Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего

образования «Южно-Уральский государственный университет (национальный

исследовательский университет)»

Высшая школа экономики и управления

Кафедра Информационных технологий в экономике

Программирование на языке Python (курс молодого бойца)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ

по дисциплине «ПРОГРАММИРОВАНИЕ»

ЮУрГУ – 380305.2022.117. ПЗ КР

Рецензент, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Руководитель, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г. «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Нормоконтролёр, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Автор, студент группы ЭУ-\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Работа защищена с оценкой

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Челябинск 20\_\_

**АННОТАЦИЯ**

Тушин И.Е.

Программирование на языке Python (курс молодого бойца)

Челябинск: ЮУрГУ, ЭУ-142,

2019. 41 с., 1 таблица, 36 рисунков.

Библиографический список – 5 наим.

В данной работе рассматриваются основные методы сортировки, их особенности, принципы работы и их эффективность на практике. Помимо этого, рассматривается библиотека Numpy, а именно: что она из себя представляет и ее применение на практике.

**ОГЛАВЛЕНИЕ**

[**ВВЕДЕНИЕ** 4](#_Toc9448247)

[**ГЛАВА 1 МЕТОДЫ СОРТИРОВКИ** 5](#_Toc9448248)

[**1.1** **Сортировка включением** 5](#_Toc9448249)

[**1.2 Обменная сортировка** 6](#_Toc9448250)

[**1.3** **Сортировка выбором** 8](#_Toc9448251)

[**1.4** **Сравнение данных методов** 9](#_Toc9448252)

[**ГЛАВА 2 БИБЛИОТЕКА NumPy** 15](#_Toc9448253)

[**2.1 Суть и значение библиотек** 15](#_Toc9448254)

[**2.2 Библиотека NumPy и примеры решения задач** 15](#_Toc9448255)

[**ГЛАВА 3 Решение СЛАУ методом Гаусса** 38](#_Toc9448256)

[**3.1 Суть метода Гаусса и решение СЛАУ** 38](#_Toc9448257)

[**ВЫВОД** 41](#_Toc9448258)

[**СПИСОК ИСТОЧНИКОВ И ЛИТЕРАТУРЫ.** 42](#_Toc9448259)

# **ВВЕДЕНИЕ**

Целью данной работы является разобрать сущность ключевых методов сортировки, понять их основную задачу и принцип работы, найти среди них самый эффективный и удобный. Помимо этого, необходимо познакомиться с такой библиотекой, как “Numpy” и решить несколько задач для наглядной демонстрации ее возможностей.

Но перед тем как начать, необходимо понять, что такое методы сортировки и для чего они нужны. Метод сортировки - это метод, который используется для упорядочивания элементов в списке. Он работает следующим образом: у нас имеется определенная последовательность, которую метод анализирует и преобразует исходную последовательность в ту же, но уже ее значения будут представлены в порядке возрастания или убывания значений ключа.

Помимо этого, в программировании существуют различные библиотеки, которые упрощают нам жизнь при написании определенного кода. Например, для того, чтобы совершить определенное действие не обязательно писать какой-либо цикл, достаточно просто воспользоваться определенной функцией из библиотеки, которая сама все сделает.

# **ГЛАВА 1 МЕТОДЫ СОРТИРОВКИ**

* 1. **Сортировка включением**

Одним из наиболее простых и естественных методов внутренней сортировки является сортировка простыми включениями или как ее еще называют- вставками. Идея алгоритма очень просто и удобна. Его суть заключается в том, что на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Для более удобного понимания, давайте представим, что мы играем в “Дурака”. Когда мы вытягиваем карту из колоды, мы анализируем уже имеющиеся у нас карты и вставляем вытянутую карту в нужное место. То есть, если у меня есть карты номера 8 и 10, а я вытянул карту под номером 9, то я вставляю ее между 8 и 10.

Данный метод сортировки очень удобен, когда у нас уже имеется частично отсортированный массив, в которой не очень много элементов, например, меньше десяти. Если же элементов очень много, то данный метод не очень удобен, так как он будет очень долго выполнять свою работу.

Я решил на практике разобрать данный метод, а именно я написал код, который сортирует одномерный массив со значениями от 0 до 20. При этом я поставил счетчик, который будет показывать мне сколько элементов он сравнил друг с другом, а сколько переставил. В итоге у меня получился следующий результат (рисунок 1):

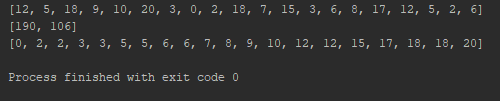


Рисунок 1- результат метода сортировки включением.

Из рисунка 1 мы видим, что данный метод сравнил одномерный массив, который состоит из 20 элементов и выстроил его значения от наименьшего к наибольшему. При этом, благодаря поставленному мною счетчику, мы видим, что количество сравнений равно 190, а количество перестановок 106. Сам метод сортировки можно посмотреть ниже.

**def** insert(arr, dim):

alg\_count = [0, 0]

**for** i **in** range(1, dim):

temp = arr[i]

j = i - 1

**while** j >= 0:

alg\_count[0] += 1

**if** arr[j] > temp:

alg\_count[1] += 1

arr[j + 1] = arr[j]

arr[j] = temp

j -= 1

**print**(alg\_count)

**import** random

arry = [random.randint(0, 20) **for** i **in** range(20)]

**print**(arry)

insert(arry, len(arry))

**print**(arry)

## **Обменная сортировка**

Вторая разобранная мною сортировка- это обменная, но чаще всего ее называют «методом пузырька». Данный метод, как и метод включением очень простой. Его суть заключается в том, что он последовательно сравнивает обменивает соседние элементы, если предшествующий оказывается больше последующего. Таким образом, в процессе выполнения данного алгоритма элементы с большим значением окажутся в конце массива, а элементы с наименьшими значениями наоборот, постепенно начнут перемещаться в начало массива. Если говорить проще, то чем тяжелее наш элемент, тем глубже он погружается под воду, а чем легче, тем он ближе находится к поверхности воды. При этом, в данном методе сортировки количество обработок данных внешнего цикла определяется длинной списка минус единица, ведь, когда второй элемент становится на свое место, то первый уже однозначно минимальный и, соответственно, находится на своем месте. А количество обработок данных внутреннего цикла полностью зависит от номера обработок внешнего цикла, потому что конец нашего массива уже отсортирован и выполнять проход по этим элементам уже не имеет смысла.

Я так же поработал с «методом пузырька» на практике, а именно: создал одномерный массив, состоявший из 20 элементов, со значениями от 0 до 20, и поставил счетчик, чтобы видеть, сколько сравнений и перестановок он сделал. Результат данного действия, можно увидеть на рисунке 2.



Рисунок 2 - результат выполнения алгоритма обмена. («метод пузырька»)

Если сравнить результат выполнения обменного метода и метода включением, то можно увидеть, что их счетчики показывают одинаковое количество перестановок, но абсолютно разное количество сравнений. А именно, количество сравнений у «метода пузырька» равна 19, а количество сравнений у метода включением равна 190, что в 10 раз больше. Но при этом, количество перестановок у них одинаковое, а если говорить точнее, то они переставили элементы ровно 106 раз. Из этого можно сделать вывод, что обменная сортировка справилась со своей задачей гораздо быстрее, чем сортировка вставками. Код обменной сортировки можно посмотреть ниже:

**def** bubble(arr, dim):

alg\_count = [0, 0]

**for** i **in** range(dim - 1):

alg\_count[0] += 1

**for** j **in** range(dim - i - 1):

**if** arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

alg\_count[1] += 1

**print**(alg\_count)

**import** random

arry = [random.randint(0, 20) **for** i **in** range(20)]

**print**(arry)

bubble(arry, len(arry))

**print**(arry)

* 1. **Сортировка выбором**

Третий метод, который я разобрал - это сортировка выбором. Суть данного алгоритма заключается в том, что он ищет в массиве минимальное значение, после чего обменивает это значение с первым элементом массива. То есть его алгоритм состоит из следующих действий:

1. Найти наименьший элемент в массиве.
2. Записать его в начало массива, а элемент, который там стоял перенести на место наименьшего.
3. Снова найти наименьший элемент в массиве, при этом не брать в расчет первый элемент.
4. Поставить данный элемент на второе место массива, а элемент, который там стоял, переместить на освободившееся место.
5. Выполнять такую последовательность до тех пор, пока массив не отсортируется.

Так же, как и с предыдущими методами сортировки, я применил его на практике все с тем же одномерным массивом, состоящий из 20 элементов. Результаты моей работы можно посмотреть на рисунке 3.

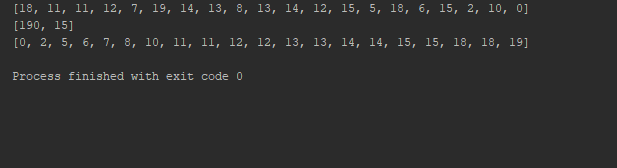


Рисунок 3- результат выполнения метода выбора.

Из рисунка 3 мы видим, что массив произвел равно столько же сравнений, что и метод включением, а именно: 190, но количество перестановок у него меньше чем у других – 15. Сам код алгоритма можно посмотреть ниже:

**def** select(arr, dim):

k = 0

alg\_count = [0, 0]

**for** k **in** range(0, dim - 1):

m = k

i = k + 1

**for** i **in** range(i, dim):

alg\_count[0] += 1

**if** arr[i] < arr[m]:

m = i

i += 1

**if** k != m:

t = arr[k]

arr[k] = arr[m]

arr[m] = t

alg\_count[1] += 1

k += 1

**print**(alg\_count)

**import** random

arry = [random.randint(0, 20) **for** i **in** range(20)]

**print**(arry)

select(arry, len(arry))

**print**(arry)

* 1. **Сравнение данных методов**

После того, как я разобрал каждый из представленных выше методов сортировки, то решил их сравнить друг с другом и выяснить, какой же из них наиболее эффективен. Для этого, я решил написать код, который будет сравнивать выше представленные алгоритмы и показывать, сколько они делают перестановок и сравнений. После чего, мне понадобилось в PyCharm создать три Python File, а именно: bubble\_2, insert\_2 и select\_2. В данных файлах я записал коды, которые будут сортировать мой массив.

Код из файла bubble\_2:

**def** bubble(arr, dim):

alg\_count = [0, 0]

**for** i **in** range(dim - 1):

**for** j **in** range(dim - i - 1):

alg\_count[0] += 1

**if** arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

alg\_count[1] += 1

**return** alg\_count

Код из файла insert\_2:

**def** insert(arr, dim):

alg\_count = [0, 0]

**for** i **in** range(1, dim):

temp = arr[i]

j = i - 1

**while** j >= 0:

alg\_count[0] += 1

**if** arr[j] > temp:

alg\_count[1] += 1

arr[j + 1] = arr[j]

arr[j] = temp

j -= 1

**return** alg\_count

Код из файла select\_2:

**def** select(arr, dim):

alg\_count = [0, 0]

**for** k **in** range(0, dim - 1):

m = k

i = k + 1

**for** i **in** range(i, dim):

alg\_count[0] += 1

**if** arr[i] < arr[m]:

m = i

**if** k != m:

t = arr[k]

arr[k] = arr[m]

arr[m] = t

alg\_count[1] += 1

**return** alg\_count

После того как были созданы данные файлы, я приступил к написанию самого кода, который будет сравнивать данные элементы на примере одномерного массива, состоящего из 65 элементов, после чего код будет показывать количество сравнений и перестановок через счетчик. При этом, массивы были как упорядоченные, так и обратно упорядоченные, и случайные. Сами же результаты я занес в сравнительную таблицу, которая представлена ниже:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Массив | Показатель | 65 элементов | | |
|  |  | Select | Insert | Bubble |
| Упорядоченный | С | 2080 | 2080 | 2080 |
| М | 0 | 0 | 0 |
| Обратно упорядоченные | С | 2080 | 2080 | 2080 |
| М | 32 | 2080 | 2080 |
| Случайные | С | 2080 | 2080 | 2080 |
| М | 59.884 | 1029.3305 | 1029.3305 |

Таблица 1- Результаты выполнения сравнений методов сортировки.

Исходя из данной таблицы, можно смело сделать вывод о том, что наиболее эффективным мотетом сортировки является “Select”, потому что в упорядоченном, обратно упорядоченном и случайном массиве он показал наименьшее количество перестановок (показатель M в сравнительной таблице эффективности). Сам алгоритм сравнения методов:

**import** random

**import** select\_2

**import** bubble\_2

**import** insert\_2

DIM = 65

bubble\_arr = []

insert\_arr = []

select\_arr = []

CTotal = [0, 0, 0]

MTotal = [0, 0, 0]

**for** i **in** range(1, DIM+1):

bubble\_arr.append(i)

insert\_arr.append(i)

select\_arr.append(i)

file = open("Result.txt", "w")

**print**("Упорядоченный массив: ")

**print**(bubble\_arr)

count = select\_2.select(select\_arr, DIM)

**print**("Упорядоченный массив: результат")

**print**(select\_arr)

CTotal[0] = count[0]

MTotal[0] = count[1]

count = insert\_2.insert(insert\_arr, DIM)

CTotal[1] = count[0]

MTotal[1] = count[1]

count = bubble\_2.bubble(bubble\_arr, DIM)

CTotal[2] = count[0]

MTotal[2] = count[1]

**print**("Размер массива:", DIM)

**print**("Сравнений:", CTotal[0], CTotal[1], CTotal[2])

**print**("Перестановок:", MTotal[0], MTotal[1], MTotal[2])

file.write("Упорядоченный массив:\n ")

file.write("Размер массива: " + str(DIM) + "\n")

file.write("Сравнений: " + str(CTotal[0]) + " " + str(CTotal[1]) + " " + str(CTotal[2]) + "\n")

file.write("Перестановок: " + str(MTotal[0]) + " " + str(MTotal[1]) + " " + str(MTotal[2]) + "\n")

select\_arr.clear()

bubble\_arr.clear()

insert\_arr.clear()

**for** i **in** range(DIM, 0, -1):

bubble\_arr.append(i)

insert\_arr.append(i)

select\_arr.append(i)

**print**("Обратно упорядоченный массив: ")

**print**(bubble\_arr)

count = select\_2.select(select\_arr, DIM)

**print**("Обратно упорядоченный массив: результат")

**print**(select\_arr)

CTotal[0] = count[0]

MTotal[0] = count[1]

count = insert\_2.insert(insert\_arr, DIM)

CTotal[1] = count[0]

MTotal[1] = count[1]

count = bubble\_2.bubble(bubble\_arr, DIM)

CTotal[2] = count[0]

MTotal[2] = count[1]

**print**("Размер массива:", DIM)

**print**("Сравнений:", CTotal[0], CTotal[1], CTotal[2])

**print**("Перестановок:", MTotal[0], MTotal[1], MTotal[2])

file.write("Обратно упорядоченный массив:\n ")

file.write("Размер массива: " + str(DIM) + "\n")

file.write("Сравнений: " + str(CTotal[0]) + " " + str(CTotal[1]) + " " + str(CTotal[2]) + "\n")

file.write("Перестановок: " + str(MTotal[0]) + " " + str(MTotal[1]) + " " + str(MTotal[2]) + "\n")

KOL = 2000

CTotal = [0, 0, 0]

MTotal = [0, 0, 0]

**for** n **in** range(0, KOL):

select\_arr.clear()

insert\_arr.clear()

bubble\_arr.clear()

select\_arr = [random.randint(0, 100) **for** i **in** range(DIM)]

**for** i **in** range(0, DIM):

bubble\_arr.append(select\_arr[i])

insert\_arr.append(select\_arr[i])

count = select\_2.select(select\_arr, DIM)

CTotal[0] += count[0]

MTotal[0] += count[1]

count = insert\_2.insert(insert\_arr, DIM)

CTotal[1] += count[0]

MTotal[1] += count[1]

count = bubble\_2.bubble(bubble\_arr, DIM)

CTotal[2] += count[0]

MTotal[2] += count[1]

**print**("\nСлучайный массив:")

**print**("Проведено экспериментов:", KOL)

**print**("Размер массива:", DIM)

**print**("Сравнений:", CTotal[0]/KOL, CTotal[1]/KOL, CTotal[2]/KOL)

**print**("Перестановок:", MTotal[0]/KOL, MTotal[1]/KOL, MTotal[2]/KOL)

file.write("\nСлучайный массив:\n")

file.write("Проведено экспериментов: " + str(KOL) + "\n")

file.write("Размер массива: " + str(DIM) + "\n")

file.write("Сравнений: " + str(CTotal[0]/KOL) + " " + str(CTotal[1]/KOL) + " " + str(CTotal[2]/KOL) + "\n")

file.write("Перестановок: " + str(MTotal[0]/KOL) + " " + str(MTotal[1]/KOL) + " " + str(MTotal[2]/KOL))

file.close()

## **ГЛАВА 2 БИБЛИОТЕКА NumPy**

## **2.1 Суть и значение библиотек**

Библиотеки в программировании- это сборники подпрограмм или объектов, которые используются для разработки программного обеспечения или решения близких по тематике задач. То есть, библиотеки представляют собой собрание функций, в которых хранятся названия каждой функции, объектный код самой это функции и информация, касающаяся “перемещаемости файла”, необходимая для редактирования связей. Когда программа делает ссылку на функцию, содержащаяся в библиотеке, то компоновщик отыскивает эту функцию и добавляет ее код к программе. Таким образом, к программе добавляются только те функции, которые будут использоваться в ней. Помимо этого, хочется сказать о том, что функции, содержащиеся в библиотеках, очень сильно облегчают жизнь при написании программ. Ведь программистам не нужно будет писать целый цикл для решения определенного вопроса, потому что они могут просто вызвать определенную функцию, которая этот вопрос решит. Например, мне необходимо посчитать элементы в массиве. Для этого мне не нужно считать отдельно элементы, ведь я могу воспользоваться функцией “sum”, которая все сделает за меня.

## **2.2 Библиотека NumPy и примеры решения задач**

В процессе написания своей курсовой работы, я познакомился с такой интересной библиотекой, как NumPy. NumPy - это библиотека языка Python, которая предоставляет общие математические и числовые операции в виде пре-скомпилированных, быстрых функций. Они объединяются в высокоуровневые пакеты, которые обеспечивают функционал, сравнимый с функционалом MatLab. То есть, NumPy предоставляет базовые методы для манипуляции с большими массивами и матрицами.

Для того, чтобы понять, как работает данная библиотека, я решил несколько задач с помощью NumPy. Сами задачи, вместе с условиями и решениями, можно посмотреть ниже:

№1. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Найти наибольший элемент столбца матрицы A, для которого сумма абсолютных значений элементов максимальна.

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

**import** random

N= random.randint (1,10)

M= random.randint (1,10)

**print**(N,M)

A=np.random.randint(0,100, (N,M))

**print**(A)

mx=0

indmx=0

**for** i **in** range (M):

**if** np.sum(A[:, i])>mx:

mx = np.sum(A[:, i])

indmx=i

**print**("Столбец с максимальной суммой элементов-"+ str(A[: , indmx]))

**print**("Сумма этого столбца-"+str(mx)+ ","+ "номер этого столбца-"+ str(indmx+1))

stmax=0

**for** b **in** A [: , indmx]:

**if** b> stmax:

stmax=b

**print**("Максимальный элемент этого столбца-"+str(stmax))

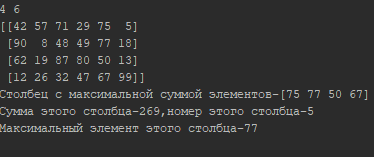


Рисунок 4- результат задачи №1.

№2. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Найти наибольшее значение среди средних значений для каждой строки матрицы.

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

**import** random

N= random.randint (1,10)

M= random.randint (1,10)

**print**(N,M)

A=np.random.randint(0,10, (N,M))

**print**(A)

mx=0

aver=0

**for** i **in** range (N):

**if** np.mean (A[i , :])> mx:

mx= np.mean(A[i , :])

aver=i

**print**("Строка с максимальным средним значением элементов-"+ str(A[aver , :]))

**print**("Среднее значение этой строки-"+str(mx)+ ","+ "номер этой строки-"+ str(aver+1))

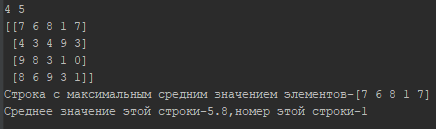


Рисунок 5- результат задачи №2.

№3. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Найти наименьший элемент столбца матрицы A, для которого сумма абсолютных значений элементов максимальна.

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

**import** random

N= random.randint (1,10)

M= random.randint (1,10)

**print**(N,M)

A=np.random.randint(0,10, (N,M))

**print**(A)

mx=0

indmx=0

**for** i **in** range (M):

**if** np.sum(A[:, i])>mx:

mx = np.sum(A[:, i])

indmx=i

**print**("Столбец с максимальной суммой элементов-"+ str(A[: , indmx]))

**print**("Сумма этого столбца-"+str(mx)+ ","+ "номер этого столбца-"+ str(indmx+1))

stmin=10

**for** b **in** A [: , indmx]:

**if** b< stmin:

stmin=b

**print**("Минимальный элемент этого столбца-"+str(stmin))

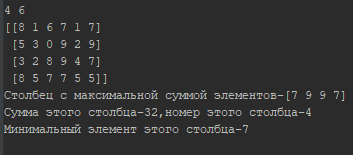


Рисунок 6- результат задачи №3.

№4. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Найти наименьшее значение среди средних значений для каждой строки матрицы.

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

**import** random

N= random.randint (1,10)

M= random.randint (1,10)

**print**(N,M)

A=np.random.randint(0,10, (N,M))

**print**(A)

mx=10

aver=0

**for** i **in** range (N):

**if** np.mean (A[i , :])< mx:

mx= np.mean(A[i , :])

aver=i

**print**("Строка с минимальным средним значением элементов-"+ str(A[aver , :]))

**print**("Среднее значение этой строки-"+str(mx)+ ","+ "номер этой строки-"+ str(aver+1))

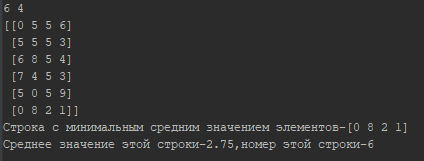


Рисунок 7- результат задачи №4.

№5. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Определить средние значения по всем строкам и столбцам матрицы. Результат оформить в виде матрицы из N + 1 строк и M + 1 столбцов.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

**print**(N,M)

A=np.random.randint(0,10, (N,M))

**print**(A)

M\_mean = A.mean(axis=0)

N\_mean = A.mean(axis=1)

**print**("Матрица со средними значениями-",A)

N\_mean = np.append(N\_mean, None)

A = np.vstack((A, M\_mean))

A = np.hstack((A, N\_mean.reshape(-1,1)))

**print**("Матрица с N и M +1-",A)

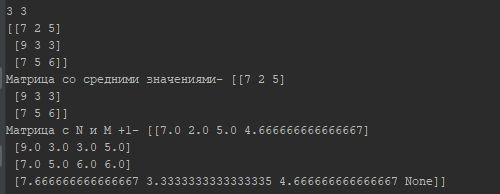


Рисунок 8- результат задачи №5.

№6. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Найти сумму элементов всей матрицы. Определить, какую долю в этой сумме составляет сумма элементов каждого столбца. Результат оформить в виде матрицы из N + 1 строк и M столбцов.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

**print**(N,M)

A=np.random.randint(0,10, (N,M))

**print**(A)

B=np.sum(A)

**print** ("Сумаа элементов матрицы A-", B)

M\_sum = np.sum(A, axis=0)/np.sum(A)

**print**("�"оля элементов каждого столбца-"+str(np.sum(A, axis=0)/np.sum(A))***)***

A = np.vstack((A,M\_sum)***)***

print("Матрица с N +1-",A***)***

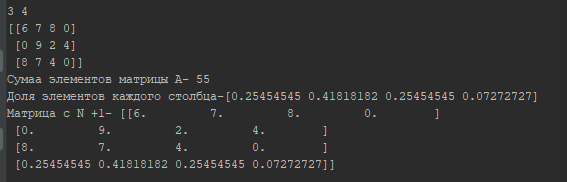


Рисунок 9- результат задачи №6.

№7. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Найти сумму элементов всей матрицы. Определить, какую долю в этой сумме составляет сумма элементов каждой строки. Результат оформить в виде матрицы из N строк и M+1 столбцов.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

**print**(N,M)

A=np.random.randint(0,10, (N,M))

**print**(A)

B=np.sum(A)

**print** ("Сумаа элементов матрицы A-", B)

N\_sum = np.sum(A, axis=1)/np.sum(A)

**print**("�"оля элементов каждой строки-"+str(np.sum(A, axis=1)/np.sum(A))***)***

A = np.hstack((A, N\_sum.reshape(-1,1))***)***

print("Матрица с M +1-",A***)***

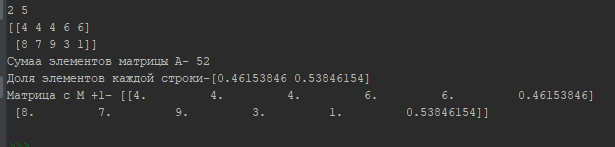


Рисунок 10- результат задачи №7.

№8. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Определить, сколько отрицательных элементов содержится в каждом столбце и в каждой строке матрицы. Результат оформить в виде матрицы из N + 1 строк и M + 1 столбцов.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

**print**(N,M)

A=np.random.randint(-10,10, (N,M))

**print**(A)

M\_sum = (A < 0).sum(axis=0)

N\_sum = (A < 0).sum(axis=1)

**print**("Кол-во отрицательных элементов в матрице-"+str(M\_sum ), str(N\_sum))

N\_sum = np.append(N\_sum, None)

A = np.vstack((A, M\_sum))

A = np.hstack((A, N\_sum.reshape(-1,1)))

**print**("Матрица с N и M +1-",A)

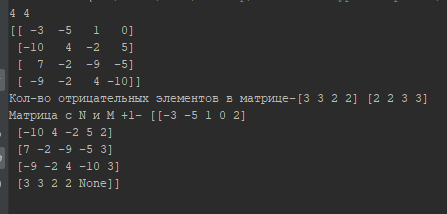


Рисунок 11- результат задачи №8.

№9. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Определить, сколько нулевых элементов содержится в верхних L строках матрицы и в левых К столбцах матрицы.

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

N = np.random.randint(2,10)

M = np.random.randint(2,10)

L = np.random.randint(1,N)

K = np.random.randint(1,M)

**print**(N,M,L,K)

A=np.random.randint(-10,10, (N,M))

New\_A = A[:L, :K]

**print**(A)

**print**(np.sum(New\_A == 0))

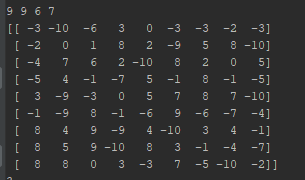


Рисунок 12- результат задачи №9.

№10. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Перемножить элементы каждого столбца матрицы с соответствующими элементами K-го столбца.

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

**import** random

N = random.randint(2, 5)

M = random.randint(2, 5)

**print**(N, M)

A = np.random.randint(-50, 50, (N, M))

**print**(str(A) + "\n")

K = random.randint(0, M)

**print**("K = " + str(K + 1))

K\_m = A[:, K].flat

**for** i **in** range(M):

**if** i != K:

**for** k **in** range(N):

A[k, i] = A[k, i] \* K\_m[k]

**print**(A)

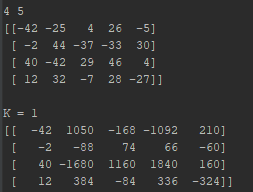


Рисунок 13- результат задачи №10

№11. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Просуммировать элементы каждой строки матрицы с соответствующими элементами L-й строки.

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

**import** random

N = random.randint(2, 5)

M = random.randint(2, 5)

**print**(N, M)

A = np.random.randint(-50, 50, (N, M))

**print**(str(A) + "\n")

L = random.randint(0, N)

**print**("L = " + str(L + 1))

L\_m = A[L, :].flat

**for** i **in** range(N):

**if** i != L:

**for** k **in** range(M):

A[i, k] = A[i, k] + L\_m[k]

**print**(A)

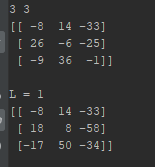


Рисунок 14- результат задачи №11.

№12. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Разделить элементы каждой строки на элемент этой строки с наибольшим значением.

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

**import** random

N = random.randint(2, 5)

M = random.randint(2, 5)

**print**(N, M)

A = np.random.randint(-50, 50, (N,M))

A = np.array(A, float)

**print**(str(A) + "\n")

**for** i **in** range(N):

mx = np.max(A[i, :])

**for** k **in** range(M):

**if** A[i, k] != mx:

**print**(A[i, k] / mx)

A[i, k] = round(A[i, k] / mx, 2)

**print**(A)

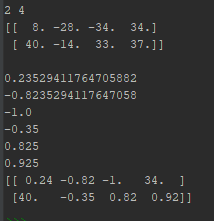


Рисунок 15- результат задачи №12.

№13. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Разделить элементы каждого столбца матрицы на элемент этого столбца с наибольшим значением.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

**print**(N,M)

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

col\_max = np.max(matrix, axis=0)

matrix = matrix / col\_max

**print**("Полученная матрица:\r\n {}".format(matrix))

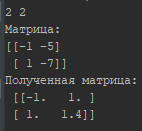


Рисунок 16- результат задачи №13.

№14. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Разделить элементы матрицы на элемент матрицы с наибольшим значением.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

**print**(N,M)

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

max\_el = np.max(matrix)

matrix = matrix / max\_el

**print**("Полученная матрица:\r\n {}".format(matrix))

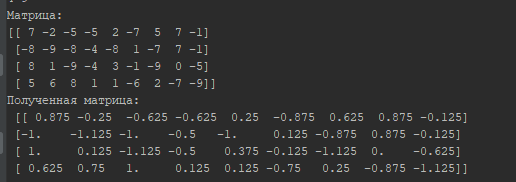


Рисунок 17- результат задачи №14.

№15. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Все элементы имеют целый тип. Дано целое число H. Определить, какие столбцы имеют хотя бы одно такое число, а какие не имеют.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

H = 2

**print**(N,M)

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

matrix\_bool = matrix == H

row\_sum = np.sum(matrix\_bool, axis=0)

**print**("Столбцы в которых встречается значение {}:".format(H))

**print**(np.argwhere(row\_sum).flatten())

**print**("Столбцы в которых нет значения {}:".format(H))

**print**(np.argwhere(row\_sum == 0).flatten())

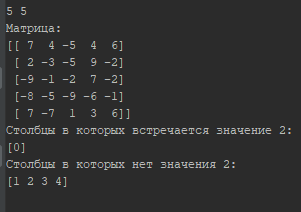


Рисунок 18- результат задачи №15.

№16. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Исключить из матрицы строку с номером L. Сомкнуть строки матрицы

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

N = 4

M = 5

L = 2

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

matrix = np.delete(matrix, L, axis=0)

**print**("Полученная матрица:\r\n {}".format(matrix))

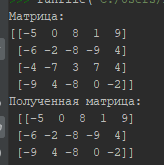


Рисунок 19- результат задачи №16.

№17. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Добавить к матрице строку и вставить ее под номером L.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

L = 1

**print**(N,M)

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

row = np.random.randint(low=-9, high=10, size=M)

**print**("Строка для вставки: {}".format(row))

matrix = np.insert(matrix, L, row, axis=0)

**print**("Полученная матрица:\r\n {}".format(matrix))

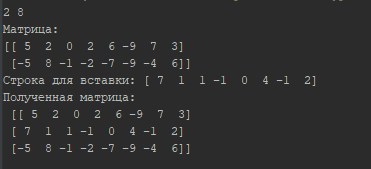


Рисунок 20- результат задачи №17.

№18. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Найти сумму элементов, стоящих на главной диагонали, и сумму элементов, стоящих на побочной диагонали (элементы главной диагонали имеют индексы от [0,0] до [N,N], а элементы побочной диагонали — от [N,0] до [0,N]).

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

**print**(N,M)

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

diagonal\_main = np.diagonal(matrix)

**print**("Элементы главной диагонали:\r\n{}".format(diagonal\_main))

sum\_diagonal\_main = np.sum(diagonal\_main)

**print**("Cумма элементов главной диагонали:\r\n{}".format(sum\_diagonal\_main))

diagonal\_side = np.diagonal(matrix[::-1])

**print**("Элементы побочной диагонали:\r\n{}".format(diagonal\_side))

sum\_diagonal\_side = np.sum(diagonal\_side)

**print**("сумму элементов побочной диагонали:\r\n{}".format(sum\_diagonal\_side))

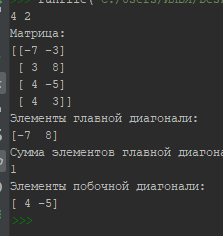


Рисунок 21- результат задачи №18.

№19. Создать квадратную матрицу A, имеющую N строк и N столбцов со случайными элементами. Определить сумму элементов, расположенных параллельно главной диагонали (ближайшие к главной). Элементы главной диагонали имеют индексы от [0,0] до [N,N].

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

N = 4

A = np.random.randint(low=-9, high=10, size=(N, N))

**print**("Матрица:\r\n{}".format(A))

diagonal\_elements = np.array([np.diagonal(A, i) **for** i **in** [1, -1]]).flatten()

**print**("Элементы, расположенные параллельно главной диагонали:")

**print**(diagonal\_elements)

**print**("Сумма элементов, расположенные параллельно главной диагонали:")

**print**(np.sum(diagonal\_elements))

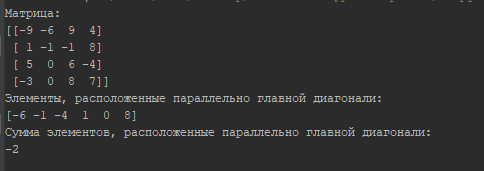


Рисунок 22- результат задачи №19.

№20. Создать квадратную матрицу A, имеющую N строк и N столбцов со случайными элементами. Определить произведение элементов, расположенных параллельно побочной диагонали (ближайшие к побочной). Элементы побочной диагонали имеют индексы от [N,0] до [0,N].

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

N = 4

A = np.random.randint(low=-9, high=10, size=(N, N))

**print**("Матрица:\r\n{}".format(A))

diagonal\_elements = np.array([np.diagonal(A[::-1], i) **for** i **in** [1, -1]]).flatten()

**print**("Элементы, расположенные параллельно побочной диагонали:")

**print**(diagonal\_elements)

**print**("Сумма элементов, расположенные параллельно побочной диагонали:")

**print**(np.prod(diagonal\_elements))

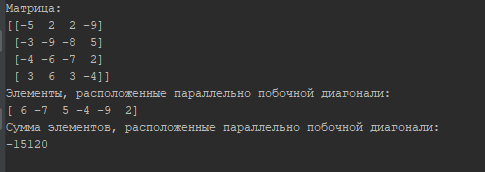


Рисунок 23- результат задачи №20.

№21. Создать квадратную матрицу A, имеющую N строк и N столбцов со случайными элементами. Каждой паре элементов, симметричных относительно главной диагонали (ближайшие к главной), присвоить значения, равные полусумме этих симметричных значений (элементы главной диагонали имеют индексы от [0,0] до [N,N]).

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

N = 4

A = np.random.randint(low=-9, high=10, size=(N, N)).astype(float)

**print**("Матрица:\r\n{}".format(A))

diagonal\_elements = [np.diagonal(A, i) **for** i **in** [1, -1]]

**print**("Элементы, расположенные параллельно главной диагонали:")

**print**(diagonal\_elements)

values = (diagonal\_elements[0] + diagonal\_elements[1])/2

**print**(values)

rng = np.arange(N-1)

A[rng, rng+1] = values[rng]

A[rng+1, rng] = values[rng]

**print**("Полученная матрица:\r\n{}".format(A))

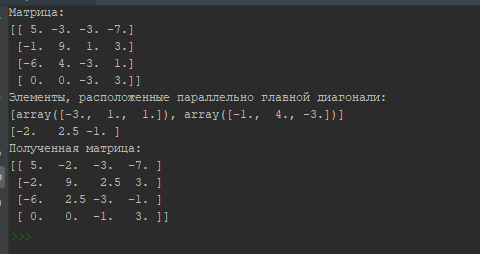


Рисунок 24- результат задачи №21.

№22. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Исходная матрица состоит из нулей и единиц. Добавить к матрице еще один столбец, каждый элемент которого делает количество единиц в каждой строке чётным.

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

**import** random

N = np.random.randint(2,10)

M = np.random.randint(2,10)

**print**(N,M)

A = np.random.randint(low=0, high=2, size=(N, M))

**print**("Матрица:\r\n{}".format(A))

col = [i % 2 **for** i **in** np.sum(A, axis=1)]

A = np.insert(A, M, col, axis=1)

**print**("Полученная матрица:\r\n {}".format(A))

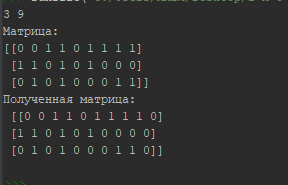


Рисунок 25- результат задачи №22.

№23. Создать квадратную матрицу A, имеющую N строк и N столбцов со случайными элементами. Найти сумму элементов, расположенных выше главной диагонали, и произведение элементов, расположенных выше побочной диагонали (элементы главной диагонали имеют индексы от [0,0] до [N,N], а элементы побочной диагонали — от [N,0] до [0,N]).

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

N = 4

A = np.random.randint(low=-9, high=10, size=(N, N))

**print**("Матрица:\r\n{}".format(A))

elements = np.diagonal(A, 1)

**print**("Элементы, расположенные выше главной диагонали:")

**print**(elements)

**print**("Сумма элементов, расположенных выше главной диагонали:")

**print**(np.sum(elements))

elements = np.diagonal(A[::-1], -1)

**print**("Элементы, расположенные выше побочной диагонали:")

**print**(elements)

**print**("Произведение элементов, расположенных выше побочной диагонали:")

**print**(np.prod(elements))

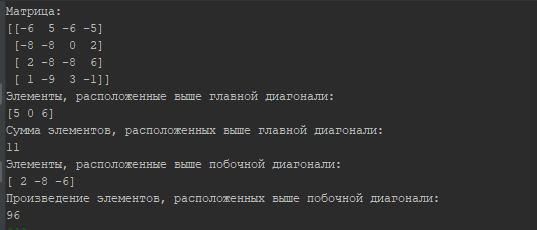


Рисунок 26- результат задачи №23.

№24. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Дан номер строки L и номер столбца K, при помощи которых исходная матрица разбивается на четыре части. Найти сумму элементов каждой части.

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

N = 4

M = 5

L = 2

K = 2

A = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(A))

parts = [

A[:L, :K],

A[:L, K:],

A[L:, :K],

A[L:, K:],

]

**for** i **in** range(len(parts)):

**print**("Cумма элементов {} части: {}".format(i+1, np.sum(parts[i])))

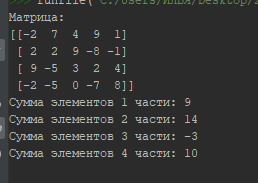


Рисунок 27- результат задачи №24.

№25. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Определить, сколько нулевых элементов содержится в каждом столбце и в каждой строке матрицы. Результат оформить в виде матрицы из N + 1 строк и M + 1 столбцов.

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

N = 4

M = 5

L = 2

K = 2

A = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(A))

matrix\_bool = A == 0

col = np.sum(matrix\_bool, axis=1)

A = np.insert(A, M, col, axis=1)

row = np.append(np.sum(matrix\_bool, axis=0), 0)

A = np.insert(A, N, row, axis=0)

**print**("Полученная матрица:\r\n {}".format(A))

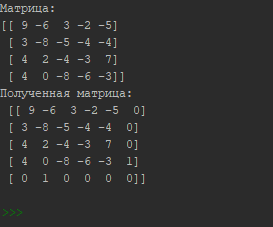


Рисунок 28- результат задачи №25.

№26. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Дан номер строки L и номер столбца K, при помощи которых исходная матрица разбивается на четыре части. Найти среднее арифметическое элементов каждой части.

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

N = 4

M = 5

L = 2

K = 2

A = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(A))

parts = [

A[:L, :K],

A[:L, K:],

A[L:, :K],

A[L:, K:],

]

**for** i **in** range(len(parts)):

**print**("Cреднее арифметическое {} части: {}".format(i+1, np.average(parts[i])))

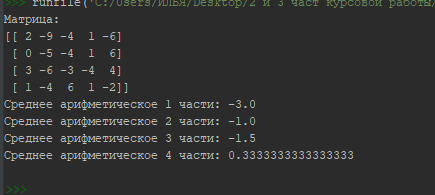


Рисунок 29- результат задачи №26.

№27. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Все элементы имеют целый тип. Дано целое число H. Определить, какие строки имеют хотя бы одно такое число, а какие не имеют.

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

N = 6

M = 7

H = 2

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

matrix\_bool = matrix == H

col\_sum = np.sum(matrix\_bool, axis=1)

**print**("Строки в которых встречается значение {}:".format(H))

**print**(np.argwhere(col\_sum).flatten())

**print**("Строки в которых нет значения {}:".format(H))

**print**(np.argwhere(col\_sum == 0).flatten())

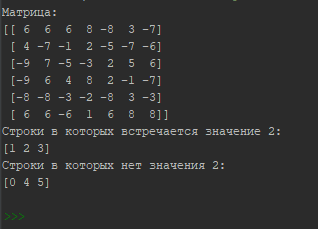


Рисунок 30- результат задачи №27.

№28. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Исключить из матрицы столбец с номером K. Сомкнуть столбцы матрицы.

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

N = 4

M = 5

K = 2

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

matrix = np.delete(matrix, K, axis=1)

**print**("Полученная матрица:\r\n {}".format(matrix))

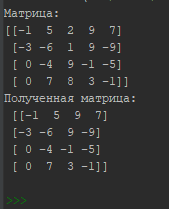


Рисунок 31- результат задачи №28.

№29. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Добавить к матрице столбец чисел и вставить его под номером K.

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

**import** random

N= random.randint (1,10)

M= random.randint (1,10)

K=1

**print**(N,M)

matrix = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(matrix))

col = np.random.randint(low=-9, high=10, size=N)

**print**("Столбец для вставки: {}".format(col))

matrix = np.insert(matrix, K, col, axis=1)

**print**("Полученная матрица:\r\n {}".format(matrix))

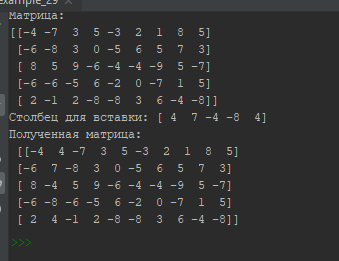


Рисунок 32- результат задачи №29.

№30. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Добавить к элементам каждого столбца такой новый элемент, чтобы сумма положительных элементов стала бы равна модулю суммы отрицательных элементов. Результат оформить в виде матрицы из N + 1 строк и M столбцов.

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

**import** random

N= random.randint (1,10)

M= random.randint (1,10)

**print**(N,M)

A = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(A))

row = np.sum(A, axis=0) \* -1

A = np.insert(A, N, row, axis=0)

**print**("Полученная матрица:\r\n {}".format(A))

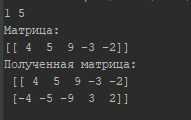


Рисунок 33- результат задачи №30.

№31. Создать прямоугольную матрицу A, имеющую N строк и M столбцов со случайными элементами. Добавить к элементам каждой строки такой новый элемент, чтобы сумма положительных элементов стала бы равна модулю суммы отрицательных элементов. Результат оформить в виде матрицы из N строк и M + 1 столбцов.

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

**import** random

N= random.randint (1,10)

M= random.randint (1,10)

**print**(N,M)

A = np.random.randint(low=-9, high=10, size=(N, M))

**print**("Матрица:\r\n{}".format(A))

col = np.sum(A, axis=1) \* -1

A = np.insert(A, M, col, axis=1)

**print**("Полученная матрица:\r\n {}".format(A))

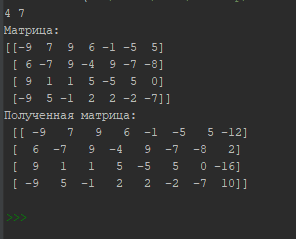


Рисунок 34- результат задачи №31.

Исходя из проделанных мною задач, я могу сделать вывод о том, что библиотека Numpy отлично подходит для работы с математическими и числовыми операциями, применяемые к большим матрицам и массивам. То есть, данную библиотеку можно рассматривать, как хорошую альтернативу MatLab.

# **ГЛАВА 3 Решение СЛАУ методом Гаусса**

## **3.1 Суть метода Гаусса и решение СЛАУ**

Метод Гаусса был предложен известнейшим немецким математиком Карлом Фридрихом Гауссом, который является одним из наиболее универсальным методов решения СЛАУ. Суть данного метода заключается в том, что расширенную матрицу приводят к ступенчатому виду, то есть по главной диагонали получают нули, после чего находятся уже все неизвестные, начиная снизу-вверх. Метод Гаусса идеально подходит для решения систем, которые содержат в себе больше трех линейных уравнений, а также для решения систем уравнений, которые не являются квадратными. То есть, данный метод наиболее универсальный для нахождения любой системы линейных уравнений. Он даже работает в случае того, что система имеет бесконечно много решений или она несовместна.

Разобравшись, что представляет из себя метод Гаусса, я решил СЛАУ методом Гаусса с помощью пакета Numpy. Всего у меня было пять матриц, которые я записал в таблице Excel, после чего сохранил ее на рабочий стол в формате CSV. Что же такое CSV? Это текстовый файл, в котором содержится определенная информация, в моем случае это матрицы, а сами поля разделяются специальными символами, или как их еще называют – разделителями. Матрицы, которые я записал в Excel, можно увидеть на рисунке 35.

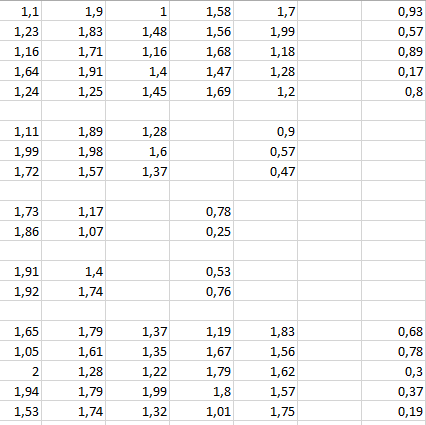


Рисунок 35 – записанные в Excel матрицы.

После этого, я перенес данный файл в PyCharm, переименовал его в “numpy-gauss” и, с помощью команды “CTRL+R”, заменил разделитель «;» на «,». Далее я начал работать над кодом решения СЛАУ методом Гаусса. Сам код, который у меня получился в итоге, находится ниже:

**import** numpy

**def** FancyPrint(A, B, selected):

**for** row **in** range(len(B)):

**print**("(", end='')

**for** col **in** range(len(A[row])):

**print**("\t{1:10.2f}{0}".format(" " **if** (selected **is** None **or** selected != (row, col)) **else** "\*", A[row][col]), end='')

**print**("\t) \* (\tX{0}) = (\t{1:10.2f})".format(row + 1,

B[row]))

data = numpy.genfromtxt('./numpy-gauss.csv', delimiter=';')

matrix\_list = []

matrix = []

**for** row **in** data:

first\_col = row[0]

**if** numpy.isnan(first\_col):

matrix\_list.append(matrix)

matrix = []

**continue**

mask = ~numpy.isnan(row)

matrix.append(row[mask])

matrix\_list.append(matrix)

f = open('numpy-gauss-slv.csv', 'wb+')

f.truncate()

**for** matrix **in** matrix\_list:

M = numpy.array(matrix)

myA = numpy.delete(M, M.shape[1] - 1, axis=1)

myB = M[:, [-1]].flatten()

**print**("�сходная система:")

FancyPrint(myA, myB, None)

slv = numpy.linalg.solve(myA, myB)

**print**("Решаем:")

**print**(slv)

numpy.savetxt(f, numpy.array([slv]), delimiter=',')

f.close()

Суть данного кода заключается в том, что он берет записанные мною матрицы из файла “numpy-gauss” и создает две переменные, а именно- “matrix” и “matrix-list”. После этого, он выполняет цикл перебора данных из “data” по строчке. Он берет первый элемент из этой строчки- “first\_col” и через инструкцию “if” проверяет следующее: если “first\_col” - это nan, то выполняется условие, в ходе которого в “matrix-list” добавляем “matrix” и затираем его, после чего условие повторяется. Если же условие у нас не сработало, то есть там лежит число, не относящееся к nan, то тогда определяем “mask” и затираем в строчке “matrix= []”. “Mask” - это те места в строчке, где нету nan. После того, как был определен “mask”, мы выбираем “row[mask]”, то есть берем только те элементы, где нету nan и добавляем их в “matrix”, после чего уже добавляем “matrix” в “matrix-list”. Далее, закончив данный цикл, создается новый CSV файл с помощью команды “f = open (‘numpy-gauss-slv.csv’, ‘wb+’)”, куда будут уже записываться ответы на решенные матрицы. После чего, благодаря циклу, записываются сами матрицы и происходит их решение. Ответы из файла “numpy-gauss-slv”, которые были получены в процессе решения можно посмотреть на рисунке 36.

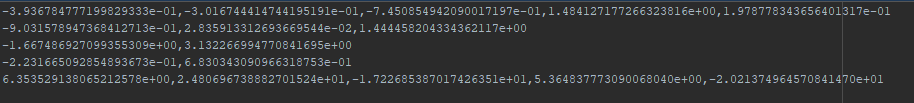


Рисунок 36- результат решения СЛАУ методом Гаусса.

# **ВЫВОД**

Какой же вывод можно сделать, исходя из того, что мы разобрали в данной работе? Всего существует три основных метода сортировки: обменная, выбором и включением. Однако, самым эффективным их них является “Select”, потому что в упорядоченном, обратно упорядоченном и случайном массиве он показал наименьшее количество перестановок. Что касается библиотеки NumPy, можно смело сказать, что она отлично подойдет для работы с математическими и числовыми операциями для больших массивов и матрица. Эту библиотеку можно даже применять для решений СЛАУ методом Гаусса. То есть, Numpy- это отличная библиотека вычислительных алгоритмов и, при этом, хорошая альтернатива MatLab.

# **СПИСОК ИСТОЧНИКОВ И ЛИТЕРАТУРЫ.**

1. NumPy в Python (Habr).  
   <https://habr.com/ru/post/352678/>
2. Краткое руководство по NumPy: “Учусь NumPy”.
3. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. − М: Изд-во «Мир», 1989. − 360 с.;
4. Книга Дж.Вандера Пласа: “Python для сложных задач: наука о данных и машинное обучение”.
5. Интернет статья на тему: базовые операции над массивами.  
   <https://pythonworld.ru/numpy/2.html>