы в этом случае. Иногда, однако, когда преобразование играет роль, мы можем использовать константу newaxis, чтобы изменить преобразование:

| import numpy as np  a = np.zeros((2,2), float)  b = np.array([-1., 3.], float)  print("A:")  print(a)  print("B:")  print(b)  print("A+B:")  print(a+b)  print("a + b[np.newaxis,:]:")  print(a + b[np.newaxis,:])  print("a + b[:,np.newaxis]:")  print(a + b[:,np.newaxis]) |
| --- |

Результат:

| A:  [[0. 0.]  [0. 0.]]  B:  [-1. 3.]  A+B:  [[-1. 3.]  [-1. 3.]]  a + b[np.newaxis,:]:  [[-1. 3.]  [-1. 3.]]  a + b[:,np.newaxis]:  [[-1. -1.]  [ 3. 3.]] |
| --- |

Вдобавок к стандартным операторам, в numpy включена библиотека стандартных математических функций, которые могут быть применены поэлементно к массивам. Собственно функции: abs, sign, sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, и arctanh.

| import numpy as np  a = np.array([1, 4, 9], float)  print("A:")  print(a)  print("sqrt(a):")  b=np.sqrt(a)  print(b) |
| --- |

Результат:

| A:  [1. 4. 9.]  sqrt(a):  [1. 2. 3.] |
| --- |

Функции floor, ceil и rint возвращают нижние, верхние или ближайшие (округлённое) значение:

| import numpy as np  a = np.array([1.1, 1.5, 1.9], float)  print("A:")  print(a)  print("floor(a):")  b=np.floor(a)  print(b)  print("ceil(a):")  c=np.ceil(a)  print(c)  print("rint(a):")  d=np.rint(a)  print(d) |
| --- |

Результат:

| A:  [1.1 1.5 1.9]  floor(a):  [1. 1. 1.]  ceil(a):  [2. 2. 2.]  rint(a):  [1. 2. 2.] |
| --- |

Также в numpy включены две важные математические константы:

| import numpy as np  print("PI:", np.pi)  print("E:", np.e) |
| --- |

Результат:

| PI: 3.141592653589793  E: 2.718281828459045 |
| --- |

**Перебор элементов массива**

Проводить итерацию массивов можно аналогично спискам:

| import numpy as np  a = np.array([1, 4, 5], int)  for x in a:      print(x) |
| --- |

Результат:

| 1  4  5 |
| --- |

Для многомерных массивов итерация будет проводиться по первой оси, так, что каждый проход цикла будет возвращать «строку» массива:

| import numpy as np  a = np.array([[1, 2], [3, 4], [5, 6]], float)  for x in a:      print(x) |
| --- |

Результат:

| [1. 2.]  [3. 4.]  [5. 6.] |
| --- |

Множественное присваивание также доступно при итерации:

| import numpy as np  a = np.array([[1, 2], [3, 4], [5, 6]], float)  for (x, y) in a:      print (x \* y) |
| --- |

Результат:

| 2.0  12.0  30.0 |
| --- |

**Базовые операции над массивами**

Для получения каких-либо свойств массивов существует много функций. Элементы могут быть суммированы или перемножены:

| import numpy as np  a = np.array([2, 4, 3], float)  S= a.sum()  P=a.prod()  print('S=',S, ' P=',P) |
| --- |

Результат:

| S= 9.0 P= 24.0 |
| --- |

В этом примере были использованы функции массива. Также можно использовать собственные функции numpy:

| import numpy as np  a = np.array([2, 4, 3], float)  S= np.sum(a)  P=np.prod(a)  print('S=',S, ' P=',P) |
| --- |

Результат:

| S= 9.0 P= 24.0 |
| --- |

Для большинства случаев могут использоваться оба варианта.  
Некие функции дают возможность оперировать статистическими данными. Это функции mean (среднее арифметическое), вариация и девиация:

| import numpy as np  a = np.array([2, 1, 9], float)  SA=a.mean()  Var=a.var()  Dev=a.std()  print("SA=", SA, " Var=", Var, " Dev=", Dev) |
| --- |

Результат:

| SA= 4.0 Var= 12.666666666666666 Dev= 3.559026084010437 |
| --- |

**Заметки**

Дисперсия или var представляет собой среднее квадратов отклонений от среднего, т . е.

Стандартное отклонение или std представляет собой квадратный корень из среднего значения квадратов отклонений от среднего значения, т . е. ,

где

Можно найти минимум и максимум в массиве:

| import numpy as np  a = np.array([2, 1, 9], float)  Min=a.min()  Max= a.max()  print("Min=", Min, "Max=", Max) |
| --- |

Результат:

| Min= 1.0 Max= 9.0 |
| --- |

Функции argmin и argmax возвращают индекс минимального или максимального элемента:

| import numpy as np  a = np.array([2, 1, 9,4,6,15], float)  Imin=a.argmin()  Imax= a.argmax()  print("Imin=", Imin, " Imax=", Imax) |
| --- |

Результат:

| Imin= 1 Imax= 5 |
| --- |

Для многомерных массивов каждая из функций может принять дополнительный аргумент axis и в зависимости от его значения выполнять функции по определенной оси, помещая результаты исполнения в массив:

| import numpy as np  a = np.array([[0, 2], [3, -1], [3, 5]], float)  print(a)  print()  M0=a.mean(axis=0)  M1=a.mean(axis=1)  print("Mean0=", M0)  print("Mean1=", M1)  Min1= a.min(axis=1)  Max0= a.max(axis=0)  print("Min1=", Min1)  print("Max0=", Max0) |
| --- |

Результат:

| [[ 0. 2.]  [ 3. -1.]  [ 3. 5.]]  Mean0= [2. 2.]  Mean1= [1. 1. 4.]  Min1= [ 0. -1. 3.]  Max0= [3. 5.] |
| --- |

Как и списки, массивы можно отсортировать:

| import numpy as np  a = np.array([6, 2, 5, -1, 0], float)  b=sorted(a)  print(b)  c=a.sort()  print(c) |
| --- |

Результат:

| [-1.0, 0.0, 2.0, 5.0, 6.0]  None |
| --- |

Значения в массиве могут быть «сокращены», чтобы принадлежать заданному диапазону. Это тоже самое что применять min(max(x, minval), maxval) к каждому элементу x:

| import numpy as np  a = np.array([6, 2, 5, -1, 0], float)  print(a)  b= a.clip(0, 5)  print(b) |
| --- |

Результат:

| [ 6. 2. 5. -1. 0.]  [5. 2. 5. 0. 0.] |
| --- |

Уникальные элементы могут быть извлечены вот так:

| import numpy as np  a = np.array([1, 1, 4, 5, 5, 5, 7], float)  print(a)  b=np.unique(a)  print(b) |
| --- |

Результат:

| [1. 1. 4. 5. 5. 5. 7.]  [1. 4. 5. 7.] |
| --- |

Для двухмерных массивов диагональ можно получить так:

| import numpy as np  a = np.array([[1, 2], [3, 4]], float)  print(a)  print()  b=a.diagonal()  print(b) |
| --- |

Результат:

| [[1. 2.]  [3. 4.]]  [1. 4.] |
| --- |

**Операторы сравнения и тестирование значений**

Булево сравнение может быть использовано для поэлементного сравнения массивов одинаковых длин. Возвращаемое значение это массив булевых True/False значений:

| import numpy as np  a = np.array([1, 3, 0], float)  b = np.array([0, 3, 2], float)  print("A:")  print(a)  print("B:")  print(b)  c=a > b  print("a>b:")  print(c)  d=a == b  print("a==b:")  print(d)  e=a <= b  print("a<=b:")  print(e) |
| --- |

Результат:

| A:  [1. 3. 0.]  B  [0. 3. 2.]  a>b:  [ True False False]  a==b:  [False True False]  a<=b:  [False True True] |
| --- |

Массивы могут быть сравнены с одиночным значением:

| import numpy as np  a = np.array([1, 3, 0], float)  print("A:")  print(a)  b=a > 2  print("a>2:")  print(b) |
| --- |

Результат:

| A:  [1. 3. 0.]  a>2:  [False True False] |
| --- |

Операторы any и all могут быть использованы для определения истинны ли хотя бы один или все элементы соответственно:

| import numpy as np  a = np.array([ True, False, False], bool)  print("A:")  print(a)  print("Any:")  b=any(a)  print(b)  print("All:")  c=all(a)  print(c) |
| --- |

Результат:

| True  False |
| --- |

Комбинированные булевы выражения могут быть применены к массивам по принципу элемент — элемент используя специальные функции logical\_and, logical\_or и logical\_not:

| import numpy as np  a = np.array([1, 3, 0], float)  aland=np.logical\_and(a > 0, a < 3)  print("A:")  print(a)  print("logical\_and(a > 0, a < 3)")  print(aland)  b = np.array([True, False, True], bool)  blnot= np.logical\_not(b)  print("B:")  print(b)  print("logical\_not(b)")  print(blnot)  c = np.array([False, True, False], bool)  clor=np.logical\_or(b, c)  print("C:")  print(c)  print("logical\_or(b, c)")  print(clor) |
| --- |

Результат:

| A:  [1. 3. 0.]  logical\_and(a > 0, a < 3)  [ True False False]  B:  [ True False True]  logical\_not(b)  [False True False]  C:  [False True False]  logical\_or(b, c)  [ True True True] |
| --- |

Функция where создает новый массив из двух других массивов одинаковых длин используя булев фильтр для выбора межу двумя элементами. Базовый синтаксис: where(boolarray, truearray, falsearray):

| import numpy as np  a = np.array([1, 3, 0], float)  b=np.where(a != 0, 1 / a, a)  print("A:")  print(a)  print("where(a != 0, 1 / a, a):")  print(b) |
| --- |

Результат:

| A:  [1. 3. 0.]  where(a != 0, 1 / a, a):  [1. 0.33333333 0. ] |
| --- |

С функцией where так же может быть реализовано «массовое сравнение»:

| import numpy as np  a = np.array([1, 3, 0], float)  b=np.where(a > 0, 3, 2)  print("A:")  print(a)  print("where(a > 0, 3, 2):")  print(b) |
| --- |

Результат:

| A:  [1. 3. 0.]  where(a > 0, 3, 2):  [3 3 2] |
| --- |

Некоторые функции дают возможность тестировать значения в массиве. Функция nonzero возвращает кортеж индексов ненулевых значений. Количество элементов в кортеже равно количеству осей в массиве:

| import numpy as np  a=np.array([[0, 1], [3, 0]], float)  b=a.nonzero()  print("A:")  print(a)  print("B:")  print(b) |
| --- |

Результат:

| A:  [[0. 1.]  [3. 0.]]  B:  (array([0, 1], dtype=int64), array([1, 0], dtype=int64)) |
| --- |

Также можно проверить значения на конечность и NaN(not a number):

| import numpy as np  a = np.array([1, np.NaN, np.Inf], float)  print("A:")  print(a)  b= np.isnan(a)  print("isNan(a):")  print(b)  c=np.isfinite(a)  print("isfinite(a):")  print(c) |
| --- |

Результат:

| A:  [ 1. nan inf]  isNan(a):  [False True False]  isfinite(a):  [ True False False] |
| --- |

Хотя здесь мы использовали константы numpy чтобы добавить значения NaN и бесконечность, они могут быть результатами применения стандартных математических операций.

**Упражнение:**

1). Создайте массив чисел от  до

2). Посчитайте сумму поэлементных квадратов синуса и косинуса для данного массива.

3). С помощью np.all проверьте, что в ответе только единицы.

**Решение:**

| import numpy as np  x = np.linspace(-2 \* np.pi, 2 \* np.pi, 20)  print("x:")  print(x)  y=np.sin(x)\*\*2 + np.cos(x)\*\*2  print("y:")  print(y)  z=np.all((np.sin(x)\*\*2 + np.cos(x)\*\*2).round() == 1)  print("z:")  print(z) |
| --- |

Рузультат:

| x:  [-6.28318531 -5.62179738 -4.96040945 -4.29902153 -3.6376336 -2.97624567  -2.31485774 -1.65346982 -0.99208189 -0.33069396 0.33069396 0.99208189  1.65346982 2.31485774 2.97624567 3.6376336 4.29902153 4.96040945  5.62179738 6.28318531]  y:  [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]  z:  True |
| --- |

**Выбор элементов массива и манипуляция с ними**

Мы уже видели, как и у списков, элементы массива можно получить используя операцию доступа по индексу. Однако, в отличии от списков, массивы также позволяют делать выбор элементов используя другие массивы. Это значит, что мы можем использовать массив для фильтрации специфических подмножеств элементов других массивов.

Булевы массивы могут быть использованы как массивы для фильтрации:

| import numpy as np  a = np.array([[6, 4], [5, 9]], float)  print("A:")  print(a)  b=a >= 6  print("a >= 6:")  print(b)  c= a[a >= 6]  print("a[a >= 6]:")  print(c) |
| --- |

Результат:

| A:  [[6. 4.]  [5. 9.]]  a >= 6:  [[ True False]  [False True]]  a[a >= 6]:  [6. 9.] |
| --- |

Стоит заметить, что когда мы передаем булев массив a>=6 как индекс для операции доступа по индексу массива a, возвращаемый массив будет хранить только True значения. Также мы можем записать массив для фильтрации в переменную:

| import numpy as np  a = np.array([[6, 4], [5, 9]], float)  print("A:")  print(a)  sel = (a >= 6)  print("a[a >= 6]:")  print(a[sel]) |
| --- |

Результат:

| A:  [[6. 4.]  [5. 9.]]  a[a >= 6]:  [6. 9.] |
| --- |

Более замысловатая фильтрация может быть достигнута использованием булевых выражений:

| import numpy as np  a = np.array([[6, 4], [5, 9]], float)  print("A:")  print(a)  b=a[np.logical\_and(a > 5, a < 9)]  print("logical\_and(a > 5, a < 9):")  print(b) |
| --- |

Результат:

| A:  [[6. 4.]  [5. 9.]]  logical\_and(a > 5, a < 9):  [6.] |
| --- |

В придачу к булеву выбору, также можно использовать целочисленные массивы. В этом случае, целочисленный массив хранит индексы элементов, которые будут взяты из массива. Рассмотрим следующий одномерный пример:

| import numpy as np  a= np.array([2, 4, 6, 8], float)  b = np.array([0, 0, 1, 3, 2, 1], int)  print("A:")  print(a)  print("B:")  print(b)  c=a[b]  print("a[b]:")  print(c) |
| --- |

Результат:

| A:  [2. 4. 6. 8.]  B:  [0 0 1 3 2 1]  a[b]:  [2. 2. 4. 8. 6. 4.] |
| --- |

Иными словами, когда мы используем b для получения элементов из a, мы берем 0-й, 0-й, 1-й, 3-й, 2-й и 1-й элементы a в этом порядке. Списки также могут быть использованы как массивы для фильтрации:

| import numpy as np  a= np.array([2, 4, 6, 8], float)  print("A:")  print(a)  c=a[[0, 0, 1, 3, 2, 1]]  print("a[[0, 0, 1, 3, 2, 1]]:")  print(c) |
| --- |

Результат:

| A:  [2. 4. 6. 8.]  a[[0, 0, 1, 3, 2, 1]]:  [2. 2. 4. 8. 6. 4.] |
| --- |

Для многомерных массивов, нам необходимо передать несколько одномерных целочисленных массивов в оператор доступа индексу (*Прим. переводчика: в нашем случае индексы это массивы*) для каждой оси. Потом каждый из массивов проходит такую последовательность: первый элемент соответствует индексу строки, который является первым элементом массива b, второй элемент соответствует индексу столбца, который является первым элементом массива c и так далее. (*Прим. переводчика: первый массив [2, 2] и второй [1, 4], имеем на выходе элементы с индексами [2, 1] и [2, 4]*) Пример:

| import numpy as np  a = np.array([[1, 4], [9, 16]], float)  b = np.array([0, 0, 1, 1, 0], int)  c = np.array([0, 1, 1, 1, 1], int)  print("A:")  print(a)  print("B:")  print(b)  print("C:")  print(c)  d=a[b,c]  print("a[b,c]:")  print(d) |
| --- |

Результат:

| A:  [[ 1. 4.]  [ 9. 16.]]  B:  [0 0 1 1 0]  C:  [0 1 1 1 1]  a[b,c]:  [ 1. 4. 16. 16. 4.] |
| --- |

Специальная функция take доступна для выполнения выборки с целочисленными массивами. Это работает также как и использования оператора взятия по индексу:

| import numpy as np  a = np.array([2, 4, 6, 8], float)  b = np.array([0, 0, 1, 3, 2, 1], int)  print("A:")  print(a)  print("B:")  print(b)  c= a.take(b)  print("a.take(b):")  print(c) |
| --- |

Результат:

| A:  [2. 4. 6. 8.]  B:  [0 0 1 3 2 1]  a.take(b):  [2. 2. 4. 8. 6. 4.] |
| --- |

Функция take также предоставляет аргумент axis (ось) для взятия подсекции многомерного массива вдоль какой-либо оси. (*Прим: по строкам или столбцам (для двумерных массивов)*).

| import numpy as np  a = np.array([[0, 1], [2, 3]], float)  b = np.array([0, 0, 1], int)  print("A:")  print(a)  print("B:")  print(b)  c=a.take(b, axis=0)  print("a.take(b, axis=0):")  print(c)  d=a.take(b, axis=1)  print("a.take(b, axis=1):")  print(d) |
| --- |

Результат:

| A:  [[0. 1.]  [2. 3.]]  B:  [0 0 1]  a.take(b, axis=0):  [[0. 1.]  [0. 1.]  [2. 3.]]  a.take(b, axis=1):  [[0. 0. 1.]  [2. 2. 3.]] |
| --- |

В противоположность к функции take есть функция put, которая будет брать значения из исходного массива и записывать их на специфические индексы в другом put-массиве.

| import numpy as np  a = np.array([0, 1, 2, 3, 4, 5], float)  b = np.array([9, 8, 7], float)  print("A:")  print(a)  print("B:")  print(b)  a.put([0, 3], b)  print("a.put([0, 3], b):")  print(a) |
| --- |

Результат:

| A:  [0. 1. 2. 3. 4. 5.]  B:  [9. 8. 7.]  a.put([0, 3], b):  [9. 1. 2. 8. 4. 5.] |
| --- |

Заметим, что значение 7 из исходного массива b не было использовано, так как только 2 индекса [0, 3] указаны. Исходный массив будет повторен если необходимо в случае не соответствия длин:

| import numpy as np  a = np.array([0, 1, 2, 3, 4, 5], float)  print("A:")  print(a)  a.put([0, 3], 5)  print("a.put([0, 3], 5):")  print(a) |
| --- |

Результат:

| A:  [0. 1. 2. 3. 4. 5.]  a.put([0, 3], 5):  [5. 1. 2. 5. 4. 5.] |
| --- |

**Векторная и матричная математика**

NumPy обеспечивает много функций для работы с векторами и матрицами. Функция dot возвращает скалярное произведение векторов:

| import numpy as np  a = np.array([1, 2, 3], float)  b = np.array([0, 1, 1], float)  c=np.dot(a, b)  print("A:")  print(a)  print("B:")  print(b)  print("np.dot(a, b):")  print(c) |
| --- |

Результат:

| A:  [1. 2. 3.]  B:  [0. 1. 1.]  np.dot(a, b):  5.0 |
| --- |

Функция dot также может умножать матрицы:

| import numpy as np  a = np.array([[0, 1], [2, 3]], float)  b = np.array([2, 3], float)  c = np.array([[1, 1], [4, 0]], float)  d=np.dot(b, a)  e=np.dot(a, b)  x= np.dot(a, c)  y= np.dot(c, a)  print("A:")  print(a)  print("\n B:")  print(b)  print("\n np.dot(b, a):")  print(d)  print("\n np.dot(a, b):")  print(e)  print("\n np.dot(a, c):")  print(x)  print("\n np.dot(c, a):")  print(y) |
| --- |

Результат:

| A:  [[0. 1.]  [2. 3.]]  B:  [2. 3.]  np.dot(b, a):  [ 6. 11.]  np.dot(a, b):  [ 3. 13.]  np.dot(a, c):  [[ 4. 0.]  [14. 2.]]  np.dot(c, a):  [[2. 4.]  [0. 4.]] |
| --- |

Также можно получить скалярное, тензорное и внешнее произведение матриц и векторов. Заметим, что для векторов внутреннее и скалярное произведение совпадает.

| import numpy as np  a = np.array([1, 4, 0], float)  b = np.array([2, 2, 1], float)  c=np.outer(a, b)  d=np.inner(a, b)  e= np.cross(a, b)  print("A:")  print(a)  print("\n B:")  print(b)  print("\n np.outer(a, b):")  print(c)  print("\n np.inner(a, b):")  print(d)  print("\n np.cross(a, b):")  print(e) |
| --- |

Результат:

| A:  [1. 4. 0.]  B:  [2. 2. 1.]  np.outer(a, b):  [[2. 2. 1.]  [8. 8. 4.]  [0. 0. 0.]]  np.inner(a, b):  10.0  np.cross(a, b):  [ 4. -1. -6.] |
| --- |

NumPy также предоставляет набор встроенных функций и методов для работы с линейной алгеброй. Это всё можно найти в под-модуле linalg. Этими модулями также можно оперировать с вырожденными и невырожденными матрицами. Определитель матрицы ищется таким образом:

| import numpy as np  a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)  d=np.linalg.det(a)  print("A:")  print(a)  print("\n np.linalg.det(a):")  print(d) |
| --- |

Результат:

| A:  [[4. 2. 0.]  [9. 3. 7.]  [1. 2. 1.]]  np.linalg.det(a):  -48.00000000000003 |
| --- |

Также можно найти собственный вектор и собственное значение матрицы:

| import numpy as np  a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)  vals, vecs = np.linalg.eig(a)  print("A:")  print(a)  print("\n vals:")  print(vals)  print("\n vecs:")  print(vecs) |
| --- |

Результат:

| A:  [[4. 2. 0.]  [9. 3. 7.]  [1. 2. 1.]]  vals:  [ 8.85591316 1.9391628 -2.79507597]  vecs:  [[-0.3663565 -0.54736745 0.25928158]  [-0.88949768 0.5640176 -0.88091903]  [-0.27308752 0.61828231 0.39592263]] |
| --- |

Невырожденная матрица (обратная матрица) может быть найдена так:

| import numpy as np  a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)  b = np.linalg.inv(a)  c=np.dot(a, b)  print("A:")  print(a)  print("\n np.linalg.inv(a):")  print(b)  print("\n np.dot(a, b):")  print(c) |
| --- |

Результат:

| A:  [[4. 2. 0.]  [9. 3. 7.]  [1. 2. 1.]]  np.linalg.inv(a):  [[ 0.22916667 0.04166667 -0.29166667]  [ 0.04166667 -0.08333333 0.58333333]  [-0.3125 0.125 0.125 ]]  np.dot(a, b):  [[1.00000000e+00 5.55111512e-17 0.00000000e+00]  [0.00000000e+00 1.00000000e+00 2.22044605e-16]  [0.00000000e+00 1.38777878e-17 1.00000000e+00]] |
| --- |

**linalg.matrix\_power**(M, n) - возводит матрицу в степень n.

| import numpy as np  a = np.array([[1, 3,-1], [5, 2,0], [0,5,1]], float)  b=np.linalg.matrix\_power(a, 2)  print("A:")  print(a)  print("\n np.matrix\_power(a, 2) :")  print(b) |
| --- |

Результат:

| A:  [[ 1. 3. -1.]  [ 5. 2. 0.]  [ 0. 5. 1.]]  np.matrix\_power(a, 2) :  [[16. 4. -2.]  [15. 19. -5.]  [25. 15. 1.]] |
| --- |

Ранг матрицы

| import numpy as np  a = np.array([[1, 3,-1], [5, 2,0], [0,5,1]], float)  b=np.linalg.matrix\_rank(a)  print("A:")  print(a)  print("\n np.linalg.matrix\_rank(a) :")  print(b) |
| --- |

Результат:

| 3 |
| --- |

Транспонированная матрица.

| import numpy as np  import numpy as np  a = np.array([[1, 3,-1], [5, 2,0], [0,5,1]], float)  b=a.T  print("A:")  print(a)  print("\n a.T :")  print(b) |
| --- |

Результат:

| A:  [[ 1. 3. -1.]  [ 5. 2. 0.]  [ 0. 5. 1.]]  a.T :  [[ 1. 5. 0.]  [ 3. 2. 5.]  [-1. 0. 1.]] |
| --- |

Cоединение матриц по горизонтали и по вертикали.

| import numpy as np  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(a)  print("B:")  print(b)  print("C:")  print(c)  print("np.hstack((a, b)):")  d=np.hstack((a, b))  print(d)  print("np.vstack((a, c)):")  e=np.vstack((a, c))  print(e) |
| --- |

Результат:

| A:  [[0 1]  [2 3]]  B:  [[4 5 6]  [7 8 9]]  C:  [[4 5]  [6 7]  [8 9]]  np.hstack((a, b)):  [[0 1 4 5 6]  [2 3 7 8 9]]  np.vstack((a, c)):  [[0 1]  [2 3]  [4 5]  [6 7]  [8 9]] |
| --- |

Сумма всех элементов; суммы столбцов; суммы строк.

| import numpy as np  a = np.array([[4, 5, 6], [7, 8, 9]])  print("A:")  print(a)  print("a.sum():")  b=a.sum()  print(b)  print("a.sum(axis=0):")  c=a.sum(axis=0)  print(c)  print("a.sum(axis=1):")  d=a.sum(axis=1)  print(d) |
| --- |

Результат:

| A:  [[4 5 6]  [7 8 9]]  a.sum():  39  a.sum(axis=0):  [11 13 15]  a.sum(axis=1):  [15 24] |
| --- |

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

След - сумма диагональных элементов.

| import numpy as np  a = np.array([[0, 1], [2, 3]])  print("A:")  print(a)  print("np.trace(a):")  b=np.trace(a)  print(b) |
| --- |

Результат:

| A:  [[0 1]  [2 3]]  np.trace(a):  3 |
| --- |

**Упражнение:**

в статистике и машинном обучении часто приходится иметь с функцией RSSRSS, которая вычисляется по формуле

где — координаты одномерного вектора  — координаты одномерного вектора . Посчитайте RSS для .

**Решение:**

| import numpy as np  y = np.arange(1, 6)  a = np.arange(3, -2, -1)  print("y:")  print(y)  print("a:")  print(a)  rss = np.sum((y - a)\*\*2)  print("RSS=", end='')  print(rss) |
| --- |

**Разложение**

Одиночное (cингулярное) разложение (аналог диагонализации не квадратной матрицы) может быть достигнут так:

| from calendar import c  import numpy as np  a = np.array([[1, 3,4], [5, 2, 3]], float)  U, s, Vh = np.linalg.svd(a)  print("A:")  print(a)  print("\n U:")  print(U)  print("\n s:")  print(s)  print("\n Vh:")  print(Vh) |
| --- |

Результат:

| A:  [[1. 3. 4.]  [5. 2. 3.]]  U:  [[-0.6113829 -0.79133492]  [-0.79133492 0.6113829 ]]  s:  [7.46791327 2.86884495]  Vh:  [[-0.61169129 -0.45753324 -0.64536587]  [ 0.78971838 -0.40129005 -0.46401635]  [-0.046676 -0.79349205 0.60678804]] |
| --- |

**linalg.cholesky**(a) - разложение Холецкого.

|  |
| --- |

Результат:

|  |
| --- |

**linalg.qr**(a[, mode]) - QR разложение.

|  |
| --- |

Результат:

|  |
| --- |

**linalg.norm**(x[, ord, axis]) - норма вектора или оператора.

| import numpy as np  a = np.array([1,2,5,7,9], float)  b=np.linalg.norm(a)  print("A:")  print(a)  print("\n Norm(a):")  print(b) |
| --- |

Результат:

| A:  [1. 2. 5. 7. 9.]  np.linalg.cholesky(a) :  12.649110640673518 |
| --- |

**Упражнение:**

В машинном обучении есть модель линейной регрессии, для которой "хорошее" решение считается по следующей формуле: . Вычислите  для ,   — единичная матрица размерности 3, λ=0.1.

**Решение:**

| import numpy as np  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  print(theta) |
| --- |

**Результат:**

| [0.68478852 2.80148778 0.81180812] |
| --- |

**Математика многочленов**

NumPy предоставляет методы для работы с полиномами. Передавая список корней, можно получить коэффициенты уравнения:

| import numpy as np  a=np.poly([-1, 1, 1, 10])  print("A:")  print(a) |
| --- |

Результат:

| A:  [ 1. -11. 9. 11. -10.] |
| --- |

Здесь, массив возвращает коэффициенты соответствующие уравнению:

.

Может быть произведена и обратная операция: передавая список коэффициентов, функция root вернет все корни многочлена:

| import numpy as np  a=np.roots([1, -11,  9, 11, -10])  print("A:")  print(a) |
| --- |

Результат:

| A:  [10.+0.0000000e+00j -1.+0.0000000e+00j 1.+9.6357437e-09j  1.-9.6357437e-09j] |
| --- |

## 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.

Решение линейной системы .

Вариант 1

| import numpy as np  a = np.array([[0, 1], [2, 3]])  v = np.array([0, 1], dtype=np.float64)  print("A:")  print(a)  print("\n V:")  print(v)  a1 = np.linalg.inv(a)  print("\n a^(-1):")  print(a1)  u=a1 @ v  print("\n U:")  print(u) |
| --- |

Вариант 2.

| import numpy as np  a = np.array([[0, 1], [2, 3]])  v = np.array([0, 1], dtype=np.float64)  print("A:")  print(a)  print("\n V:")  print(v)  u = np.linalg.solve(a, v)  print("\n U:")  print(u) |
| --- |

Проверим:

| import numpy as np  a = np.array([[0, 1], [2, 3]])  v = np.array([0, 1], dtype=np.float64)  print("A:")  print(a)  print("\n V:")  print(v)  u = np.linalg.solve(a, v)  print("\n U:")  print(u)  print("a @ u - v :")  x=a @ u - v  print(x) |
| --- |

Результат:

| A:  [[0 1]  [2 3]]  V:  [0. 1.]  U:  [0.5 0. ]  a @ u - v :  [0. 0.] |
| --- |

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

| import numpy as np  a = np.array([[0, 1], [2, 3]])  print("A:")  print(a)  print("diag(a):")  L = np.diag(a)  print(L)  print("diag(L): ")  print(np.diag(L)) |
| --- |

Рузультат

| A:  [[0 1]  [2 3]]  diag(a):  [0 3]  diag(L):  [[0 0]  [0 3]] |
| --- |

Все уравнения  можно собрать в одно матричное уравнение , где  — диагональная матрица с собственными значениями  по диагонали.

Поэтому .

Литература

1. <https://engineering.ucsb.edu/~shell/che210d/numpy.pdf>

**100 задач**

1. Импортировать NumPy под именем np

**import** **numpy** **as** **np**

1. Напечатать версию и конфигурацию
2. print(np.\_\_version\_\_)

np.show\_config()

1. Создать вектор (одномерный массив) размера 10, заполненный нулями
2. Z = np.zeros(10)

print(Z)

1. Создать вектор размера 10, заполненный единицами
2. Z = np.ones(10)

print(Z)

1. Создать вектор размера 10, заполненный числом 2.5
2. Z = np.full(10, 2.5)

print(Z)

1. Как получить документацию о функции numpy.add из командной строки?

python3 -c "import numpy; numpy.info(numpy.add)"

1. Создать вектор размера 10, заполненный нулями, но пятый элемент равен 1
2. Z = np.zeros(10)
3. Z[4] = 1

print(Z)

1. Создать вектор со значениями от 10 до 49
2. Z = np.arange(10,50)

print(Z)

1. Развернуть вектор (первый становится последним)
2. Z = np.arange(50)

Z = Z[::-1]

1. Создать матрицу (двумерный массив) 3x3 со значениями от 0 до 8
2. Z = np.arange(9).reshape(3,3)

print(Z)

1. Найти индексы ненулевых элементов в [1,2,0,0,4,0]
2. nz = np.nonzero([1,2,0,0,4,0])

print(nz)

1. Создать 3x3 единичную матрицу
2. Z = np.eye(3)

print(Z)

1. Создать массив 3x3x3 со случайными значениями
2. Z = np.random.random((3,3,3))

print(Z)

1. Создать массив 10x10 со случайными значениями, найти минимум и максимум
2. Z = np.random.random((10,10))
3. Zmin, Zmax = Z.min(), Z.max()

print(Zmin, Zmax)

1. Создать случайный вектор размера 30 и найти среднее значение всех элементов
2. Z = np.random.random(30)
3. m = Z.mean()

print(m)

1. Создать матрицу с 0 внутри, и 1 на границах
2. Z = np.ones((10,10))

Z[1:-1,1:-1] = 0

1. Выяснить результат следующих выражений
2. 0 \* np.nan
3. np.nan == np.nan
4. np.inf > np.nan
5. np.nan - np.nan

0.3 == 3 \* 0.1

1. Создать 5x5 матрицу с 1,2,3,4 под диагональю
2. Z = np.diag(np.arange(1, 5), k=-1)

print(Z)

1. Создать 8x8 матрицу и заполнить её в шахматном порядке
2. Z = np.zeros((8,8), dtype=int)
3. Z[1::2,::2] = 1
4. Z[::2,1::2] = 1

print(Z)

1. Дан массив размерности (6,7,8). Каков индекс (x,y,z) сотого элемента?

print(np.unravel\_index(100, (6,7,8)))

1. Создать 8x8 матрицу и заполнить её в шахматном порядке, используя функцию tile
2. Z = np.tile(np.array([[0,1],[1,0]]), (4,4))

print(Z)

1. Перемножить матрицы 5x3 и 3x2
2. Z = np.dot(np.ones((5,3)), np.ones((3,2)))

print(Z)

1. Дан массив, поменять знак у элементов, значения которых между 3 и 8
2. Z = np.arange(11)

Z[(3 < Z) & (Z <= 8)] \*= -1

1. Создать 5x5 матрицу со значениями в строках от 0 до 4
2. Z = np.zeros((5,5))
3. Z += np.arange(5)

print(Z)

1. Есть генератор, сделать с его помощью массив
2. **def** generate():
3. **for** x **in** xrange(10):
4. **yield** x
5. Z = np.fromiter(generate(),dtype=float,count=-1)

print(Z)

1. Создать вектор размера 10 со значениями от 0 до 1, не включая ни то, ни другое
2. Z = np.linspace(0,1,12)[1:-1]

print(Z)

1. Отсортировать вектор
2. Z = np.random.random(10)
3. Z.sort()

print(Z)

1. Проверить, одинаковы ли 2 numpy массива
2. A = np.random.randint(0,2,5)
3. B = np.random.randint(0,2,5)
4. equal = np.allclose(A,B)

print(equal)

1. Сделать массив неизменяемым
2. Z = np.zeros(10)
3. Z.flags.writeable = **False**

Z[0] = 1

1. Дан массив 10x2 (точки в декартовой системе координат), преобразовать в полярную
2. Z = np.random.random((10,2))
3. X,Y = Z[:,0], Z[:,1]
4. R = np.hypot(X, Y)
5. T = np.arctan2(Y,X)
6. print(R)

print(T)

1. Заменить максимальный элемент на ноль
2. Z = np.random.random(10)
3. Z[Z.argmax()] = 0

print(Z)

1. Создать структурированный массив с координатами x, y на сетке в квадрате [0,1]x[0,1]
2. Z = np.zeros((10,10), [('x',float),('y',float)])
3. Z['x'], Z['y'] = np.meshgrid(np.linspace(0,1,10),
4. np.linspace(0,1,10))

print(Z)

1. Из двух массивов сделать матрицу Коши C (Cij = 1/(xi - yj))
2. X = np.arange(8)
3. Y = X + 0.5
4. C = 1.0 / np.subtract.outer(X, Y)

print(np.linalg.det(C))

1. Найти минимальное и максимальное значение, принимаемое каждым числовым типом numpy
2. **for** dtype **in** [np.int8, np.int32, np.int64]:
3. print(np.iinfo(dtype).min)
4. print(np.iinfo(dtype).max)
5. **for** dtype **in** [np.float32, np.float64]:
6. print(np.finfo(dtype).min)
7. print(np.finfo(dtype).max)

print(np.finfo(dtype).eps)

1. Напечатать **все** значения в массиве
2. np.set\_printoptions(threshold=np.nan)
3. Z = np.zeros((25,25))

print(Z)

1. Найти ближайшее к заданному значению число в заданном массиве
2. Z = np.arange(100)
3. v = np.random.uniform(0,100)
4. index = (np.abs(Z-v)).argmin()

print(Z[index])

1. Создать структурированный массив, представляющий координату (x,y) и цвет (r,g,b)
2. Z = np.zeros(10, [ ('position', [ ('x', float, 1),
3. ('y', float, 1)]),
4. ('color', [ ('r', float, 1),
5. ('g', float, 1),
6. ('b', float, 1)])])

print(Z)

1. Дан массив (100,2) координат, найти расстояние от каждой точки до каждой
2. **import** **scipy.spatial**
3. Z = np.random.random((10,2))
4. D = scipy.spatial.distance.cdist(Z,Z)

print(D)

1. Преобразовать массив из float в int
2. Z = np.arange(10, dtype=np.int32)

Z = Z.astype(np.float32, copy=**False**)

1. Дан файл:
2. 1,2,3,4,5
3. 6,,,7,8
4. ,,9,10,11

Как прочитать его?

Z = np.genfromtxt("missing.dat", delimiter=",")

1. Каков эквивалент функции enumerate для numpy массивов?
2. Z = np.arange(9).reshape(3,3)
3. **for** index, value **in** np.ndenumerate(Z):
4. print(index, value)
5. **for** index **in** np.ndindex(Z.shape):

print(index, Z[index])

1. Сформировать 2D массив с распределением Гаусса
2. X, Y = np.meshgrid(np.linspace(-1,1,10), np.linspace(-1,1,10))
3. D = np.hypot(X, Y)
4. sigma, mu = 1.0, 0.0
5. G = np.exp(-((D - mu) \*\* 2 / (2.0 \* sigma \*\* 2)))

print(G)

1. Случайно расположить p элементов в 2D массив
2. n = 10
3. p = 3
4. Z = np.zeros((n,n))

np.put(Z, np.random.choice(range(n\*n), p, replace=**False**), 1)

1. Отнять среднее из каждой строки в матрице
2. X = np.random.rand(5, 10)

Y = X - X.mean(axis=1, keepdims=**True**)

1. Отсортировать матрицу по n-ому столбцу
2. Z = np.random.randint(0,10,(3,3))
3. n = 1 *# Нумерация с нуля*
4. print(Z)

print(Z[Z[:,n].argsort()])

1. Определить, есть ли в 2D массиве нулевые столбцы
2. Z = np.random.randint(0,3,(3,10))

print((~Z.any(axis=0)).any())

1. Дан массив, добавить 1 к каждому элементу с индексом, заданным в другом массиве (осторожно с повторами)
2. Z = np.ones(10)
3. I = np.random.randint(0,len(Z),20)
4. Z += np.bincount(I, minlength=len(Z))

print(Z)

1. Дан массив (w,h,3) (картинка) dtype=ubyte, посчитать количество различных цветов
2. w,h = 16,16
3. I = np.random.randint(0, 2, (h,w,3)).astype(np.ubyte)
4. F = I[...,0] \* 256 \* 256 + I[...,1] \* 256 + I[...,2]
5. n = len(np.unique(F))

print(np.unique(I))

1. Дан четырехмерный массив, посчитать сумму по последним двум осям
2. A = np.random.randint(0,10, (3,4,3,4))
3. sum = A.reshape(A.shape[:-2] + (-1,)).sum(axis=-1)

print(sum)

1. Найти диагональные элементы произведения матриц
2. *# Slow version*
3. np.diag(np.dot(A, B))
4. *# Fast version*
5. np.sum(A \* B.T, axis=1)
6. *# Faster version*

np.einsum("ij,ji->i", A, B).

1. Дан вектор [1, 2, 3, 4, 5], построить новый вектор с тремя нулями между каждым значением
2. Z = np.array([1,2,3,4,5])
3. nz = 3
4. Z0 = np.zeros(len(Z) + (len(Z)-1)\*(nz))
5. Z0[::nz+1] = Z

print(Z0)

1. Поменять 2 строки в матрице
2. A = np.arange(25).reshape(5,5)
3. A[[0,1]] = A[[1,0]]

print(A)

1. Рассмотрим набор из 10 троек, описывающих 10 треугольников (с общими вершинами), найти множество уникальных отрезков, составляющих все треугольники
2. faces = np.random.randint(0,100,(10,3))
3. F = np.roll(faces.repeat(2,axis=1),-1,axis=1)
4. F = F.reshape(len(F)\*3,2)
5. F = np.sort(F,axis=1)
6. G = F.view( dtype=[('p0',F.dtype),('p1',F.dtype)] )
7. G = np.unique(G)

print(G)

1. Дан массив C; создать массив A, что np.bincount(A) == C
2. C = np.bincount([1,1,2,3,4,4,6])
3. A = np.repeat(np.arange(len(C)), C)

print(A)

1. Посчитать среднее, используя плавающее окно
2. **def** moving\_average(a, n=3):
3. ret = np.cumsum(a, dtype=float)
4. ret[n:] = ret[n:] - ret[:-n]
5. **return** ret[n - 1:] / n

print(moving\_average(np.arange(20), 3))

1. Дан вектор Z, построить матрицу, первая строка которой (Z[0],Z[1],Z[2]), каждая последующая сдвинута на 1 (последняя (Z[-3],Z[-2],Z[-1]))
2. **from** **numpy.lib** **import** stride\_tricks
3. **def** rolling(a, window):
4. shape = (a.size - window + 1, window)
5. strides = (a.itemsize, a.itemsize)
6. **return** stride\_tricks.as\_strided(a, shape=shape, strides=strides)
7. Z = rolling(np.arange(10), 3)

print(Z)

1. Инвертировать булево значение, или поменять знак у числового массива без создания нового
2. Z = np.random.randint(0,2,100)
3. np.logical\_not(arr, out=arr)
4. Z = np.random.uniform(-1.0,1.0,100)

np.negative(arr, out=arr)

1. Рассмотрим 2 набора точек P0, P1 описания линии (2D) и точку р, как вычислить расстояние от р до каждой линии i (P0[i],P1[i])
2. **def** distance(P0, P1, p):
3. T = P1 - P0
4. L = (T\*\*2).sum(axis=1)
5. U = -((P0[:,0] - p[...,0]) \* T[:,0] + (P0[:,1] - p[...,1]) \* T[:,1]) / L
6. U = U.reshape(len(U),1)
7. D = P0 + U \* T - p
8. **return** np.sqrt((D\*\*2).sum(axis=1))
9. P0 = np.random.uniform(-10,10,(10,2))
10. P1 = np.random.uniform(-10,10,(10,2))
11. p = np.random.uniform(-10,10,( 1,2))

print(distance(P0, P1, p))

1. Дан массив. Написать функцию, выделяющую часть массива фиксированного размера с центром в данном элементе (дополненное значением fill если необходимо)
2. Z = np.random.randint(0,10, (10,10))
3. shape = (5,5)
4. fill = 0
5. position = (1,1)
6. R = np.ones(shape, dtype=Z.dtype)\*fill
7. P = np.array(list(position)).astype(int)
8. Rs = np.array(list(R.shape)).astype(int)
9. Zs = np.array(list(Z.shape)).astype(int)
10. R\_start = np.zeros((len(shape),)).astype(int)
11. R\_stop = np.array(list(shape)).astype(int)
12. Z\_start = (P - Rs//2)
13. Z\_stop = (P + Rs//2)+Rs%2
14. R\_start = (R\_start - np.minimum(Z\_start, 0)).tolist()
15. Z\_start = (np.maximum(Z\_start, 0)).tolist()
16. R\_stop = np.maximum(R\_start, (R\_stop - np.maximum(Z\_stop-Zs,0))).tolist()
17. Z\_stop = (np.minimum(Z\_stop,Zs)).tolist()
18. r = [slice(start,stop) **for** start,stop **in** zip(R\_start,R\_stop)]
19. z = [slice(start,stop) **for** start,stop **in** zip(Z\_start,Z\_stop)]
20. R[r] = Z[z]
21. print(Z)

print(R)

1. Посчитать ранг матрицы
2. Z = np.random.uniform(0,1,(10,10))

rank = np.linalg.matrix\_rank(Z)

1. Найти наиболее частое значение в массиве
2. Z = np.random.randint(0,10,50)

print(np.bincount(Z).argmax())

1. Извлечь все смежные 3x3 блоки из 10x10 матрицы
2. Z = np.random.randint(0,5,(10,10))
3. n = 3
4. i = 1 + (Z.shape[0] - n)
5. j = 1 + (Z.shape[1] - n)
6. C = stride\_tricks.as\_strided(Z, shape=(i, j, n, n), strides=Z.strides + Z.strides)

print(C)

1. Создать подкласс симметричных 2D массивов (Z[i,j] == Z[j,i])
2. *# Note: only works for 2d array and value setting using indices*
3. **class** **Symetric**(np.ndarray):
4. **def** \_\_setitem\_\_(self, (i,j), value):
5. super(Symetric, self).\_\_setitem\_\_((i,j), value)
6. super(Symetric, self).\_\_setitem\_\_((j,i), value)
7. **def** symetric(Z):
8. **return** np.asarray(Z + Z.T - np.diag(Z.diagonal())).view(Symetric)
9. S = symetric(np.random.randint(0,10,(5,5)))
10. S[2,3] = 42

print(S)

1. Рассмотрим множество матриц (n,n) и множество из p векторов (n,1). Посчитать сумму p произведений матриц (результат имеет размерность (n,1))
2. p, n = 10, 20
3. M = np.ones((p,n,n))
4. V = np.ones((p,n,1))
5. S = np.tensordot(M, V, axes=[[0, 2], [0, 1]])
6. print(S)
7. *# It works, because:*
8. *# M is (p,n,n)*
9. *# V is (p,n,1)*
10. *# Thus, summing over the paired axes 0 and 0 (of M and V independently),*

*# and 2 and 1, to remain with a (n,1) vector.*

1. Дан массив 16x16, посчитать сумму по блокам 4x4
2. Z = np.ones((16,16))
3. k = 4
4. S = np.add.reduceat(np.add.reduceat(Z, np.arange(0, Z.shape[0], k), axis=0),

np.arange(0, Z.shape[1], k), axis=1)

1. Написать игру "жизнь"
2. **def** iterate(Z):
3. *# Count neighbours*
4. N = (Z[0:-2,0:-2] + Z[0:-2,1:-1] + Z[0:-2,2:] +
5. Z[1:-1,0:-2] + Z[1:-1,2:] +
6. Z[2: ,0:-2] + Z[2: ,1:-1] + Z[2: ,2:])
7. *# Apply rules*
8. birth = (N == 3) & (Z[1:-1,1:-1]==0)
9. survive = ((N == 2) | (N == 3)) & (Z[1:-1,1:-1] == 1)
10. Z[...] = 0
11. Z[1:-1,1:-1][birth | survive] = 1
12. **return** Z
13. Z = np.random.randint(0,2,(50,50))
14. **for** i **in** range(100):
15. print(Z)

Z = iterate(Z)

1. Найти n наибольших значений в массиве
2. Z = np.arange(10000)
3. np.random.shuffle(Z)
4. n = 5

print (Z[np.argpartition(-Z,n)[:n]])

1. Построить прямое произведение массивов (все комбинации с каждым элементом)
2. **def** cartesian(arrays):
3. arrays = [np.asarray(a) **for** a **in** arrays]
4. shape = map(len, arrays)
5. ix = np.indices(shape, dtype=int)
6. ix = ix.reshape(len(arrays), -1).T
7. **for** n, arr **in** enumerate(arrays):
8. ix[:, n] = arrays[n][ix[:, n]]
9. **return** ix

print(cartesian(([1, 2, 3], [4, 5], [6, 7])))

1. Даны 2 массива A (8x3) и B (2x2). Найти строки в A, которые содержат элементы из каждой строки в B, независимо от порядка элементов в B
2. A = np.random.randint(0,5,(8,3))
3. B = np.random.randint(0,5,(2,2))
4. C = (A[..., np.newaxis, np.newaxis] == B)
5. rows = (C.sum(axis=(1,2,3)) >= B.shape[1]).nonzero()[0]

print(rows)

1. Дана 10x3 матрица, найти строки из неравных значений (например [2,2,3])
2. Z = np.random.randint(0,5,(10,3))
3. E = np.logical\_and.reduce(Z[:,1:] == Z[:,:-1], axis=1)
4. U = Z[~E]
5. print(Z)

print(U)

1. Преобразовать вектор чисел в матрицу бинарных представлений
2. I = np.array([0, 1, 2, 3, 15, 16, 32, 64, 128], dtype=np.uint8)

print(np.unpackbits(I[:, np.newaxis], axis=1))

1. Дан двумерный массив. Найти все различные строки
2. Z = np.random.randint(0, 2, (6,3))
3. T = np.ascontiguousarray(Z).view(np.dtype((np.void, Z.dtype.itemsize \* Z.shape[1])))
4. \_, idx = np.unique(T, return\_index=**True**)
5. uZ = Z[idx]

print(uZ)

1. Даны векторы A и B, написать einsum эквиваленты функций inner, outer, sum и mul
2. *# Make sure to read: http://ajcr.net/Basic-guide-to-einsum/*
3. np.einsum('i->', A) *# np.sum(A)*
4. np.einsum('i,i->i', A, B) *# A \* B*
5. np.einsum('i,i', A, B) *# np.inner(A, B)*

np.einsum('i,j', A, B) *# np.outer(A, B)*