Практическая работа с NumPy

1. Введение

Библиотека NumPy для языка программирования Python [Python, Python-Ru1] предоставляет эффективные структуры и алгоритмы для хранения и обработки данных. Все данные представляются в виде многомерных массивов (ndarray), для которых существуют быстрые процедуры изменения размеров и обработки.

Благодаря своей эффективности, библиотека получила широкое распространение в научном и инженерном сообществе и является стандартом де-факто для расчетов на языке Python. На основе NumPy было создано большое число других библиотек для машинного обучения, отображения данных и многого другого.

Официальный сайт библиотеки [NumPy] содержит обширную документацию по функциям и возможностям с примерами и учебниками. Часть этой документации на русском языке можно найти на переводных сайтах [NumPy-Ru1] и [NumPy-Ru2].

Для использования библиотеки в программах нужно ее предварительно установить командой в командной строке операционной системы:

```
pip install numpy
```

После этого последняя версия библиотеки будет установлена для текущего пользователя компьютера. Если необходимо установить библиотеку для всех пользователей, то команду выполняют от имени администратора (в Windows) или sudo (в Linux).

После установки в любой программе на языке Python можно подключить библиотеку, поставив в начале файла строку:

```
import numpy
```

или

```
import numpy as np
```

Последний вариант чаще всего встречается, поскольку позволяет обращаться к функциям библиотеки через короткий префикс np. В дальнейшем будем предполагать, что библиотека импортирована именно так.

2. Создание массивов

Для создания массивов определенного размера в библиотеке имеется множество функций: можно создать ndarray из стандартных структур данных языка Python (списков, множеств), можно создать пустые, единичные или нулевые массивы, можно сгенерировать массив с заданными или случайными числами.

Для создания ndarray из списка предназначена функция array. Например, создание одномерного массива из списка:

```
    list1d = [1, 0.5, 4, 2]
    arr1d = np.array(list1d)
```

В переменной arr1d содержится одномерный массив numpy.

Команда

```
print(arr1d)
```

выведет

```
[1.  0.5 4.  2. ]
```

Можно создавать двумерные массивы из списков:

```
    list2d = [[1, 2, 3], [4, 5, 6]]
    arr2d = np.array(list2d)
    print(arr2d)
```

Вывод будет следующий:

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

Не обязательно объявлять переменные для списков. Инициализирующие

выражения можно передать непосредственно в функцию array:

```
    arr1d = np.array([1, 0.5, 4, 2])
    arr2d = np.array([[1, 2, 3], [4, 5, 6]])
```

Вторым параметром функции array (под именем dtype) можно передать тип данных, который будет храниться в массиве. Например, создание массива чисел с плавающей точкой:

```
    arr1d = np.array([1, 0.5, 4, 2], dtype=float)
    print(arr1d)
```

Результат:

```
[1.  0.5 4.  2. ]
```

Можно указать целочисленный тип (int):

```
    arr1d = np.array([1, 0.5, 4, 2], int)
    print(arr1d)
```

Результат будет следующий:

```
[1 0 4 2]
```

Если тип данных не указан, то он определяется исходя из типа данных, хранящегося в списке.

Можно создать массив булевых значений:

```
    arr1d = np.array([2, 0.5, 0, -7], bool)
    print(arr1d)
```

Результат:

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

При преобразовании численный значений в булевы действует правило: если число равно нулю, то результат будет ложный (False), иначе результат будет истинный (True).

Массивы можно создавать заполненными нулевыми значениями при помощи функции zeros. Первым параметром ей указывается размер массива, а вторым тип данных. Если тип данных не указан, то он принимается float (число с плавающей точкой). Например:

```
    z1 = np.zeros(5, float)
    print(z1)
```

Результат:

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

Аналогичный результат будет получен, если указать без типа данных:

```
    z1 = np.zeros(5)
```

Если необходимо задать двумерный массив, тогда размер указывается перечислением через запятую в скобках (кортеж языка Python). Первым указывается число строк, затем число столбцов. Например, массив целых чисел из 2 строк и 3 столбцов задается:

```
    z2 = np.zeros((2, 3), int)
```

Вывод:

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

(Примечание: чтобы вывести значение переменной в терминал, необходимо воспользоваться функцией print, как в примерах выше. Для упрощения в некоторых случаях далее будем эту строчку пропускать).

NumPyне ограничивает количество измерений в массиве (тип ndarray многомерный). Например, можно создать следующий массив:

```
    z3 = np.zeros((2, 3, 4), int)
```

Будет получено два двумерных массива размером 3 на 4 (или трехмерный массив 2x3x4, что равнозначно):

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

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

Существует аналогичная функция ones для создания массивов, заполненных единицами, с теми же параметрами:

```
    e1 = np.ones(3, int)
    e2 = np.ones((3, 2))
```

Вывод:

```
[1 1 1]

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

Для формирования массива, заполненного одним конкретным значением, можно воспользоваться функцией full. Первым ее параметром указывается размер массива, вторым – значение, которым заполнить массив, третьим – тип данных (также не обязательно). Например:

```
    f1 = np.full((2, 3), 5.1, float)
```

Вывод:

```
[[5.1 5.1 5.1]
 [5.1 5.1 5.1]]
```

Одним из частых типов матриц, используемых в линейной алгебре, является единичная диагональная матрица, у которой на диагонали расположены единицы, а остальные элементы равны нулю. Чаще всего такая матрица квадратная. Для создания такой матрицы имеется функция eye:

```
    np.eye(3)
```

Результат:

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

Первым параметром функции указывается размер (количество строк). Если необходима матрица с другим числом столбцов (не квадратная), то вторым параметром (под именем M) можно указать число столбцов:

```
    np.eye(3, 4)
```

Вывод:

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

Третьим параметром (k) можно указать индекс диагонали, на которой должны располагаться единицы (по умолчанию, он равен 0 – главная диагональ). Можно заполнить другую:

```
    np.eye(3, 4, -1)
```

Результат:

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

Четвертым параметром (dtype) можно задать тип данных для функции. Если не указан, то тип выбирается float.

Все параметры, кроме первого, являются не обязательными. Если необходимо задать, например, только диагональ, или только тип данных, то можно воспользоваться возможностью языка Python указывать параметры функции через имя параметра, пропуская ненужные. Например, если необходимо создать матрицу 3x3 с целочисленными элементами, то можно вызвать:

```
    np.eye(3, dtype=int)
```

Результат:

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

Аналогичного результата можно добиться, если вызвать:

```
np.eye(3, 3, 0, int)
```

Однако первый вариант проще и понятнее.

Функция arange позволяет сгенерировать массив с последовательностью чисел. Например:

```
np.arange(5)
```

Результат:

```
[0 1 2 3 4]
```

Параметры функции:

arange([start,]stop, [step,] [dtype])

start – начальное значение диапазона (если не указано, то равно 0);

stop – конечное значение диапазона (обязательный параметр), не входит в результат;

step – шаг (если не задан, то равен 1);

dtype – тип данных (если не задан, то определяется по типу первых параметров).

Пример:

```
np.arange(1, 9, 2, float)
np.arange(start=1, stop=9, step=2, dtype=float)
```

Результат:

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

Еще пример:

```
np.arange(1, step=0.2)
```

Результат:

```
[0.  0.2 0.4 0.6 0.8]
```

Если необходимо создать последовательность чисел, в которой известен диапазон и число элементов, но не известен шаг, то можно вместо arrange воспользоваться linspace:

linspace(start, stop, num=50, endpoint=True, dtype=None)

start – начальное значение (обязательный параметр);

stop – конечное значение (обязательный параметр);

num – число элементов (если не указано, 50);

endpoint – признак включения значения stop в последовательность (по умолчанию, True);

dtype – тип данных, если не указан, то определяется по параметрам.

Например:

```
np.linspace(1, 2, 5)
```

Результат:

```
[1.   1.25 1.5  1.75 2.  ]
```

Обратите внимание, что значение 2 включено в последовательность. Чтобы его не включать:

```
np.linspace(1, 2, 5, endpoint=False)
```

Результат:

```
[1.  1.2 1.4 1.6 1.8]
```

Одной из частых задач при работе с массивами является генерация массива заданного размера со случайными числами. Для этого можно воспользоваться функцией randint из подмодуля random:

randint(low, high=None, size=None)

low – если задан только этот параметр, то указывает максимальное значение для случайных чисел (но само значение не входит в диапазон);

high – если заданы оба (low и high), то они задают диапазон генерации чисел [low, high), то есть high не входит;

size – длина последовательности.

Например, генерация массива из пяти чисел от 0 до 4:

```
np.random.randint(5, size=5)
```

Результат:

```
[0 0 2 1 4]
```

Генерация массива из пяти чисел от 10 до 19:

```
np.random.randint(10, 20, size=5)
```

Результат:

```
[16 17 16 18 18]
```

Если необходимо сгенерировать массив случайных чисел с плавающей точкой, необходимо использовать функцию rand:

random.rand(d0, d1, ...)

Она всегда генерирует случайные числа в диапазоне от 0 до 1 (но 1 не входит в диапазон). Параметрами функции задаются размеры массива. Если указан один размер, то будет одномерный массив. Два размера зададут матрицу и т.д.

Пример:

```
np.random.rand(4)
```

Результат:

```
[0.1836634  0.46449058 0.88689603 0.32688258]
```

Пример:

```
np.random.rand(2, 3)
```

Результат:

```
[[0.77549814 0.83479098 0.71086936]
 [0.04089972 0.83923191 0.33830922]]
```

Если необходимо, чтобы диапазон случайных чисел отличался от [0, 1), тогда нужно использовать математические операции над массивом. Например, сгенерировать матрицу 2x3 чисел в диапазоне [-1, 1):

```
2 * np.random.rand(2, 3) - 1
```

Результат:

```
[[ 0.35214644 -0.32754867 -0.28123389]
 [ 0.97920488 -0.81045682 -0.77036573]]
```

Общее правило, как задавать генерацию чисел:

D * rand() + M

где D – ширина диапазона случайных чисел; M – нижняя граница диапазона.

3. Типы данных

Массив ndarray может содержать элементы следующих типов:

- целочисленный (int)
- с плавающей точкой (float)
- булевый (bool)
- комплексный (complex)
- символьный (str)
- и другие

Каждый тип имеет различные варианты реализации (размер, диапазон значений). На данный момент ограничимся перечисленными вариантами. Более подробную информацию можно найти в документации.

Узнать тип данных массива можно, обратившись к его свойству dtype:

```
d1 = np.array([1, 2, 3])
print(d1.dtype)
```

Результат:

```
int32
```

Данные в примере целочисленного типа длиной 32 бит.

Можно изменить тип данных массива, использовав метод astype(тип):

```
d1 = np.array([1, 0, 3])
print('Тип исходный:', d1.dtype)
print('Значение: ', d1)
d2 = d1.astype(bool)
print('Тип после преобразования:', d2.dtype)
print('Значение: ', d2)
```

Результат:

```
Тип исходный: int32
Значение:  [1 0 3]
Тип после преобразования: bool
Значение:  [ True False  True]
```

4. Размеры массива

У переменной типа ndarray есть свойство shape, которое хранит текущие размеры массива в виде кортежа Python свойство ndim, которое хранит размерность массива (число измерений), т.е. длину кортежа shape. Например:

```
d1 = np.ones((4, 3))
print('Размерность:', d1.ndim, 'Размер:', d1.shape)
```

Результат:

```
Размерность: 2 Размер: (4, 3)
```

К элементам кортежа shape можно обращаться по отдельности:

```
print('Строк:', d1.shape[0], 'Столбцов:', d1.shape[1])
```

Результат:

```
Строк: 4 Столбцов: 3
```

У любого массива можно изменить размер при помощи метода reshape. Его можно вызвать двумя способами: указав первым параметром исходный массив, а втором параметром кортеж нового размера, или указав его как метод от исходного массива и передав кортеж нового размера в качестве параметра.

Например, имеется массив d1 размером (3, 4). Следующие два вызова выполняют одинаковую работу по преобразованию его в массив размером (2, 6):

```
d2 = np.reshape(d1, (2, 6))
d2 = d1.reshape((2, 6))
```

В результате d2 будет иметь следующий вид:

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

При изменении размера можно менять и число измерений, например, преобразовать одномерный массив из 16 элементов в матрицу 4x4 или матрицу размеров 3x6 в трехмерный массив 2x3x3. Главное условие, чтобы при преобразовании число элементов совпадало.

Одно из измерений для метода reshape можно задавать значением -1, тогда метод сам рассчитает сколько элементов должно в нем быть. Например:

```
d1 = np.ones((3, 4))
d2 = d1.reshape((-1, 2))
d3 = d2.reshape(-1)
print(d2.shape)
print(d3.shape)
```

Результат:

```
(6, 2)
(12,)
```

Метод flatten позволяет превратить массив любой размерности в одномерный

(«плоский») массив с тем же числом элементов:

```
    d1 = np.ones((3, 4, 2))
    d2 = d1.flatten()
    print(d2.shape)
```

Результат:

```
(24,)
```

5. Индексация, срезы, присвоение и копирование массивов

Для обращения к элементам массива необходимо указать индекс элемента в квадратных скобках. Элементы нумеруются начиная с нуля. Например, для обращения к первому элементу одномерного массива d1 необходимо указать:

```
    d1[0]
```

Для многомерных массивов индексы по каждой оси измерений перечисляются через запятую, например, для обращения к элементу на второй строке и третьем столбце матрицы d1, необходимо написать:

```
    d1[1, 2]
```

Если обращаться только к одному индексу многомерного массива, то будет возвращен массив меньшей размерности, находящийся по данному индексу. Например, для двумерных массивов:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    print(d1[1])
[4 5 6]
```

Присвоение переменных массивов друг другу не создает новых массивов, а создает ссылки на уже существующий массив. Например, после присвоения d2= d1 изменения в каждом из этих массивов будут отражаться в обоих, потому что d2 является ссылкой на тот же самый массив, а не новым массивом.

Например:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    print(d1)
    d2 = d1
    d2[0, 1] = 9
    print(d1)
 [[1 2 3]
 [4 5 6]]
[[1 9 3]
 [4 5 6]]
```

Как видно, изменение в массиве d2 отразилось на d1.

Если необходимо создать независимую копию массива, для этого служит метод copy:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    print(d1)
    d2 = d1.copy()
    d2[0, 1] = 9
    print(d1)
```

Результат:

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

Изменение в массиве d2 не отразилось на d1, потому что функция copy новый массив.

Функция array_equal(a1, a2) позволяет проверить на равенство два массива a1, a2:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = d1.copy()
```

```
    print(np.array_equal(d1, d2))
True
```

Мощным инструментом ndarray, который широко используется в алгоритмах, являются *срезы*. Срезы – представления части массива, с которыми можно работать, как с отдельными массивами. Срез может иметь другие размеры или количество измерений. Для создания среза используется символ ":".

Чтобы задать срез, нужно указать:

start : stop

где start – начальный индекс среза; stop – конечный индекс (в диапазон не входит).

Самый простой пример среза – поддиапазон массива, например:

```
    d1 = np.arange(10)
    d2 = d1[2:5]
    print(d2)
[2 3 4]
```

Если индекс start не задан, то массив берется с самого начала. Аналогично, если индекс stop не задан, то срез действует до конца:

```
    d1 = np.arange(10)
    d2 = d1[:]
    print(d2)
[0 1 2 3 4 5 6 7 8 9]
```

Срез можно задать по нескольким измерениям многомерного массива:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = d1[:, 1:3]
    print(d2)
[[2 3]
 [5 6]]
```

Срез является представлением, а не копией массива. Изменения в срезе отражаются и в самом массиве:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = d1[:2, 1:3]
    d2[0, 0] = 9
    print(d1)
[[1 9 3]
 [4 5 6]]
```

Но можно сделать копию среза и поместить в отдельный массив при помощи функции copy:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = d1[:2, 1:3].copy()
    d2[0, 0] = 9
    print(d1)
    print(d2)
[[1 2 3]
 [4 5 6]]
[[9 3]
 [5 6]]
```

Срезы можно использовать для присвоения значений части массива. Например, проставить всему первому столбцу матрицы значение 10:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d1[:, 0] = 10
    print(d1)
[[10  2  3]
 [10  5  6]]
```

Можно заменить часть матрицы частью другой матрицы:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    d2 = np.array([[11, 12, 13], [14, 15, 16], [17, 18, 19]])
    d1[:2, :2] = d2[1:3, 1:3]
```

```
    print(d1)
```
```
[[15 16  3]
 [18 19  6]
 [ 7  8  9]]
```

Можно взять столбец из матрицы:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    d2 = d1[:, 1]
    print(d2)
```
```
[2 5 8]
```

Функция append позволяет дополнить один массив другим:

append(arr, values, axis=None)

arr – исходный массив;

values – дополнительны массив или значения;

axis – индекс оси, вдоль которой дополнять, если ось не указана, то arr и values будут превращены в одномерный массив. Для матриц ось 0 соответствует вертикальной оси (по строкам), а ось 1 – по столбцам.

Функция делает новый массив из исходных, т. е. копирует все данные.

Например, можно добавить колонку в матрицу:

```
    d1 = np.array([[1, 2], [3, 4]])
    d2 = np.array([[11], [12]])
    d3 = np.append(d1, d2, axis=1)
    print(d3)
```
```
[[ 1  2 11]
 [ 3  4 12]]
```

При использовании функции необходимо, чтобы размеры складываемых массивов были совместимы.

Когда нужно сконструировать матрицу из нескольких, удобнее использовать функции vstack и hstack.

vstack(tup, dtype=None) – соединить по вертикали;

hstack(tup, dtype=None) – соединить по горизонтали.

Параметр tup – кортеж с перечислением всех складываемых массивов;

dtype – тип данных результирующего массива (если не указан, определяется исходя из типов исходный массивов.

Например, можно создать матрицу из строк:

```
    d1 = np.array([1, 2, 3])
    d2 = np.array([4, 5, 6])
    d3 = np.array([7, 8, 9])
    d4 = np.vstack((d1, d2, d3))
    print(d4)
```
```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Или из столбцов:

```
    d1 = np.array([[1], [2], [3]])
    d2 = np.array([[4], [5], [6]])
    d3 = np.array([[7], [8], [9]])
    d4 = np.hstack((d1, d2, d3))
    print(d4)
```
```
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

Альтернативный вариант предыдущего примера:

```
    d1 = np.array([1, 2, 3]).reshape(-1, 1)
    d2 = np.array([4, 5, 6]).reshape(-1, 1)
    d3 = np.array([7, 8, 9]).reshape(-1, 1)
```

```
    d4 = np.hstack((d1, d2, d3))
    print(d4)
```
```
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

6. Логические условия

С массивами ndarray можно выполнять логические операции (сравнение на равенство/неравенство, больше/меньше и т.д.). Результатом сравнения является булевый массив той же размерности, элементы которого равны True при выполнении условия и False в противном случае.

Пример:

```
   d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = d1 == 3
    print(d2)
```
```
 [[False False  True]
 [False False False]]
```

Пример:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = (d1 >= 2) & (d1 < 5)
    print(d2)
```
```
[[False  True  True]
 [ True False False]]
```

С использованием булевых массивов можно выполнять операцию логического индексирования. Если передать в качестве индекса массива булевый массив того же размера, то результатом будет другой массив, в котором сохранены элементы, соответствующие значениям True, и удалены элементы на позициях False:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = (d1 >= 2) & (d1 < 5)
    d3 = d1[d2]
    print(d3)
```
```
[2 3 4]
```

Можно использовать логические операции прямо в квадратных скобках, миную объявление дополнительных переменных:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = d1[(d1 >= 2) & (d1 < 5)]
    print(d2)
```
```
[2 3 4]
```

В отличие от среза, в результате логической индексации создается новый массив, в который копируются элементы. Изменения в массиве d2 не затронут значения в d1.

Функция where позволяет реализовать тернарный оператор над элементами массивов:

where(condition, x, y)

condition – условие;

x, y – массивы одинакового размера.

Результатом функции является массив того же размера, что x и y, в котором значение элемента при выполнении условия берется из x, в противном случае из y. Следующий пример берет четный числа из массива d1, а нечетные из d2:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = np.array([[11, 12, 13], [14, 15, 16]])
    d3 = np.where(d1 % 2 == 0, d1, d2)
    print(d3)
```
```
[[11  2 13]
 [ 4 15  6]]
```

Условие d1 % 2 == 0 фактически проверяет остаток от деления на 2. Он равен нулю для четного числа.

7. Математические Функции

В библиотеке реализовано множество математических функций, реализующих статистические, тригонометрические, алгебраические и прочие операции. Почти все, что может пригодится, скорее всего есть в библиотеке. Полный список можно найти в документации, а сейчас рассмотрим несколько часто используемых функция для примера.

Вычислить логарифм каждого элемента массива можно функцией log:

```
    d1 = np.array([1, 2, 3, 4, 5])
    print(np.log(d1))
```

```
[0.         0.69314718 1.09861229 1.38629436 1.60943791]
```

Вычислить квадратный корень:

```
    np.sqrt([4, 9, 16, 25])
```

```
[2. 3. 4. 5.]
```

Эти функции возвращают массив того же размера, что и исходный. Есть также функции, которые возвращают одно значение. Например, статистические функции, такие как максимум:

```
    np.max([4, 9, 16, 25])
```

```
25
```

Среднее значение и стандартное отклонение:

```
    np.mean(np.random.rand(10))
    np.std(np.random.rand(10))
```

```
0.4727853049269539
0.2428845235788801
```

В таблице ниже перечислены некоторые из функций.

Таблица 1. Функции NumPy

abs	Вычисляет абсолютные значения каждого элементов массива
sqrt	Вычисляет квадратный корень из каждого элемента массива (эквивалентно arr ** 0.5)
exp	Вычисляет экспоненту (ex) от каждого элемента массива
log, log10, log2, log1p	Вычисляет натуральный, десятичный логарифмы, логарифм по основанию 2 и log(1 + x), соответственно
ceil	Вычисляет наименьшее целое число большее либо равное каждого элемента массива
floor	Вычисляет наибольшее целое число меньшее либо равное каждого элемента массива
cos, cosh, sin, sinh, tan, tanh	Обычные и гиперболические тригонометрические функции
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Обратные тригонометрические функции

Еще одной полезной функцией является sort, которая сортирует и возвращает массив, переданный в качестве параметра:

```
    d1 = np.random.randint(100, size=10)
    print(d1)
    d2 = np.sort(d1)
    print(d2)
```

```
[11 85 79 35  0 15 60 73 10 70]
[ 0 10 11 15 35 60 70 73 79 85]
```

8. Операции с множествами

Библиотека NumPy имеет ряд функций для обработки массивов как множеств. Функция unique возвращает массив из одних только уникальных значений исходного массива:

```
    d1 = np.array([[1, 2, 2], [3, 3, 4]])
    d2 = np.unique(d1)
    print(d2)
```

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

Функция in1d позволяет проверить вхождение элементов одного массива в другой массив. Функция возвращает булевый массив:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = np.array([1, 3, 5])
    d3 = np.in1d(d1, d2)
    print(d3)
```

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

В таблице ниже перечислены функции работы с множествами.

Таблица 2. Функции работы с множествами

unique(x)	Возвращает отсортированные единственные элементы из x
intersect1d(x, y)	Возвращает общие элементы массивов x и y
union1d(x, y)	Возвращает объединение элементов массивов x и y
in1d(x, y)	Возвращает булев массив, указывающий содержится ли каждый элемент массива x в y
setdiff1d(x, y)	Возвращает элементы массива x, которых нет в y
setxor1d(x, y)	Возвращает элементы, которые есть либо в x, либо в y, но не в обоих массивах

9. Линейная алгебра

Библиотека NumPy содержит необходимый набор функций для вычислений из области линейной алгебры: транспонирование, сложение и перемножение матриц, вычисление собственных значений, обратных матриц и т.д.

Часть этих функций доступно в самой библиотеке, а часть реже используемых – в подпространстве linalg.

Для двумерной матрицы транспонирование означает зеркальное отображение матрицы относительно главной диагонали. Любой массив ndarray содержит метод T, который возвращает транспонированный массив^

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = d1.T
    print(d2)
```

Результат:

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

Также ровно того же результат можно добиться вызовом метода transpose и передачей ему исходной матрицы в качестве параметра:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = np.transpose(d1)
    print(d2)
```

Сложение матриц выполняется с использованием символов операций + и –:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
```

```
    d2 = np.array([[11, 12, 13], [14, 15, 16]])
    print(d1 + d2)
    print(d2 - d1)
```
```
[[12 14 16]
 [18 20 22]]
[[10 10 10]
 [10 10 10]]
```

Операции * и / выполняет поэлементное перемножение или деление матриц:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = np.array([[2, 1, 2], [1, 2, 1]])
    print(d1 * d2)
    print(d1 / d2)
```
```
[[ 2  2  6]
 [ 4 10  6]]
[[0.5 2.  1.5]
 [4.  2.5 6. ]]
```

Возможны арифметические операции между матрицами и скалярами:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    print(1 - d1)
    print(d1 / 2)
```
```
[[ 0 -1 -2]
 [-3 -4 -5]]
[[0.5 1.  1.5]
 [2.  2.5 3. ]]
```

Операции матричного умножения матрицы на вектор:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = np.array([1, 2, 3])
    print(np.dot(d1, d2))
```
```
[14 32]
```

Или матрицы на матрицу в соответствии с правилами линейной алгебры (на соответствие размеров перемножаемых матриц):

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = np.array([[11, 12], [13, 14], [15, 16]])
    print(np.dot(d1, d2))
```
```
[[ 82  88]
 [199 214]]
```

Также можно вызывать функцию матричного умножения от первой матрицы с передачей ей параметром второй матрицы:

```
    print(d1.dot(d2))
```

Или с использованием оператора @:

```
    print(d1 @ d2)
```

Результат будет тот же самый:

```
[[ 82  88]
 [199 214]]
```

В задачах машинного обучения нам могут понадобиться функции нахождения определителя (det), обратной матрицы (inv) , псевдообратной матрицы (pinv) из подпространства linalg:

```
    d1 = np.random.rand(3, 3)
    print(np.linalg.det(d1))
    print(np.linalg.inv(d1))
    print(np.linalg.pinv(d1))
```
```
0.3457249105553521

[[-1.17032681  2.06019789 -0.15938304]
 [-0.46795666 -0.09955335  1.2684454 ]
 [ 1.55842401 -0.29567334 -0.64258227]]
```

```
[[-1.17032681  2.06019789 -0.15938304]
 [-0.46795666 -0.09955335  1.2684454 ]
 [ 1.55842401 -0.29567334 -0.64258227]]
```

10. Чтение и запись файлов

Массив ndarray можно сохранить в файл функцией save, которой передается имя файла (по умолчанию файлу устанавливается расширение .npy) и переменная массива:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    np.save('ndarr-d1', d1)
```

Файл можно загрузить в переменную:

```
    d1l = np.load('ndarr-d1.npy')
    print(d1l)
```

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

В один файл можно записать несколько массивов. В этом случае файл отдельно открывается функцией open языка Python в бинарном режиме на запись (wb), а затем с использованием переменной файла выполняется запись:

```
    d1 = np.array([[1, 2, 3], [4, 5, 6]])
    d2 = np.array([[11, 12, 13], [14, 15, 16]])
    with open('ndarr-2.npy', 'wb') as f:
        np.save(f, d1)
        np.save(f, d2)
```

Чтение из этого файла выполняется аналогично (только файл открывается на чтение в бинарном режиме – rb) в том же порядке:

```
    with open('ndarr-2.npy', 'rb') as f:
        d1l = np.load(f)
        d2l = np.load(f)
    print(d1l)
    print(d2l)
```

```
[[1 2 3]
 [4 5 6]]
[[11 12 13]
 [14 15 16]]
```

11. Отображение графиков функций

Библиотека matplotlib.pyplot языка Python позволяет рисовать, отображать и сохранять в различные графические форматы графики функций. Она обладает богатым функционалом и поддерживает множество типов исходных данных, в том числе и массивы ndarray библиотеки NumPy.

Для использования библиотеки в программах нужно ее предварительно установить командой в командной строке операционной системы:

```
pip install matplotlib
```

Затем подключить к программе на языке Python:

```
import matplotlib.pyplot as plt
```

Следующий пример иллюстрирует использование pyplot для отображения графика синуса:

```
    t = np.linspace(0, 1, 100)
    y = np.sin(2 * np.pi * t)
    plt.plot(t, y)
    plt.title('График синуса')
    plt.xlabel('Время')
    plt.ylabel('Амплитуда')
    plt.show()
```

После выполнения этой программы на экране должно появиться окно следующего вида (рис. 1).

Нажатием кнопки «Save figure» (самая права снизу с изображением дискеты) можно сохранить график в файл на диск.

В примере выше функция plot рисует график, ее первый аргумент – значения по оси x, второй параметр – значения по оси y.

Функция title позволяет задать заголовок графика, xlabel – подпись оси x, ylabel – подпись оси y. Функция show выводит все на экран.

Функции plotможно передать больше параметров, задающих цвет линии, толщину, вид (сплошная, пунктирная и т.д.), а также многое другое.

Можно вывести на один рисунок несколько графиков, например:

```
t = np.linspace(0, 1, 100)
y1 = np.sin(2 * np.pi * t)
y2 = np.sin(4 * np.pi * t)
plt.plot(t, y1, 'r', label='sin 2*pi*t')
plt.plot(t, y2, 'g', label='sin 4*pi*t')
plt.title('Графики синусов')
plt.xlabel('Время')
plt.ylabel('Амплитуда')
plt.legend()
plt.show()
```

Результат показан на рис. 2.

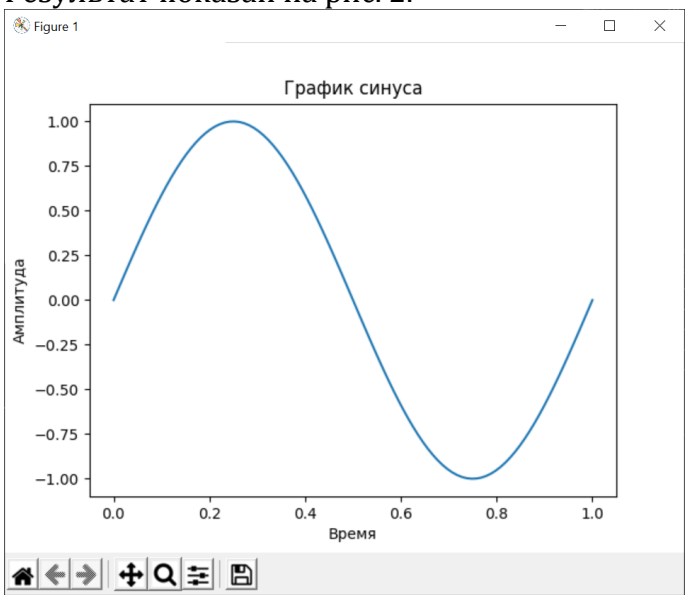

Рис. 1. Пример графика

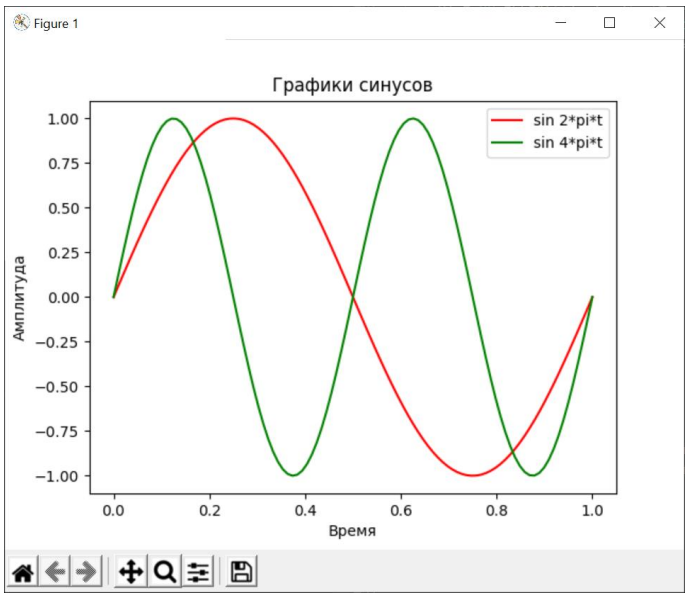

Рис. 1. Пример нескольких графиков

Библиотека matplotlib очень обширная и мощная. Подробное ее описание может занять целую книгу. Документацию по ее отдельным функция можно найти на официальном сайте [MPL]. Множество примеров и руководств, в том числе и на русском языке, имеет в сети Интернет, например [MPL-Ru-1] и [MPL-Ru-2]. Ознакомление с одним из подобных руководств будет полезным для выполнения дальнейших работ по предмету.

Литература

[Python] Python – URL: https://www.python.org/

[Python-Ru1] Учебник Python – URL: https://pymanual.github.io/

[NumPy] NumPy – URL: https://numpy.org/

[NumPy-Ru1] NumPy – русскоязычный сайт библиотеки. – URL: https://numpy.ru/

[NumPy-Ru2] NumPy documentation – URL: https://runebook.dev/ru/docs/numpy/

[MPL] Matplotlib: Visualization with Python – URL: https://matplotlib.org/

[MPL-Ru-1] Project School Matplotlib – URL: https://indico-hlit.jinr.ru/event/151/attachments/340/492/Project_school_Matplotlib_original.pdf

[MPL-Ru-2] Библиотека Matplotlib для построения графиков – URL: https://skillbox.ru/media/code/biblioteka-matplotlib-dlya-postroeniya-grafikov/