

Министерство науки и высшего образования Российской Федерации
Казанский национальный исследовательский
технологический университет

А. Н. Титов, Р. Ф. Тазиева

ОСНОВЫ РАБОТЫ С БИБЛИОТЕКОЙ NUMPY

Учебно-методическое пособие

Казань
Издательство КНИТУ
2024

УДК 004.42(075)
ББК 32.97я7
Т45

*Печатается по решению редакционно-издательского совета
Казанского национального исследовательского технологического университета*

*Рецензенты:
д-р пед. наук, доц. Ю. В. Торкунова
канд. экон. наук, доц. О. С. Семичева*

Титов А. Н.
Т45 Основы работы с библиотекой NumPy : учебно-методическое пособие / А. Н. Титов, Р. Ф. Тазиева; Минобрнауки России, Казан. нац. исслед. технол. ун-т. – Казань : Изд-во КНИТУ, 2024. – 112 с.

ISBN 978-5-7882-3470-0

Рассмотрены возможности библиотеки NumPy в области решения задач линейной алгебры, работы с массивами и генерирования случайных чисел и последовательностей. Для оценки уровня усвоения студентами пройденного материала предложены варианты заданий для самостоятельной работы.

Предназначено для бакалавров, обучающихся по направлениям подготовки 09.03.02 «Информационные системы и технологии», 22.03.01 «Материаловедение и технологии материалов», 28.03.02 «Наноинженерия», 18.03.01 «Химическая технология», 29.03.04 «Технология художественной обработки материалов», 29.03.05 «Конструирование изделий легкой промышленности», изучающих дисциплины «Алгебра и геометрия», «Информатика», «Информационные технологии», «Вычислительная математика», «Теория вероятностей и математическая статистика».

Подготовлено на кафедре информатики и прикладной математики.

УДК 004.42(075)
ББК 32.97я7

ISBN 978-5-7882-3470-0

© Титов А. Н., Тазиева Р. Ф., 2024
© Казанский национальный исследовательский
технологический университет, 2024

ОГЛАВЛЕНИЕ

Введение	5
1. NUMPY. НАЧАЛО РАБОТЫ.....	7
1.1. Объекты библиотеки NumPy	10
1.2. Типы данных	15
1.3. Вывод массивов на экран.....	16
2. РАБОТА С МАССИВАМИ	19
2.1. Атрибуты массивов	19
2.2. Создание массивов.....	20
2.3. Доступ к элементам массива	32
2.4. Сортировка элементов массива	37
2.5. Операции над множествами из массивов.....	42
2.6. Преобразования массивов. Функции <code>ravel()</code> , <code>reshape()</code> , <code>resize()</code> , <code>transpose()</code> , <code>swapaxes()</code> и <code>rot90()</code>	46
2.7. Удаление, добавление строк и столбцов	48
2.8. Разбиение массива	53
2.9. Математические операции над элементами массива.....	54
2.10. Логические операции	57
2.11. Статистические операции	59
2.12. Строковые функции.....	74
2.13. Дополнительные возможности. Чтение данных из файла	78
3. МАТРИЧНЫЕ ОПЕРАЦИИ С МАССИВАМИ. РАБОТА С МОДУЛЕМ LINALG.....	81
3.1. Умножение векторов и матриц	81
3.2. Работа с модулем <code>numpy.linalg</code>	84

4. ГЕНЕРИРОВАНИЕ СЛУЧАЙНЫХ ЧИСЕЛ.....	93
4.1. Генерирование чисел из дискретных распределений	93
4.2. Генерирование чисел из непрерывных распределений	97
4.3. Генерирование случайной выборки из заданного массива	102
Задания для самостоятельной работы	107
Литература.....	110

ВВЕДЕНИЕ

Главным разработчиком библиотек NumPy и SciPy, предназначенных для проведения научных и технических вычислений, является Трэвис Олифант (Travis Oliphant). NumPy (Numerical Python) – это библиотека с открытым исходным кодом, отделившаяся от проекта SciPy. NumPy является наследником пакетов Numeric и NumArray и была выпущена в 2005 году. Основана NumPy на библиотеке LAPACK, которая написана на Fortran. Официальный сайт библиотеки – www.numpy.org.

В силу того что NumPy базируется на Fortran, это быстрая библиотека. Библиотека поддерживает векторные операции с многомерными массивами. Эти два обстоятельства делают ее крайне удобной. Кроме базового варианта (многомерные массивы в базовом варианте), NumPy включает в себя набор модулей для решения специализированных задач:

- модуль `numpy.linalg` реализует операции линейной алгебры (простое умножение векторов и матриц есть в базовом варианте);
- модуль `numpy.random` реализует функции для работы со случайными величинами;
- модуль `numpy.fft` реализует прямое и обратное преобразование Фурье.

NumPy активно используется в сочетании с различными библиотеками, такими как Pandas, Scipy, Matplotlib, scikit-learn и т. д.

Пособие включает в себя 4 главы. В первой рассмотрены объекты библиотеки и типы данных, предоставляемые библиотекой, приведен список основных математических функций. Во второй главе показано, как можно создать основной объект библиотеки – n -мерный массив, приведены его атрибуты. Показано, как можно осуществить доступ к элементам массива, как проводить операции над массивами: арифметические, логические, статистические. Рассмотрены вопросы преобразования массивов: удаления и добавления строк и столбцов, сортировки массивов, разбиения его на несколько частей. Третья глава посвящена работе с модулем `numpy.linalg`. Показано, как можно умножать вектора и матрицы, вычислять определитель матрицы, ее ранг и норму, собственные значения и собственные вектора, находить обратную матрицу, решать системы линейных алгебраических уравнений (СЛАУ), находить приближенное решение СЛАУ методом наименьших квадра-

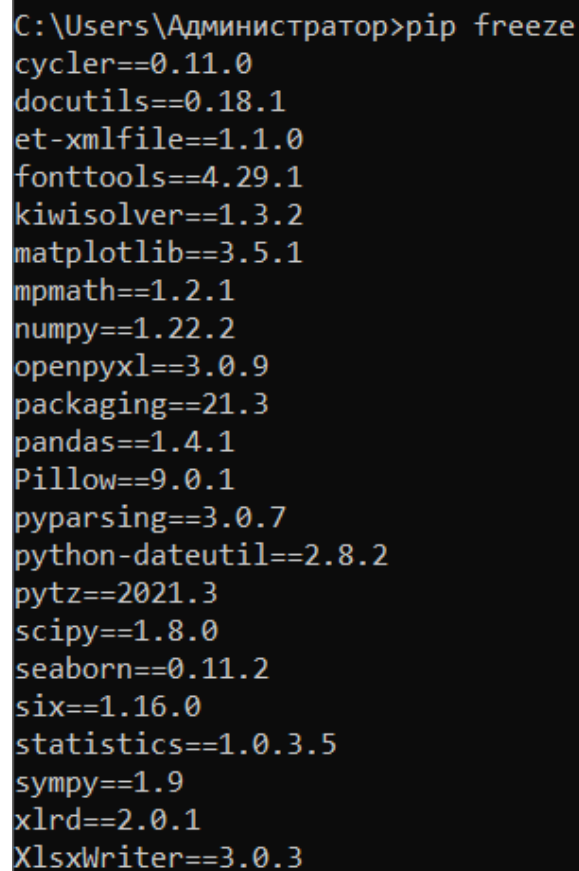
тов в случае, когда система несовместна. На большом количестве примеров продемонстрировано назначение параметров каждого рассмотренного метода или функции. В четвертой главе рассмотрена работа с модулем `numpy.random`: генерирование случайных чисел с заданным дискретным или непрерывным законом распределения, генерирование случайной выборки заданного объема из имеющегося массива.

Материал, изложенный в пособии, может быть использован при проведении лабораторных занятий по информационным технологиям, обработке экспериментальных данных, линейной алгебре, теории вероятностей и математической статистике. Используемая в пособии версия NumPy – 1.23.3.

Вся работа была проведена в Jupyter Notebook и Google Colab, кроме специально оговоренных случаев.

1. NUMPY. НАЧАЛО РАБОТЫ

Для работы с библиотекой NumPy ее нужно установить на компьютер. Некоторые среды разработки Python, такие как, например, Google Colab, уже включают в себя различные библиотеки, в том числе и NumPy. Их устанавливать не надо, а следует только импортировать. Проверить, установлена ли данная библиотека (модуль), можно, набрав в командной строке *pip freeze*. Получится список, аналогичный приведенному на рис. 1.1.



```
C:\Users\Администратор>pip freeze
cyclr==0.11.0
docutils==0.18.1
et-xmlfile==1.1.0
fonttools==4.29.1
kiwisolver==1.3.2
matplotlib==3.5.1
mpmath==1.2.1
numpy==1.22.2
openpyxl==3.0.9
packaging==21.3
pandas==1.4.1
Pillow==9.0.1
pyparsing==3.0.7
python-dateutil==2.8.2
pytz==2021.3
scipy==1.8.0
seaborn==0.11.2
six==1.16.0
statistics==1.0.3.5
sympy==1.9
xlrd==2.0.1
XlsxWriter==3.0.3
```

Рис. 1.1. Список установленных библиотек Python

Если в этом списке нет нужной библиотеки, установить ее в Python можно несколькими способами. Самый простой – установить библиотеку из репозитория PyPI (Python Package Index). Репозиторий – это место, где хранятся и поддерживаются какие-либо данные. В Python это коллекция дополнительных библиотек, хранящаяся на сервере. В настоящий момент количество библиотек в репозитории составляет более 400 тысяч.

Для установки библиотек из репозитория необходимо подключение к сети Интернет. Далее в консоли (терминале) нужно выполнить команду: `pip install <название библиотеки>`.

Установить библиотеку NumPy можно командой

```
pip install numpy
```

После ввода команды начнется загрузка установочного пакета и дополнительных библиотек, от которых зависит NumPy. Затем начнется процесс установки. Если установка пройдет успешно, в командной строке появится сообщение:

```
Successfully installed numpy
```

По умолчанию установится последняя версия библиотеки. Узнать версию, если вы работаете в Jupyter Notebook или в другой среде разработки, можно, выполнив команды

```
import numpy as np # импорт библиотеки
np.version.version
```

В программе обращаться к NumPy можно по псевдониму *np*. Это упростит чтение кода. Такой импорт широко используется сообществом программистов, поэтому рекомендуется его придерживаться, чтобы код был понятен каждому. С помощью команды *from* можно импортировать либо все функции (методы) библиотеки (символ ***), либо лишь некоторые из них (например, *sin*, *cos*). Воспользоваться какой-либо функцией библиотеки, например вычислить *sin(3)*, можно одним из способов:

```
import numpy;
y=numpy.sin(3); y
import numpy as np;
y=np.sin(3); y
from numpy import sin;
y=sin(3); y
from numpy import * ;
y=sin(3); y
import numpy as abc;
y=abc.sin(3); y
```

Использовать *abc* или любое другое имя вместо *np* не рекомендуется.

Набрав после импортирования библиотеки в строке ввода редактора IDLE *np.* и нажав клавишу <Tab>, получим отображения содержимого пространства имен NumPy:



Выбрав, например, пункты `add_newdoc_ufunc` и `count_nonzero`, получим следующую информацию:

```
>>> import numpy as np
>>> np.add_newdoc_ufunc
<built-in function _add_newdoc_ufunc>
>>> np.count_nonzero
<function count_nonzero at 0x0000020575419E10>
```

Для отображения встроенной документации можно набрать и выполнить строки:

```
import numpy as np ; np?
```

В результате получим краткую характеристику библиотеки (модуля), фрагмент которой представлен далее:

Type: module

String form: <module 'numpy' from 'C:\\Users\\Администратор\\AppData\\Local\\Programs\\Python\\Python310\\lib\\site-packages\\numpy__init__.py'>

File: c:\\users\\администратор\\appdata\\local\\programs\\python\\python310\\lib\\site-packages\\numpy__init__.py

Docstring:

NumPy

Provides

1. An array object of arbitrary homogeneous items

2. Fast mathematical operations over arrays
 3. Linear Algebra, Fourier Transforms, Random Number Generation
- How to use the documentation

Более детальную документацию вместе с учебниками и другими ресурсами можно найти по адресу – <https://numpy.org/>.

1.1. Объекты библиотеки NumPy

Основным объектом NumPy является однородный многомерный массив (в NumPy он называется `NumPy.ndarray`). Многомерность массива означает, что у него может быть несколько *осей* (*axes*). Число осей называется *рангом* (*rank*). Массив в Python – это упорядоченная структура данных, используемая для хранения однотипных объектов. По своему функциональному назначению массивы схожи со списками, но обладают некоторыми ограничениями на тип и размер входных данных. Главным отличием списка и массива является то, что списки могут хранить разнородные данные, а массивы – только данные одного типа. Такое ограничение позволяет многократно увеличить скорость вычислений, а также избежать ненужных ошибок с приведением и обработкой типов. Список – это динамическая структура. Размер списка можно изменять во время выполнения программы (удалять, добавлять элементы), чего нельзя делать с массивами. Следует также помнить, что одни и те же операции со списками и массивами могут привести к разным результатам. Массивы NumPy оптимизированы для сложных математических и статистических операций. С массивами можно проводить числовые операции с большим объемом информации в разы быстрее и намного эффективнее, чем со списками.

В NumPy можно выделить три вида массивов:

- произвольные многомерные массивы (*array*);
- матрицы (*matrix*) – двумерные квадратные массивы, для которых определены матричные операции. Для работы с матрицами можно вместо обычных массивов создавать объекты, принадлежащие к классу *matrix*;
- сетки (*grid*) – массивы, в которых записаны значения координат точек сетки (обычно ортогональной). Сетки позволяют удобно вычислять значение функций многих переменных.

В Python массив – это объект, содержащий набор данных и информацию о форме, размерности, типе данных и т. д. Как и у любого объекта, у массива можно менять атрибуты напрямую или через вызов соответствующей функции.

Кроме массивов, в NumPy имеются константы – предопределенные фиксированные значения, используемые для вычислений. Например, `np.pi` – это математическая константа, которая возвращает значение числа π , т. е. 3.141592653589793. Использование констант делает код кратким и более удобным для чтения. Константы и примеры обращения к ним представлены в табл. 1.1.

Таблица 1.1

Константы библиотеки NumPy

Описание функции	Имя функции	Примеры
Число Эйлера (e)	e	np.e
Число π	pi	np.pi
Бесконечность	Inf	np.Inf, np.inf

Для выполнения таких операций, как сложение, вычитание и деление между константой и всеми элементами массива, можно использовать арифметические операторы:

```
import numpy as np
a = np.array([0, -2, 4])
a + np.pi
```

Результат:

```
array([3.14159265, 1.14159265, 7.14159265])
```

Кроме массивов и констант, в библиотеке имеется большой набор универсальных функций. Универсальные функции – это простые математические функции, выполняющие поэлементные операции над элементами массива. NumPy предоставляет функции, которые охватывают широкий спектр операций: стандартные тригонометрические функции, функции для арифметических операций, обработки комплексных чисел, статистические функции и т. д. Некоторые наиболее часто используемые *унарные* функции (т. е. функции, принимающие на вход один массив или скаляр) приведены в табл. 1.2.

Унарные универсальные функции

Описание функции	Имя функции	Примеры
<i>Логарифмы, экспоненты, степени и корни</i>		
Логарифм по основанию e	<code>log(x)</code>	<code>np.log(x)</code>
Логарифм по основанию 2	<code>log2(x)</code>	<code>np.log2(x)</code>
Десятичный логарифм $lg(x)$	<code>log10(x)</code>	<code>np.log10(y)</code>
Логарифм по основанию n от x	<code>emath.logn(n,x)</code>	<code>np.emath.logn(2,10)</code>
Экспонента e^x	<code>exp()</code>	<code>np.exp(y)</code>
x в степени p	<code>power(x, p)</code>	<code>np.power([2, 4], 2)</code>
Квадратный корень	<code>sqrt()</code>	<code>np.sqrt(y)</code>
<i>Тригонометрические функции</i>		
Синус	<code>sin()</code>	<code>np.sin(y)</code>
Косинус	<code>cos()</code>	<code>np.cos(y)</code>
Тангенс	<code>tan()</code>	<code>np.tan(y)</code>
Кардинальный синус, $\sin(\pi x)/(\pi x)$	<code>sinc(x)</code>	<code>np.sinc(2)</code>
Арктангенс	<code>arctan()</code>	<code>np.arctan(y)</code>
Арсинус	<code>arcsin()</code>	<code>np.arcsin(0.8)</code>
Арккосинус	<code>arccos()</code>	<code>np.arccos(0.8)</code>
Преобразование углов из радианов в градусы	<code>degrees(x)</code>	<code>np.degrees(np.pi)</code>
Преобразование углов из градусов в радианы	<code>radians(x)</code>	<code>np.radians(180)</code>
<i>Гиперболические функции</i>		
Гиперболический синус	<code>sinh(x)</code>	<code>np.sinh(3)</code>
Гиперболический косинус	<code>cosh(x)</code>	<code>np.cosh(2)</code>
Гиперболический тангенс	<code>tanh(x)</code>	<code>np.tanh(7.5)</code>
Гиперболический арксинус	<code>arcsinh(x)</code>	<code>np.arcsinh(7.5)</code>
Гиперболический арккосинус	<code>arccosh(x)</code>	<code>np.arccosh(7.5)</code>
Гиперболический арктангенс	<code>arctanh(x)</code>	<code>np.arctanh(0.5)</code>

Эти функции можно применять как к скалярам, так и к последовательностям. В последнем случае функция будет применена ко всем элементам последовательности:

```
import numpy as np
c=6; v=[ 1, 2, 5]; t=np.array([1, 2, 5])
x1=np.sin(c); x2=np.log10(v); x3=np.cosh(t)
x1, x2, x3, type(np.sin)
```

Результаты:

```
(-0.27941549819892586,
 array([0.    , 0.30103, 0.69897]),
```

```
array([ 1.54308063,  3.76219569, 74.20994852]),
numpy.ufunc)
```

Универсальные функции (*ufunc*) принимают опциональный аргумент *out*, который позволяет выполнять операции прямо в заданном массиве:

```
import numpy as np
v=np.array([ 1, 2, 5], dtype='float64')
x2=np.log10(v, out=v) ; x2, v
```

Результат:

```
(array([0.    , 0.30103, 0.69897]), array([0.    , 0.30103, 0.69897]))
```

Во избежание ошибок пришлось изменить тип данных массива *v*, установленный по умолчанию (*int32*), на *float*.

В библиотеке имеются функции для определения частного и остатка от деления (табл. 1.3).

Таблица 1.3

Функции для вычисления частного и остатка от деления

Описание функции	Имя функции	Примеры
Остаток от деления	<code>mod(x,y)</code>	<code>np.mod(7,3)</code>
Остаток от деления	<code>remainder(x1, x2)</code>	<code>np.remainder(7, 3)</code>
Получение дробной и целой части аргумента	<code>modf()</code>	<code>np.modf(9.67)</code>
Частное и остаток от деления	<code>divmod(x1, x2)</code>	<code>np.divmod(7, 3)</code>

Примеры применения:

```
import numpy as np
c=6; v=[ 23, 73, 51]; t=np.array([5, 8, 17])
np.divmod(v, t)
```

После выполнения кода получили два массива. Первый из них содержит частное от деления v_i на t_i ($i = 0, 1, 2$), второй – остаток от деления:

```
(array([4, 9, 3]), array([3, 1, 0]))
```

Для округления чисел в библиотеке предусмотрены функции, представленные в табл. 1.4.

Таблица 1.4

Функции округления чисел

Описание функции	Имя функции	Примеры записи
Округление вверх	ceil()	np.ceil(y)
Округление вниз	floor()	np.floor(y)
Округление до ближайшего целого	around(x)	np.around(x)

Эти функции также работают и со скалярами, и с последовательностями:

```
import numpy as np ; c=6; v=[ 23, 73, 51]; t=np.array([5,8, 17])
v/t, np.floor(v/t), np.ceil(v/t), np.around(v/t)
```

Результаты:

```
(array([4.6 , 9.125, 3.  ]),
 array([4., 9., 3.]),
 array([ 5., 10., 3.]),
 array([5., 9., 3.]))
```

Результаты применения разных функций округления отличаются. При применении функции *floor()* результаты были округлены в меньшую, а при применении функции *ceil()* – в большую сторону. Функция *around()* произвела округления по принятым в математике правилам.

Сумму и произведение элементов массивов можно вычислить с помощью функций, представленных в табл. 1.5.

Таблица 1.5

Функции для вычисления сумм и произведений

Описание функции	Имя функции	Примеры записи
Сумма элементов массива по заданной оси	sum()	a.sum(1) np.sum(a,1)
Произведение элементов	prod()	np.prod(w)
Кумулятивная сумма элементов по заданной оси	cumsum(a)	a.cumsum(1) np.cumsum(a,1)
Кумулятивное произведение элементов по заданной оси	cumprod(a)	a.cumprod(1) np.cumprod(a,1)

Можно выделить еще одну функцию, с помощью которой можно вычислить факториал ($n!$): *math.factorial()*. Пример обращения к ней: *np.math.factorial(5)*.

1.2. Типы данных

NumPy предлагает более широкий диапазон типов данных, чем тот, который доступен в Python.

Все доступные типы можно получить, выполнив код:

```
import numpy as np; np.sctypes
```

Результат выполнения кода – словарь:

```
{'int': [numpy.int8, numpy.int16, numpy.int32, numpy.int64],  
'uint': [numpy.uint8, numpy.uint16, numpy.uint32, numpy.uint64],  
'float': [numpy.float16, numpy.float32, numpy.float64],  
'complex': [numpy.complex64, numpy.complex128],  
'others': [bool, object, bytes, str, numpy.void]}
```

Описание некоторых приведенных типов:

- `int8`, `int16`, `int32`, `int64` – целочисленные типы со знаком и разным количеством бит, выделяемым под хранение числа. Диапазоны возможных значений таких целых чисел лежат в пределах $[-2^7, 2^7-1]$, $[-2^{15}, 2^{15}-1]$, $[-2^{31}, 2^{31}-1]$, $[-2^{63}, 2^{63}-1]$;

- `uint8`, `uint16`, `uint32`, `uint64` – целочисленные типы без знака, различающиеся количеством бит, выделяемых под хранение числа. Целые числа без знака, лежащие в пределах от 0 до 2^8-1 , $2^{16}-1$, $2^{32}-1$ и $2^{64}-1$ соответственно;

- `float32`, `float64` – типы с плавающей запятой с разными уровнями точности. 1 бит на знак, 8-битная экспонента, 23-битная мантисса в первом случае и 1 бит на знак, 11-битная экспонента и 52-битная мантисса – во втором;

- `complex64`, `complex128` – типы комплексных чисел с разными уровнями точности. Комплексное число, представленное двумя 32- или 64-битными float (с действительной и мнимой частями).

Создать массив с определенным типом данных можно, передав параметр *dtype* при вызове функции *np.array()*. Создадим, например, целочисленный 32-битный массив:

```
import numpy as np  
a = np.array([1, 3, 7], dtype='int32')  
print(a, a.dtype)
```

Результат:

```
[1 3 7] int32
```

Поскольку тип данных массива установлен равным *int32*, каждый элемент массива представлен в виде 32-разрядного целого числа.

Если при создании массива тип данных не задан явно, подходящий тип подбирается автоматически:

```
import numpy as np
a = np.array([0, -2, 4])
a.dtype
```

Результат:

```
dtype('int32')
```

Попытки присвоить элементам массива значения, выходящие за пределы области возможных значений, приводят к «неожиданным» результатам. Выполнив код

```
import numpy as np ; a = np.array([0, -10, 4, 258], dtype='uint8'); a
```

получили:

```
array([ 0, 246,  4,  2], dtype=uint8)
```

Данные были преобразованы, потому что в массиве *a* присутствовали числа, выходящие за допустимые пределы. Вместо отрицательного числа -10 появилось число 246 (256-10), а вместо числа 258 число 2 (258-256).

В NumPy можно преобразовать тип данных массива с помощью метода *astype()*. Например:

```
import numpy as np
a = np.array([1, 3, 5, 7])
b = a.astype('float')
print(a, a.dtype, b, b.dtype)
```

Результаты:

```
[1 3 5 7] int32 [1. 3. 5. 7.] float64
```

1.3. Вывод массивов на экран

Если массив слишком большой, при его выводе на экран NumPy автоматически скрывает центральную часть массива и выводит только его крайние элементы. Выведем, например, на экран содержимое массива целых чисел от 0 до 9999:

```
import numpy as np; print(np.arange(0, 10000, 1))
```


Результат:

```
[ 0  1  2 ... 9997 9998 9999]
```

При необходимости увидеть весь массив используют метод `set_printoptions()`. С помощью этой функции можно настроить вывод массивов «под себя».

Синтаксис метода:

```
numpy.set_printoptions(precision=None, threshold=None, edgeitems=None,  
linewidth=None, suppress=None, nanstr=None, infstr=None, formatter=  
None, sign=None, floatmode=None, *, legacy=None)
```

Параметры метода определяют способ отображения чисел с плавающей запятой, массивов и других объектов NumPy:

- *precision* – целое число или значение None. Количество разрядов точности для вывода числа с плавающей запятой (по умолчанию 8);

- *threshold* – целое число. Если указано значение, меньшее количества элементов массива, то на экране будут показаны не все элементы. Например, выполнение операторов `np.set_printoptions(threshold=4); np.arange(11)` приведет к выводу на экран `array([0, 1, 2, ..., 8, 9, 10])`, тогда как `np.set_printoptions(threshold=12); np.arange(11)` приведет к появлению на экране всех элементов массива: `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])`;

- *edgeitems* – целое число. Количество элементов в начале и в конце каждой размерности массива (по умолчанию 3). Например, выполнив код: `np.set_printoptions(threshold=1, edgeitems=2); np.arange(11)` мы увидим на экране два первых и два последних значения массива: `array([0, 1, ..., 9, 10])`;

- *linewidth* – целое число. Количество символов в строке, после которых осуществляется перенос (по умолчанию 75). Так, выполнение операторов `np.set_printoptions(threshold=20, linewidth=60); np.arange(20)` приведет к появлению на экране двух строк:

```
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,  
       13, 14, 15, 16, 17, 18, 19])
```

В первой из них 60 символов (включая пробелы, запятые);

- *suppress* – булева переменная. Если значение параметра равно True, числа выводятся на экран с плавающей запятой. В этом случае числа, равные нулю при заданной точности, будут печататься как ноль. Если значение параметра равно False, то используется экспоненциальное представление, когда абсолютное значение наименьшего

числа меньше 10^{-4} или отношение максимального абсолютного значения к минимальному абсолютному значению больше 10^3 . Значение параметра по умолчанию – False;

– *nanstr* – строка символов. Строковое представление NaN (по умолчанию 'nan'). Например, выполнение кода `np.set_printoptions(precision=3); np.array([1, .98, np.nan])` приведет к появлению на экране `array([1. , 0.98, nan])`, а выполнение кода `np.set_printoptions(precision=3, nanstr='b'); np.array([1.,98, np.nan])` – к появлению `array([1. , 0.98, b])`;

– *infstr* – строка символов. Строковое представление inf (по умолчанию 'inf'). Например, выполнение кода `np.set_printoptions(precision=3); np.array([1, .98, np.inf])` приведет к появлению на экране `array([1. , 0.98, inf])`, а выполнение кода `np.set_printoptions(precision=3, nanstr='бесконечность'); np.array([1.,98, np.nan])` – к появлению `array([1., 0.98, бесконечность])`;

– *sign* – строка символов. Принимает одно из значений: '-', '+' или ' '. Управляет печатью знака чисел с плавающей запятой. Если значение равно '+', перед такими положительными числами печатается знак плюс. Например, выполнив код: `np.set_printoptions(sign='+'); np.array([1., 98, -78])`, получим: `array([+1., +98., -78.])`. Если значение параметра равно ' ', в позиции знака положительных значений всегда печатает пробел (символ пробела). Если значение равно '-', знак + перед положительными числами не печатается. Значение по умолчанию '-';

– *formatter* – позволяет более тонко управлять печатью массивов. Более подробную информацию об этих и других параметрах можно найти по адресу – <https://numpy.org/>.

Установленные опции печати будут действовать до их изменения. Вернуть параметры по умолчанию можно следующим образом:

```
np.set_printoptions(edgeitems=3, infstr='inf',  
linewidth=75, nanstr='nan', precision=8, sign='-',  
suppress=False, threshold=1000, formatter=None)
```

Чтобы временно переопределить параметры, можно использовать *printoptions()* в менеджере контекста *with*:

```
with np.printoptions(precision=2, suppress=True, threshold=3):  
    t = np.linspace(0,1,10); print(t)  
print(t)
```

Результаты:

```
[0.    0.11 0.22 ... 0.78 0.89 1.    ]  
[0.    0.11111111 0.22222222 0.33333333 0.44444444 0.55555556  
 0.66666667 0.77777778 0.88888889 1.    ]
```

2. РАБОТА С МАССИВАМИ

Массив в Python – это упорядоченная структура данных, используемая для хранения однотипных объектов. В NumPy реализовано множество операций для работы с массивами:

- создание массивов и их модификация (изменение формы, транспонирование, поэлементные операции);
- выбор отдельных элементов массива;
- арифметические, логические и строковые операции над элементами массивов, вычисление функций от элементов массива;
- решение задач линейной алгебры (системы линейных уравнений, собственные вектора, собственные значения);
- создание массивов из случайных чисел;
- быстрое преобразование Фурье.

2.1. Атрибуты массивов

Массивы NumPy являются объектами класса *ndarray*. Наиболее важными атрибутами класса *ndarray* являются атрибуты *ndim*, *shape*, *size*, *dtype*, *itemsize*.

Число осей (измерений) массива можно узнать с помощью атрибута *ndim*:

```
import numpy as np
d=np.array([[2,5,7,-2],[3,-1.23,67,1e-02]])
d.ndim # 2 – двумерный массив
```

Атрибут *shape* – это кортеж натуральных чисел, показывающий длину массива по каждой оси. Число элементов кортежа *shape* равно рангу массива, т. е. *ndim*. Добавив к коду строку *d.shape* и выполнив его, получим кортеж (2, 4): в массиве 2 строки и 4 столбца.

Общее количество элементов в массиве можно узнать с помощью атрибута *size*: *d.size* # 8. Оно равно произведению всех элементов атрибута *shape*.

Объект, описывающий тип данных в массиве, можно получить с помощью атрибута *dtype*: *d.dtype* # *dtype('float64')*.

Размер каждого элемента в массиве в байтах можно узнать с помощью атрибута *itemsize*:

```
import numpy as np
d=np.array([[2,5,7,-2],[3,-1.23,67,1e-02]])
d.dtype, d.itemsize # (dtype('float64'), 8)
```

d представляет собой массив, содержащий 64-разрядные десятичные числа, который использует 8 байт памяти на элемент. Атрибут *itemsize* возвращает 8 как размер каждого элемента.

Атрибут *data* содержит элементы массива. Обычно этот атрибут не используется, так как обращаться к элементам массива проще всего с помощью индексов.

2.2. Создание массивов

В NumPy существует много способов создания массивов: из списка, кортежа, из функции, с помощью сеток, из файла.

Самый простой из них – создать массив из вложенного списка Python. Для этого используют функцию *array()*. Список может быть любой формы: одномерный, двумерный, трехмерный и т. д. Двумерный массив состоит из одномерных, трехмерный – из двумерных и т. д. В данном пособии мы ограничились одномерными и двумерными массивами. Тип элементов массива зависит от типа элементов исходной последовательности, но его можно и переопределить в момент создания.

Одномерный массив можно создать таким образом:

```
import numpy as np
a=np.array([1,3,7])
a
```

Результат: array([1, 3, 7])

Покажем, как можно создать двумерный массив D:

$$D = \begin{pmatrix} 2 & 5 & 7 & -2 \\ 3 & -1.23 & 67 & 1e-02 \end{pmatrix}.$$

Выполнив код

```
import numpy as np ; d=np.array([[2,5,7,-2],[3,-1.23,67,1e-02]]); d
```

получим:

```
array([[ 2.00e+00,  5.00e+00,  7.00e+00, -2.00e+00],
       [ 3.00e+00, -1.23e+00,  6.70e+01,  1.00e-02]])
```

Добавление в исходные данные числа с плавающей запятой ($1e-02$) привело к тому, что все числа в массиве d представлены в экспоненциальной форме.

В терминологии NumPy приведенный в примере массив a имеет одну ось (термин «axis» из документации) длиной 3 элемента, а массив d имеет две оси: первая имеет длину 2 (количество столбцов, номер оси 0), а длина второй оси равна 4 (количество строк, номер оси $axis = 1$).

Аналогично создают массив из кортежа:

```
import numpy as np ; a=np.array((1,3,7)) ; a # array([1, 3, 7])
```

Можно преобразовать список или кортеж в массив с помощью функции `asarray()`: `t=(1,3,7); a=np.asarray(t)`.

Массив можно создать с помощью функции. Для этого используют метод `fromfunction()`. Следующим образом можно создать массив из трех строк и трех столбцов:

```
import numpy as np
def z(x, y):
    return x**2+ + y**2
np.fromfunction(z, (3, 3))
```

Результат:

```
array([[0., 1., 4.],
       [1., 2., 5.],
       [4., 5., 8.]])
```

Еще один способ создания массивов – из сеток. Для этого используют функции `arange()`, `linspace()`, `logspace()`, `meshgrid()`, `mgrid[]`, `ogrid[]`.

Функции `arange()` имеет следующий синтаксис:

```
np.arange ( [ start , ] stop , [ step , ] dtype=None , * , like=None )
```

Функция возвращает равномерно распределенные значения в пределах заданного интервала и может вызываться с различным количеством позиционных аргументов:

- `arange(stop)`: значения генерируются в пределах полуоткрытого интервала, т. е. интервала $[0, stop)$;
- `arange(start, stop)`: значения генерируются в пределах полуоткрытого интервала $[start, stop)$ с шагом 1;

– `arange(start, stop, step)`: значения генерируются в пределах полуоткрытого интервала с шагом `step`.

Функция `arange()` содержит два необязательных параметра: `dtype` и `like`. Первый задает тип выходного массива. Вторым – ссылочный объект, позволяющий создавать последовательности структуры `array_like`.

Создать массив чисел, начиная с числа `start` (включая) до числа `stop` (не включая) с шагом `step`, например от 3 до 7 с шагом 0.5, можно таким образом:

```
import numpy as np
a=np.arange(3,7,.5)
a
```

Результат:

```
array([3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5])
```

По умолчанию `start = 0`, `step = 1`, поэтому возможны варианты с одним и двумя параметрами:

```
import numpy as np ; c=np.arange(7)
c # array([0, 1, 2, 3, 4, 5, 6])
d=np. arange(7,11) ; d # array([ 7,  8,  9, 10])
```

Чтобы задать массив, состоящий из n равноотстоящих точек от `start` до `stop`, можно воспользоваться функцией `linspace()`. Синтаксис функции:

```
linspace ( start , stop , n = 50 , endpoint = True , retstep = False ,
dtype = None , axis = 0 )
```

Другие параметры функции:

– `endpoint` – параметр, определяющий, входит ли точка `stop` в сгенерированный набор значений. Если значение параметра равно `True`, `stop` – это последняя точка выборки. В противном случае значение `stop` в выборку не включается. Значение параметра по умолчанию – `True`;

– `retstep` – параметр, определяющий, выводить ли на экран значение шага. Значение параметра по умолчанию – `False`;

– `dtype` – параметр, определяющий тип выходного массива. Если значение `dtype` не указано, тип данных определяется значениями `start` и `stop`.

Создадим с помощью функции `linspace()` массив из 7 чисел, равномерно распределенных на интервале `[0, 3]`:

```
import numpy as np ; p=np.linspace(0,3,7)
p # array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. ])
```

Добавим в функцию *linspace()* параметры *endpoint*, *retstep* и *dtype*:

```
import numpy as np
p=np.linspace(0,3,7,False,True,dtype=str) ; p
```

Результат:

```
(array(['0.0', '0.42857142857142855', '0.8571428571428571',
'1.2857142857142856', '1.7142857142857142', '2.142857142857143',
      '2.571428571428571'], dtype='<U32'),
0.42857142857142855)
```

Было сгенерировано 7 значений, начиная с 0, с шагом $(3-0)/7=0.42857142857142855$. Результат преобразован в символьный массив. На экран выведено значение шага. Значение шага можно вывести отдельно. В данном примере это элемент массива *p* с индексом 1: *p*[1].

Используя метод *astype()*, можно привести массив к другому типу. В качестве параметра метода *astype()* в этом случае указывается желаемый тип:

```
import numpy as np ; p=np.linspace(0,3,7)
t=p.astype(int)
t # array([0, 0, 1, 1, 2, 2, 3])
```

В результате работы кода из массива *p* с элементами 0, 0.5, 1, 1.5, 2, 2.5, 3 был получен массив *t* с целыми элементами.

Числа, равномерно распределенные по логарифмической шкале, можно получить с помощью функции *logspace()*. Синтаксис функции:

```
numpy.logspace(start, stop, num=50, endpoint=True, base=10.0,
dtype=None, axis=0)
```

Параметры функции:

- *start* – начальное значение последовательности. Определяется по формуле $\text{base}^{\text{start}}$;

- *stop* – конечная точка последовательности, если *endpoint* = True. Вычисляется по формуле $\text{base}^{\text{stop}}$;

- *num* – количество точек сетки;

- *endpoint* – параметр, определяющий, входит ли точка *stop* в сгенерированный набор значений. Если значение параметра равно True,

stop – последняя точка выборки. В противном случае значение *stop* в выборку не включается. Значение параметра по умолчанию – True;
– *base* – основание лог-пространства. По умолчанию 10.0;
– *dtype* – параметр, определяющий тип выходного массива. Если значение *dtype* не указано, тип данных определяется значениями *start* и *stop*, но никогда не будет целым числом. Будет выбран тип *float*, даже если аргументы будут создавать массив целых чисел.

Пример:

```
import numpy as np ; np.logspace(2, 4, num=5, base=2)
```

Результат:

```
array([ 4.        ,  5.65685425,  8.        , 11.3137085 , 16.        ])
```

Первое значение массива равно $2^2=4$, второе – $2^{2.5}$, третье – 2^3 , четвертое – $2^{3.5}$ и пятое – 2^4 .

Векторы, описывающие точки ортогональной сетки, можно создать с помощью функции *meshgrid()*. Синтаксис функции:

```
numpy.meshgrid(*xi, copy=True, sparse=False, indexing='xy')
```

Параметры функции:

– *x1, x2, ... , xn* – структуры данных *array_like*. *array_like* – это входные данные в любой форме, которые можно преобразовать в массив. Одномерные массивы, задающие координаты сетки;

– *sparse* – булева переменная. При значении параметра, равном True, меняется вид выводимых массивов;

– *indexing* – параметр, принимающий одно из значений: 'xy' или 'ij'. Декартово ('xy' по умолчанию) или матричное ('ij') индексирование вывода. В двумерном случае с входными данными длины M и N выходные данные имеют форму (N, M) для индексации 'xy' и (M, N) для индексации 'ij'.

Примеры:

```
import numpy as np
nx, ny = (3, 5) # количество элементов сетки по осям
x = np.linspace(0, 6, nx); y = np.linspace(0, 3, ny);
[a, b]=np.meshgrid(x, y)
a,b
```

В результате работы кода получили два массива. Первый содержит 5 строк со значениями x, второй – 3 столбца со значениями y:


```
[array([[0., 3., 6.],
        [0., 3., 6.],
        [0., 3., 6.],
        [0., 3., 6.],
        [0., 3., 6.])),
 array([[0. ,  0. ,  0. ],
        [0.75, 0.75, 0.75],
        [1.5 ,  1.5 ,  1.5 ],
        [2.25, 2.25, 2.25],
        [3. ,   3. ,  3. ]])]
```

Каждому элементу $a[i,j]$ соответствует свой элемент $b[i,j]$.

Подключив библиотеку Matplotlib, получим графическую интерпретацию полученного результата:

```
import matplotlib.pyplot as plt; plt.plot(a, b, 'go'); plt.show()
```

Результат работы кода – график, представленный на рис. 2.1.

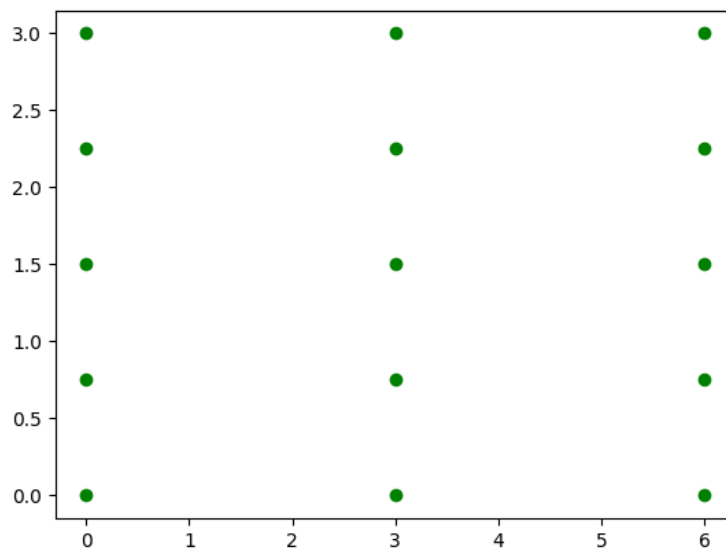


Рис. 2.1. Координатная сетка, соответствующая функции *meshgrid()*

Добавив в функцию *meshgrid()* параметр *sparse* (`np.meshgrid(x, y, sparse=True)`) и выполнив код, получим более компактный вывод:

```
[array([[0., 3., 6.])),
 array([[0. ],
        [0.75],
        [1.5 ],
        [2.25],
        [3. ]])]
```

Полный набор данных, описывающий многомерную равномерную ортогональную сетку (X, Y) или (X, Y, Z), можно получить с помощью функции *mgrid[]*:

```
np.mgrid[0:3, -3:6]
```

Результат:

```
array([[[ 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [ 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [ 2, 2, 2, 2, 2, 2, 2, 2, 2]],
       [[-3, -2, -1, 0, 1, 2, 3, 4, 5],
        [-3, -2, -1, 0, 1, 2, 3, 4, 5],
        [-3, -2, -1, 0, 1, 2, 3, 4, 5]])
```

Сокращенный набор данных, описывающий многомерную равномерную ортогональную сетку (X, Y) или (X, Y, Z), можно получить с помощью функции *ogrid[]*:

```
np.ogrid[0:3,-3:6]
```

Результат:

```
[array([[0],
        [1],
        [2]]),
 array([[-3, -2, -1, 0, 1, 2, 3, 4, 5]])]
```

Создать массив можно из файла. Создадим, например, в Word, файл с содержимым:

```
1 -2 13 2 15 7 112
2 7 -1 7 34 41 56
6 17 60 81 17 13 4
```

Сохраним его в той же директории, что и блокнот Jupyter Notebook как файл с именем *data.txt* (тип файла «Обычный текст»). Эти данные можно записать в NumPy-массив с помощью функции *genfromtxt()*:

```
import numpy as np
fdata = np.genfromtxt('data.txt', delimiter=' ', dtype='int32'); fdata
```

В результате работы кода получили двумерный массив:

```
array([[ 1, -2, 13,  2, 15,  7, 112],
       [ 2,  7, -1,  7, 34, 41,  56],
       [ 6, 17, 60, 81, 17, 13,  4]])
```

Краткий комментарий к коду. С помощью функции `genfromtxt()` были получены данные из файла *data.txt*. В функции указаны имя считываемого файла, разделитель, чтобы NumPy понимал, где начинаются и заканчиваются числа, и нужный нам тип. В качестве разделителя указан пробел. Числа приведены к формату `int32`.

В библиотеке предусмотрено множество функций для создания массивов специального вида, из которых рассмотрим функции `zeros()`, `diag()`, `diagonal()`, `trace()`, `eye()`, `identity()`, `ones()`, `full()`, `tri()`, `tril()`, `triu()`.

Для создания массива NumPy с заданным количеством измерений, где каждый элемент будет равен 0, используют функцию `zeros()`. Синтаксис функции:

```
numpy.zeros(shape, dtype=float, order='C', *, like=None)
```

Параметры функции:

- *shape* – целое число или кортеж целых чисел. Например: 3 или (2,3);

- *dtype* – тип данных. Желательный тип данных для массива, например `numpy.int8` (или `'int8'`). Значение по умолчанию – `numpy.float64`;

- *order* – переменная, определяющая способ хранения данных, принимающая одно из значений `{'C', 'F'}`. Значение по умолчанию – `'C'`. Указывает, хранить ли многомерные данные в памяти по строкам (стиль C) или по столбцам (стиль Fortran);

- *like* – ссылочный объект, позволяющий создавать массивы *array_like*, которые не являются массивами NumPy.

Пример:

```
np.zeros((2,3),np.int8)
```

Результат:

```
array([[0, 0, 0],  
       [0, 0, 0]], dtype=int8)
```

Функция `diag()` предназначена для извлечения из двумерного массива (матрицы) диагональных элементов. Синтаксис функции: `numpy.diag(v, k=0)`

Параметры функции:

- *v* – структура данных *array_like*. Если *v* – двумерный массив, то результатом работы функции является копия его *k*-й диагонали. Если *v* является одномерным массивом, то функция возвращает двумерный массив с элементами вектора *v* на *k*-й диагонали;

– k – целое число, необязательный параметр. Если необходима диагональ, находящаяся выше главной диагонали, то задается $k > 0$. Если требуется диагональ ниже главной диагонали, то $k < 0$.

Примеры:

```
import numpy as np
a=np.array([2,6,8,12])
np.diag(a,2)
```

Результатом будет матрица, вторая диагональ которой состоит из элементов массива a :

```
array([[ 0,  0,  2,  0,  0,  0],
       [ 0,  0,  0,  6,  0,  0],
       [ 0,  0,  0,  0,  8,  0],
       [ 0,  0,  0,  0,  0, 12],
       [ 0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0]])
```

Пример функции с двумерным массивом в качестве аргумента:

```
import numpy as np
d=np.array([[2,5,7],[3,-1,4],[-4,6,12]])
d,np.diag(d,-1)
```

Результат:

```
(array([[ 2,  5,  7],
       [ 3, -1,  4],
       [-4,  6, 12]]),
 array([3, 6]))
```

Матрица v не обязательно должна быть квадратной:

```
import numpy as np
d=np.array([[2,5,7],[3,-1,4]]) ; d, np.diag(d)
```

Результат:

```
(array([[ 2,  5,  7],
       [ 3, -1,  4]]),
 array([ 2, -1]))
```

В библиотеке есть аналогичная функция *diagonal()*, имеющая схожий синтаксис:

```
numpy.diagonal(a, offset=0, axis1=0, axis2=1)
```

Здесь *offset* – целое число, задающее смещение диагонали относительно главной диагонали. Может быть положительным или отрицательным. По умолчанию главная диагональ – (0). Параметры *axis1* и *axis2* используются, когда *a* имеет более двух измерений.

Приведем пример, демонстрирующие работу функции *diagflat()*:

```
import numpy as np
np.diagflat([[1,2], [3,4]],2)
```

Результатом работы кода будет матрица:

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

Элементы исходной матрицы $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ стали элементами второй диагонали созданной матрицы.

С помощью функции *trace()* вычисляют сумму элементов диагонали матрицы. Синтаксис функции:

```
numpy.trace(a, offset=0, axis1=0, axis2=1, dtype=None, out=None)
```

Параметры функции:

- *a* – структура данных *array_like*. Входной массив, из которого берутся диагонали;

- *offset* – целое число, задающее смещение диагонали относительно главной диагонали. Может быть как положительным, так и отрицательным. По умолчанию значение – *offset*=0 (главная диагональ);

- *axis1*, *axis2* – целые числа, необязательный параметр. Оси, которые будут использоваться в качестве первой и второй осей двумерных подмассивов, из которых должны быть взяты диагонали. По умолчанию используются первые две оси *a*;

- *dtype* – определяет тип данных возвращаемого массива;

- *out* – массив, в который помещается результат.

Пример:

```
import numpy as np
d=np.array([[2,5,7],[3,-1,4],[-4,6,12]]) ; d, np.trace(d,-1)
```

Результат:

```
(array([[ 2,  5,  7],
       [ 3, -1,  4],
       [-4,  6, 12]]),
```

9)

Выведен исходный массив и полученная сумма элементов диагонали, расположенной под главной диагональю (9).

Для получения двумерного массива с единицами по диагонали и нулями в других местах используют функцию *eye()*. Ее синтаксис:

```
numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C', *,
like=None)
```

Параметры функции:

- N , M – целые числа, задающие количество строк и столбцов получаемого массива. Если M не задано, то предполагается $M=N$;
- k – целое число. Номер диагонали;
- *dtype* – тип исходных данных возвращаемого массива;
- *order* – переменная, определяющая способ хранения данных, принимающая одно из значений {'C', 'F'};
- *like* – ссылочный объект, позволяющий создавать массивы, которые не являются массивами NumPy.

Пример:

```
import numpy as np ; np.eye(3, k=1, dtype=int)
```

Результат:

```
array([[0, 1, 0],
       [0, 0, 1],
       [0, 0, 0]])
```

Получить единичную матрицу можно с помощью функции *identity()*. Синтаксис функции:

```
numpy.identity(n, dtype=None, *, like=None)
```

Здесь n – количество строк (и столбцов) в выводе n на n .

Пример:

```
import numpy as np
np.identity(4)
```

Результат:

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Получить массив, состоящий из единиц, можно с помощью функции *ones()*. Ее синтаксис:

```
numpy.ones(shape, dtype=None, order='C', *, like=None)
```

Назначение параметров этой функции идентично параметрам функции *zeros()*.

Создать массив заданной формы и типа, заполненный заданными значениями можно с помощью функции *full()*. Синтаксис функции:

```
numpy.full(shape, fill_value, dtype=None, order='C', *, like=None)
```

Создается массив с заданным количеством строк и столбцов, заполненный значениями *fill_value*.

Пример:

```
import numpy as np
np.full((2,4),[7, 3, -5, 33])
```

Результат:

```
array([[ 7,  3, -5, 33],
       [ 7,  3, -5, 33]])
```

Для создания массива с единицами на заданной диагонали и ниже нее и нулями в других местах используют функцию *tri()*. Ее синтаксис:

```
numpy.tri(N, M=None, k=0, dtype=<class 'float'>, *, like=None)
```

Параметры функции идентичны параметрам функции *eye()*.

Пример:

```
import numpy as np; np.tri(3, 4, 1, dtype=int)
```

Результат:

```
array([[1, 1, 0, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 1]])
```

Функция *tril()* возвращает копию массива с обнуленными элементами выше *k*-й диагонали:

```
numpy.tril(m, k=0)
```

Параметры функции:

- *m* – структура данных *array_like*. Входной массив;
- *k* – целое число. Номер диагонали, выше которой обнуляются элементы. По умолчанию *k* = 0 (главная диагональ).

Пример:

```
import numpy as np
d=np.array([[2,5,7],[3,-1,4],[-4,6,12]]) ; d,np.tril(d,-1)
```

Результат:

```
(array([[ 2,  5,  7],
        [ 3, -1,  4],
        [-4,  6, 12]]),
array([[ 0,  0,  0],
        [ 3,  0,  0],
        [-4,  6,  0]]))
```

Для получения копии массива с обнуленными элементами ниже k -й диагонали используют функцию *triu()*:

```
numpy.triu(m, k=0)
```

Параметры функции аналогичные параметрам функции *tril()*.

Еще один способ создания массивов – с помощью функций *numpy.zeros_like()*, *numpy.ones_like()*, *numpy.empty_like()*, которые возвращают массив той же структуры, что массив, переданный в функцию. Например:

```
import numpy as np
a=[[2,3,4], [5, 6, 7]]
np.ones_like(a)
```

Результат – массив из единиц, состоящий из двух строк и трех столбцов:

```
array([[1, 1, 1],
        [1, 1, 1]])
```

2.3. Доступ к элементам массива

После того как массив создан, доступ к его элементам осуществляется по индексам, которые начинаются с нуля:

```
import numpy as np
c=np.arange(7,11)
print(c)
c[0],c[-2]
```


Результат:

[7 8 9 10]

(7, 9)

Индекс «-2» означает: второй с конца.

При обращении к элементам массива можно использовать срезы. Для одномерных массивов эта операция аналогична стандартному срезу в Python:

```
import numpy as np ; c=np.arange(7,16)
c,c[0:8:2]
```

Результат:

(array([7, 8, 9, 10, 11, 12, 13, 14, 15]), array([7, 9, 11, 13]))

На экран были выведены массив *c* и массив, содержащий нулевой, второй, четвертый и шестой элементы массива *c*.

Если шаг равен 1, то его в срезе можно не указывать:

```
import numpy as np
c=np.arange(7,11)
c[1:3] # array([8, 9])
```

Можно пропустить индекс:

```
import numpy as np
c=np.arange(7,11)
c[:3] # array([7, 8, 9])
```

c[:3] означает: элементы массива *c* с номерами до 3, т. е. 0, 1 и 2.

Индексация возможна и с использованием списка или массива целочисленных индексов вдоль каждой оси (*fancy indexing*):

```
import numpy as np
c=np.arange(7,11)
c,c[[1,0,-1]]
```

Результат:

(array([7, 8, 9, 10]), array([8, 7, 10]))

На экран были выведены массив *c* и массив с тремя элементами, содержащий первый, нулевой и последний элемент массива *c*. Элементы в последнем массиве расположены в заданном порядке: сначала первый элемент массива *c*, затем нулевой и последний.

Доступ к элементам двумерного массива возможен как к конкретному элементу, так и к строке. Пусть задана матрица $E = \begin{pmatrix} 2 & 5 & 7 & -2 \\ 3 & -6 & 67 & 12.6 \end{pmatrix}$. Покажем, как вывести на экран отдельные элементы матрицы и всю строку:

```
import numpy as np
e=np.array([[2,5,7,-2],[3,-6,67,12.6]])
e[1],e[1,1],e[1][0] # (array([ 3. , -6. , 67. , 12.6]), -6.0, 3.0)
```

На экран были выведены первая строка, а также элементы $e_{1,1}$ и $e_{1,0}$ матрицы E .

Применение индексации для двумерных массивов позволяет брать подмножество строк и вывести его в определенном порядке:

```
import numpy as np
e=np.array([[2, 5],[7, -23],[-6, 67],[12.6, 45]])
e, e[[3,2]], e[np.array([1,3])]
```

Результаты:

```
(array([[ 2. ,  5. ],
        [ 7. , -23. ],
        [-6. , 67. ],
        [12.6, 45. ]]),
 array([[12.6, 45. ],
        [-6. , 67. ]]),
 array([[ 7. , -23. ],
        [12.6, 45. ]]))
```

Можно передать несколько массивов индексов. В результате получим одномерный массив с элементами, соответствующими заданному массиву индексов:

```
import numpy as np
e=np.array([[2, 5],
            [7, -23],
            [-6, 67],
            [12.6, 45]])
e[[3, 0],
   [1, 0]]
```

Результат:

```
array([45., 2.])
```

На экран выведен массив с элементами, соответствующими кортежам индексов (3, 1) и (0, 0) – элементы $e[3, 1]$ и $e[0, 0]$.

Для доступа к многомерным массивам также можно использовать срезы, задавая их диапазон отдельно для каждой оси. Таким образом, можно взять срез отдельной части матрицы, указав, какие строки и столбцы должны в него попасть:

```
import numpy as np
e=np.array([[2,5,7,-2],[3,-6,67,12.6]])
e[1,:],e[:,1] # (array([ 3. , -6. , 67. , 12.6]), array([ 5., -6.]))
```

Выполнение кода

```
import numpy as np
e=np.array([[2,5,7,-2],[3,-6,67,12.6]])
e[:,::-1]
```

приведет к выводу массива, в котором элементы каждой строки будут идти в обратном порядке:

```
array([[ -2. ,  7. ,  5. ,  2. ],
       [12.6, 67. , -6. ,  3. ]])
```

Аналогично можно «развернуть» элементы столбцов:

```
import numpy as np
e=np.array([[2,5,7,-2],[3,-6,67,12.6]])
e[:,::-1,:]
```

Результат:

```
array([[ 3. , -6. , 67. , 12.6],
       [ 2. ,  5. ,  7. , -2. ]])
```

Оставить в массиве два первых столбца двумерного массива e можно следующим образом:

```
import numpy as np
e=np.array([[2,5,7,-2],[3,-6,67,12.6]])
g=e[:,0:2]; e, g
```

Результат:

```
(array([[ 2. ,  5. ,  7. , -2. ],
        [ 3. , -6. , 67. , 12.6]]),
 array([[ 2.,  5.],
        [ 3., -6.]])
```

Можно изменять значения в NumPy-массиве с помощью той же операции доступа к элементам. Выполним код:

```
import numpy as np
e=np.array([[2,5,7,-2],[3,-6,67,12.6]])
z=e
z[0,1]=100
e
```

Результат:

```
array([[ 2. , 100. ,  7. , -2. ],
       [ 3. , -6. , 67. , 12.6]])
```

Исходный массив *e* изменился, хотя непосредственно с этим массивом мы ничего не делали. Дело в том, что массивы в NumPy – это только ссылки на области в памяти. Операция присваивания $z = e$ не копирует значение, а лишь прикрепляет имя к объекту, содержащему нужные данные. Имя – это ссылка на объект, а не сам объект.

Для того чтобы избежать ситуации, возникшей при решении последнего примера, можно было создать копию массива *e*:

```
import numpy as np
e=np.array([[2,5,7,-2],[3,-6,67,12.6]])
z = e.copy() ; z[0,1]=100
e
```

В результате работы этого кода исходный массив не изменился: на выходе получили массив `array([[2. , 5. , 7. , -2.], [3. , -6. , 67. , 12.6]])`.

С помощью срезов можно заменять любые последовательности элементов:

```
import numpy as np
d=np.array([[2,5,7,-2],[3,-1.23,67,1]])
d[:,2]=[55, 56] ; d
```

В результате мы поменяли содержимое второго столбца:

```
array([[ 2. ,  5. , 55. , -2. ],
       [ 3. , -1.23, 56. ,  1. ]])
```

Можно всем элементам среза присвоить одно и то же число:

```
import numpy as np
e=np.array([2,5,7,-23,-6,67,12.6])
e[1:4]=15
e
```

Результат:

```
array([ 2. , 15. , 15. , 15. , -6. , 67. , 12.6])
```

Срезы массива в отличие от списков Python являются *представлениями* исходного массива. Это означает, что данные не копируются и любые изменения в представлении будут отражены в исходном массиве. Создадим массив и список с одинаковыми данными. В каждом из них сделаем срез, а затем изменим в нем одно из значений:

```
import numpy as np
e=np.array([2, 5, 7, -23, -6, 67, 12.6])
t=[2, 5, 7, -23, -6, 67, 12.6]
z=t[1:4]
z[1]=456
r=e[1:4] # [5, 7, -23]
r[1]=456
r,e,z,t
```

Результаты:

```
(array([ 5., 456., -23.]),
 array([ 2. ,  5. , 456. , -23. , -6. , 67. , 12.6]),
 [5, 456, -23],
 [2, 5, 7, -23, -6, 67, 12.6])
```

Элементы списка не изменились, а в массиве изменился элемент с индексом 1.

2.4. Сортировка элементов массива

Для сортировки элементов массива используют функции *sort()*, *argsort()*, *lexsort()*, *sort_complex()*, *partition()*, *argpartition()*.

Отсортированную копию массива можно получить с помощью функции *sort()*. Синтаксис функции:

```
numpy.sort(a, axis=-1, kind=None, order=None)
```

Параметры функции:

- *a* – структура данных *array_like*. Входные данные;
- *axis* – целое число или значение None. Ось, по которой производится сортировка. Если значение параметра равно None, перед сорти-

ровкой массив сглаживается. Сглаживание массива – процесс объединения группы вложенных массивов, присутствующих внутри данного массива. Значение параметра *axis* по умолчанию равно -1, что означает сортировку по последней оси;

– *kind* – алгоритм сортировки. По умолчанию используется «быстрая сортировка». Параметр может принимать одно из значений: 'quicksort', 'mergesort', 'heapsort', 'stable';

– *order* – строка или список строк. Если массив *a* является структурированным массивом с определенными полями, этот аргумент указывает порядок полей, по которым нужно провести сортировку.

Примеры:

```
import numpy as np
a = np.array([[1, 7, 5],[3, 1, 8], [5, 3, 6]])
a, np.sort(a, None), np.sort(a, 0), np.sort(a)
```

Результаты:

```
(array([[1, 7, 5],
        [3, 1, 8],
        [5, 3, 6]]),
array([1, 1, 3, 3, 5, 5, 6, 7, 8]),
array([[1, 1, 5],
        [3, 3, 6],
        [5, 7, 8]]),
array([[1, 5, 7],
        [1, 3, 8],
        [3, 5, 6]]))
```

Первым выведен исходный массив, вторым – отсортированный сглаженный массив, третьим и четвертым – массивы, отсортированные по нулевой и последней оси.

Отсортировать массив можно и по убыванию:

```
import numpy as np
a = np.array([[1, 7, 5],[3, 1, 8], [5, 3, 6]])
np.sort(a, None)[::-1]
```

Результат:

```
array([8, 7, 6, 5, 5, 3, 3, 1, 1])
```

[::-1] означает обратный порядок элементов. Так, результатом выполнения кода

```
import numpy as np
A=np.array([[2, 7, -3], [3, 1, 12], [-1, 7, -4], [1, 8, 6]])
A, A[::-1]
```

будут две матрицы: исходная и та, в которой строки будут идти в обратном порядке:

```
(array([[ 2,  7, -3],
        [ 3,  1, 12],
        [-1,  7, -4],
        [ 1,  8,  6]]),
array([[ 1,  8,  6],
        [-1,  7, -4],
        [ 3,  1, 12],
        [ 2,  7, -3]]))
```

Сортировать можно и структурированные массивы. Структурированные массивы — это массивы, тип данных которых представляет собой композицию более простых типов данных, организованных в виде последовательности именованных полей. Создадим, например, одномерный массив длины два, тип данных которого представляет собой структуру с тремя полями: строки длиной 10 или меньше с именем «Name», 32-битным целым числом с именем «Age» и 32-битным числом с плавающей запятой с именем «Height». Доступ к таким элементам возможен как по номеру индекса, так и по имени:

```
import numpy as np
x = np.array([('Paul', 19, 172.0), ('Glen', 33, 165.0)],
             dtype=[('Name', 'U10'), ('Age', 'i4'), ('Height', 'f4')])
x['Age']=[20, 34] ; x[1];x['Name']
x
```

Результаты:

```
(('Glen', 34, 165.),
array(['Paul', 'Glen'], dtype='<U10'),
array([('Paul', 20, 172.), ('Glen', 34, 165.)],
      dtype=[('Name', '<U10'), ('Age', '<i4'), ('Height', '<f4')]))
```

Таким образом можно отсортировать структурированный массив:

```
import numpy as np
```

```
x = np.array([('Paul', 19, 172.0), ('Glen', 33, 165.0)],
             dtype=[('Name', 'U10'), ('Age', 'i4'), ('Height', 'f4')])
np.sort(x, order='Height')
```

Результат:

```
array([('Glen', 33, 165.), ('Paul', 19, 172.)],
      dtype=[('Name', '<U10'), ('Age', '<i4'), ('Height', '<f4')])
```

Если необходимо, чтобы после сортировки массива результат сортировки был записан в исходный массив, используют конструкцию `ndarray.sort()`, имеющую синтаксис:

```
ndarray.sort(axis=-1, kind=None, order=None)
```

Параметры функции:

- *axis* – целое число или значение `None`. Ось, по которой производится сортировка. Значение параметра *axis* по умолчанию равно -1, что означает сортировку по последней оси;

- *kind* – алгоритм сортировки. По умолчанию используется 'quicksort' («быстрая сортировка»). Параметр может принимать одно из значений: 'quicksort', 'mergesort', 'heapsort', 'stable';

- *order* – строка или список строк. Если массив *a* является структурированным массивом с определенными полями, этот аргумент указывает порядок полей, по которым нужно провести сортировку.

Пример:

```
import numpy as np
a = np.array([[1, 7, 5], [3, 1, 8], [5, 3, 6]])
a.sort(1); a
```

Результат:

```
array([[1, 5, 7],
       [1, 3, 8],
       [3, 5, 6]])
```

Видим, что массив *a* изменился.

Если необходимо получить индексы, по которым можно отсортировать исходный массив, используют функцию `argsort()` (косвенная сортировка). Пусть, например, исходный массив *x* состоит из элементов -3, 4, 2, 1. Отсортированный по возрастанию он будет состоять из элементов, идущих в следующем порядке: -3, 1, 2, 4. Первый элемент в отсортированном массиве имел в массиве *x* индекс 0, второй – индекс 3, третий – индекс 2 и четвертый – индекс 1. Результатом работы

функции `argsort()` и будет массив индексов: 0, 3, 2, 1. Функция имеет следующий синтаксис:

```
numpy.argsort(a, axis=-1, kind=None, order=None)
```

Параметры функции:

- *a* – структура данных *array_like*. Входные данные;
- *axis* – целое число или значение `None`. Ось, по которой производится сортировка. Если значение параметра равно `None`, перед сортировкой массив сглаживается. Значение параметра *axis* по умолчанию равно -1, что означает сортировку по последней оси;
- *kind* – алгоритм сортировки. По умолчанию используется 'quicksort' («быстрая сортировка»). Параметр может принимать одно из значений: 'quicksort', 'mergesort', 'heapsort', 'stable';
- *order* – строка или список строк. Если массив *a* является структурированным массивом с определенными полями, этот аргумент указывает порядок полей, по которым нужно провести сортировку.

Пример:

```
import numpy as np; x = np.array([-3, 4, 2, 1]); np.argsort(x)
```

Результат:

```
array([0, 3, 2, 1], dtype=int64)
```

Если теперь выполнить оператор `x[np.argsort(x)]`, получим отсортированный массив: `array([-3, 1, 2, 4])`.

Применим косвенную сортировку с разными значениями параметра *axis* к двумерному массиву $A =$

$$A = \begin{pmatrix} 2 & 7 & -3 \\ 3 & 1 & 12 \\ -1 & 7 & -4 \\ 1 & 8 & 6 \end{pmatrix} :$$

```
import numpy as np
A=np.array([[2, 7, -3], [3, 1, 12], [-1, 7, -4], [1, 8, 6]])
ind=np.argsort(A)
ind1=np.argsort(A,0)
ind2=np.argsort(A,None)
ind, ind1, ind2
```

Результаты сортировки:

```
(array([[2, 0, 1],
        [1, 0, 2],
        [2, 0, 1],
```

```

        [0, 2, 1]], dtype=int64),
array([[2, 1, 2],
       [3, 0, 0],
       [0, 2, 3],
       [1, 3, 1]], dtype=int64),
array([ 8, 2, 6, 4, 9, 0, 3, 11, 1, 7, 10, 5], dtype=int64))

```

Можно отсортировать двумерный массив по любому столбцу. Отсортируем массив *A* по второму столбцу:

```

import numpy as np
A=np.array([[2, 7, -3], [3, 1, 12], [-1, 7, -4], [1, 8, 6]])
A[A[:, 2]. argsort ()]

```

Результат:

```

array([[ -1,  7, -4],
       [  2,  7, -3],
       [  1,  8,  6],
       [  3,  1, 12]])

```

Следующий код можно использовать для сортировки строк массива NumPy в порядке убывания на основе значений во втором столбце:

```

import numpy as np
A=np.array([[2, 7, -3], [3, 1, 12], [-1, 7, -4], [1, 8, 6]])
A[A[:, 2]. argsort ()[::-1]]

```

Результат сортировки:

```

array([[ 3,  1, 12],
       [ 1,  8,  6],
       [ 2,  7, -3],
       [-1,  7, -4]])

```

Описание остальных функций сортировки см. в документации по NumPy.

2.5. Операции над множествами из массивов

Функции библиотеки позволяют находить уникальные элементы массива, изменять порядок элементов вдоль заданной оси, производить

логические операции, находить пересечение и разность двух массивов, получать отсортированное объединение двух массивов и т. д.

Отсортированные общие элементы двух массивов можно получить с помощью метода *intersect1d()*. Массивы могут быть разной размерности:

```
import numpy as np
A=np.array([[2, 7, -3], [3, 1, 12], [-1, 2, -4], [1, 8, 6]])
b=np.array([3, 2, -6, 78, 6, 8, 2])
np.intersect1d(A, b)
```

Результат:

```
array([2, 3, 6, 8])
```

Можно получить индексы общих элементов (учитывается первое вхождение), добавив в метод параметр *return_indices=True*.

Отсортированное объединение двух массивов можно получить с помощью метода *union1d()*:

```
import numpy as np
A=np.array([[2, 7, -3], [3, 1, 12], [-1, 2, -4], [1, 8, 6]])
b=np.array([3, 2, -6, 78, 6, 8, 2])
np.union1d(A, b)
```

Результат:

```
array([-6, -4, -3, -1, 1, 2, 3, 6, 7, 8, 12, 78])
```

Получить булев массив, указывающий, содержится ли каждый элемент массива *A* в *b*, можно с помощью метода *in1d()*:

```
import numpy as np
A=np.array([[2, 7, -3], [3, 1, 12], [-1, 2, -4], [1, 8, 6]])
b=np.array([3, 2, -6, 78, 6, 8, 2])
np.in1d(A, b)
```

Результат:

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

Нумерация первого (двумерного) массива ведется слева направо по строкам.

Разность двух множеств, т. е. элементы массива *x*, которых нет в *y*, можно получить методом *setdiff1d()*:

```
import numpy as np
x=np.array([2, 7, -33, 1, 6, -1, 2, -6])
y=np.array([3, 2, -6, 78, 6, 8, 2])
np.setdiff1d(x, y)
```

Результат:

```
array([-33, -1, 1, 7])
```

Симметричную разность, т. е. элементы, которые есть либо в *x*, либо в *y*, но не в обоих массивах, можно получить методом *setxor1d()*:

```
import numpy as np
x=np.array([2, 7, -33, 1, 6, -1, 2, -6]); y=np.array([3, 2, -6, 78, 6, 8, 2])
np.setxor1d(x, y)
```

Результат:

```
array([-33, -1, 1, 3, 7, 8, 78])
```

С помощью функции *unique()* можно найти уникальные элементы массива и получить массив, состоящий из отсортированных уникальных элементов. В дополнение к уникальным элементам можно получить индексы входного массива, которые дают уникальные значения, индексы уникального массива, с помощью которых можно восстановить исходный массив и частоту каждого уникального элемента. Синтаксис функции:

```
numpy.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None, *, equal_nan=True)
```

Параметры функции:

- *ar* – структура данных *array_like*. Входной массив;
- *return_index* – булева переменная. Если значение параметра равно *True*, то функция возвращает индексы уникальных элементов в массиве *ar*;
- *return_inverse* – булева переменная. Если значение параметра равно *True*, то функция возвращает индексы, которые можно использовать для 4 массива *ar*;
- *return_counts* – булева переменная. Если значение параметра равно *True*, то функция возвращает частоту вхождения уникального элемента в массив *ar*;
- *axis* – ось. Необязательный параметр.

Пример:

```
import numpy as np; a = np.array([1, 2, 6, 6, 2, 1, 2, 3])
u, indices = np.unique(a, return_inverse=True)
u, indices, u[indices]
```

Выполнив код, получили три массива:

```
(array([1, 2, 3, 6]),
 array([0, 1, 3, 3, 1, 0, 1, 2], dtype=int64),
 array([1, 2, 6, 6, 2, 1, 2, 3]))
```

Первый массив содержит отсортированные по возрастанию уникальные элементы массива *a*. Во втором указаны номера, на которых находятся уникальные элементы (элементы массива *u*) в массиве *a*. Так, например, второй элемент массива *u* (элемент с индексом 1 и со значением 2) в исходном массиве встречается трижды (элементы массива *a* с индексами 1, 4 и 6), поэтому в массиве индексов *indices* на первом, четвертом и шестом месте указан индекс 1 – индекс элемента 2 в массиве *u*. Добавим к коду строки

```
counts = np.unique(a, return_counts=True)
counts
```

и выполним код. На экран будут выведены значения двух массивов: массивы, содержащие уникальные элементы массива *a* и частоту этих элементов в массиве *a*:

```
(array([1, 2, 3, 6]), array([2, 3, 1, 2], dtype=int64))
```

Обратный порядок элементов в массиве вдоль заданной оси можно получить, используя функцию *flip()*.

Примеры:

```
import numpy as np
a = np.array([1, 2, 6, 6, 2, 1, 2, 3]); b = np.array([[1,3,7], [-3, 8, 2]])
u = np.flip(a) ; s=np.flip(b); v=np.flip(b,1)
u, b, s, v
```

Результаты:

```
(array([3, 2, 1, 2, 6, 6, 2, 1]),
 array([[ 1,  3,  7],
        [-3,  8,  2]]),
 array([[ 2,  8, -3],
        [ 7,  3,  1]]),
 array([[ 7,  3,  1],
        [ 2,  8, -3]]))
```

«Развернуть» элементы по оси 1 в многомерном массиве можно с помощью функции *fliplr()*:

```
import numpy as np
b = np.array([[1,3,7], [-3, 8, 2]])
b, np.fliplr(b)
```

Результаты:

```
(array([[ 1,  3,  7],
        [-3,  8,  2]]),
 array([[ 7,  3,  1],
        [ 2,  8, -3]]))
```

Элементы строки в новом массиве идут в порядке, обратном их порядку в исходном массиве *b*.

2.6. Преобразования массивов. Функции *ravel()*, *reshape()*, *resize()*, *transpose()*, *swapaxes()* и *rot90()*

В библиотеке предусмотрены функции, позволяющие изменить форму массива и проводить операции, подобные транспонированию.

Двумерный массив можно преобразовать в одномерный:

```
import numpy as np
d=np.array([[2,5,7,12],[3,-1,6,1]])
v=d.ravel() ; v
```

Результат:

```
array([ 2,  5,  7, 12,  3, -1,  6,  1])
```

Для изменения размерности массива используется функция *reshape()*. Она принимает кортеж, значения которого задают новые размеры массива по осям, и возвращает новый массив. Так, например, одномерный массив размера *n* можно превратить в двумерный с желаемым числом строк (*k*) и столбцов (*m*). При этом необходимо, чтобы $n=k*m$:

```
import numpy as np ; d=np.array([ 2,  5,  7, 12,  3, -1,  6,  1])
v=d.reshape(4,2) ; v
```

Результат – матрица размерности 4 на 2:

```
array([[ 2,  5],
       [ 7, 12],
       [ 3, -1],
       [ 6,  1]])
```

Размер вдоль одной оси можно задать отрицательным числом (например, -1), и тогда он будет вычислен автоматически. Изменив строку `v=d.reshape(4,2)` на строку `v=d.reshape(-1,2)` и запустив получившийся код, получим тот же результат.

Метод *resize()* меняет размерность исходного массива:

```
import numpy as np
d=np.array([ 2,  5,  7, 12,  3, -1,  6,  1])
d.resize(4,2) ; d
```

Результат:

```
array([[ 2,  5],
       [ 7, 12],
       [ 3, -1],
       [ 6,  1]])
```

Порядок осей в двумерном массиве можно поменять, используя операцию транспонирования:

```
import numpy as np; A = np.array([[1, 2, 3], [4, 5, 6]]); A.T
```

Результат:

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Транспонировать матрицу можно, применив функцию *transpose()*:

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
np.transpose(A)
```

Поменять местами две оси *axis1* и *axis2* массива *a* можно с помощью метода *swapaxes()*, имеющего синтаксис:

```
swapaxes(a, axis1, axis2)
```

Результат предыдущего примера можно получить, выполнив код:

```
import numpy as np
```

```
A = np.array([[1, 2, 3], [4, 5, 6]])
np.swapaxes(A, 1, 0)
```

Матрицы можно поворачивать функцией *rot90()* против часовой стрелки (по умолчанию) на 90 градусов. При повороте вторым аргументом можно указать направление поворота:

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]]);
A, np.rot90(A), np.rot90(A, -1)
```

Результаты:

```
(array([[1, 2, 3],
        [4, 5, 6]]),
 array([[3, 6],
        [2, 5],
        [1, 4]]),
 array([[4, 1],
        [5, 2],
        [6, 3]]))
```

2.7. Удаление, добавление строк и столбцов

Массивы можно объединять, создавать массивы с добавленными, удаленными или повторенными заданное число раз элементами исходного массива, получать копии массива со значениями, добавленными к заданной оси, удалять нули из начала и/или конца одномерного массива или последовательности.

Для объединения массивов можно использовать функцию *concatenate()*. Объединяемые массивы должны иметь одинаковое количество осей. Объединять массивы можно с образованием новой оси либо вдоль уже существующей. Для объединения с образованием новой оси исходные массивы должны иметь одинаковые размеры вдоль всех осей.

Синтаксис функции *concatenate()*:

```
numpy.concatenate((a1, a2, ...), axis=0, out=None, dtype=None,
casting="same_kind")
```


Параметры функции:

- *a1, a2, ...* – последовательность структур данных *array_like*;
- *axis* – целое значение. Ось, вдоль которой будут объединяться массивы. Если значение параметра равно *None*, массивы выравниваются перед использованием. По умолчанию значение *axis* равно 0;
- *out* – структура данных *ndarray*. Необязательный параметр. Используется для размещения результата. Форма должна соответствовать той, которую операция конкатенации вернула бы, если бы аргумент *out* не был указан;
- *dtype* – тип результата. Если параметр задан, то целевой массив будет иметь указанный тип;
- *casting* – одно из значений: {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}. Определяет тип приведения данных. Значение по умолчанию – 'same_kind'.

Пример:

```
import numpy as np
a = np.array([[1,3,7], [-3, 8, 2]])
b = np.array([[5, 0, 6]]) ; c = np.array([[90,67]])
x1 = np.concatenate((a, b), axis=0),
x2 = np.concatenate((a, c.T), axis=1)
x3 = np.concatenate((a, b), axis=None)
a, x1, x2, x3
```

Результат:

```
(array([[ 1,  3,  7],
        [-3,  8,  2]]),
 (array([[ 1,  3,  7],
        [-3,  8,  2],
        [ 5,  0,  6]]),),
 array([[ 1,  3,  7, 90],
        [-3,  8,  2, 67]]),
 array([ 1,  3,  7, -3,  8,  2,  5,  0,  6]))
```

При первом применении функции *concatenate()* к первоначальному массиву *a* добавилась строка *b*, во втором случае к исходному массиву добавился столбец *c*. В последнем случае был образован одномерный массив.

Элементы массива можно повторить нужное число раз вдоль заданной оси с помощью функции *repeat()*:

```
import numpy as np
x = np.array([[1,2],[3,4]])
x, np.repeat(x, 2), np.repeat(x, [4, 2], axis=0), np.repeat(x, [4, 2], axis=1)
```

Результат:

```
(array([[1, 2],
        [3, 4]]),
 array([1, 1, 2, 2, 3, 3, 4, 4]),
 array([[1, 2],
        [1, 2],
        [1, 2],
        [1, 2],
        [3, 4],
        [3, 4]]),
 array([[1, 1, 1, 1, 2, 2],
        [3, 3, 3, 3, 4, 4]]))
```

Вставить значения в массив вдоль данной оси в заданное место можно с помощью функции *insert()*. Синтаксис функции:

```
numpy.insert(arr, obj, values, axis=None)[source]
```

Параметры функции:

- *arr* – структура данных *array_like*. Исходный массив;
- *obj* – целое, срез или последовательность целых чисел. Объект, определяющий индекс или индексы, перед которыми вставляются значения *values*. Значения для вставки в *arr*. Если тип значений отличается от типа *arr*, *values* преобразуются в тип массива *arr*;
- *axis* – целое число. Ось, вдоль которой вставляются значения. Если значение переменной *axis* равно *None*, то *arr* сначала сглаживается.

Пример:

```
import numpy as np; x = np.array([[1,2],[3,4]])
display(x,np.insert(x,1,[7,6],axis=1),np.insert(x,1,[7,6]))
np.insert(x,1,[7,6],axis=None),np.insert(x,1,[7,6],axis=0)
```

Результаты:

```
array([[1, 2],
        [3, 4]])
array([[1, 7, 2],
```

```

        [3, 6, 4]))
array([1, 7, 6, 2, 3, 4])
(array([1, 7, 6, 2, 3, 4]),
 array([[1, 2],
        [7, 6],
        [3, 4]]))

```

Исходный массив был преобразован следующим образом. Сначала был получен массив с добавленным столбцом, затем одномерный массив, в котором значения 7 и 6 были добавлены после элемента с индексом 0. В последнем случае был получен массив, в котором 7 и 6 стали элементами первой строки.

Таким образом можно поставить каждое из чисел на свое место, а не подряд:

```

import numpy as np; b = np.array([1,2,3,4])
b, np.insert(b, [2, 0], [5, 6])

```

Результат:

```

(array([1, 2, 3, 4]), array([6, 1, 2, 5, 3, 4]))

```

Число 5 в новом массиве имеет индекс 2, а число 6 – индекс 0.

Добавить значения в конец массива можно с помощью функции *append()*:

```

import numpy as np; b = np.array([1,2,3,4])
display(b,np.append(b, [5, 6]))
x = np.array([[1,2],[3,4]])
display(x,np.append(x, [[5, 6]],axis=0))

```

Результаты:

```

array([1, 2, 3, 4])
array([1, 2, 3, 4, 5, 6])
array([[1, 2],
       [3, 4]])
array([[1, 2],
       [3, 4],
       [5, 6]])

```

Удалить из массива строку или столбец можно с помощью функции *delete()*:

```
import numpy as np
a = np.array([[1,3,7,6], [-3, 8, 2,5],[5, 0, 6, 8],[6,12,-5,12]])
a, np.delete(a,1,0), np.delete(a,2,1)
```

Результат:

```
(array([[ 1,  3,  7,  6],
        [-3,  8,  2,  5],
        [ 5,  0,  6,  8],
        [ 6, 12, -5, 12]]),
array([[ 1,  3,  7,  6],
        [ 5,  0,  6,  8],
        [ 6, 12, -5, 12]]),
array([[ 1,  3,  6],
        [-3,  8,  5],
        [ 5,  0,  8],
        [ 6, 12, 12]]))
```

В первом случае из массива *a* была удалена строка с номером 1, а во втором – столбец с номером 2.

Удалить нули из начала и/или конца одномерного массива или последовательности можно, используя функцию *trim_zeros()*. Функция имеет следующий синтаксис:

```
trim_zeros(filt, trim='fb')
```

Параметры функции:

- *filt* – входной одномерный массив или последовательность;
- *trim* – строка, принимающая одно из значений: 'f' – в случае, когда необходима обрезка в начале массива; 'b' – если обрезка нужна в конце; 'fb' – если удаляются нули и в начале, и в конце массива. Значение по умолчанию – 'fb'.

Функция возвращает одномерный массив или последовательность.

Пример:

```
import numpy as np
a = np.array((0, 0, 0, 1, 2, 3, 0, 2, 1, 0))
np.trim_zeros(a), np.trim_zeros(a,'f')
```

В результате работы кода получили два массива:

```
(array([1, 2, 3, 0, 2, 1]), array([1, 2, 3, 0, 2, 1, 0]))
```

2.8. Разбиение массива

Разделить массив на несколько подмассивов одинакового или почти равного размера можно с помощью функции *array_split()*:

```
import numpy as np
a=np.array([ 1, 3, 7, -3, 8, 2, 5, 0, 6])
np.array_split(a, 3),np.array_split(a, 4)
```

Результат:

```
([array([1, 3, 7]), array([-3, 8, 2]), array([5, 0, 6])],
 [array([1, 3, 7]), array([-3, 8]), array([2, 5]), array([0, 6])])
```

Разделить массив на несколько подмассивов по горизонтали (по столбцам) или по вертикали (по строкам) можно с помощью функций *hsplit()* и *vsplit()* соответственно:

```
import numpy as np
a = np.array([[1,3,7,6], [-3, 8, 2,5],[5, 0, 6, 8],[6,12,-5,12]])
a,np.hsplit(a,2),np.vsplit(a,4)
```

Результат:

```
(array([[ 1, 3, 7, 6],
        [-3, 8, 2, 5],
        [ 5, 0, 6, 8],
        [ 6, 12, -5, 12]]),
 [array([[ 1, 3],
        [-3, 8],
        [ 5, 0],
        [ 6, 12]]),
 array([[ 7, 6],
        [ 2, 5],
        [ 6, 8],
        [-5, 12]])],
 [array([[1, 3, 7, 6]]),
 array([[ -3, 8, 2, 5]]),
 array([[ 5, 0, 6, 8]]),
 array([[ 6, 12, -5, 12]])])
```

2.9. Математические операции над элементами массива

Для работы с массивами применимы все стандартные арифметические операции, а также тригонометрические, экспоненциальная и другие функции. Массивы позволяют выполнять операции без использования циклов. Выполнение математических операций над массивами происходит поэлементно. При этом создается новый массив, который заполняется результатами действия оператора.

Если массивы A и B одинакового размера, то с ними можно производить те же арифметические операции, что и с обычными числами: их можно складывать, умножать, вычитать, делить и возводить в степень. Эти операции выполняются *поэлементно*, результирующий массив будет совпадать по геометрии с исходными массивами, а каждый его элемент будет результатом выполнения соответствующей операции над парой элементов из исходных массивов.

Пусть имеется два массива A и B : $A = \begin{pmatrix} 4 & 2 & 3 \\ 0 & 2 & 1 \\ 6 & 9 & -3 \end{pmatrix}$ и $B = \begin{pmatrix} 2 & 1 & 4 \\ 3 & 2 & 2 \\ 3 & \frac{1}{3} & 6 \end{pmatrix}$.

Вычислим сумму, разность, произведение, частное от деления этих массивов и возведем каждый элемент массива A в степень, заданную массивом B :

```
import numpy as np
A = np.array([[4, 2, 3], [0, 2, 1], [6, 9, -3]])
B = np.array([[2, 1, 4], [3, 2, 2], [3, 1/3, 6] ])
C = A + B ; D = A - B
E = A * B
F = A / B
G = A ** B
display(A,B)
C, D, E, F, G
```

Результаты:

```
array([[ 4,  2,  3],
       [ 0,  2,  1],
       [ 6,  9, -3]])
```

```

array([[2.    , 1.    , 4.    ],
       [3.    , 2.    , 2.    ],
       [3.    , 0.33333333, 6.    ]])

(array([[6.    , 3.    , 7.    ],
       [3.    , 4.    , 3.    ],
       [9.    , 9.33333333, 3.    ]]),
array([[ 2.    , 1.    , -1.    ],
       [-3.    , 0.    , -1.    ],
       [ 3.    , 8.66666667, -9.    ]]),
array([[ 8.,  2., 12.],
       [ 0.,  4.,  2.],
       [18.,  3., -18.]]),
array([[ 2. ,  2. ,  0.75],
       [ 0. ,  1. ,  0.5 ],
       [ 2. , 27. , -0.5 ]]),
array([[16.    ,  2.    , 81.    ],
       [ 0.    ,  4.    ,  1.    ],
       [216.    ,  2.08008382, 729.    ]]))

```

Для выполнения каждой операции можно использовать либо соответствующий оператор, либо встроенные функции *add()*, *subtract()*, *multiply()*, *divide()*, *power()*, *mod()*. Последняя функция определяет остаток от деления. Приведенные функции – *бинарные*. Они принимают на вход два аргумента.

Пример нахождения остатков от деления:

```

f=np.array([[11,6],[8,12]]); b=np.array([[3,5], [4, 9]])
S=np.mod(f, b)
S

```

Результат:

```

array([[2, 1],
       [0, 3]])

```

Можно выполнить любую арифметическую операцию над массивом и числом. В этом случае операция также выполнится над каждым из элементов массива:

```

import numpy as np
a=np.array([1,3,7])
b=[1,3,7]
print(2*a); print(2*b)

```

Результаты:

```
[ 2  6 14]
[1, 3, 7, 1, 3, 7]
```

В первом случае, когда число умножается на *массив* (2 на *a*), мы получили вектор той же размерности, каждый элемент которого в два раза больше соответствующего элемента вектора *a*. Во втором случае, когда на 2 мы умножили *список* *b*, получили список из повторяющихся элементов исходного списка, длина которого стала в два раза больше длины исходного: 6 вместо 3. В этом состоит еще одно различие между списками и массивами.

Если атрибуты *ndarray.shape* двух массивов не совпадают, действия над массивами производятся в соответствии с концепцией транслирования (*broadcasting*). Операция транслирования – это расширение одного или обоих массивов операндов до массивов с равной размерностью.

Пример:

```
a = np.array([16, 20])
b = np.array([ [10, 12], [30, 4] ])
a + b
```

Результат:

```
array([[26, 32],
       [46, 24]])
```

От каждого элемента массива можно вычислить какую-либо функцию. Вычислим, например, синус от каждого элемента матрицы $F = \begin{pmatrix} 1 & 6 \\ 8 & 0 \end{pmatrix}$:

```
import numpy as np; f=np.array([[1,6],[8,0]]); np.sin(f)
```

Результат:

```
array([[ 0.84147098, -0.2794155 ],
       [ 0.98935825,  0.        ]])
```

Многие унарные операции, такие как, например, вычисление суммы всех элементов массива, представлены также и в виде методов класса *ndarray*:

```
a = np.array([16, 20, 34])
a.sum(), np.sum(a), a.min(), np.min(a)
```


Результаты:

(70, 70, 16, 16)

Наиболее часто используемые функции и константы библиотеки NumPy приведены в табл. 1.1–1.4.

2.10. Логические операции

Список доступных в NumPy операторов для сравнения элементов массивов тот же, что и в Python: < (меньше), <= (меньше или равно), > (больше, чем), >= (больше или равно), == (равно), != (не равно). NumPy также предоставляет встроенные функции для выполнения всех операций сравнения: *less()*, *less_equal()*, *greater()*, *greater_equal()*, *equal()*, *not_equal()*.

Примеры:

```
import numpy as np
f=np.array([[1,6],[8,12]]); b=np.array([[3,6], [4, 9]])
w= np.less(f, b); y= f == b; z= f != b
w, y, z
```

Результаты представляют собой массивы, где каждый элемент равен либо True, либо False в зависимости от сравнения элементов массивов. *f*[0, 0] сравнивается с *b*[0, 0], *f*[0, 1] сравнивается с *b*[0, 1] и т. д.:

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

Над массивами определены логические операции *logical_and*, *logical_or* и *logical_not*, выполняющие логические операции И, ИЛИ и НЕ поэлементно. *logical_and* и *logical_or* принимают 2 операнда, *logical_not* – один. Можно использовать операторы &, | и ~ для выполнения И, ИЛИ и НЕ соответственно с любым количеством операндов:

```
import numpy as np
```

```

A = np.array([[15, 32, -3], [12, -4, 8]])
f=A[np.logical_or(A < 10, A > 30)]
h=A[np.logical_and(A > 0, A < 40)]
t=A[(A>-4) & (A<15) | (A>30)]
f,h,t

```

В результате работы кода получили три массива. Первый (f) содержит элементы массива *A*, которые либо меньше 10, либо больше 30, второй – положительные элементы массива *A*, меньшие 40, третий – элементы массива *A*, которые либо больше 30, либо принадлежат открытому интервалу (-4, 15):

```

(array([32, -3, -4, 8]), array([15, 32, 12, 8]), array([32, -3, 12, 8]))

```

Покажем, как можно получить, например, массив *h* в рассмотренном ранее примере с помощью цикла. Первый способ:

```

import numpy as np
Z=[]
A = np.array([[15, 32, -3], [12, -4, 8]])
for x in A:
    for i in x:
        if i>0 & i<40:
            Z.append(i)
r=np.array(Z)
r,Z

```

Выполнив код, получим кортеж, содержащий ответ в виде массива и списка:

```

(array([15, 32, 12, 8]), [15, 32, 12, 8])

```

Второй вариант программы более традиционен:

```

import numpy as np
Z=[]
A = np.array([[15, 32, -3], [12, -4, 8]])
for i in range(2):
    for j in range(3):
        if A[i][j]>0 & A[i][j]<40:
            Z.append(A[i,j])
r=np.array(Z)
r,Z

```

Результат будет таким же.

2.11. Статистические операции

В табл. 2.1 приведены основные статистические функции, применяемые для работы с массивами.

Таблица 2.1

Статистические функции

Описание функции	Имя функции	Примеры
Вычисление среднего значения	<code>mean(x)</code>	<code>np.mean(x)</code>
Вычисление средневзвешенного значения	<code>average()</code>	<code>np.average(x)</code>
Вычисление медианы x	<code>median(x)</code>	<code>np.median(x)</code>
Дисперсия	<code>var(x)</code>	<code>np.var(x)</code>
Среднеквадратическое отклонение	<code>std(x)</code>	<code>np.std(x)</code>
Диапазон значений (максимум-минимум) по оси	<code>ptp()</code>	<code>np.ptp(y,1)</code>
Вычисление q-го перцентиля по указанной оси	<code>percentile()</code>	<code>np.percentile(a, 30, axis=0)</code>
Вычисление q-го квантиля вдоль указанной оси	<code>quantile()</code>	<code>np.quantile(a, .3, axis=0)</code>
Коэффициент корреляции Пирсона	<code>corrcoef()</code>	<code>np.corrcoef(x, y)</code>
Взаимная корреляция двух одномерных последовательностей	<code>correlate()</code>	<code>np.correlate(x, y)</code>
Ковариационная матрица	<code>cov()</code>	<code>np.cov(x, y)</code>
Получение данных для построения гистограммы	<code>histogram()</code>	<code>np.histogram(x, bins=4)</code>

Агрегирующие функции используются для получения обобщающих значений. Это функции, выполняющие вычисления над набором значений и возвращающие одно значение. В библиотеке Numpy есть множество агрегатных (или статистических) функций для работы с одномерными или многомерными массивами: `sum()`, `prod()`, `cumsum()`, `cumprod()`, `min()`, `max()`, `argmin()`, `argmax()`, `mean(x)`, `average()`, `median(x)`, `percentile()`, `var()`, `std()` и `corrcoef()`.

Использовать агрегатные функции можно либо вызывая метод экземпляра массива, либо используя функцию NumPy верхнего уровня.

Сумму значений массива вдоль заданной оси можно вычислить с помощью функции `sum()`. Например, `np.sum(a, 0)` (или `a.sum(axis=0)`, или `a.sum(0)`) возвращает сумму каждого столбца в массиве *a*:

```
import numpy as np
a=np.array([[2,4],[4,8],[5,12]])
a, a.sum(0)
```

Результат:

```
(array([[ 2,  4],
        [ 4,  8],
        [ 5, 12]]),
 array([11, 24]))
```

Для нахождения суммы элементов строки нужно воспользоваться оператором `a.sum(1)`, для вычисления суммы всех элементов массива – оператором `a.sum()`.

Произведение значений массива вдоль заданной оси можно вычислить с помощью функции *prod()*. `a.prod(axis=0)` возвращает произведение элементов каждого столбца в массиве *a*:

```
import numpy as np
a=np.array([[2,4],[4,8],[5,12]])
a.prod(0)
```

Результат: array([40, 384])

Для нахождения произведений элементов строки нужно воспользоваться оператором `a.prod(1)`, для вычисления суммы всех элементов массива – оператором `a.prod()`.

Для вычислений накопленных сумм и произведений используют функции *cumsum()* и *cumprod()* соответственно.

Примеры:

```
b=np.array([[2,4,1],[4,3,-3]])
display(b,b.cumsum(0), b.cumsum(1), b.cumsum())
b.cumprod(0), b.cumprod(1), b.cumprod())
```

Выполнив код, получили:

```
array([[ 2,  4,  1],
        [ 4,  3, -3]])
array([[ 2,  4,  1],
        [ 6,  7, -2]])
array([[2, 6, 7],
        [4, 7, 4]])
array([ 2,  6,  7, 11, 14, 11])
```

```
(array([[ 2,  4,  1],
        [ 8, 12, -3]]),
array([[ 2,  8,  8],
        [ 4, 12, -36]]),
array([ 2,  8,  8, 32, 96, -288]))
```

С помощью функций *min()* и *max()* вычисляют минимальные элементы по заданной оси:

```
b=np.array([[2,4,1],[5,3,-3]])
display(b,b.min(0), b.min(1), b.min())
b.max(0), b.max(1), b.max()
```

Результаты работы кода:

```
array([[ 2,  4,  1],
        [ 5,  3, -3]])
array([ 2,  3, -3])
array([ 1, -3])
-3
(array([5, 4, 1]), array([4, 5]), 5)
```

Индексы минимальных и максимальных элементов массива находят с помощью функций *argmin()* и *argmax()*.

Примеры вычисления индексов максимальных элементов:

```
import numpy as np
b=np.array([[2,4,1],[5,3,-3]])
b,b.argmax(0), b.argmax(1), b.argmax()
```

Результаты:

```
(array([[ 2,  4,  1],
        [ 5,  3, -3]]),
array([1, 0, 0], dtype=int64),
array([1, 0], dtype=int64),
3)
```

Сначала выведен исходный двухмерный массив. Второй массив – массив индексов максимальных элементов по столбцам: в нулевом столбце максимальный элемент находится в первой строке, в остальных – в нулевой. Третий массив содержит индексы максимальных элементов строк матрицы: максимальный элемент нулевой строки стоит в первом столбце, а максимальный элемент первой строки – в ну-

левом столбце. Вместо номера строки и столбца максимального элемента получен скаляр – число 3. Дело в том, что при использовании функции *argmax()* для поиска максимума в двумерном массиве элементы массива перенумеровываются: элементы нулевой строки в данном примере имеют номера 0, 1 и 2 (нумерация ведется слева направо), элементы первой строки – номера 3, 4 и 5 и т. д.

Сгенерируем двумерный массив *A* со случайным количеством строк и столбцов (например, от 3 до 5), состоящий из случайных целых чисел из интервала от 1 до 101, и выведем на экран полученный массив, его максимальный элемент и позицию этого элемента в массиве:

```
import numpy as np
n=np.random.randint(3,5)
m=np.random.randint(3,5)
A=np.random.randint(1,110,size=(n,m))
Imax= A.argmax()
num=A.shape[1] # или num=m
z=np.divmod(Imax,num)
print ("A=")
print(A)
print("Максимальный элемент массива A равен %i" % A.max())
print("Он находится во %i-й строке и %i-м столбце" % (z[0],z[1]))
```

Результаты одного из запусков кода:

```
A=
[[ 36  37  30  35]
 [ 65  80  68  52]
 [ 34  38 102  76]]
```

Максимальный элемент массива *A* равен 102. Он находится во 2-й строке и 2-м столбце.

Создать из двух одномерных массивов *a* и *b* массив *c* с элементами $c_i=\min(a_i, b_i)$ или $c_i=\max(a_i, b_i)$ можно с помощью функций *fmin()* или *fmax()* соответственно. Создадим, например, массив $c_i=\min(a_i, b_i)$:

```
import numpy as np
a=np.array([58, 16, 72, 79]); b=np.array([36, 40, 28, 83])
np.fmin(a,b)
```

Результат работы кода:

```
array([36, 16, 28, 79])
```

Для вычисления средних значений в NumPy имеются функции *mean()* и *average()*. Первая из них имеет следующий синтаксис:

```
numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>, *, where=<no value>)
```

С ее помощью можно вычислить среднее арифметическое по каждой оси либо всего массива.

Параметры функции:

- *a* – структура данных *array_like*. Массив, содержащий числа, среднее значение которых требуется вычислить;
- *axis* – ось, вдоль которой вычисляется среднее значение. Параметр может принимать значение *None*, быть целым числом или кортежем целых чисел. Если это кортеж целых чисел, среднее значение вычисляется по нескольким осям вместо одной оси или для всех осей;
- *dtype* – тип данных. Для целочисленных входных данных значение по умолчанию равно *float64*; для входных данных с плавающей запятой – тот же тип, что и входной массив;
- *out* – альтернативный выходной массив, в который следует поместить результат. По умолчанию значение параметра равно *None*; если массив указан, то он должен иметь ту же форму, что и ожидаемый результат;
- *keepdims* – булева переменная. С этой опцией результат будет корректно транслироваться относительно входного массива;
- *where* – структура данных *array_like* из значений *False* или *True*. Указание на элементы, которые необходимо включить в среднее значение.

При значении параметра *out = None* функция *mean()* возвращает новый массив, содержащий средние значения. В противном случае возвращается ссылка на указанный выходной массив.

Примеры:

```
import numpy as np; a=np.array([58, 16, 72, 79])
np.mean(a) # или a.mean(). Вычисляет (58+16+72+79)/4 = 56.25
b=np.array(
[[58, 16, 72, 79],
 [77, 16, 2, 79],
 [36, 40, 28, 83],
 [68, 33, 79, 39]])
np.mean(b,0),np.mean(b,1),np.mean(b),\
np.mean(b,where=[[True], [False], [False],[True]])
```

Результаты работы кода:

```
(array([59.75, 26.25, 45.25, 70.  ]),  
 array([56.25, 43.5 , 46.75, 54.75]),  
 50.3125,  
 55.5)
```

В последнем случае было вычислено среднее арифметическое строк с номерами 0 и 3.

С помощью функции *average()* вычисляют средневзвешенное значение вдоль указанной оси. Синтаксис функции:

```
numpy.average(a, axis=None, weights=None, returned=False, *,  
keepdims=<no value>)
```

Параметры функции:

– *weights* – структура данных *array_like*. Массив весов. Каждое значение в *a* вносит свой вклад в среднее значение в соответствии с соответствующим весом. Массив весов может быть либо одномерным (в этом случае его длина должна быть равна размеру *a* по заданной оси), либо иметь ту же форму, что и исходный массив *a*. Если значение параметра *weights=None*, предполагается, что все данные в массиве *a* имеют вес, равный единице;

– *returned* – переменная, принимающая значения True или False. Значение по умолчанию – False. Если значение параметра равно True, возвращается кортеж (*average*, *sum_of_weights*), в противном случае возвращается только среднее значение. Если значение параметра *weights=None*, сумма весов *sum_of_weights* эквивалентна количеству элементов, по которым берется среднее значение.

Примеры:

```
import numpy as np  
c=np.array([[0, 1],  
            [2, 3],  
            [4, 5]])  
np.average(c, axis=1, weights=[1./4, 3./4],returned=True),\  
np.average(c, weights=[[1./4, 3./4],[1, 2], [0.5, 0.7]],returned=True)
```

Результаты:

```
((array([0.75, 2.75, 4.75]), array([1., 1., 1.])), (2.7403846153846154,  
5.2))
```


В первом случае был получен кортеж $((0 \cdot 1/4 + 1 \cdot 3/4)/(1/4 + 3/4), (2 \cdot 1/4 + 3 \cdot 3/4)/(1/4 + 3/4), (4 \cdot 1/4 + 5 \cdot 3/4)/(1/4 + 3/4))$, во втором – значение $(0 \cdot 0.25 + 1 \cdot 0.75 + 2 \cdot 1 + 3 \cdot 2 + 4 \cdot 0.5 + 5 \cdot 0.7)/(1/4 + 3/4 + 1 + 2 + 0.5 + 0.7) = 2.74038$.

Медиану в NumPy вычисляют с помощью функции *median()*. Медианой называют варианту, которая делит вариационный ряд (упорядоченную последовательность) на две равные по числу элементов части.

Синтаксис функции *median()*:

```
numpy.median(a, axis=None, out=None, overwrite_input=False,
keepdims=False)
```

Если значение параметра *overwrite_input* = True, то для вычислений разрешается использовать память входного массива *a*. Это позволит сэкономить память, когда нет необходимости сохранять содержимое входного массива. Входной массив будет изменен. Убедимся в этом на примерах. Сначала параметр *overwrite_input* не используется:

```
import numpy as np
a = np.array([-3, 3, 4, 8], [12, 2, 1, 7])
a, np.median(a), a, np.median(a, 1)
```

Результаты:

```
(array([-3, 3, 4, 8],
      [12, 2, 1, 7]),
 3.5,
array([-3, 3, 4, 8],
      [12, 2, 1, 7]),
array([3.5, 4.5]))
```

Массив *a* не изменился.

Выполним код:

```
import numpy as np
a = np.array([-3, 3, 4, 8], [12, 2, 1, 7])
np.median(a, axis=None, overwrite_input=True), a
```

В результате выполнения кода массив *a* изменился:

```
(3.5,
array([-3, 1, 2, 3],
      [ 4, 7, 12, 8]))
```

Оценить дисперсию массива *x* с элементами $x_i, i = \overline{1, n}$ можно с помощью функции *var()*. Синтаксис функции:

`numpy.var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=<no value>, *, where=<no value>)`

При значении параметра `ddof = 0` получают смещенную оценку дисперсии, вычисляемую по формуле $\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$, а при значении `ddof = 1` – несмещенную $\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$. Здесь \bar{x} – среднее значение.

Примеры:

```
import numpy as np
x=np.array([90, 92, 93, 95, 96, 98])
a = np.array([[1, 2], [3, 4]])
np.var(x), np.var(x,ddof=1), np.var(a)
```

Результаты:

(7.0, 8.4, 1.25)

Оценку среднеквадратического отклонения (корень из дисперсии) вычисляют с помощью функции `std()`. Синтаксис функции:

`numpy.std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=<no value>, *, where=<no value>)`

Примеры:

```
import numpy as np
a = np.array([[4, -8, 1, 10, 5], [2, -9, 1, 1, 12], [3, 5, 9, 2, 4]])
np.std(a), np.std(a, where=[[True], [False], [True]])
```

Результаты:

(5.503937984146745, 4.674398357008098)

Размах (разность между максимальным и минимальным значениями) по оси вычисляют с использованием функции `ptp()`. Ее синтаксис:

`numpy.ptp(a, axis=None, out=None, keepdims=<no value>)`

Примеры использования функции ptp():

```
import numpy as np
y = np.array([ [1, 27, 7],
               [ 0, -27, 5],
               [ 4, 12, 7],
               [-2,  3, 4]])
np.ptp(y), np.ptp(y,1)
```

Результаты:

```
(54, array([26, 32, 8, 6]))
```

Для вычисления процентилей применяют функцию *percentile()*. Процентиль (англ. percentile) – это значение, которое заданная случайная величина не превышает с фиксированной вероятностью, заданной в процентах. Термин впервые был использован в 1885 году статистиком Фрэнсисом Гальтоном. Синтаксис функции:

```
numpy.percentile(a, q, axis=None, out=None, overwrite_input=False,  
method='linear', keepdims=False)
```

С помощью этой функции вычисляют q -й процентиль.

Параметр *method* указывает метод, используемый для оценки процентиля. Подробнее о функции см. в документации NumPy¹.

Примеры:

```
import numpy as np  
a = np.array([[7, 17, 14, 24], [3, 2, 1, 7]])  
m = np.percentile(a, 30, axis=0)  
out = np.zeros_like(m)  
t=np.percentile(a, 30, axis=0, out=out)  
t, out, np.percentile(a, 50, axis=1, keepdims=True), np.median(a, 1)  
# Медиана и 50 % перцентиль – это одно и то же
```

Результаты:

```
(array([ 4.2,  6.5,  4.9, 12.1]),  
array([ 4.2,  6.5,  4.9, 12.1]),  
array([[15.5],  
       [ 2.5]]),  
array([15.5,  2.5]))
```

В примерах использовались параметры *out* и *keepdims*.

Квантиль можно вычислить с помощью одноименной функции *quantile()*. Квантиль в математической статистике – это значение, которое заданная случайная величина не превышает с фиксированной вероятностью. Если вероятность задана в процентах, то квантиль называется процентилем или перцентилем. Синтаксис функции *quantile()*:

¹ Документация NumPy. URL: <https://numpy.org/doc/>

`numpy.quantile(a, q, axis=None, out=None, overwrite_input=False, method='linear', keepdims=False)`

Параметры функции аналогичны параметрам функции *percentile()*.

Примеры:

```
import numpy as np
a = np.array([[7, 17, 14, 24], [3, 2, 1, 7]])
m = np.quantile(a, .3, axis=0)
out = np.zeros_like(m)
t=np.quantile(a, .3, axis=0, out=out)
t, out, np.quantile(a, .5, axis=1, keepdims=True), np.median(a, 1)
```

Результаты будут такими же, как в предыдущем примере.

Предположим, что в результате n взаимно независимых экспериментов с двумерной случайной величиной (X, Y) получена ее реализация $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. Выборочную ковариацию между величинами X и Y вычисляют по следующей формуле:

$$\text{cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}), \quad \text{где} \quad \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i.$$
$$\text{cov}(X, X) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2, \quad \text{cov}(Y, Y) = \frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2. \quad \text{cov}(X, Y) = \text{cov}(Y, X).$$

Для вычисления ковариационной матрицы в NumPy используют функцию *cov()*. Элемент ковариационной матрицы C_{ij} является ковариацией величин X_i и X_j . Элемент C_{ii} это дисперсия X_i . Синтаксис функции:

`numpy.cov(m, y=None, rowvar=True, bias=False, ddof=None, fweights=None, aweights=None, *, dtype=None)`

Параметры функции:

- m – структура данных *array_like*. Одномерный или двумерный массив, содержащий несколько переменных и наблюдений. Каждая строка данных m представляет переменную, а каждый столбец – одно наблюдение всех этих переменных;

- y – дополнительный набор переменных и наблюдений с той же формой, что и m ;

- *rowvar* – булева переменная. Если *rowvar* = True (по умолчанию), то каждая строка представляет собой переменную с наблюдениями в столбцах. В противном случае каждый столбец представляет собой переменную, а строки содержат наблюдения;

– *bias* – булева переменная. Для вычисления ковариации по формуле $\text{cov}(X, Y) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$ (смещенная оценка) нужно задать

bias = True;

– *ddof* – целое число или значение None. С помощью этого параметра можно переопределить значение параметра *bias*. При *ddof* = 1 будет возвращена несмещенная оценка, при значении *ddof* = 0 – смещенная оценка;

– *fweights* – одномерный массив из целых чисел. Частота каждого вектора наблюдений;

– *aweights* – структура данных *array_like*. Одномерный массив относительных весов. Если *ddof* = 0, массив весов можно использовать для назначения вероятностей векторам наблюдений.

Вычислим ковариационную матрицу и убедимся, что по диагонали стоят оценки дисперсии:

```
import numpy as np
x = np.array([-2.1, -1, 4.3, 8])
y = np.array([3, 1.1, 0.12, 2])
np.cov(x, y), np.var(x, ddof=1), np.var(y, ddof=1)
```

Результаты:

```
(array([[22.24666667, -1.73    ],
        [-1.73    , 1.51743333]]),
22.246666666666667,
1.5174333333333332)
```

Вычислим матрицу со смещенными оценками ковариации по тем же данным:

```
np.cov(x, y, ddof=0), np.cov(x, y, bias=True)
```

Результаты применения функции *cov()* одинаковы:

```
(array([[16.685  , -1.2975 ],
        [-1.2975 , 1.138075]]),
array([[16.685  , -1.2975 ],
        [-1.2975 , 1.138075]]))
```

Зададим веса элементам каждого столбца:

```
import numpy as np
x = np.array([ [-2.1, -1, 4.3, 8],
               [3, 1.1, 0.12, 2]])
np.cov(x, fweights=[2, 3, 1, 4]), np.cov(x, aweights=[.2, .3, .1, .4])
```

Результаты:

```
(array([[22.40322222, -0.23068889],  
       [-0.23068889, 0.81097333]]),  
array([[28.80414286, -0.2966   ],  
       [-0.2966   , 1.04268   ]]))
```

Выборочный коэффициент корреляции вычисляется по формуле

$$r = \rho_{xy} = \frac{\sum_{i=0}^{n-1} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=0}^{n-1} (x_i - \bar{x})^2 \sum_{i=0}^{n-1} (y_i - \bar{y})^2}}.$$

Он характеризует меру линейной зависимости двух случайных величин X и Y . Соотношение между матрицей коэффициентов корреляции R и матрицей ковариаций C : $R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii}C_{jj}}}$.

Значения R находятся в диапазоне от -1 до 1 включительно.

Для вычисления коэффициентов корреляции Пирсона между векторами X и Y используют функцию `corrcoef()`. Ее синтаксис:

```
numpy.corrcoef(x, y=None, rowvar=True, dtype=None)
```

Параметры функции:

- m – структура данных *array_like*. Одномерный или двумерный массив, содержащий несколько переменных и наблюдений. Каждая строка данных m представляет переменную, а каждый столбец – одно наблюдение всех этих переменных;

- y – дополнительный набор переменных и наблюдений с той же формой, что и m ;

- $rowvar$ – булева переменная. Если $rowvar = \text{True}$ (по умолчанию), то каждая строка представляет собой переменную с наблюдениями в столбцах. В противном случае каждый столбец представляет собой переменную, а строки содержат наблюдения.

Пример:

```
import numpy as np  
x = np.array([ [-2.1, -1, 4.3, 8],  
              [3, 1.1, 0.12, 2]])  
np.corrcoef(x)
```

Результат:

```
array([[ 1.          , -0.29775483],  
       [-0.29775483, 1.          ]])
```

Получить данные для построения гистограммы можно с помощью функции *histogram()*. Синтаксис функции:

```
numpy.histogram(a, bins=10, range=None, density=None, weights=None)
```

Параметры функции:

- *a* – структура данных *array_like*. Входные данные;
- *bins* – либо целое число, задающее количество интервалов равной длины, на которые разбивается интервал значений, либо последовательность возрастающих чисел – концов интервалов, либо строка. Все интервалы, кроме последнего (самого правого), полуоткрыты. В качестве значения переменной можно задать метод, использующийся для вычисления ширины интервала: 'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges' или 'sqrt'. Подробнее см. в документации по NumPy;
- *range* – диапазон. Задается, если нужно изменить определяемые автоматически значения начала (a_{\min}) и конца (a_{\max}) интервала разбиения;
- *weights* – структура данных *array_like*. Массив весов той же формы, что *a*. Если для *density* установлено значение True, веса нормализуются;
- *density* – переменная, принимающая значения True, False или None. Если значение равно False, результат будет содержать частоту попадания в каждый интервал. Если значение параметра равно True, результатом будет массив, состоящий из относительных частот, деленных на ширину соответствующего интервала.

Примеры:

```
import numpy as np
x=np.array([2, 2.3, 3, 4, 4.2, 4.3, 4.5, 6.2, 7, 7.1, 8, 8])
np.histogram(x, bins=4)
```

Результаты:

```
(array([3, 4, 1, 4], dtype=int64), array([2. , 3.5, 5. , 6.5, 8. ]))
```

Комментарий к коду. Интервал значений *x* (от минимального, 2, до максимального, 8) разбили на 4 равные части. Шаг равен $(8-2)/4=1.5$. Три значения из *x* попало в интервал [2, 3.5), четыре значения попало в интервал [3.5, 5), одно – в интервал [5, 6.5) и четыре – в интервал [6.5, 8].

Разобьем теперь интервал значений на три неравные части:

```
import numpy as np
x=np.array([2,2.3,3,4,4.2,4.3,4.5,6.2,7,7.1,8,8])
np.histogram(x, bins=[2, 3, 6, 8])
```

Результат:

```
(array([2, 5, 5], dtype=int64), array([2, 3, 6, 8]))
```

Два значения попало в интервал [2, 3) и по пять значений – в интервалы [3, 6) и [6, 8].

Проведем нормализацию:

```
import numpy as np
x=np.array([2,2.3,3,4,4.2,4.3,4.5,6.2,7,7.1,8,8])
np.histogram(x, bins=[2, 3, 6, 8], density=True)
```

Результаты:

```
(array([0.16666667, 0.13888889, 0.20833333]), array([2, 3, 6, 8]))
```

Каждая частота была поделена на количество элементов в массиве и на ширину соответствующего интервала: [2/12/1, 5/12/3, 5/12/2].

Предположим, что в массиве *x* значение 2 встречается 4 раза, значение 3 – 2 раза, значение 5 – 1 раз и значение 7 – 3 раза. Разобьем интервал значений на 2 подынтервала [2, 4) и [4, 7] и вычислим частоту попадания значений *x* в каждый из них:

```
import numpy as np
x=np.array([2, 3, 5, 7])
np.histogram(x, bins=[2, 4, 7], weights=[4, 2, 1, 3])
```

Результат:

```
(array([6, 4]), array([2, 4, 7]))
```

В интервал [2, 4) попало 6 значений, а в интервал [4, 7] – 4 значения. Выполнение приведенного далее кода приведет к тому же результату:

```
import numpy as np
x=np.array([2, 2, 2, 2, 3, 3, 5, 7, 7, 7])
np.histogram(x, bins=[2, 4, 7])
```

Получить только границы интервалов можно с помощью функции *histogram_bin_edges()*. Синтаксис функции:

```
numpy.histogram_bin_edges(a, bins=10, range=None, weights=None)
```

Параметры функции:

- *a* – структура данных *array_like*. Входные данные;
- *bins* – либо целое число, задающее количество интервалов равной длины, на которые разбивается интервал значений, либо последовательность возрастающих чисел – концов интервалов, либо строка. Все интервалы, кроме последнего (самого правого), полуоткрыты;

– *range* – диапазон. Задается, если нужно изменить определяемые автоматически значения начала (a_{\min}) и конца (a_{\max}) интервала разбиения;
– *weights* – структура данных *array_like*. Массив весов той же формы, что *a*. Если для *density* установлено значение True, веса нормализуются.

Примеры:

```
import numpy as np
a = np.array([0, 2, -3, 1, -2, 9, 4, 3, 5])
np.histogram_bin_edges(a, bins=4),\
np.histogram_bin_edges(a, bins='auto')
```

Результаты:

```
(array([-3., 0., 3., 6., 9.]), array([-3., -0.6, 1.8, 4.2, 6.6, 9. ]))
```

Индексы интервалов, которым принадлежит каждое значение во входном массиве, можно получить с помощью функции *digitize()*. Синтаксис функции:

```
numpy.digitize(x, bins, right=False)
```

Параметры функции:

– *x* – структура данных *array_like*. Входные данные;
– *bins* – структура данных *array_like*. Массив, содержащий значения концов интервалов. Должен быть одномерным и монотонным;
– *right* – переменная, принимающая одно из значений True или False. Указывает, является ли интервал закрытым справа. Поведение по умолчанию (*right=False*) указывает, что интервал не включает правый край, т. е. $\text{bins}[i-1] \leq x < \text{bins}[i]$ является поведением по умолчанию для монотонно возрастающих значений массива *bins*.

Нумерация интервалов начинается с единицы.

Примеры:

```
import numpy as np
x = np.array([0, 5, 10, 15.5, 20.])
bins = np.array([0, 5, 10, 15, 20])
np.digitize(x,bins), np.digitize(x,bins,right=True)
```

Результаты:

```
(array([1, 2, 3, 4, 5], dtype=int64), array([0, 1, 2, 4, 4], dtype=int64))
```

Краткий комментарий. В первом случае (*right=False* по умолчанию) получены интервалы [0, 5), [5, 10), [10, 15), [15, 20). Значение 0

попало в первый интервал, значение 5 – во второй, значение 10 – в третий, значение 15.5 – в четвертый, а значение 20 вышло за пределы, заданные параметром *bins*. Номер интервала, в котором находится это число, считается равным 5. Во втором случае получены такие интервалы: (0, 5], (5, 10], (10, 15], (15, 20]. Поэтому первое значение не попало ни в один из этих интервалов (номер интервала 0), последнее значение попало в четвертый интервал.

2.12. Строковые функции

Рассмотрим некоторые из наиболее часто используемых строковых функций NumPy: *char.add()*, *char.multiply()*, *char.capitalize()*, *char.lower()*, *char.center()*, *char.join()*, *ljust()*, *rjust()*, *char.partition()*, *char.replace()*, *char.split()*, *swapcase()*, *char.title()*, *char.upper()*.

Объединить два массива из строковых значений можно с помощью функции *char.add()*. Массивы должны иметь одинаковую форму.

Пример:

```
import numpy as np
a = np.array(['Фамилия: ', 'Год рождения: '])
b = np.array(['Иванов', '2001'])
result = np.char.add(a, b)
result
```

Результат:

```
array(['Фамилия: Иванов', 'Год рождения: 2001'], dtype='<U21')
```

Повторить строку заданное число раз можно с помощью функции *char.multiply()*:

```
import numpy as np; print(np.char.multiply('-',50))
```

Результат:

Аргументы функции могут быть массивами:

```
import numpy as np
a = np.array(["a", "b", "c"])
```

```
i = np.array([1, 2, 3])
np.char.multiply(a, i)
```

Результат:

```
array(['a', 'bb', 'ccc'], dtype='<U3')
```

Преобразовать первую букву строки в заглавную можно с помощью функции *char.capitalize()*:

```
a = np.array(["paul", "john", "george", 'ringo'])
np.char.capitalize(a)
```

Результат:

```
array(['Paul', 'John', 'George', 'Ringo'], dtype='<U6')
```

Преобразовать все символы верхнего регистра в строке в нижний регистр помогает функция *char.lower()*:

```
a = np.array(["СЛОН", "КОТ"])
np.char.lower(a)
```

Результат:

```
array(['слон', 'кот'], dtype='<U4')
```

Получить копию строки *a* с элементами, центрированными в строке длины *width* можно с помощью функции *char.center(a, width)*.

Пример:

```
c=np.array(["СЛОН", "КОТ"])
np.char.center(c, width=9, fillchar='*')
```

Результат:

```
array(['***СЛОН**', '***КОТ***'], dtype='<U9')
```

Возвратить строку, которая представляет собой конкатенацию строк в последовательности, можно с помощью функции *char.join()*:

```
np.char.join(['-', '*'], ['лето', '1234567'])
```

Результат:

```
array(['л-е-т-о', '1*2*3*4*5*6*7'], dtype='<U13')
```

Получить массив с элементами, выровненными по левому или правому краю, в виде строки длины *width* можно с помощью функций *ljust(a, width [, fillchar])* или *rjust(a, width [, fillchar])* соответственно:

```
c=np.array(["СЛОН"])
np.char.ljust(c, width=9), np.char.rjust(c, 12, fillchar='-')
```

Результаты:

```
(array(['СЛОН'], dtype='<U9'), array(['-----СЛОН'], dtype='<U12'))
```

Разбить строку на 3 части, указав разделитель, можно с помощью функции *char.partition()*. Первая строка будет содержать часть исходной строки перед разделителем, вторая – сам разделитель, третья – часть после разделителя. Если разделитель не найден, возвращаются три строки, содержащие саму строку, за которыми следуют две пустые строки:

```
a = np.array(['His rival, it seems, had broken his dreams'])
np.char.partition(a, 'it seems')
```

Результат:

```
array(['His rival', ' ', 'it seems', ' ', 'had broken his dreams']),
dtype='<U23')
```

Получить копию строки, в которой все вхождения подстроки *old* заменены на *new*, можно с помощью функции *char.replace()*:

```
a = np.array(["The meat is fresh. That is good"])
np.char.replace(a, 'is', 'was', count=1)
```

Результат:

```
array(['The meat was fresh. That is good'], dtype='<U32')
```

В строке было заменено только одно (*count=1*) вхождение подстроки 'is'.

Получить список слов в строке, задав строку-разделитель, можно с помощью функции *char.split()*:

```
import numpy as np
a = np.array(["Now somewhere in the Black Mountain Hills of Dakota \
There lived a young boy named Rocky Raccoon"])
np.char.split(a, ' ', 8)
```

Список слов мы ограничили 8. Дальше разбивка не производилась.

Результат:

```
array([list(['Now', 'somewhere', 'in', 'the', 'Black', 'Mountain', 'Hills', 'of', 'Da-
kota There lived a young boy named Rocky Raccoon'])], dtype=object)
```

Если нужно получить копию строки с символами верхнего регистра, преобразованными в нижний регистр, и наоборот, используют функцию *swapcase()*:

```
a = np.array(['His rival, it seems, Had broken his dreams'])
np.char.swapcase(a)
```

Результат:

```
array(['hIS RIVAL, IT SEEMS, hAD BROKEN HIS DREAMS'],
      dtype='<U42')
```

Получить поэлементную версию строки или юникода в заглавном регистре можно с помощью функции *char.title()*:

```
a = np.array(['his rival, it seems, had broken his dreams'])
np.char.title(a)
```

Результат:

```
array(['His Rival, It Seems, Had Broken His Dreams'], dtype='<U42')
```

Массив с элементами, преобразованными в верхний регистр, можно с помощью функции *char.upper()*:

```
a = np.array(["paul", "john", "george", 'ringo'])
np.char.upper(a)
```

Результат:

```
array(['PAUL', 'JOHN', 'GEORGE', 'RINGO'], dtype='<U6')
```

Подробнее о строковых функциях библиотеки см. в документации по NumPy.

Над массивами из символьных строк или символов юникода (unicode) можно производить операции сравнения: *equal()*, *greater_equal()*, *less_equal()*, *greater()*, *less()*.

В отличие от стандартных операторов сравнения NumPy, операторы в модуле *char* удаляют конечные пробельные символы перед выполнением сравнения. Приведем несколько примеров:

```
import numpy as np
a = np.array(['C', 'Python ', 'Swift'])
b = np.array(['C++', 'Python', 'Java'])
a1 = np.array(['Color', 'cream ', 'Swift'])
b1 = np.array(['Dream', 'clean', 'Team'])
np.char.equal(a, b) # array([False,  True, False])
np.char.not_equal(a, b) # array([ True, False,  True])
np.char.greater(a1, b1) # array([False,  True, False])
```

Так как буква «C» находится перед «D», слово *Color* будет находиться перед словом *Dream*, т. е. *Dream* «больше», чем *Color*). При использовании операций сравнения нужно помнить, что в Python одинаковые буквы, имеющие разный регистр, считаются разными символами, поэтому, например, результат сравнения `np.char.equal(['Python'], ['python'])` равен `False`.

2.13. Дополнительные возможности. Чтение данных из файла

NumPy предлагает функции ввода-вывода (I/O) для загрузки и сохранения данных в файлы и из файлов. Функции ввода-вывода поддерживают множество форматов файлов, включая двоичный и текстовый форматы. Первый из них предназначен для эффективного хранения и извлечения больших массивов. Текстовый формат более удобочитаем для человека и может быть легко отредактирован в текстовом редакторе.

К числу наиболее часто используемых функций ввода-вывода в NumPy относятся функции *save()*, *load()*, *savetxt()*, *loadtxt()*, которые сохраняют/загружают массив в двоичный/текстовый файл.

В NumPy функция *save()* используется для сохранения массива в двоичный файл в формате *.npy*. Синтаксис функции:

```
np.save(file, array)
```

Параметры функции:

- *file* – указывает имя файла. При необходимости указывается путь к файлу;

- *array* – указывает имя массива NumPy, который будет сохранен.

Пример:

```
import numpy as np
# создаем массив
a = np.array([[-1, 7, 5, 9],
               [12, .9, 1.1, 45]])
# сохраняем массив в файл
np.save('file1.npy', a)
```

Был создан массив *a* и сохранен в текущем каталоге в виде двоичного файла с именем *file1.npy*.

Загрузить сохраненный файл можно с помощью функции *load()*:

```
import numpy as np
# загружаем сохраненный файл
b = np.load('file1.npy') ; display(b, b.dtype)
```

Результат:

```
array([[ -1. ,  7. ,  5. ,  9. ],
       [12. ,  0.9,  1.1, 45. ]])
dtype('float64')
```

Функции *savetxt()* и *loadtxt()* имеют аналогичные параметры. Создадим массив и сохраним его в файле с расширением *.txt*:

```
import numpy as np
a1 = np.array([[ -1,  7,  5,  9],
               [12, .9, 1.1, 45]]) # создаем массив
np.savetxt('file2.txt', a1) # сохраняем массив в файл
```

Считать сохраненный файл можно аналогично:
np.loadtxt('file2.txt').

loadtxt() по умолчанию считывает значения как числа с плавающей запятой. При необходимости полученный тип данных можно преобразовать, например, к целому типу с помощью атрибута *astype*:

```
import numpy as np
a2 = np.array([[ -1,  7,  5,  9],
               [12,  9,  1, 45]])
np.savetxt('file2.txt', a1)
b3 = np.loadtxt('file2.txt').astype(np.int64)
display(b3)
```

Результат:

```
array([[ -1,  7,  5,  9],
       [12,  0,  1, 45]], dtype=int64)
```

С помощью функции *np.savez()* можно сохранить несколько массивов. При загрузке файла возвращается объект типа словаря, который содержит отдельные массивы:

```
import numpy as np
a2 = np.array([[ -1,  7,  5,  9], [12,  9,  1, 45]])
```

```
b=np.array([3, 45, -8.9, 8])
np.savez('Two_arr.npz', c=a2, d=b)
arch = np.load('Two_arr.npz')
arch['c']
```

Результат:

```
array([[ -1,  7,  5,  9],
       [12,  9,  1, 45]])
```

Для сжатия данных можно использовать функцию *np.savez_compressed()*.

3. МАТРИЧНЫЕ ОПЕРАЦИИ С МАССИВАМИ. РАБОТА С МОДУЛЕМ LINALG

Рассмотренные во второй главе операции над массивами выполнялись поэлементно. При необходимости выполнения операций умножения по правилам линейной алгебры в библиотеке существуют специальные методы и операторы. Аргументами при выполнении умножения при этом могут быть скаляры, векторы (одномерные массивы), двумерные массивы или матрицы. Для выполнения умножения у аргументов должны совпадать соответствующие размеры.

Функции для разложения матриц, вычисления определителей, обратных матриц, ранга и нормы матриц, собственных значений и собственных векторов содержатся в модуле `numpy.linalg`. В модуле также содержатся функции для решения систем линейных уравнений и многие другие функции для решения задач линейной алгебры.

3.1. Умножение векторов и матриц

В отличие от некоторых языков программирования, таких как MATLAB, в NumPy операция `*` – это поэлементное умножение матриц, а не стандартное умножение матриц в соответствии с правилами линейной алгебры. В связи с этим в NumPy для умножения матриц реализована функция `dot()` как в виде метода объекта типа `ndarray`, так и в виде функции из пространства имен NumPy. Продемонстрируем работу функции в случае, когда хотя бы один из ее аргументов является скаляром:

```
import numpy as np
a,d=2,4
np.dot(a,d) # 2*4=8
b=np.array([3,6,-2])
np.dot(a,b) # array([ 6, 12, -4])
e=np.array([[1,3,2],[4,7,2]])
np.dot(a,e) # array([[ 2,  6,  4], [ 8, 14,  4]])
```

В приведенных примерах для достижения того же результата достаточно было воспользоваться стандартной операцией умножения «*»: $a*d$; $a*b$; $a*e$.

Иная ситуация, когда аргументы являются одномерными или двумерными массивами. Создадим одномерный массив c и умножим его на одномерный массив b :

```
import numpy as np
b=np.array([3,6,-2])
c=np.array([1,3,2])
np.dot(c,b), c*b # (17, array([ 3, 18, -4]))
```

В первом случае результатом умножения является число 17 ($1*3+3*6+2*(-2)=17$), во втором получен массив, элементы которого – результат поэлементного умножения двух массивов. Далее приведены

результаты умножения вектора на матрицу $(2 \ 3) \cdot \begin{pmatrix} 1 & 3 & 2 \\ 4 & 7 & 2 \end{pmatrix}$ и мат-

рицы на матрицу $\begin{pmatrix} 1 & 3 & 2 \\ 4 & 7 & 2 \end{pmatrix} \cdot \begin{pmatrix} 2 & 4 \\ 4 & 8 \\ 5 & 12 \end{pmatrix}$:

```
import numpy as np
e=np.array([[1,3,2],[4,7,2]])
np.dot([2,3],e) # array([14, 27, 10])=[2*1+4*3, 2*3+7*3, 2*2+3*2]
v=np.array([[2,4],[4,8],[5,12]])
np.dot(e,v) # или e.dot(v) результат: array([[24, 52], [46, 96]])
```

Вместо функции *dot()* можно было использовать оператор @:

```
import numpy as np
b=np.array([3,6,-2])
c=np.array([1,3,2])
e=np.array([[1,3,2],[4,7,2]])
v=np.array([[2,4],[4,8],[5,12]])
b @ c # 17
[2,3] @ e; e @ v
```

С помощью функции *vdot()* можно найти скалярное произведение двух векторов:

```
import numpy as np
a=np.array([1, -2, 1, -1])
```

```
b=np.array([5, 7, 12, 4])
np.vdot(a,b)
```

Результат: -1

Для проведения матричных операций можно преобразовать массивы в матрицы с помощью функции библиотеки NumPy *asmatrix()* и воспользоваться операцией умножения. Результатом будет уже не массив, а матрица:

```
v=np.array([[2,4],[4,8],[5,12]])
e=np.array([[1,3,2],[4,7,2]])
t=np.asmatrix(v)
z=np.asmatrix(e)
z*t
```

Результат:

```
matrix([[24, 52],
        [46, 96]])
```

Вычислим обратную матрицу и проверим правильность результата, убедившись в том, что произведение исходной матрицы на обратную равно единичной матрице:

```
import numpy as np
a = np.array([[1,3,7], [-3, 8, 2],[5, 0, 6]])
t=np.asmatrix(a)
display( t**(-1))
t*t**(-1)
```

Результаты:

```
matrix([[-0.32432432, 0.12162162, 0.33783784],
        [-0.18918919, 0.19594595, 0.15540541],
        [ 0.27027027, -0.10135135, -0.11486486]])
matrix([[ 1.00000000e+00, -6.93889390e-17, -2.77555756e-17],
        [-2.22044605e-16, 1.00000000e+00, 5.55111512e-17],
        [ 0.00000000e+00, -2.77555756e-17, 1.00000000e+00]])
```

Данную задачу можно было решить, сразу создав матрицу и применив к ней операцию возведения в степень:

```
import numpy as np
a = np.matrix([[1,3,7], [-3, 8, 2], [5, 0, 6]])
a, a**-1
```

Результаты:

```
(matrix([[ 1,  3,  7],
        [-3,  8,  2],
        [ 5,  0,  6]]),
matrix([[-0.32432432, 0.12162162, 0.33783784],
        [-0.18918919, 0.19594595, 0.15540541],
        [ 0.27027027, -0.10135135, -0.11486486]]))
```

3.2. Работа с модулем `numpy.linalg`

Модуль `numpy.linalg` имеет стандартный набор функций для разложения матриц, вычисления определителя, обратной матрицы, ее ранга и нормы, собственных значений и собственных векторов. В модуле можно вычислить знак и натуральный логарифм определителя матрицы, что бывает удобно, когда значение определителя очень мало. Предусмотрены функции для решения систем линейных уравнений и многие другие функции.

Для умножения нескольких матриц в модуле имеется функция `linalg.multi_dot()`. Синтаксис функции:

```
linalg.multi_dot(arrays, *, out=None)
```

Параметры функции:

- *arrays* – последовательность данных *array_like*. Если первый аргумент – одномерный массив, он обрабатывается как вектор-строка. Если последний аргумент является одномерным массивом, он рассматривается как вектор-столбец. Другие аргументы должны быть двумерными;

- *out* – необязательный выходной аргумент.

Пример:

Вычислить произведение матриц *a*, *b*, *c* и *d*.

$$a = \begin{pmatrix} 1 & -2 \\ 1 & -1 \end{pmatrix}, b = \begin{pmatrix} 5 & 7 \\ 12 & 4 \end{pmatrix}, c = \begin{pmatrix} -3 & 2 \\ 6 & 7 \end{pmatrix}, d = \begin{pmatrix} -1 & 2 \\ -1 & 1 \end{pmatrix}.$$

```
from numpy.linalg import multi_dot
```

```
a=np.array([[1, -2],[ 1, -1]]) ; b=np.array([[5, 7],[ 12, 4]])
```

```
c=np.array([[-3, 2],[ 6, 7]]) ; d=np.array([[-1, 2],[ -1, 1]])
```

```
Z=multi_dot([a, b, c, d])
```

```
Z
```

Результат:

```
array([[ -6, 57],  
       [-46, 85]])
```

Тот же результат можно было получить с помощью метода *dot()*:
`a.dot(b).dot(c).dot(d)`.

Возвести квадратную матрицу в целую степень *n* можно с помощью функции модуля *linalg matrix_power()*:

```
from numpy.linalg import matrix_power  
a=np.array([[1, 2],[ 8, 1]]) ; matrix_power(a, 4)
```

Результат:

```
array([[353, 136],  
       [544, 353]])
```

Продemonстрируем некоторые другие функции модуля на примерах.

Пример. Вычислить определитель матрицы, ее ранг и норму, собственные вектора и собственные значения.

Решение:

```
import numpy as np  
A=np.array([[2,9,9],[2, -3, 12],[4,8,3]]);  
опредетель=np.linalg.det(A);  
print("Определитель матрицы A равен %.3f" % определитель)  
ранг=np.linalg.matrix_rank(A)  
print("Ранг матрицы A равен %i\n" % ранг)  
обр=np.linalg.inv(A);  
print("Матрица, обратная матрице A: ")  
display(обр)  
знач, вект = np.linalg.eig(A)  
print("Вектор собственных значений матрицы A: ")  
display(знач)  
print("Собственные векторы матрицы A: ")  
display(вект)  
норма= np.linalg.norm(A)  
print("Норма Фробениуса: %.5f" % норма)
```

Результаты:

Определитель матрицы A равен 420.000.

Ранг матрицы A равен 3.

Матрица, обратная матрице A:

```
array([[ -0.25      ,  0.10714286,  0.32142857],
       [  0.1      , -0.07142857, -0.01428571],
       [ 0.06666667,  0.04761905, -0.05714286]])
```

Вектор собственных значений матрицы A:

```
array([14.73001151, -2.90089227, -9.82911924])
```

Собственные векторы матрицы A:

```
array([[-0.70537021, -0.92478477,  0.3176012 ],
       [-0.45015909,  0.34659232, -0.84590119],
       [-0.54754878,  0.15699329,  0.42846314]])
```

Норма Фробениуса: 20.29778

Поясним и проверим некоторые результаты.

Поскольку определитель матрицы 3×3 отличен от нуля (420), ранг матрицы равен 3. Обратная матрица A^{-1} это такая матрица, что $AA^{-1}=E$, где E – единичная матрица. Проверим это:

A @ обр

В результате получили единичную матрицу:

```
array([[ 1.00000000e+00, -1.17961196e-16,  7.63278329e-17],
       [ 5.55111512e-17,  1.00000000e+00,  2.77555756e-17],
       [-6.93889390e-17,  3.46944695e-17,  1.00000000e+00]])
```

Собственные значения матрицы это такие значения λ , при которых $\det(A-\lambda E)=0$.

Проверяем:

```
for i in range(A.shape[1]):
    B=np.array([[2-знач[i],9,9],[2, -3-знач[i], 12],[4,8,3-знач[i]])
    f=np.linalg.det(B) ;    print(f)
```

Результаты – числа, близкие к 0:

2.213730537942425e-12

1.64186143010507e-13

3.345721874216797e-13

Когда определитель близок к нулю, для его вычисления бывает удобным использовать функцию *slogdet()*, результатом работы которой

является кортеж из двух чисел: знака числа (1 – в случае положительного значения определителя или -1 – в случае, если определитель отрицательный) и натуральный логарифм абсолютного значения определителя. В рассмотренном ранее примере можно было заменить предпоследнюю строку на `t=np.linalg.slogdet(B)`. Выполнив код, получили:

```
(1.0, -26.836341997295573)
(1.0, -29.437775592376084)
(1.0, -28.725923731684624)
```

Собственный вектор матрицы – это вектор V , умножение матрицы на который дает тот же вектор, умноженный на некоторое число: $AV=\lambda V$, где A – матрица; V – собственный вектор; λ – собственное число.

Убедимся, что собственные векторы найдены правильно.

Собственные векторы – это элементы соответствующих столбцов матрицы *вект*, λ – соответствующие им собственные числа. Делаем проверку:

$$A \cdot V = \begin{pmatrix} 2 & 9 & 9 \\ 2 & -3 & 12 \\ 4 & 8 & 3 \end{pmatrix} \cdot \begin{pmatrix} -0.70537021 \\ -0.45015909 \\ -0.54754878 \end{pmatrix} = (-10.39011124, -6.63084854, -8.06539987).$$

$$\lambda \cdot V = 14.73001151 \cdot (-0.70537021 \quad -0.45015909 \quad -0.54754878) = (-10.39011124, -6.63084854, -8.06539987).$$

Результаты совпали, следовательно, вектор $(-0.70537021 \quad -0.45015909 \quad -0.54754878)$ является собственным вектором матрицы A . Остальные проверки сделаем в Python:

```
for i in range(A.shape[0]):
    print(A @ вект[:,i])
    print(знач[i]*вект[:,i])
```

Результаты совпали для всех собственных векторов:

```
[-10.39011124 -6.63084854 -8.06539987]
[-10.39011124 -6.63084854 -8.06539987]
[ 2.68270099 -1.00542698 -0.45542063]
[ 2.68270099 -1.00542698 -0.45542063]
[-3.12174003  8.31446363 -4.21141529]
[-3.12174003  8.31446363 -4.21141529]
```

В модуле предусмотрена возможность вычисления различных норм. p -Норма (норма Гельдера) для вектора $x = (x_1, \dots, x_n) \in R_n$ вычисляется по формуле $\|x\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$, $p \geq 1$. L_1 -норма (также известная как норма манхэттенское расстояние) для вектора $x = (x_1, \dots, x_n) \in R_n$ вычисляется по формуле $\|x\|_1 = \sum_{i=1}^n |x_i|$. L_2 -норма (евклидова норма) вычисляется по формуле $\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$. Для выбора нормы в функции *linalg.norm()* нужно задать параметр *ord*. Значение *ord*=1 соответствует норме L_1 , *ord*=2 – норме L_2 . Подробнее о том, какие еще нормы (в том числе матричные) можно вычислить, см. в документации по NumPy.

В качестве примера нормы для матрицы в коде была выбрана норма Фробениуса, вычисляемая по формуле $\sqrt{\sum_{i=1}^n \sum_{j=1}^n a_{ij}^2}$. В Python ее можно вычислить и таким образом: `sum(sum(A**2))**.5`.

Для двух векторов $x = (x_1, \dots, x_n)$ и $y = (y_1, \dots, y_n)$ L_1 и L_2 расстояния вычисляются по следующим формулам:

$$\rho_1(x, y) = \|x - y\|_1 = \sum_{i=1}^n |x_i - y_i|,$$

$$\rho_2(x, y) = \|x - y\|_2 = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}.$$

Вычислим, например, L_1 и L_2 расстояния между векторами $x=(2, 6, -3, 4, 12, 3)$ и $y=(2, 5, 1, -4, 6, 3)$:

```
from numpy.linalg import *
x = np.array([2, 6, -3, 4, 12, 3])
y = np.array([2, 5, 1, -4, 6, 3])
print ('Вектор x:', x)
print ('Вектор y:', y)
print ('L1 расстояние между ними: %.3f' % norm(x - y, ord=1))
print ('L2 расстояние между ними: %.3f' % norm(x - y, ord=2))
```

Результаты:

Вектор x: [2 6 -3 4 12 3]

Вектор y: [2 5 1 -4 6 3]

L1 расстояние между ними: 19.000.

L2 расстояние между ними: 10.817.

Для решения систем линейных уравнений в модуле имеются функции *solve()* и *lstsq()*. Последняя из них предназначена для поиска приближенного решения методом наименьших квадратов в случае, когда система уравнений несовместна или имеет множество решений.

Предположим, что исходная система линейных уравнений записана в матричном виде, т. е. в виде $ax = b$, где a – матрица коэффициентов, b – вектор-столбец свободных членов. Пусть, например, дана система из трех уравнений с тремя неизвестными:

$$\begin{cases} 8x_1 - x_2 + 2x_3 = 2 \\ -2x_2 + 5x_3 = 7 \\ -x_1 + 4x_2 - 2x_3 = -25 \end{cases}$$

Перепишем эту систему уравнений в матричном виде:

$$\begin{pmatrix} 8 & -1 & 2 \\ 0 & -2 & 5 \\ -1 & 4 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ -25 \end{pmatrix}.$$

Здесь $a = \begin{pmatrix} 8 & -1 & 2 \\ 0 & -2 & 5 \\ -1 & 4 & -2 \end{pmatrix}$, $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$, $b = \begin{pmatrix} 2 \\ 7 \\ -25 \end{pmatrix}$.

Код для решения этой системы:

```
import numpy as np
a=np.array([[8, -1, 2],[0, -2, 5],[-1, 4, -2]]) #массив a
b=np.array([2, 7,-25]) # массив b
x=np.linalg.solve(a,b)
print("Решение системы уравнений:")
for i in range(len(b)):
    print("x(%i)=%0.5f" % (i+1,x[i]))
```

Результат:

Решение системы уравнений:

x(1)=-0.27559

x(2)=-7.02362

x(3)=-1.40945

Проверяем правильность решения:

```
np.allclose(np.dot(a, x), b)
```

Результат:

True

То есть результат умножения $a @ x$ равен вектору b .

Если система уравнений $ax = b$ несовместна, то можно найти приближенное решение, которое минимизирует норму невязки $d = \|ax - b\|$.

Для поиска приближенного решения, минимизирующего выражение $\|Ax - b\|^2$, в NumPy применяют функцию `linalg.lstsq()`. Если матрица a квадратная и имеет полный ранг, то x (за исключением ошибки округления) является точным решением уравнения. Если существует несколько минимизирующих решений, то возвращается решение с наименьшей нормой L_2 .

Функцию `linalg.lstsq()`, в частности, можно использовать для решения задачи аппроксимации с помощью алгебраических полиномов.

Покажем, как это можно сделать на примере. Пусть некоторая функция задана в виде таблицы:

x	-2	-1	1	2
y	-5.5	0.5	3.5	3

Методом наименьших квадратов найдем для нее уравнение аппроксимирующей параболы $y = a + bx + cx^2$.

Решение:

Если бы парабола проходила через все точки заданной таблицы, то для нахождения ее коэффициентов можно было составить и решить

систему уравнений:

$$\begin{cases} a - 2b + 4c = -5.5 \\ a - b + c = 0.5 \\ a + b + c = 3.5 \\ a + 2b + 4c = 3 \end{cases}$$

Левая часть каждого уравнения получена подстановкой табличных значений x в уравнение параболы.

Теперь решим эту систему уравнений с помощью функции `linalg.lstsq()`:

```
import numpy as np
a=np.array([[1, -2, 4],[ 1, -1, 1],[1, 1, 1],[1, 2, 4]]) #массив a
b=np.array([-5.5, .5, 3.5, 3]) # массив b
f=np.linalg.lstsq(a, b, rcond=None)
print("Приближенное решение системы уравнений МНК:")
for i in range(a.shape[1]):
```

```

    print("x(%i)=%8.5f" % (i+1,f[0][i]))
d=b-a@f[0]
print("Вектор невязок b-ax:")
for i in range(a.shape[0]):
    print("d(%i)=%8.5f" % (i+1,d[i]))
print("Сумма площадей квадратов отклонений: %.5f" % f[1])

```

Результаты:

Приближенное решение системы уравнений МНК:

x(1)= 3.08333

x(2)= 2.00000

x(3)=-1.08333

Вектор невязок b-ax:

d(1)=-0.25000

d(2)= 0.50000

d(3)=-0.50000

d(4)= 0.25000

Сумма площадей квадратов отклонений: 0.62500.

Таким образом, уравнение аппроксимирующей параболы $y=3.08333 + 2x - 1.08333x^2$.

Сумму площадей квадратов отклонений можно найти и таким образом: `sum((a@f-b)**2)`.

Графическое решение задачи аппроксимации представлено на рис. 3.1.

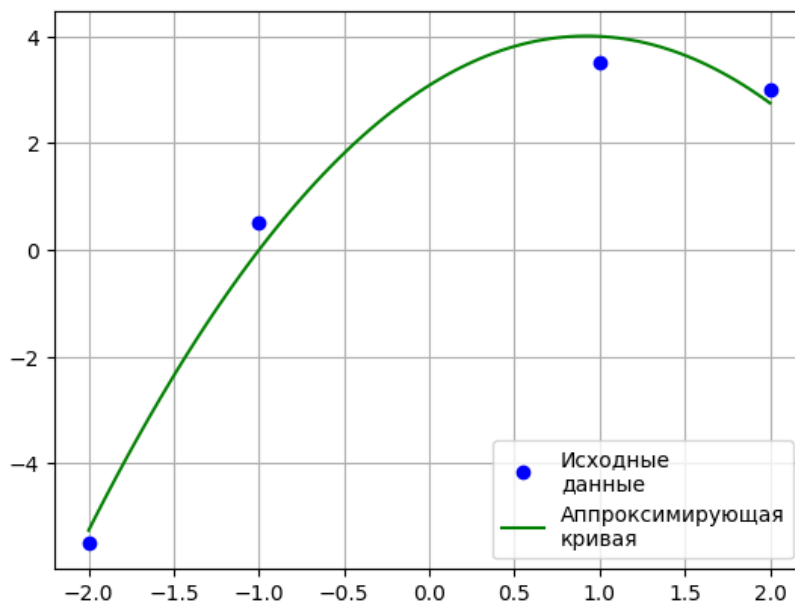


Рис. 3.1. Графическое решение задачи аппроксимации

Код для построения графика:

```
import numpy as np
import matplotlib.pyplot as plt
x=[-2,-1, 1, 2]
y=np.array([-5.5, 0.5, 3.5, 3])
a=np.array([[1, -2, 4],[ 1, -1, 1],[1, 1, 1],[1, 2, 4]]) #массив a
b=np.array([-5.5, .5, 3.5, 3]) # массив b
f=np.linalg.lstsq(a, b, rcond=None)
a, b, c = f[0][0], f[0][1], f[0][2]
x1 = np.linspace(min(x), max(x), 100);
ym= a+b*x1+c*x1**2
plt.plot(x, y, 'bo', label="Исходные\нданные")
plt.plot(x1,ym, label="Аппроксимирующая\пкривая",c='g')
plt.legend()
plt.grid()
plt.show()
```

4. ГЕНЕРИРОВАНИЕ СЛУЧАЙНЫХ ЧИСЕЛ

Модуль `numpy.random` реализует генераторы псевдослучайных чисел с возможностью получения выборок из различных вероятностных распределений. Как правило, для этого нужно создать конструктор `default_rng` и вызвать для него различные методы для получения последовательностей из разных распределений. Их можно применять для математического моделирования. Можно получать и случайные перестановки с помощью `RandomArray.permutation()`.

4.1. Генерирование чисел из дискретных распределений

Дискретной случайной величиной называется величина, принимающая конечное или счетное множество значений. Оценка студента на экзамене – дискретная случайная величина.

В табл. 4.1 приведены функции, генерирующие числа с заданным дискретным распределением.

Таблица 4.1

Функции для генерации чисел из дискретных распределений

Функция	Назначение
<code>integers(low, high, size, endpoint)</code>	Генерирует случайные целые числа от <code>low</code> (включая) до <code>high</code> (не включая или включая)
<code>binomial(n, p[, size])</code>	Генерирует числа с биномиальным распределением
<code>poisson([lam, size])</code>	Генерирует числа с распределением Пуассона
<code>geometric(p[, size])</code>	Генерирует числа с геометрическим распределением
<code>hypergeometric(ngood, nbad, nsample[, size])</code>	Генерирует числа с гипергеометрическим распределением ($n = \text{ngood}$, $m = \text{nbad}$, and $N = \text{number of samples}$)
<code>negative_binomial(n, p[, size])</code>	Генерирует числа с отрицательным биномиальным распределением

Рекомендуемый конструктор для класса случайных чисел *Generator* – *default_rng*. Рассмотрим несколько способов, которыми можно построить генератор случайных чисел, используя *default_rng* и класс *Generator*.

Для генерации случайного целочисленного числа в диапазоне от *low* до *high* можно использовать метод *random.Generator.integers()*.

Синтаксис метода:

```
random.Generator.integers(low, high=None, size=None, dtype=
np.int64, endpoint=False)
```

Другие параметры метода:

- *size* – форма выходной выборки;
- *endpoint* – булева переменная, включается ли число *high* в диапазон. По умолчанию принимает значение False: интервал [*low*, *high*).

Сгенерируем массив из трех целых чисел из интервала [6, 12]:

```
rng = np.random.default_rng()
rng.integers(6,12,3, endpoint=True)
```

Результат:

```
array([10, 11, 12], dtype=int64)
```

Таким образом можно создать массив заданного размера с целыми числами от 0 до 7 включительно:

```
rng = np.random.default_rng()
rng.integers(8, size=(2, 5))
```

Результат:

```
array([[5, 0, 5, 2, 3],
       [2, 2, 4, 0, 0]], dtype=int64)
```

Для получения выборок из биномиального распределения используют функцию *binomial()*. Выборки берутся из биномиального распределения с параметрами *n* и *p*, где *n* – число последовательных независимых испытаний, *p* – вероятность наступления интересующего нас события *A*. Функция выдает количество $X=t$ наступлений события *A* в *n* опытах. Пример на применение функции:

```
rng = np.random.default_rng();rng.binomial(10, 0.6, size=4)
```

Результат:

```
array([5, 6, 8, 3], dtype=int64)
```

Чтобы получить выборку из распределения Пуассона, используют функцию *poisson()*. В качестве параметра надо передать *lam* – ожидаемое количество событий, происходящих в фиксированный интервал времени. $lam=np$, где n – число последовательных независимых испытаний, p – вероятность наступления интересующего нас события.

Пример на применение функции poisson():

```
import numpy as np
rng = np.random.default_rng()
rng.poisson(5, 3)
```

Результат:

```
array([5, 9, 8], dtype=int64)
```

Для получения выборок из геометрического распределения используют функцию *geometric()*. Выборки берутся из геометрического распределения с параметром p , где p – вероятность наступления интересующего нас события A . Геометрическое распределение моделирует количество испытаний, которые необходимо выполнить для достижения успеха.

Пример на применение функции geometric():

```
np.random.default_rng().geometric(p=0.4, size=5)
```

Результат:

```
array([3, 5, 1, 2, 4], dtype=int64)
```

Для получения выборки из гипергеометрического распределения используют функцию *hypergeometric()*. Типичный пример этого распределения: осуществлена поставка из $ngood + nbad$ объектов, $nbad$ из которых имеют дефект. Гипергеометрическое распределение описывает вероятность того, что в выборке из $nsample$ различных объектов, случайным образом выбранных из поставки, ровно k объектов являются бракованными. Функция имеет 4 параметра: $ngood$, $nbad$, $nsample$, $size$.

Для иллюстрации работы функции предположим, что имеется 100 «хороших» образцов и 5 «плохих». Сгенерируем 1000 случайных выборок, каждая из 40 образцов, и построим распределение числа «хороших» образцов:

```
import numpy as np
rng = np.random.default_rng()
ngood, nbad, nsamp = 100, 5, 40
s = rng.hypergeometric(ngood, nbad, nsamp, 1000)
```

```

val, freq = np.unique(s, return_counts=True)
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(8,5))
ax.bar(val, freq)
ax.bar_label(ax.containers[0])

```

Результат распределения «хороших» образцов в выборке представлен на рис. 4.1.

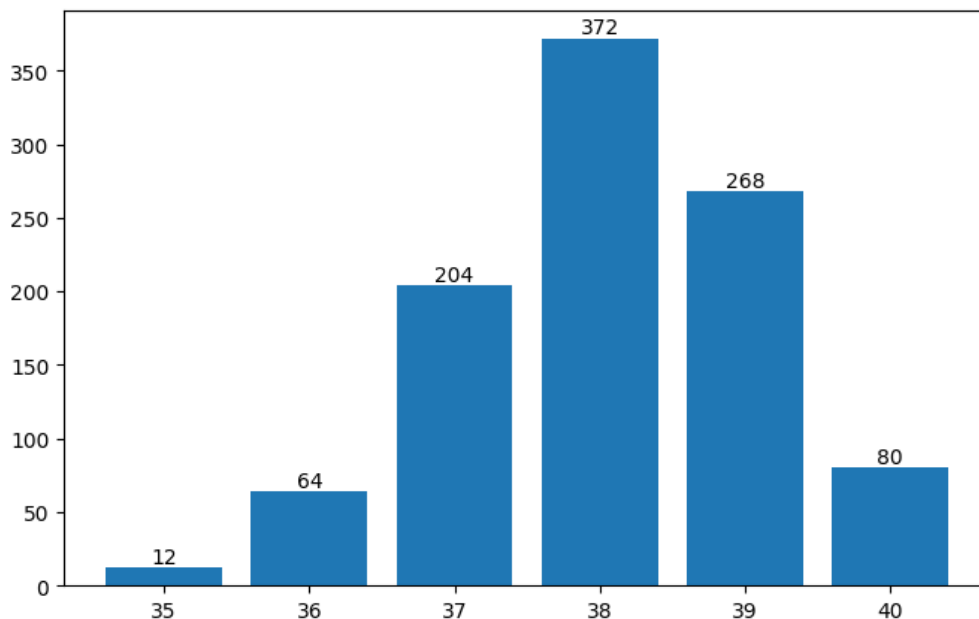


Рис. 4.1. Распределение «хороших» образцов в выборке

Поясним полученные результаты. 12 раз встречалась выборка, в которой из 40 образцов «хороших» было 35, 64 раза – в которой небракованных экземпляров было 36, ..., 80 раз – в которой не было «плохих».

Для получения чисел с отрицательным биномиальным распределением используют функцию *negative_binomial()*. Выборки берутся из отрицательного биномиального распределения с параметрами n (количество успехов) и p (вероятность успеха), где $n > 0$, а p находится в интервале $(0, 1)$.

Пример на применение функции:

```
np.random.default_rng().negative_binomial(10, 0.1, 5)
```

Результат:

```
array([ 73, 85, 87, 138, 120], dtype=int64)
```

Элементы выборки – количество неудач, которые произошли до того, как было достигнуто общее количество n успехов (10).

4.2. Генерирование чисел из непрерывных распределений

Непрерывной случайной величиной называется величина, принимающая любые значения из интервала, конечного или бесконечного. Время проведения экзамена – непрерывная случайная величина.

Для того чтобы сгенерировать последовательности из случайных чисел, элементы которой подчиняются заданному закону, в NumPy включены генераторы для таких непрерывных распределений, как равномерное, нормальное (распределение Гаусса), бета, хи-квадрат, Дирихле, экспоненциальное, Фишера, Гамма, Гумбеля, Лапласа, логистическое, логнормальное, логарифмическое, многомерное нормальное, нецентральное хи-квадрат, нецентральное Фишера, Парето, степенное, Рэлея, Коши, Стюдента, треугольное, фон Мизеса, Вальда, Вейбулла, Зипфа и некоторых других.

В табл. 4.2 приведены функции для генерирования случайных чисел с некоторыми из имеющихся в библиотеке законами распределения. Для применения этих функций в них, как правило, нужно передать параметры требуемого распределения. Вид плотности распределений, приведенных в таблице, приведен в документации по функции. Формулы для вычисления плотности можно найти и в соответствующей литературе по теории вероятностей и математической статистике.

Таблица 4.2

Функции для генерации чисел из непрерывных распределений

Функция	Назначение
1	2
<code>rand(d0, d1, ..., dn)</code>	Генерирует набор случайных чисел заданной формы
<code>randn([d1, ..., dn])</code>	Генерирует набор (или наборы) случайных чисел со стандартным нормальным распределением
<code>beta(a, b[, size])</code>	Генерирует числа с β -распределением
<code>chisquare(df[, size])</code>	Генерирует числа с распределением хи-квадрат
<code>dirichlet(alpha[, size])</code>	Генерирует числа с распределением Дирихле (alpha – массив параметров)

1	2
<code>exponential([scale, size])</code>	Генерирует числа с экспоненциальным распределением
<code>f(dfnum, dfden[, size])</code>	Генерирует числа с F-распределением (dfnum – число степеней свободы числителя > 0; dfden – число степеней свободы знаменателя > 0)
<code>gamma(shape[,scale, size])</code>	Генерирует числа с гамма-распределением
<code>gumbel([loc, scale, size])</code>	Генерирует числа с распределением Гумбеля
<code>laplace([loc, scale, size])</code>	Генерирует числа с распределением Лапласа
<code>logistic([loc, scale, size])</code>	Генерирует числа с логистическим распределением
<code>lognormal([mean, sigma, size])</code>	Генерирует числа с логарифмическим нормальным распределением
<code>logseries(p[, size])</code>	Генерирует числа с распределением логарифмического ряда
<code>noncentral_chisquare(df, nonc[, size])</code>	Генерирует числа с нецентральным распределением хи-квадрат
<code>noncentral_f(dfnum, dfden, nonc[, size])</code>	Генерирует числа с нецентральным F-распределением (dfnum – целое > 1; dfden – целое > 1; nonc : действительное >= 0)
<code>normal([loc, scale, size])</code>	Генерирует числа с нормальным распределением
<code>pareto(a[, size])</code>	Генерирует числа с распределением Парето
<code>power(a[, size])</code>	Генерирует числа со степенным распределением [0, 1]
<code>rayleigh([scale, size])</code>	Генерирует числа с распределением Релея
<code>standard_cauchy ([size])</code>	Генерирует числа со стандартным распределением Коши
<code>standard_exponential ([size])</code>	Генерирует числа со стандартным экспоненциальным распределением
<code>standard_gamma(shape[, size])</code>	Генерирует числа с гамма-распределением
<code>standard_normal([size])</code>	Генерирует числа со стандартным нормальным распределением (среднее=0, сигма=1)
<code>standard_t(df[, size])</code>	Генерирует числа со стандартным распределением Стьюдента с df степенями свободы
<code>triangular(left, mode, right[, size])</code>	Генерирует числа из треугольного распределения

1	2
<code>uniform([low, high, size])</code>	Генерирует числа с равномерным распределением
<code>vonmises(mu, kappa [, size])</code>	Генерирует числа с распределением фон Мизеса
<code>wald(mean, scale [, size])</code>	Генерирует числа с распределением Вальда
<code>weibull(a[, size])</code>	Генерирует числа с распределением Вейбулла
<code>zipf(a[, size])</code>	Генерирует числа с распределением Зипфа (зетта функция Римана)

Массив случайных чисел из интервала $[0.0, 1.0)$ может быть сгенерирован с помощью функции `random.rand()`. Можно сгенерировать как одно значение, так и получить массив нужной формы из нескольких значений:

```
import numpy as np
d=np.random.rand
d(), d(3), d(2,3), d(6).reshape(3,2)
```

Результаты:

```
(0.8594711651095508,
array([0.78743264, 0.81044604, 0.36260353]),
array([[0.63863103, 0.89647501, 0.9518258 ],
        [0.25643834, 0.04718807, 0.67106264]]),
array([[0.47256315, 0.01100635],
        [0.35332992, 0.8599897 ],
        [0.54788058, 0.98269617]]))
```

Чтобы не писать каждый раз перед скобками `np.random.rand`, для обозначения метода в коде была введена переменная *d*.

Для генерирования числа с равномерным распределением на интервале $[low, high)$ используют функцию `uniform()`. В нее нужно передать параметры распределения (концы отрезка) и количество требуемых значений. Таким образом можно сгенерировать 5 чисел, равномерно распределенных на отрезке $[-2, 3)$:

```
np.random.default_rng().uniform(-2, 3, 5)
```

Результат:

```
array([-0.07549432,  1.22000135, -0.83840045,  1.82861636,
 2.93504561])
```

Получить два массива из четырех чисел каждый, распределенных по нормальному закону с параметрами $\text{loc}=0$, $\text{scale}=1$ (математическое ожидание и среднеквадратическое отклонение соответственно) и $\text{loc}=3$, $\text{scale}=2$ можно следующим образом:

```
s=np.random.default_rng().normal(3, 2, 4),  
z=np.random.default_rng().standard_normal(4)  
s, z
```

Результат:

```
((array([4.15990665, 4.76133444, 6.89614925, 0.89173674]),),  
 array([ 1.19735578, -1.61984364, 0.32317691, -0.624929 ]))
```

Во втором случае (функция *standard_normal()*) параметры распределения в функцию не передавались, так как стандартное нормальное распределение предполагает, что $\text{loc}=0$, $\text{scale}=1$. Хотя можно было воспользоваться и функцией *normal()*, передав в нее эти параметры.

Качество работы генератора можно оценить, получив выборку большого объема и построив нормированную гистограмму. В приведенном далее примере наряду с гистограммами для 4 распределений были построены графики функций плотности каждого из них:

```
import matplotlib.pyplot as plt ; import numpy as np  
from scipy.stats import f, gamma, chi2, t  
rng = np.random.default_rng(); plt.figure(figsize=[11, 9])  
# создаем общий заголовок  
plt.suptitle('Гистограммы, построенные \n \n по сгенерированным данным,\n \n и графики функции плотности', fontsize=19, fontweight='bold')  
plt.subplot(2, 2, 1) # Первый график  
x = np.linspace(.01,5,100)  
plt.plot(x, f.pdf(x,6,60), 'k-', lw=3, alpha=0.6)  
plt.title('Распределение Фишера', fontsize=15)  
plt.grid(); plt.legend(['v1=6, v2=60']); y=rng.f(6, 60, 1000)  
plt.hist(y, 10,density = True,edgecolor='k',rwidth=.9,  
         color='white')  
plt.subplot(2, 2, 2) # Второй график  
x = np.linspace(.01,10,100)  
plt.plot(x, gamma.pdf(x,5,0,1/3), 'k-', lw=3, alpha=0.6)  
plt.title('Гамма-распределение', fontsize=15)  
t1=rng.gamma(5, 1/3, 1000)  
plt.hist(t1, 10,density = True,edgecolor='k',rwidth=.9,
```

```

        color='white')
plt.grid(); plt.legend(['k=5,  $\lambda=3$ '])
plt.subplot(2, 2, 3) # Третий график
x = np.linspace(.01,27,100); plt.plot(x, chi2.pdf(x,10),'k--', lw=3)
plt.title('Распределение хи-квадрат', fontsize=15)
y1=rng.chisquare(10, 1000)
plt.hist(y1, 10,density = True,edgecolor='k',
        color='white',rwidth=.9)
plt.grid(); plt.legend(['n=10']);
plt.subplot(2, 2, 4) # Четвертый график
x = np.linspace(-6,6,100); y2=rng.standard_t(15, 1000)
plt.title('Распределение Стьюдента', fontsize=15)
plt.plot(x, t.pdf(x,15), 'k-.', lw=3, alpha=0.6)
plt.grid(); plt.hist(y2, 10,density = True,edgecolor='k',rwidth=.9,
        color='white')
plt.legend(['v=15']); plt.show()

```

**Гистограммы, построенные по сгенерированным данным,
и графики функции плотности**

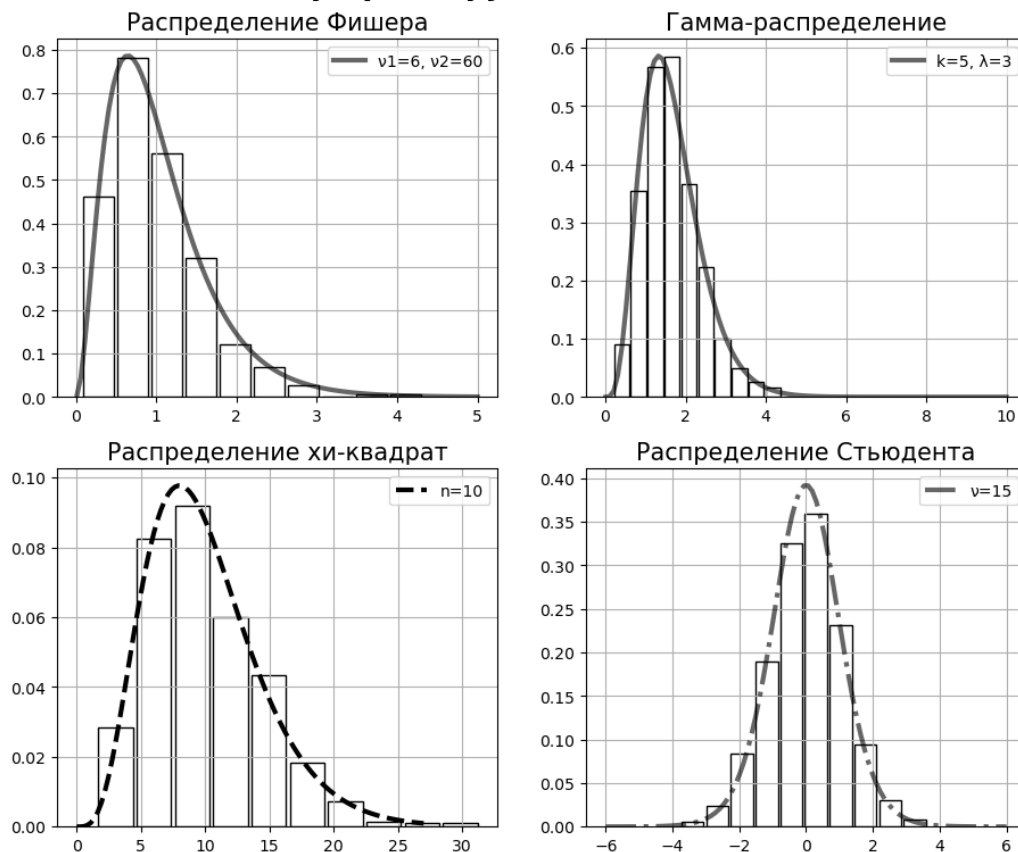


Рис. 4.2. Гистограммы чисел, распределенных по указанному закону с приведенными параметрами, и графики функций плотности этих распределений

Краткий комментарий к коду. Кроме библиотеки NumPy, нам понадобились библиотеки Matplotlib (для построения графика) и SciPy (для получения данных для построения функций плотности). Они были импортированы. В принципе без последней библиотеки можно было обойтись, но тогда пришлось бы составлять программы для вычислений функций плотностей каждого из распределений. Графическое окно было разбито на 4 подокна и в каждом из них построен свой график. Задавались пределы изменения аргумента x , строился подписанный график, на который наносилась сетка. В легенде указаны параметры распределений.

По взаимному расположению гистограмм и соответствующих функций распределения можно сделать вывод о хорошей работе соответствующих генераторов.

4.3. Генерирование случайной выборки из заданного массива

Модуль содержит функции создания случайной последовательности из элементов заданного массива – функции *shuffle()* и *permutation()*. Функции имеют два параметра. Первый из них – массив, список или последовательность для перемешивания, второй – ось, вдоль которой необходимо провести перемешивание элементов. Основное различие между *Generator.shuffle* и *Generator.permutation* заключается в том, что *Generator.shuffle* работает на месте, а *Generator.permutation* возвращает перемешанную копию массива.

Таблица 4.3

Функции для перестановки элементов массива

Функция	Назначение
<i>shuffle(x)</i>	Переставляет элементы на месте
<i>permutation(x)</i>	Возвращает последовательность элементов, переставленных случайным образом
<i>choice(x, size, replace, p, axis, shuffle)</i>	Генерирует случайную выборку из заданного одномерного массива

Пусть, например, нам задан массив из чисел: [0, -5, 9.45, 15, 34].
Переставим элементы массива в случайном порядке:

```
import numpy as np
y=np.array([0, -5, 9.45, 15, 34])
print(y)
np.random.shuffle(y)
print(y)
```

Результаты:

```
[ 0. -5.  9.45 15. 34. ]
[15.  9.45 -5. 34.  0. ]
```

Исходный массив изменился.

Перестановки можно проводить по любой оси:

```
import numpy as np
rng = np.random.default_rng() # Создаем новый генератор
A=np.array([[2,9,9],[2, -3, 12],[4,8,3]])
print(A);rng.shuffle(A,axis=1);print('\n',A)
```

Результаты:

```
[[ 2  9  9]
 [ 2 -3 12]
 [ 4  8  3]]
```

```
[[ 9  9  2]
 [12 -3  2]
 [ 3  8  4]]
```

Использование для перестановок функции *permutation()* не приводит к изменению исходного массива:

```
y=np.array([ 0, -5, 9.45, 15, 34])
d=np.random.permutation (y)
print(y)
print(d)
```

Результаты:

```
[ 0. -5.  9.45 15. 34. ]
[-5. 15. 34.  0.  9.45]
```

С помощью функции *choice()* можно сгенерировать случайную выборку из заданного одномерного массива. Числа в полученном массиве могут повторяться. Синтаксис функции:

```
random.Generator.choice(a, size=None, replace=True, p=None, axis=0, shuffle=True)
```

Параметры функции:

- *a* – структура данных *array_like* или целое число. Если *a* ndarray, то из его элементов генерируется случайная выборка. Если *a* – целое число, случайная выборка генерируется из массива *np.arange(a)*;

- *size* – размер выходного массива. Целое число или кортеж из целых чисел. Если размер задан как кортеж (*m, n, k*), тогда будут выбраны *m·n·k* образцов из *a*. По умолчанию принимает значение None. В этом случае выбирается 1 элемент;

- *replace* – булева переменная. Параметр, определяющий, может ли элемент из *a* встречаться в сгенерированной выборке больше одного раза. По умолчанию установлено значение True, что означает, что значение из *a* можно выбирать несколько раз;

- *p* – вероятность появления каждого элемента из *a*. Если параметр не указан, предполагается равномерное распределение;

- *axis* – ось, по которой осуществляется выбор. Значение параметра по умолчанию 0: выбор осуществляется по строке;

- *shuffle* – параметр, определяющий, перетасовывается ли полученная выборка при значении *replace = False*.

Следующим образом можно выбрать три случайных целых числа из отрезка [0, 5]:

```
rng = np.random.default_rng()
rng.choice(5, 3) # array([0, 2, 0], dtype=int64)
```

Таким образом – три случайных числа из массива *y*:

```
rng = np.random.default_rng()
y=np.array([ 0, -5, 9.45, 15, 34])
rng.choice(y, 3) # array([34., 15., 15.])
```

Так можно создать из элементов массива *y* двумерный массив с заданным числом строк и столбцов:

```
import numpy as np
rng = np.random.default_rng()
y=np.array([ 0, -5, 9.45, 15, 34, 76])
rng.choice(y, (3, 5))
```


Результат:

```
array([[15. , 0. , 15. , 9.45, 0. ],  
       [34. , 0. , 76. , 15. , -5. ],  
       [34. , 76. , 76. , -5. , 76. ]])
```

Приведем два примера с генерированием одного и того же набора чисел со значением параметров `replace=False`, `shuffle=False` и `replace=False`, `shuffle=True`:

```
import numpy as np ; rng = np.random.default_rng(55555555)  
y=np.array([ 0, -5, 9.45, 15, 34, 76])  
g=rng.choice(y, (3, 2), replace=False, shuffle=False) ; g
```

Результат:

```
array([[ 0. , -5. ],  
       [ 9.45, 15. ],  
       [34. , 76. ]])
```

Запустим генератор еще один раз:

```
import numpy as np ; rng = np.random.default_rng(55555555)  
y=np.array([ 0, -5, 9.45, 15, 34, 76])  
g=rng.choice(y, (3, 2), replace=False) ; g
```

Результат:

```
array([[76. , -5. ],  
       [ 9.45, 34. ],  
       [ 0. , 15. ]])
```

И в том, и другом случае получены одни и те же наборы чисел, но порядок их в массиве `g` разный. Одинаковые наборы получились, потому что мы указали одно и то же значение (55555555) в качестве параметра генератора.

Если мы хотим, чтобы одни значения появлялись чаще других, можно задать параметр `p`:

```
import numpy as np ; rng = np.random.default_rng()  
y=np.array([ 0, -5, 9.45, 15, 34]); rng.choice(y, 8,[0.1, .4, .4, 0.05, 0.05])
```

Результат:

```
array([ 9.45, -5. , 9.45, 15. , 15. , -5. , 9.45, 15. ])
```

Чаще всего появляются числа 9.45 и -5, так как вероятность их появления выше.

Случайную выборку можно осуществлять из последовательностей символов:

```
import numpy as np; rng = np.random.default_rng()
animals = ['cat', 'dog', 'wolf', 'fox']
rng.choice(animals, 5, p=[0.5, 0.2, 0.1, 0.2])
```

Результат:

```
array(['fox', 'cat', 'cat', 'wolf', 'dog'], dtype='<U4')
```

Более подробно о работе модуля `numpy.random` см. документацию по NumPy.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

В приведенных заданиях α – это номер факультета, β – последняя цифра года поступления, γ и θ – две последние цифры номера группы, μ – первая цифра номера студента по списку, ν – последняя цифра номера. Так, у пятого студента группы 731312 $\alpha = 7$, $\beta = 3$, $\gamma = 1$, $\theta = 2$, $\mu = 0$, $\nu = 5$. N_{st} – это номер студента по списку (5, 21 и т. д.).

Задание 1. Без использования циклов средствами NumPy решить следующие задачи:

- а) создайте случайный вектор из 10 целых чисел, каждое из которых лежит в пределах от N_{st} до $N_{st} + 20$;
- б) создайте вектор из нулей длины $N_{st} + 10$, но его элемент с номером β должен быть равен 1;
- в) создайте матрицу $(\mu+2, 3)$, заполненную целыми числами из диапазона $[0, \mu + \nu + 1]$;
- г) найдите все положительные числа в `np.array([1,2,0,42,4,0])`;
- д) умножьте матрицу размерности $(4, 3)$ на матрицу размерности $(3, 2)$;
- е) создайте случайный вектор из $N_{st} + 4$ элементов и отсортируйте его;
- ж) для заданного числа найдите ближайший к нему элемент в векторе;
- з) создайте массив из файла;
- и) создайте массив из функции;
- й) создайте одномерный массив и выведите на экран два его последних элемента;
- к) создайте одномерный массив из $N_{st} + 8$ элементов и выведите на экран 4 элемента с заданными номерами;
- л) создайте матрицу 4 на 4 и выведите на экран две ее строки;
- м) из массива 5 на 5 создайте массив, в котором элементы каждой строки будут идти в обратном порядке;
- н) измените два значения в одномерном массиве;
- о) отсортируйте одномерный массив по возрастанию и убыванию;
- п) вычислите минимальные и максимальные элементы каждого столбца матрицы 5 на 4;
- р) вычислите разность и симметричную разность двух массивов;
- с) выведите на экран уникальные элементы массива и их частоту;
- т) преобразуйте одномерный массив из 15 элементов в двумерный;

- у) создайте и транспонируйте матрицу 4 на 4;
- ф) объедините два одномерных массива;
- х) добавьте строку в двумерный массив;
- ц) удалите строку из двумерного массива;
- ч) создайте двумерный массив и выведите на экран те его элементы, которые больше среднего арифметического.

Задание 2. Перемножьте матрицы A , B и C :

$$A = \begin{pmatrix} 2 & 3+\alpha & 4-\mu \\ \beta & \gamma & \nu \\ 5 & 10\mu & -2 \end{pmatrix}, B = \begin{pmatrix} \beta & -3 & 4+\mu \\ \alpha & 4 & \nu \\ 5\mu & 10 & -2 \end{pmatrix}, C = \begin{pmatrix} -1 & 3+\mu & 4-\nu \\ 2 & 1 & \nu \\ 5 & 5-\mu & -2 \end{pmatrix}.$$

Задание 3. Вычислите определитель, ранг, норму, вектор собственных значений матрицы G и матрицу, обратную ей:

$$G = \begin{pmatrix} 2 & 3+\alpha & 4+\mu & 6 \\ \beta+1 & \gamma & \nu & 12 \\ 5 & -2 & 16 & \mu+1 \\ 2 & 7 & 11 & -6 \end{pmatrix}.$$

Задание 4. Решите систему линейных уравнений:

$$\begin{cases} 3x_1 + x_2 + x_3 + \gamma x_4 = \alpha \\ x_1 - \mu x_2 + \nu x_3 + 4x_4 = \beta \\ -5x_1 - x_3 - 7x_4 = -5 \\ x_1 - 6x_2 + \alpha x_3 + 6x_4 = 3 \end{cases}.$$

Задание 5. Найдите решение системы линейных алгебраических уравнений методом наименьших квадратов:

$$\begin{cases} 3x_1 + x_2 + x_3 + \gamma x_4 = \alpha \\ x_1 - \mu x_2 + \nu x_3 + 4x_4 = \beta \\ -5x_1 - x_3 - 7x_4 = -5 \end{cases}.$$

Задание 6. Сгенерируйте массив из $n = 500 + 10\mu + \nu$ чисел, распределенных по нормальному закону с математическим ожиданием, равным ν и дисперсией, равной $\mu + 2$. Разбейте интервал значений на k подынтервалов. Количество подынтервалов выберите по правилу

Стерджеса: $k = 1 + 3.322 \lg n$. Значение k округлите в большую сторону. Подсчитайте абсолютные и относительные частоты попадания в каждый интервал. Результаты представьте в виде датафрейма, столбцы которого должны содержать следующую информацию: номер интервала, левый конец интервала, его середина, правый конец, абсолютные и относительные частоты попадания в интервал. Подсчитайте накопленную сумму частот (функция `cumsum()`), среднее значение элементов массива, медиану, выборочную дисперсию (смещенную и несмещенную), среднеквадратическое отклонение и 30 % квантиль. В отчете приведите формулы для расчета указанных характеристик. Поясните полученные результаты. Сведения о работе с датафреймами можно найти, например, в пособии А. Н. Титова и Р. Ф. Тазиевой².

² Титов, А. Н., Тазиева Р. Ф. Обработка данных в Python. Основы работы с библиотекой Pandas. Казань: Изд-во КНИТУ, 2022. 116 с.

ЛИТЕРАТУРА

1. Документация NumPy. – URL: <https://numpy.org/doc/>
2. Нескучный tutorial по NumPy. – URL: <https://habr.com/ru/articles/469355/>
3. Титов, А. Н. Python. Обработка данных / А. Н. Титов, Р. Ф. Тазиева. – Казань: Изд-во КНИТУ, 2022. – 104 с.
4. Титов, А. Н. Обработка данных в Python. Основы работы с библиотекой Pandas / А. Н. Титов, Р. Ф. Тазиева. – Казань: Изд-во КНИТУ, 2022. – 116 с.
5. Руководство по использованию Python-библиотеки NUMPY. – URL: <https://pythonru.com/biblioteki/rukovodstvo-po-ispolzovaniju-python-biblioteki-numpy>
6. Модули math и numpy – основы Python. – URL: <https://academy.yandex.ru/handbook/python/article/moduli-math-i-numpy>
7. Numpy. Матричные вычисления. – URL: <https://physics.susu.ru/vorontsov/language/numpy.html>
8. Основы NumPy: массивы и векторные вычисления. – URL: <https://slemeshevsky.github.io/python-course/numpy/pdf/numpy.pdf>
9. NumPy String Functions. – URL: <https://www.programiz.com/python-programming/numpy/string-functions>
10. Борю, С. Ю. Модули и пакеты в Python: учеб.-метод. пособие для студентов естественно-научных специальностей / С. Ю. Борю. – Запорожье: ЗНУ, 2021. – 286 с.
11. NumPy: матрицы и операции над ними. – URL: http://cs.mipt.ru/advanced_python/lessons/lab16.html#section-13
12. Вадзинский, Р. Н. Справочник по вероятностным распределениям / Р. Н. Вадзинский. – Санкт-Петербург: Наука, 2001. – 295 с.
13. Кобзарь, А. И. Прикладная математическая статистика для инженеров и научных работников / А. И. Кобзарь. – Москва: ФИЗМАТЛИТ, 2006. – 816 с.

УЧЕБНОЕ ИЗДАНИЕ

*Андрей Николаевич Титов
Рамиля Фаридовна Тазиева*

ОСНОВЫ РАБОТЫ С БИБЛИОТЕКОЙ NUMPY

*Редактор Л. А. Муравьева
Компьютерная верстка и макет – А. К. Рахманкулова*

Подписано в печать 15.03.2024

Бумага офсетная

7,0 уч.-изд. л.

Печать цифровая

Тираж 400 экз.

Формат 60×84 1/16

6,51 усл. печ. л.

Заказ 15/24

Издательство Казанского национального исследовательского
технологического университета

Отпечатано в офсетной лаборатории Казанского национального
исследовательского технологического университета

420015, Казань, К. Маркса, 68