



Урок 3

Массивы. Кортежи. Множества. Списки

Понятие массива, кортежа, множества и списков. Обработка последовательностей, одномерных и двумерных массивов. Работа с ассоциативными массивами (таблицами данных). Двоичный (бинарный) поиск элемента в массиве.

[Введение](#)

[Структуры хранения данных](#)

[Понятие массива, кортежа, множества и списков](#)

[Список](#)

[Кортеж](#)

[Множества](#)

[Алгоритмы обработки последовательностей, одномерных и двумерных массивов \(списков\)](#)

[Разложить положительные и отрицательные числа по разным массивам](#)

[Вставка элемента в произвольное место массива](#)

[Реверс массива](#)

[Вывести неповторяющиеся элементы массива](#)

[Суммы строк и столбцов матрицы](#)

[Обмен значений главной и побочной диагоналей квадратной матрицы](#)

[Запись в матрицу результатов побитовых операций](#)

[Алгоритм решения задачи:](#)

[Работа с ассоциативными массивами \(таблицами данных\)](#)

[Двоичный \(бинарный\) поиск элемента в массиве.](#)

[Алгоритм поиска](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Введение

В предыдущих уроках мы изучили основы алгоритмизации, рассмотрели задачи линейных, разветвляющихся и циклических алгоритмов, познакомились с понятием рекурсии и особенностями ее применения.

На этом уроке вы освоите основные классические алгоритмы работы с одномерными и двумерными массивами и другими структурами данных.

В повседневной жизни мы часто сталкиваемся с большими множествами однородных объектов. Люди живут в многомиллионных городах, где поведение сообщества качественно отличается от поведения конкретного индивида. Мы сталкиваемся с ситуацией, когда совокупность объектов обретает новые качества, совершенно не присущие ее отдельным объектам. Программирование – одна из сфер, где большое внимание уделяется способам организации различных множеств объектов и методам работы с ними.

Структуры хранения данных

По сути своей данные – это абстракции реальных объектов, и формулируются они как абстрактные структуры. В процессе разработки той или иной программы представление данных постепенно уточняется вслед за уточнением алгоритма. Они все более подчиняются ограничениям, которые складываются в конкретных системах программирования. Поэтому стоит определить фундаментальные структуры данных, на основе которых были созданы все остальные.

К фундаментальным структурам данных относят:

- массивы (структура с фиксированной длиной);
- множества (динамическая структура).

Краеугольный камень в теории структур данных – это различие между фундаментальными и усложненными структурами. Фундаментальные структуры – это «молекулы», компоненты, из которых состоят усложненные.

В Python все структуры данных уже высокоуровневые – это не классические массивы с фиксированной длиной. Поэтому представленные ниже алгоритмы вы изучите для понимания всего процесса работы с различными структурами данных.

Понятие массива, кортежа, множества и списков

Массив – это поименованная последовательность однотипных данных, имеющая фиксированную длину, пронумерованная от 0.

Список – поименованная последовательность данных (возможно, разного типа), имеющая динамическую длину.

Кортежи (tuple) – это неизменная последовательность объектов разного типа. Они обычно записываются в круглых скобках. Если неоднозначности не возникает, то скобки можно опустить.

Множества – это структура данных, позволяющая хранить одновременно объекты разных типов без определенного порядка. Со множеством можно производить все стандартные операции: добавление и удаление элементов, перебор, – а также такие логические операции, как объединение, пересечение и разность.

Список

Не только в программировании, но и в повседневной жизни мы часто наблюдаем ситуации, когда объекты (данные) выстраиваются в очереди для получения доступа к тому или иному ресурсу. Например, самолеты перед выходом на взлетно-посадочную полосу.

Такие очереди (списки) характеризуются тем, что каждый объект состоит из отдельных и четко различимых элементов. Они могут быть разной природы, но при попадании в список становятся элементами одного множества.

Всем подобным структурам характерна дисциплина: определенный порядок добавления и удаления объектов. Если вернуться к примеру с самолетами, то можно увидеть следующую организацию: каждый лайнер знает, за кем он следует – у него есть «указатель» на следующего за ним. Именно такая организация свойственна списку.

В классическом понимании есть два вида списков:

1. Однонаправленный список, все элементы которого связаны в одном направлении. Предыдущий имеет «указатель» на следующий, но не наоборот;
2. Двухнаправленный список, все элементы которого связаны между собой. Предыдущий связан с последующим, и наоборот.

Для хранения таких данных можно использовать структуру, называемую в Python списком.

Но важно понимать разницу между массивом и списком. Массив – это поименованная пронумерованная совокупность однотипных данных, имеющая конечную длину. Список же объединяет данные различного типа и имеет бесконечную длину.

Перейдем к практике: научимся создавать списки в Python и работать с ними.

Список можно задать явно – с помощью перечисления элементов в квадратных скобках:

```
primes = [2, 3, 5, 7, 11, 13]
rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
```

В списке Primes – 6 элементов:

```
primes[0] == 2,
primes[1] == 3,
primes[2] == 5,
primes[3] == 7,
primes[4] == 11,
primes[5] == 13.
```

Список Rainbow состоит из 7 элементов, каждый из которых является строкой.

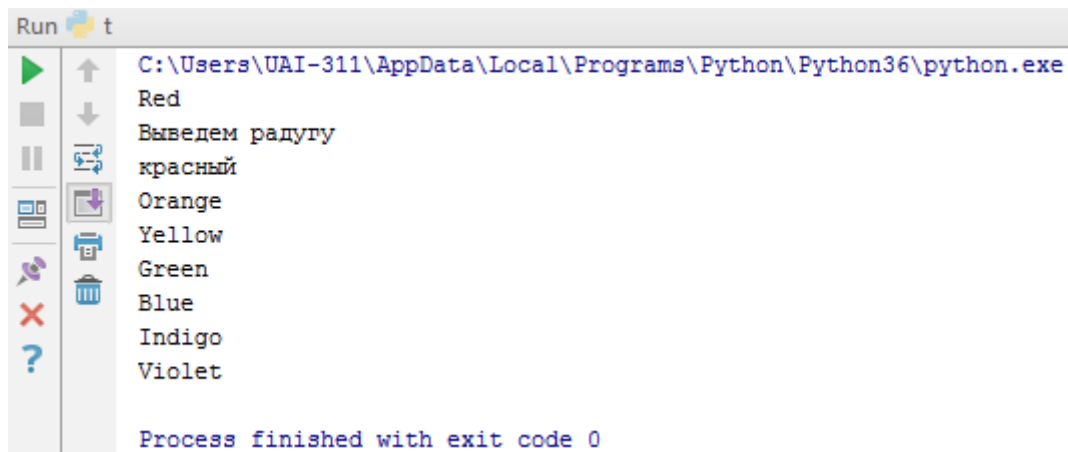
В Python элементы списка, как и символы в строке, можно индексировать отрицательными числами с конца: например, `primes[-1] == 13`, `primes[-6] == 2`.

Длину списка (количество элементов в нем) можно узнать при помощи функции `len`: например, `len(primes) == 6`.

В отличие от строк, элементы списка можно изменять, присваивая им новые значения.

```
rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
print(rainbow[0])
rainbow[0] = 'красный'
print('Выведем радугу')
for i in range(len(rainbow)):
```

```
print(rainbow[i])
```



Рассмотрим организацию методов создания и чтения списков. Возможен следующий сценарий работы: создадим пустой список и будем добавлять элементы в конец, используя метод `append()`.

Например, пользователь будет задавать с клавиатуры размер списка (максимальное количество элементов). Переменную, которая будет отвечать за количество, обозначим как `n`.

Входные данные в таком формате:

```
i = 0
n = int(input('Размер списка: '))
spisok = []
for i in range(n):
    spisok.append(5)

print(spisok)
```

Получение среза

Так же как и к строкам, к спискам можно применить операцию среза. Его виды:

1. Срез от `i`-ого до `j`-ого элемента;
2. Срез от `i`-ого до `j`-ого элемента в обратном порядке;
3. Срез от `i`-ого до `j`-ого элемента с определенным шагом – допустим, каждый третий. Если указать шаг со знаком минус, последовательность будет выводиться в обратном порядке.

Списки, в отличие от массивов, являются изменяемыми объектами: можно отдельному элементу списка присвоить новое значение. Но можно менять и срезы целиком:

```
a = [1, 2, 3, 4, 5]
a[2:4] = [7, 8, 9]

a = [1, 2, 3, 4, 5, 6, 7]
a[::-2] = [10, 20, 30, 40]
```

В первом примере получается список, у которого вместо двух элементов среза `a[2:4]` вставлен новый список, уже из трех элементов. Теперь список стал равен `[1, 2, 7, 8, 9, 5]`.

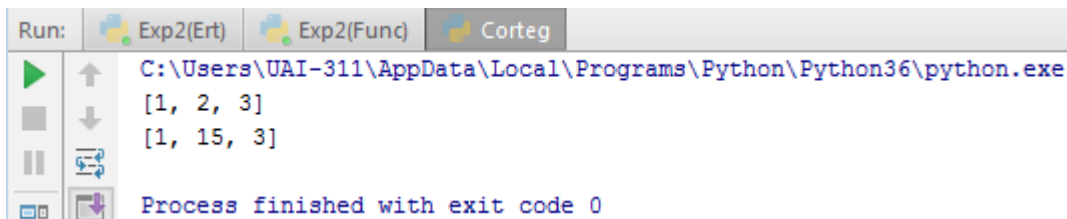
Во втором примере – список `[40, 2, 30, 4, 20, 6, 10]`. Здесь `a[::-2]` – это список из элементов `a[-1]`, `a[-3]`, `a[-5]`, `a[-7]`, которым присваиваются значения 10, 20, 30, 40 соответственно.

Обратите внимание, `a[i]` — это элемент списка, а не срез.

Кортеж

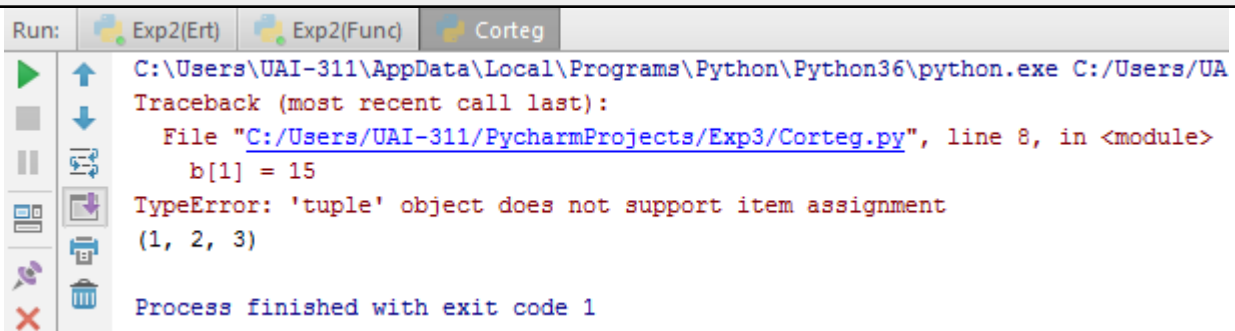
Кортеж (tuple) — это поименованная последовательность неизменяемых данных с динамически изменяющимся размером. В основе организации работы с кортежем лежит список. Если у нас есть список `a = [1, 2, 3]` и мы хотим заменить второй элемент с 2 на 15, то мы можем это сделать, напрямую обратившись к элементу списка.

```
a = [1, 2, 3]
print(a)
a[1] = 15
print(a)
```



А с кортежем мы уже не можем производить такие операции, так как его элементы изменять нельзя.

```
b = (1, 2, 3)
print(b)
b[1] = 15
```



Существует несколько причин, по которым стоит использовать кортежи вместо списков:

1. Обеспечивается целостность данных, защита от потери и случайного изменения. Это удобно, если имеются данные, на основе которых необходимо провести вычисления, но при этом непосредственно менять данные не нужно;
2. Экономится место. Кортежи занимают меньший объем памяти, чем списки;
3. Повышается производительность работы алгоритма. На операции перебора элементов и подобные будет тратиться меньше времени;
4. В словарях кортеж также можно использовать как ключ.

```
lst = [10, 20, 30]
tpl = (10, 20, 30)
print(lst.__sizeof__())
print(tpl.__sizeof__())
```

```
Run: Exp2(Ert) Exp2(Func) Corteg
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
64
48
Process finished with exit code 0
```

Рассмотрим на практике способы работы с кортежами.

Создание, удаление кортежей и работа с его элементами

Для создания пустого кортежа можно воспользоваться одной из следующих команд.

```
a = ()
print(type(a))
b = tuple()
print(type(b))
```

```
Run: Exp2(Ert) Exp2(Func) Corteg
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
<class 'tuple'>
<class 'tuple'>
Process finished with exit code 0
```

Кортеж, содержащий исходные данные, создается с помощью перечисления данных в круглых скобках.

```
a = (1, 2, 3, 4, 5)
print(type(a))
print(a)
```

```
Run: Exp2(Ert) Exp2(Func) Corteg
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
<class 'tuple'>
(1, 2, 3, 4, 5)
Process finished with exit code 0
```

Доступ к элементам кортежа осуществляется так же, как к элементам списка, – через указание индекса.

```
a = (1, 2, 3, 4, 5)
print(a[0])
print(a[1:3])
a[1] = 3
```

```
Run: Exp2(Ert) Exp2(Func) Corteg
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe C:/Users/UAI-:
1
(2, 3)
Traceback (most recent call last):
  File "C:/Users/UAI-311/PycharmProjects/Exp3/Corteg.py", line 27, in <module>
    a[1] = 3
TypeError: 'tuple' object does not support item assignment
Process finished with exit code 1
```

Удалить элемент кортежа невозможно.

```
a = (1, 2, 3, 4, 5)
del a[0]
del a
print(a)
```

```
Run t
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe C:/User
Traceback (most recent call last):
  File "C:/Users/UAI-311/PycharmProjects/Exp3/t.py", line 10, in <module>
    del a[0]
TypeError: 'tuple' object doesn't support item deletion
Process finished with exit code 1
```

Преобразование кортежа в список и обратно

Так как в основе организации кортежа лежит список, то можно преобразовать кортежи в списки и обратно. Для превращения списка в кортеж достаточно передать его в качестве аргумента функции `tuple()`.

```
lst = [1, 2, 3, 4, 5]
print(type(lst))
print(lst)
tpl = tuple(lst)
print(type(tpl))
print(tpl)
```

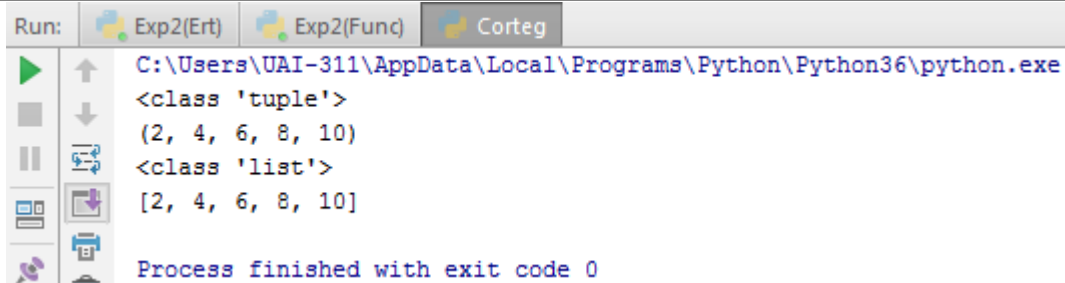
```
Run: Exp2(Ert) Exp2(Func) Corteg
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
<class 'list'>
[1, 2, 3, 4, 5]
<class 'tuple'>
(1, 2, 3, 4, 5)
Process finished with exit code 0
```

Обратная операция также является корректной.


```

tpl = (2, 4, 6, 8, 10)
print(type(tpl))
print(tpl)
lst = list(tpl)
print(type(lst))
print(lst)

```



Множества

У множества есть существенное отличие от вышеупомянутых структур хранения данных: его элементы организованы не в виде последовательности, а в виде групп (если хотите, подмножеств). Каждая группа имеет определенный алгоритм связи элементов. Это позволяет классифицировать объекты по набору признаков, а не по одному, как это было в списках, массивах и кортежах. Мы можем производить проверку объекта на принадлежность к определенному множеству.

Множество может состоять из объектов только неизменяемых типов данных: чисел, строк и кортежей.

Пустое множество можно создать с помощью функции `set()`. Также в функцию можно передать строку или кортеж, и тогда она вернет сформированное множество.

Простое множество задается перечислением элементов в фигурных скобках.

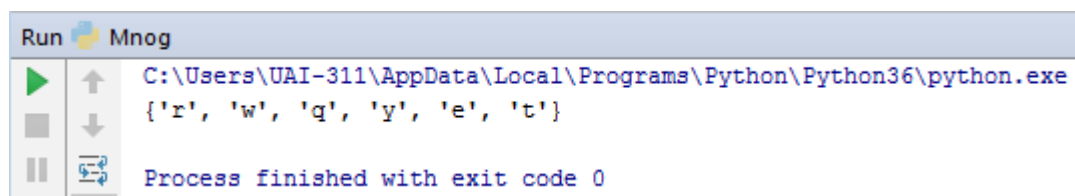
Пример 1:

```

a = {1, 2, 3}
a = set('qwerty')
print(a)

```

Результат работы:



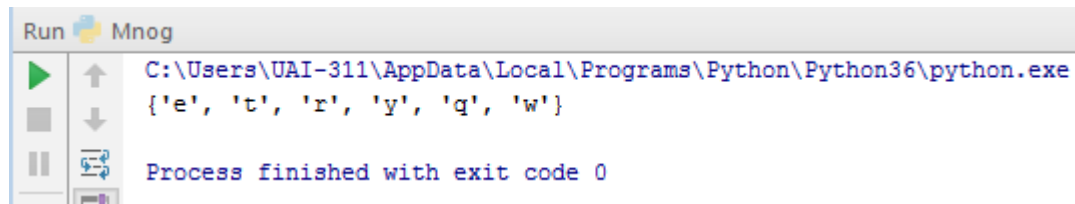
Пример 2:

```

a = {1, 2, 3}
a = set('qwerty')
print(a)

```

Результат работы:



```
Run Mnog
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
{'e', 't', 'r', 'y', 'q', 'w'}
Process finished with exit code 0
```

Работа с элементами множеств

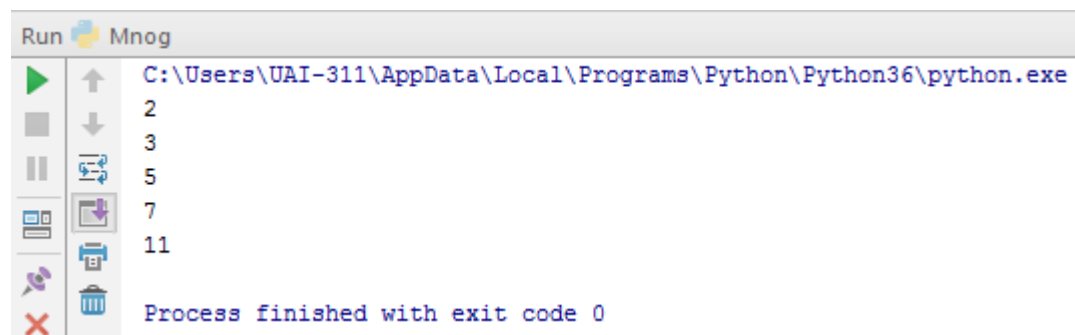
Для старта работы с множеством стоит определить изначальное количество его элементов. Это можно сделать с помощью метода **len()**.

Для перебора элементов множества используется цикл **for**:

Пример 1:

```
primes = {2, 3, 5, 7, 11}
for num in primes:
    print(num)
```

Результат работы:

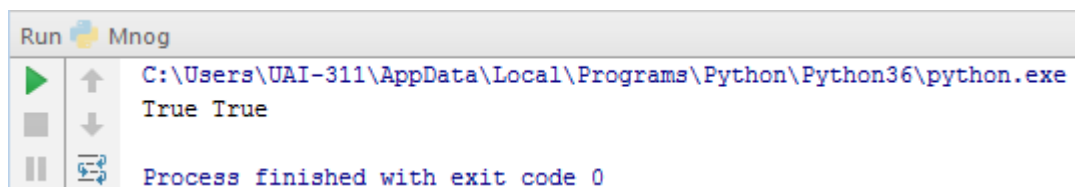


```
Run Mnog
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
2
3
5
7
11
Process finished with exit code 0
```

Пример 2:

```
a = {1, 2, 3}
print(1 in a, 4 not in a)
a.add(4)
```

Результат работы:



```
Run Mnog
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
True True
Process finished with exit code 0
```

Для удаления элемента «x» из множества есть два метода: **discard()** и **remove()**. Если удаляемый элемент отсутствует в множестве, то метод **discard()** не делает ничего, а метод **remove()** генерирует исключение **KeyError**.

Метод `pop()` удаляет из множества один случайный элемент и возвращает его значение. Если же множество пусто, то генерируется исключение `KeyError`.

Из множества можно сделать список при помощи функции `list()`.

Алгоритмы обработки последовательностей, одномерных и двумерных массивов (списков)

Разложить положительные и отрицательные числа по разным массивам

Дано два массива, в них необходимо распределить случайные числа в диапазоне от -5 до 5. Положительные числа будем помещать в первый массив, отрицательные во второй. Числа, равные нулю, не учитываем.

В итоге представим случайно сгенерированные числа и результат распределения по массивам.

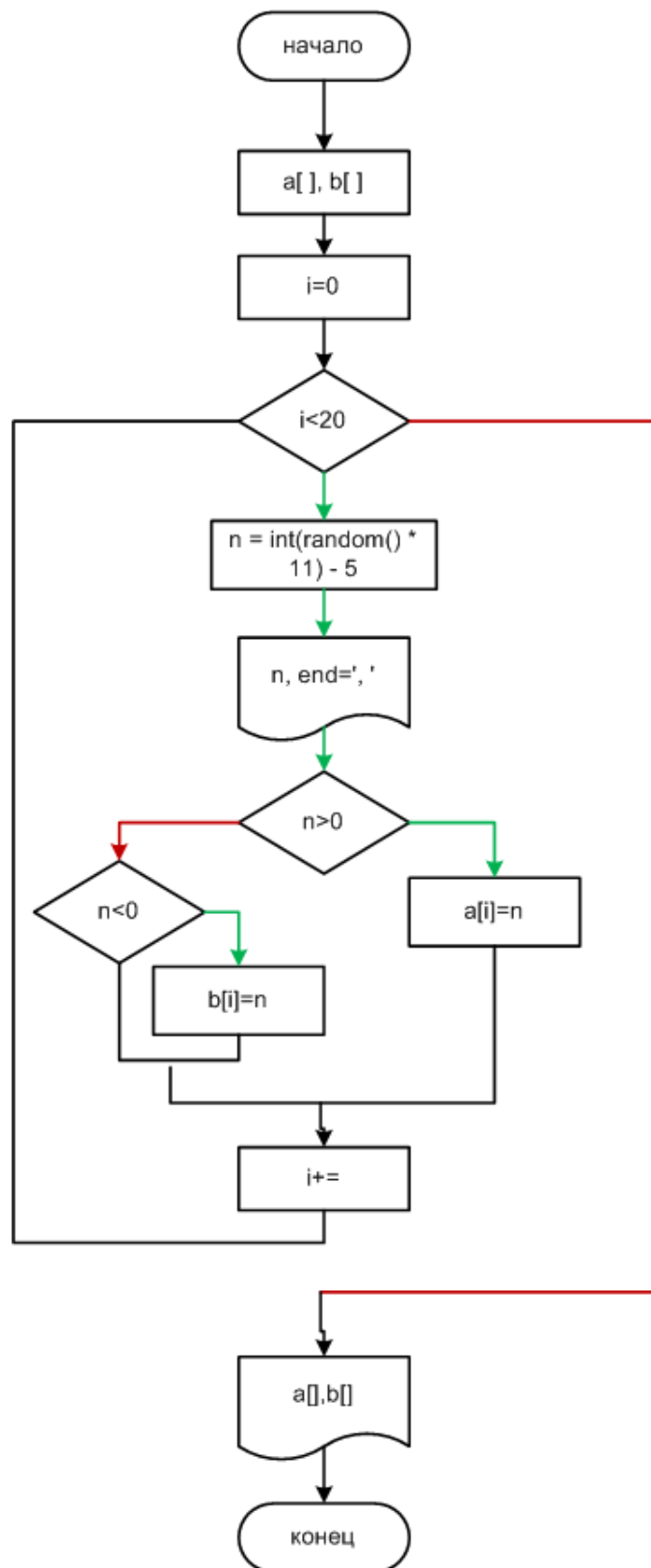
Алгоритм сводится к следующим шагам:

1. Создаем два пустых массива. Не забываем ввести переменные индексы для каждого из массивов, начальное значение которых равно нулю;

Следующие действия выполняются в теле цикла.

2. Генерируем случайное число и выводим его на экран;
3. Если сгенерированное число положительное, то увеличиваем индекс первого массива и записываем число в массив;
4. Если сгенерированное число отрицательное, то увеличиваем индекс второго массива и записываем число в массив;
5. Организуем циклы для вывода обоих получившихся массивов. Переменные индекса в итоге – это переменные, определяющие размер массива.

Блок-схема алгоритма:



Программная реализация:

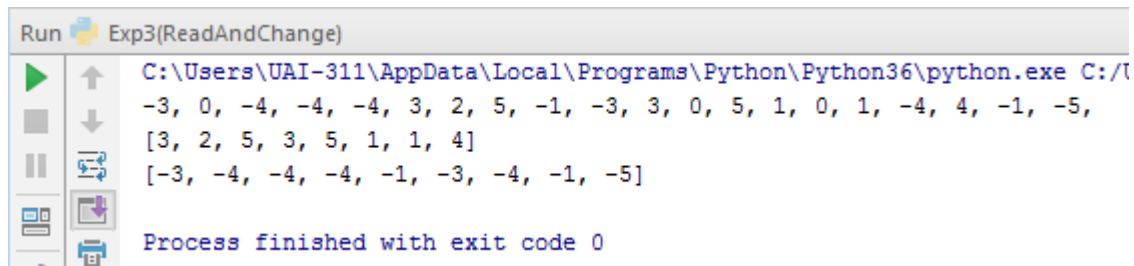
```
from random import random

a = []
b = []

for i in range(20):
    n = int(random() * 11) - 5
    print(n, end=', ')
    if n > 0:
        a.append(n)
    elif n < 0:
        b.append(n)

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

Результат работы программы:



```
Run Exp3(ReadAndChange)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe C:/I
-3, 0, -4, -4, -4, 3, 2, 5, -1, -3, 3, 0, 5, 1, 0, 1, -4, 4, -1, -5,
[3, 2, 5, 3, 5, 1, 1, 4]
[-3, -4, -4, -4, -1, -3, -4, -1, -5]
Process finished with exit code 0
```

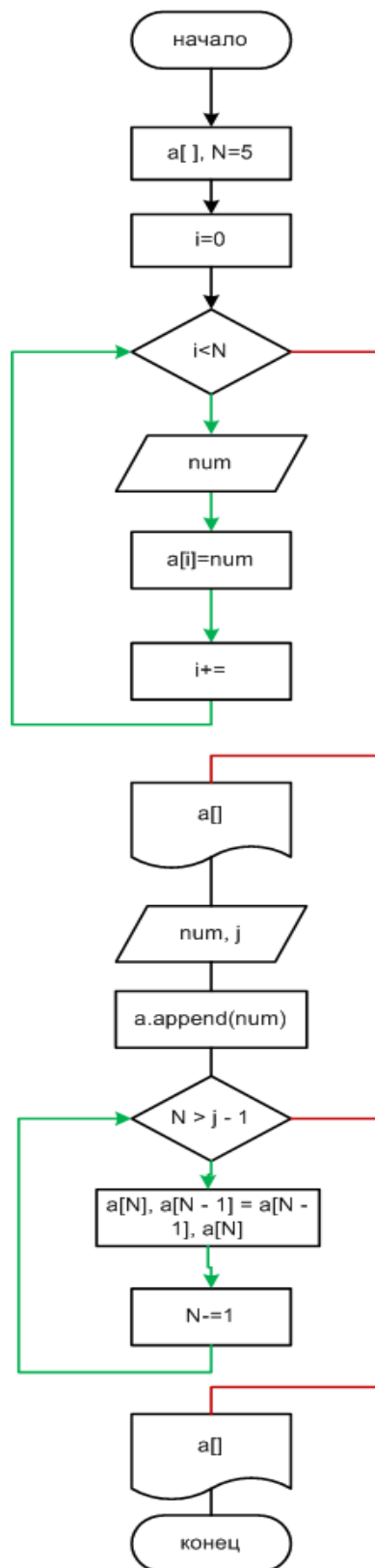
На Python данная задача решается проще, так как ее можно реализовать, просто добавляя элементы в конец списков. Также можно вывести весь список на экран, вызвав переменную, с которой он связан, а не обращаться к каждому элементу отдельно.

Вставка элемента в произвольное место массива

Заполним пустой массив данными, введенными с клавиатуры, кроме последнего элемента. Он понадобится для вставки нового элемента в произвольное место.

Чтобы поставить новый элемент на желаемую позицию в массиве, необходимо сдвинуть все его элементы, начиная с этой позиции, на один. Таким образом ячейка с указанным номером становится свободной, и в нее можно вставить заданное число.

Блок-схема алгоритма:



Программная реализация:

```
# 1-й вариант:

a = []
N = 5
for i in range(N):
    num = int(input())
    a.append(num)
print(a)
num = int(input("Число: "))
j = int(input("Позиция: "))
a.append(num)
while N > j - 1:
    a[N], a[N - 1] = a[N - 1], a[N]
    N -= 1
print(a)

# 2-й вариант:

a = []
N = 5
for i in range(N):
    num = int(input())
    a.append(num)
print(a)
num = int(input("Число: "))
j = int(input("Позиция: "))

a.insert(j - 1, num)

print(a)

# 3-й вариант:

a = []
N = 5
for i in range(N):
    num = int(input())
    a.append(num)
print(a)
num = int(input("Число: "))
j = int(input("Позиция: "))

a = a[0:j - 1] + [num] + a[j - 1:]

print(a)
```

Результат работы программы:

```
Run Exp3(Insert1,2,3)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
4
5
6
7
8
[4, 5, 6, 7, 8]
Число: 6
Позиция: 3
[4, 5, 6, 6, 7, 8]
7
8
9
3
5
[7, 8, 9, 3, 5]
Число: 6
Позиция: 5
[7, 8, 9, 3, 6, 5]
4
5
6
7
8
[4, 5, 6, 7, 8]
Число: 11
Позиция: 2
[4, 11, 5, 6, 7, 8]
Process finished with exit code 0
```

Мы рассмотрели три реализации классического алгоритма вставки элемента на необходимую позицию. Теперь вы понимаете, как он работает.

В Python данный алгоритм уже реализован с помощью стандартной функции **insert(index, obj)**. Второй вариант реализации показывает работу этой функции.

Реверс массива

Реверс – вывод элементов массива в обратном порядке.

Алгоритм сводится к следующему:

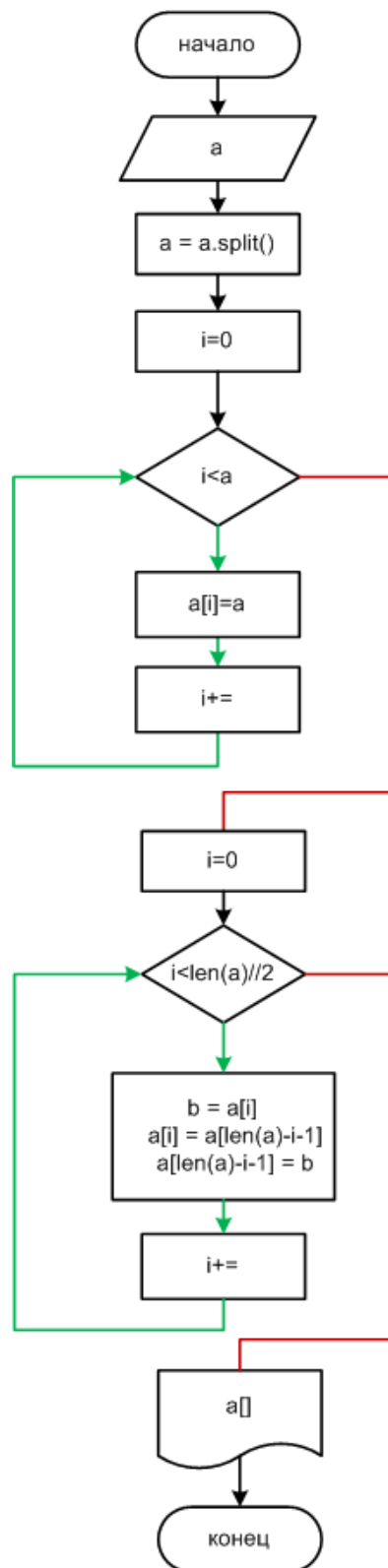
1. Первый элемент массива меняется местами с последним;
2. Второй элемент массива меняется с предпоследним;
3. Третий элемент меняется местами с элементом, третьим с конца, и так далее – пока мы не дойдем до элемента (или элементов), являющегося серединой массива.

Аналогично мы можем передвигаться из середины массива к границам (к первому и последнему элементам массива). Но первый вариант проще в реализации: если в массиве нечетное количество элементов, то в середине находится один элемент – у него нет пары и обменивать его не надо. Если же в массиве четное количество элементов, то в середине будет пара, которая также должна

поменяться местами. В любом случае, количество обменов будет равно числу элементов массива, нацело деленному на 2.

Этот алгоритм можно реализовать с помощью циклического перебора. Количество итераций равно целому среднему количеству элементов массива. В теле цикла происходит обмен элементов. Если индекс массива начинается с единицы, а количество элементов N , то индекс элемента, с которым должен происходить обмен, будет вычисляться по формуле $N-i+1$. Если же индексация идет с нуля, то противоположный для i элемент находится как $N-i-1$.

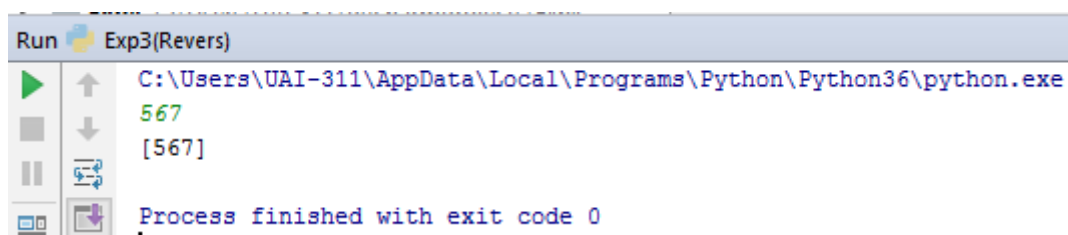
Блок-схема алгоритма:



Программная реализация:

```
a = input('Элементы через пробел: ')
a = a.split()
a = [int(i) for i in a]
for i in range(len(a)//2):
    b = a[i]
    a[i] = a[len(a)-i-1]
    a[len(a)-i-1] = b
#a.reverse()
#a = a[::-1]
print(a)
```

Результат работы программы:



В Python есть метод, позволяющий сделать реверс списка – **reverse()**. Также список можно перевернуть с помощью операций взятия среза `a[::-1]`: он берет все элементы из списка, начиная с конца.

В коде ниже эти способы указаны в виде комментариев.

Вывести неповторяющиеся элементы массива

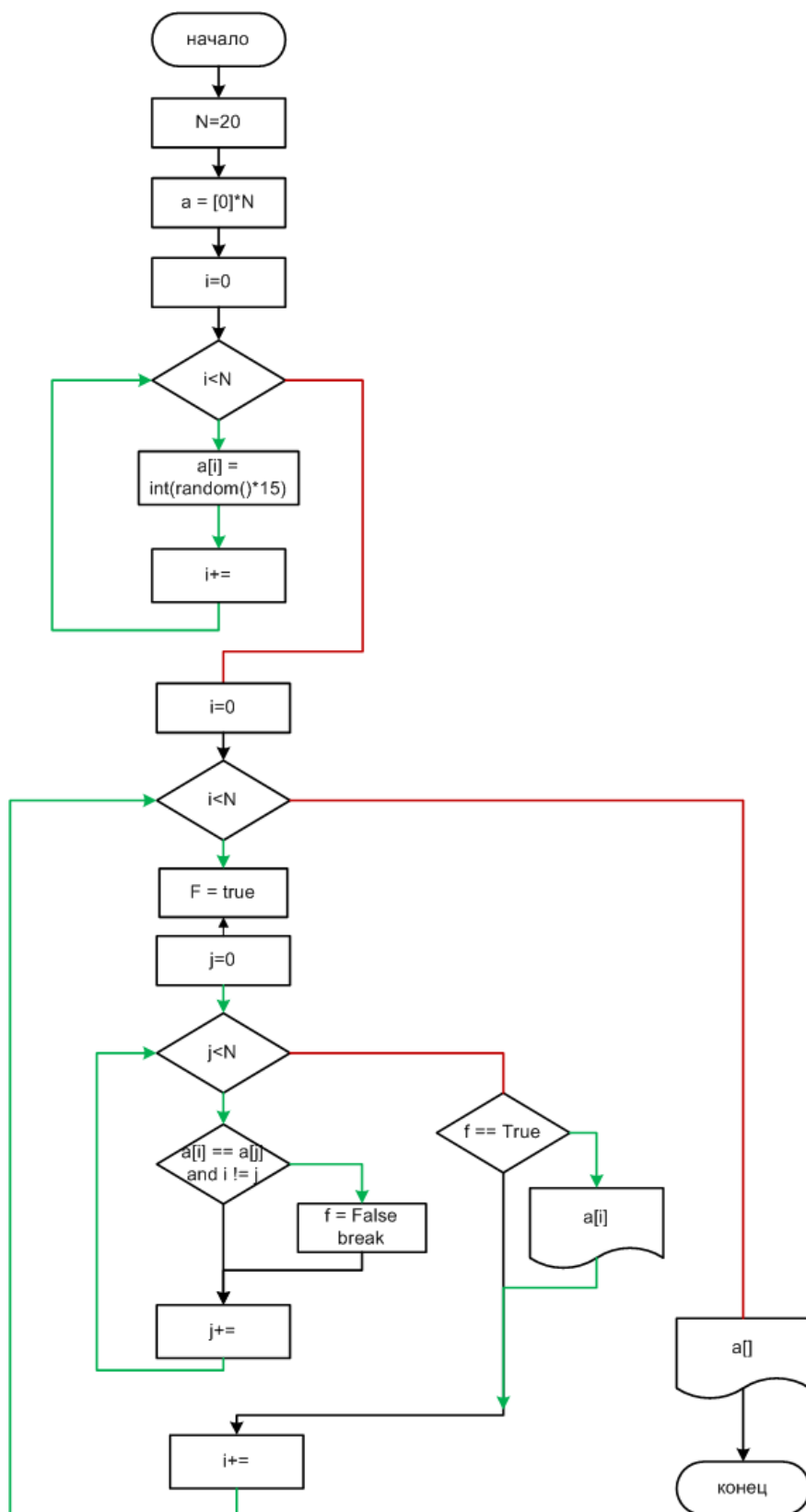
Задан массив чисел, необходимо вывести только уникальные элементы.

Алгоритм сводится к следующему решению:

1. Берем первый элемент массива и сравниваем со всеми остальными. Если окажется, что хотя бы один из следующих элементов будет равен первому, значит он не уникален. Если совпадений не было, то выводим первый элемент;
2. Берем второй элемент массива и сравниваем со всеми остальными. Если окажется, что хотя бы один из следующих элементов будет равен второму, значит он не уникален. Если совпадений не было, то выводим второй элемент;
3. Таким образом проверяем все элементы массива.

Для определения повторяющихся элементов массива стоит ввести переменную «флажок», которая будет менять свое значение при выполнении заданного условия – совпадения элементов. Допустим, перед проверкой очередного элемента эта переменная будет принимать значение 1. При ситуации совпадения значение будет равно 0. Если после всех сравнений флажок остался равен 1, значит элемент не повторяется в массиве.

Блок-схема алгоритма:



Программная реализация:

```

from random import random

N = 20
a = [0] * N
for i in range(N):

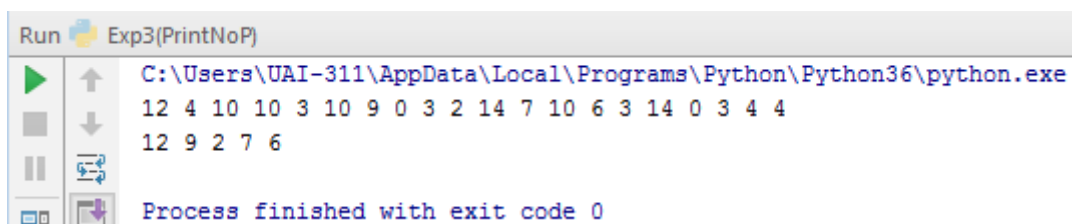
```

```

a[i] = int(random()*15)
print(a[i],end=' ')
print()
for i in range(N):
    f = True
    for j in range(N):
        if a[i] == a[j] and i != j:
            f = False
            break
    if f:
        print(a[i],end=' ')
print()

```

Результат работы программы:



```

Run Exp3(PrintNoP)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
12 4 10 10 3 10 9 0 3 2 14 7 10 6 3 14 0 3 4 4
12 9 2 7 6
Process finished with exit code 0

```

Данную задачу можно реализовать с помощью уже существующих в Python функции `list(set(old_list))`.

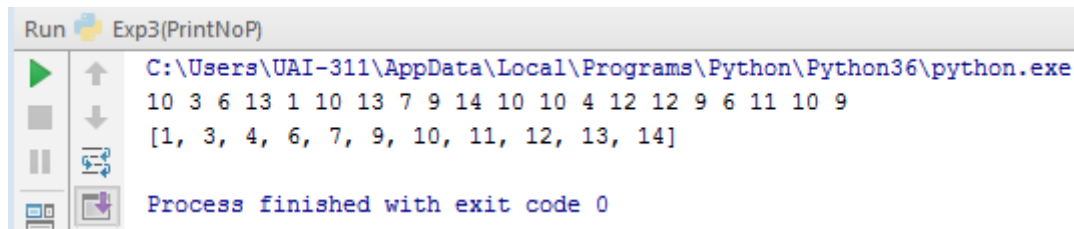
```

from random import random

N = 20
a = [0] * N
for i in range(N):
    a[i] = int(random()*15)
    print(a[i],end=' ')
print()
#for i in range(N):
#    f = True
#    for j in range(N):
#        if a[i] == a[j] and i != j:
#            f = False
#            break
#    if f == True:
#        print(a[i],end=' ')
print(list(set(a)))

```

В итоге мы увидим:



```
Run Exp3(PrintNoP)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
10 3 6 13 1 10 13 7 9 14 10 10 4 12 12 9 6 11 10 9
[1, 3, 4, 6, 7, 9, 10, 11, 12, 13, 14]
Process finished with exit code 0
```

Мы рассмотрели несколько примеров работы с одномерными массивами. Перейдем к практике с двумерными.

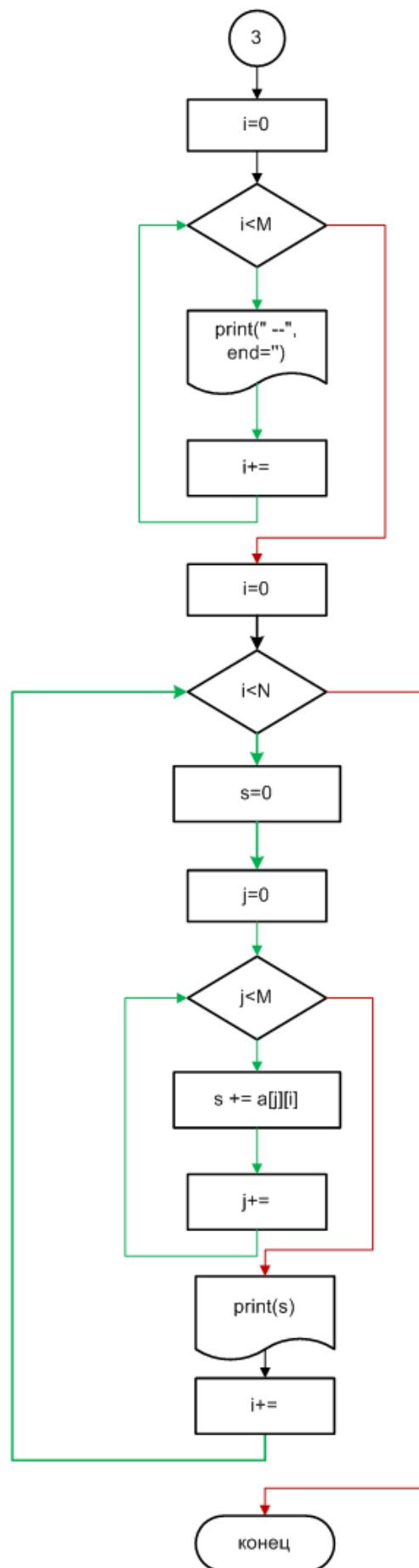
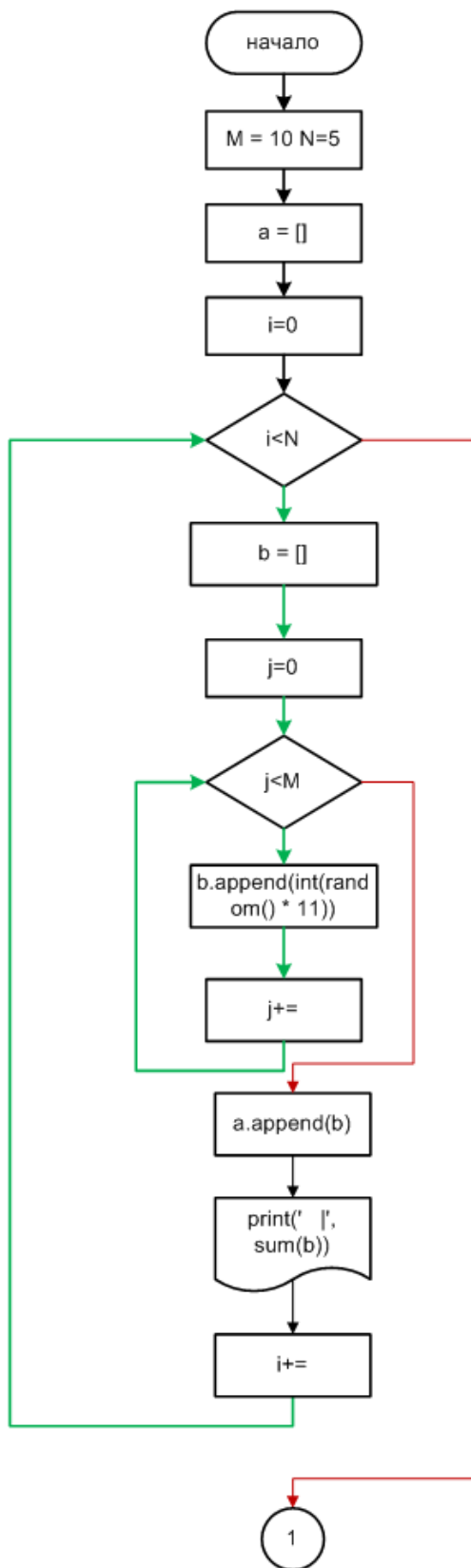
Суммы строк и столбцов матрицы

Дана матрица целых или вещественных чисел. Необходимо посчитать сумму ее столбцов и строк. Для наглядности введем сумму каждой строки после нее, аналогично покажем суммы столбцов.

Алгоритм сводится к следующему решению:

1. Начнем построчно заполнять матрицу элементами и запоминать сумму введенных элементов в отдельную переменную;
2. После заполнения конкретной строки будем выводить сумму на экран, а после этого обнулять значение переменной, которая запоминает сумму элементов строки;
3. Когда матрица заполнена, мы можем пройти по ее столбцам, и внутри цикла по столбцу считать сумму его элементов. После прохода по каждому столбцу выводим его сумму.

Блок-схема алгоритма:



Программная реализация:

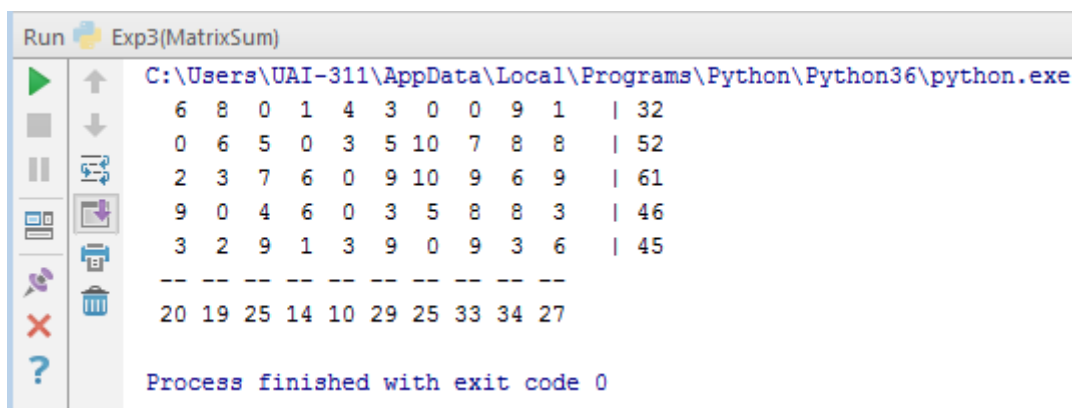
```
from random import random

M = 10
N = 5
a = []
for i in range(N):
    b = []
    for j in range(M):
        b.append(int(random() * 11))
        print(b[j], end='')
    a.append(b)
    print(' |', sum(b))

for i in range(M):
    print(" --", end='')
print()

for i in range(M):
    s = 0
    for j in range(N):
        s += a[j][i]
    print(s, end='')
print()
```

Результат работы программы:



```
Run Exp3(MatrixSum)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
6 8 0 1 4 3 0 0 9 1 | 32
0 6 5 0 3 5 10 7 8 8 | 52
2 3 7 6 0 9 10 9 6 9 | 61
9 0 4 6 0 3 5 8 8 3 | 46
3 2 9 1 3 9 0 9 3 6 | 45
-----
20 19 25 14 10 29 25 33 34 27
Process finished with exit code 0
```

В Python данный алгоритм реализуется немного другим способом. Сначала создается пустой список (наша матрица). Далее в нее в цикле добавляются вложенные списки (строки матрицы).

Суммы строк матрицы вычисляются с помощью функции **sum()**.

Суммы столбцов вычисляются путем прохода по каждому столбцу матрицы.

Обмен значений главной и побочной диагоналей квадратной матрицы

Дана квадратная матрица 10x10. Необходимо поменять местами главную и побочную диагонали.

Алгоритм

Обратите внимание, что диагонали можно определить только в квадратных матрицах. Главная диагональ – из верхнего левого угла матрицы в нижний правый. Побочная – из верхнего правого в нижний левый. Соответственно, можно определить закономерность изменения индексов как на главной, так и на побочной диагонали. На главной индексы строки и столбца элемента равны между собой, а на побочной – противоположны.

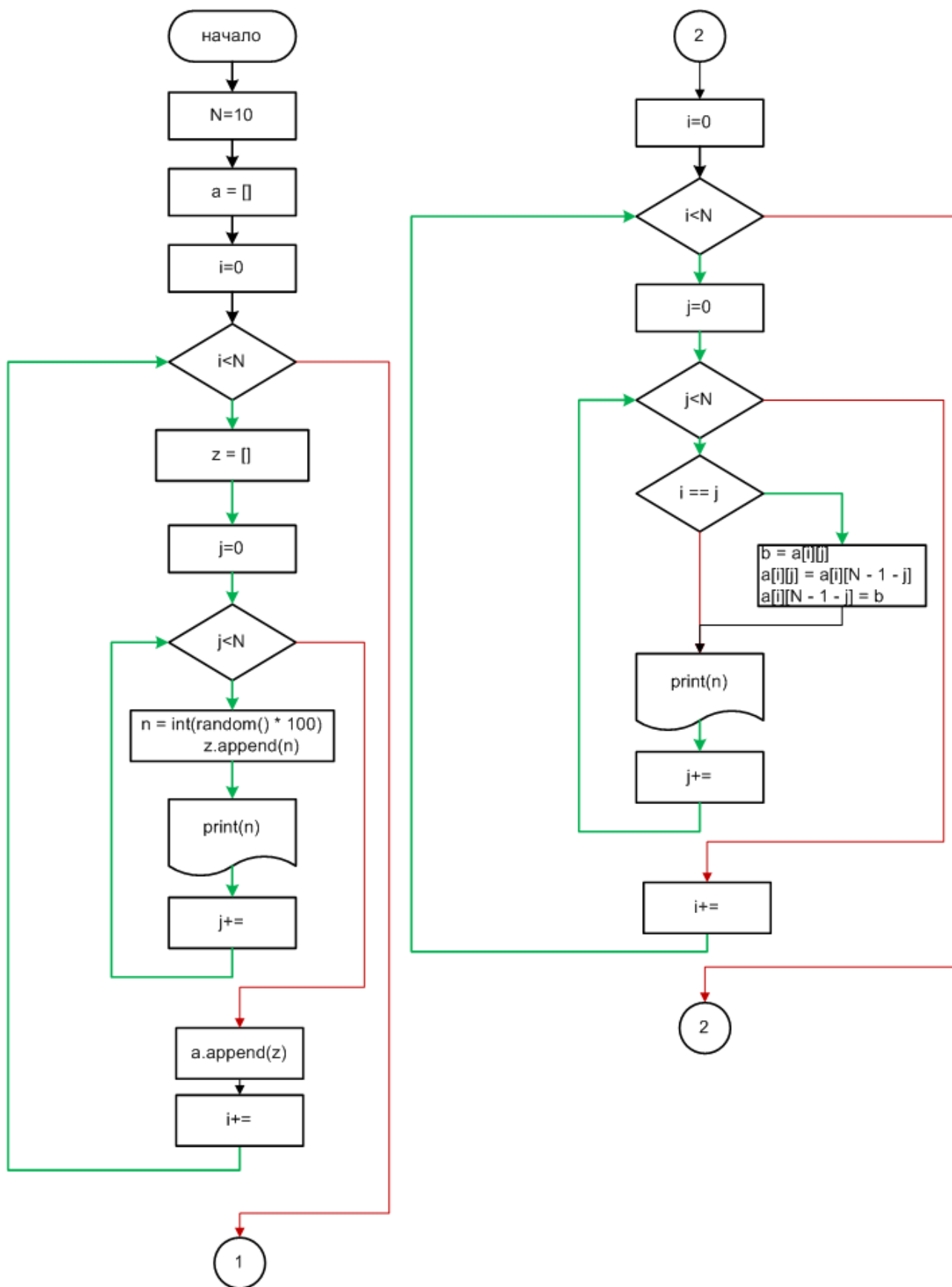
Так как менять элементы диагоналей будем построчно, выделим формулы расчета индексов:

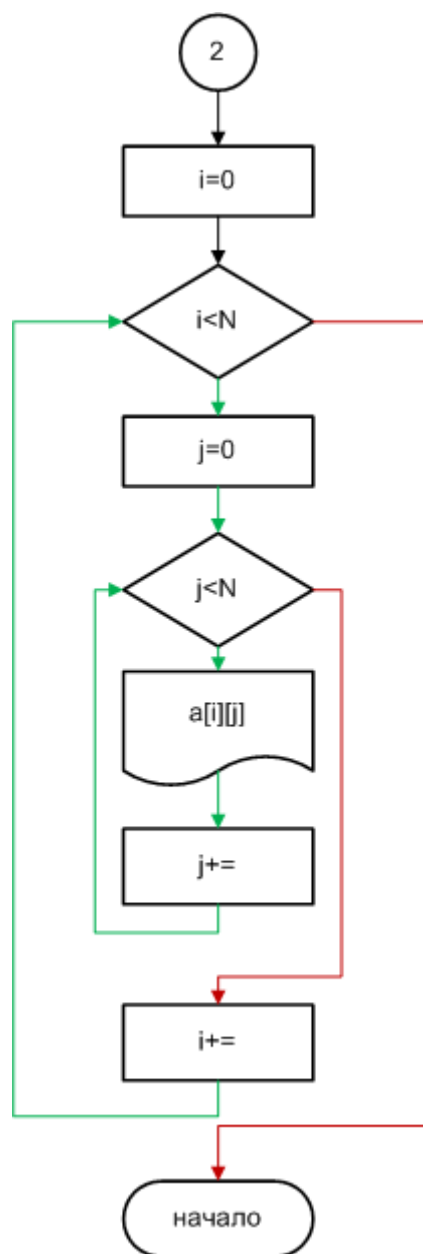
1. Элементы главной диагонали будут иметь индексы $i=j$;
2. У элементов побочной индекс строки будет i , а индекс столбца рассчитывается по формуле $N-1-j$.

Обратите внимание, что расчет индекса столбца у побочной диагонали будет выглядеть так в случае индексации с нуля.

Производить обмен элементов будем в случае выполнения условия $i = j$.

Блок-схема алгоритма:





Программная реализация:

```
from random import random

N = 10
a = []
for i in range(N):
    z = []
    for j in range(N):
        n = int(random() * 100)
        z.append(n)
        print(n, end='')
    print()
    a.append(z)
print()

for i in range(N):
    for j in range(N):
        if i == j:
            b = a[i][j]
            a[i][j] = a[i][N - 1 - j]
            a[i][N - 1 - j] = b

for i in range(N):
    for j in range(N):
        print(a[i][j], end='')
    print()
```

Результат работы программы:

```

Run Exp3(MatrixObmen)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
38  9 20 83  5 79 30 92  5 77
56 84 35 77 64  5 92 64 86 95
96 80 69 80 37 55 28 56  4 61
12 74 90 47 86 32 96 22 87  3
92 90 16 87  6 10 29 88 90 74
99 20 91 83 93 14 88 33 79 36
50 81 98 56 44 74 98  2 21 28
70 19 84 73 16 91 75 42 89 56
26 14 65  5 91 97 69  9 18 71
96 45 38 69 11 76 61 64 69 45

77  9 20 83  5 79 30 92  5 38
56 86 35 77 64  5 92 64 84 95
96 80 56 80 37 55 28 69  4 61
12 74 90 96 86 32 47 22 87  3
92 90 16 87 10  6 29 88 90 74
99 20 91 83 14 93 88 33 79 36
50 81 98 98 44 74 56  2 21 28
70 19 42 73 16 91 75 84 89 56
26 18 65  5 91 97 69  9 14 71
45 45 38 69 11 76 61 64 69 96

Process finished with exit code 0

```

Запись в матрицу результатов побитовых операций

Дана матрица размером 4x8. Необходимо заполнить первые две строки случайным образом 0 и 1. В третью запишем результат побитовой операции «И» над двоичными числами, которые получились в первой и второй строке матрицы. А в четвертую поместим результат логической операции «ИЛИ» над теми же числами.

Вспомним, как работают эти логические операции:

A	B	И	ИЛИ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

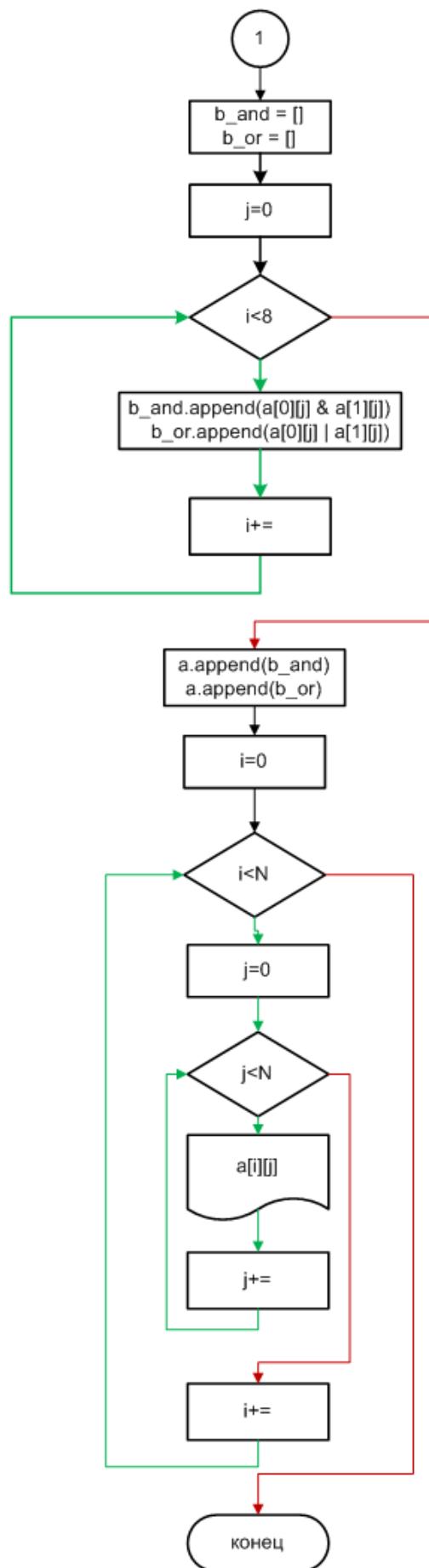
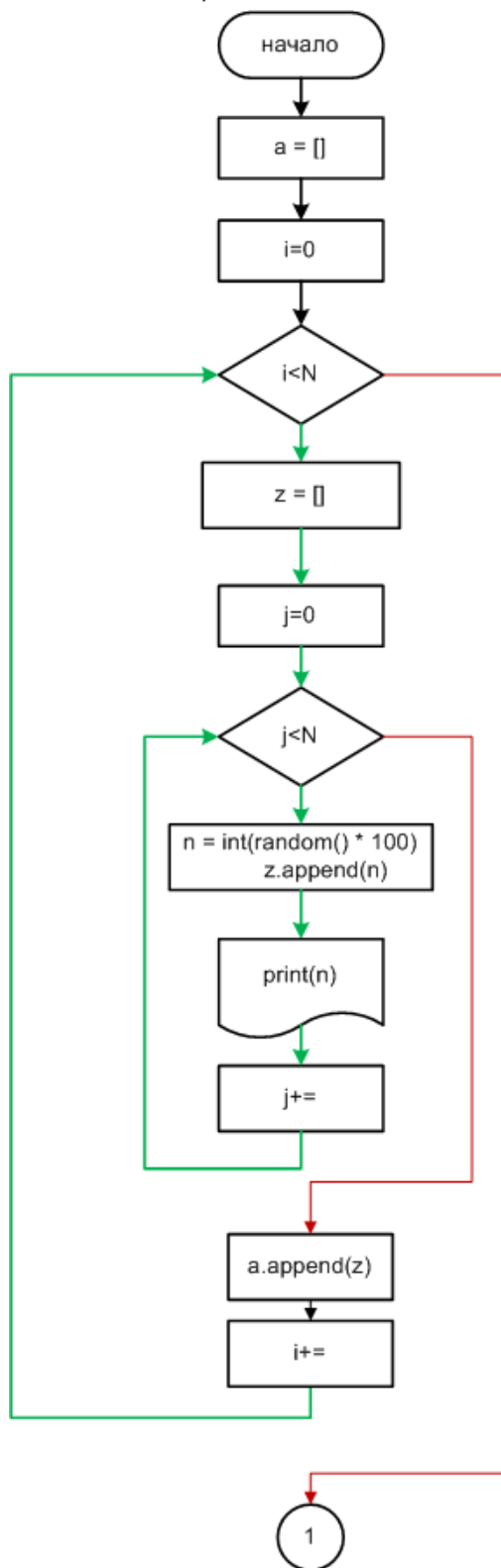
По сути, логическая операции И – это логическое умножение, а логическая операция ИЛИ – логическое сложение.

Алгоритм решения задачи:

1. Сначала заполняем нулями и единицами первые две строки. Выводим получившиеся строки;
2. В цикле по столбцам матрицы заполняем третью и четвертую строки результатами побитовых операций «И» и «ИЛИ»;

3. Выводим получившуюся матрицу.

Блок-схема алгоритма:



Программная реализация:

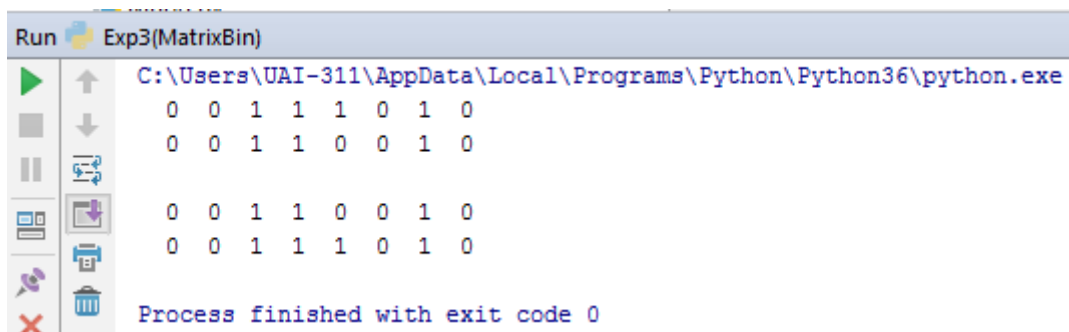
```
from random import random

a = []
for i in range(2):
    z = []
    for j in range(8):
        n = int(random() * 2)
        z.append(n)
        print(n, end='')
    print()
    a.append(z)
print()

b_and = []
b_or = []
for j in range(8):
    b_and.append(a[0][j] & a[1][j])
    b_or.append(a[0][j] | a[1][j])
a.append(b_and)
a.append(b_or)

for i in range(2, 4):
    for j in range(8):
        print(a[i][j], end='')
    print()
```

Результат работы программы:



```
Run Exp3(MatrixBin)
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
0 0 1 1 1 0 1 0
0 0 1 1 0 0 1 0

0 0 1 1 0 0 1 0
0 0 1 1 1 0 1 0

Process finished with exit code 0
```

Работа с ассоциативными массивами (таблицами данных)

Ассоциативный массив лучше всего описывается табличным представлением данных, когда каждая строка таблицы содержит характеристики какого-то объекта из множества однородных. Типичный пример – список учеников, их домашних телефонов и адресов. По значению из первого столбца такой таблицы (ключу) можно однозначно определить значения из остальных столбцов. Значение ключа ассоциируется с остальными характеристиками объекта. В случае ученика – по фамилии можно найти другую информацию.

Если в ассоциативном массиве только два столбца («ключ» и «значение»), то такой массив называется «хэш». Они часто используются в современных информационных системах: например, пары «логин-пароль».

В области моделирования процессов и явлений часто встречаются задачи, в которых значению «ключа» соответствует несколько параметров. Например, номеру химического элемента однозначно соответствует название, атомный вес, валентность, количество протонов и прочее. В таких задачах простые хэш-массивы использовать уже неудобно.

Эффективный алгоритм обработки ассоциативных массивов (поиска значений, добавления и удаления значений и ключей, сортировки и прочего) в значительной степени зависит от используемого языка программирования, определенных в нем типов и структур данных. Так, в Basic ассоциативный массив образуется из нескольких согласованных одномерных массивов. В Pascal для представления ассоциативных массивов используется структура данных «запись» (record).

В Python для ассоциативных массивов определена специальная структура данных – словарь, но мы рассмотрим работу с помощью списков и соответствующих функций.

Разберем задачу из области экономического анализа.

Оценим экономическую деятельность нескольких предприятий. Известны их названия, значения планового и фактического объемов розничного товарооборота.

Требуется определить:

1. Процент выполнения плана каждым предприятием;
2. Количество предприятий, не выполнивших план;
3. Наибольший плановый товарооборот;
4. Упорядочить предприятия по возрастанию планового товарооборота.

Обозначим количество предприятий как k и сформируем три списка: список названий предприятий (name), список значений планового товарооборота (plan), список значений фактического товарооборота (fact). На основании этих данных создадим список процентных значений выполнения плана (procent).

Количество предприятий, не выполнивших план, будем определять в результате сравнения процента выполнения со 100 % в цикле по всем предприятиям.

Текст программы на Python может выглядеть так:

```
# k - количество предприятий
# name - список названий предприятий
# plan - список значений планового товарооборота
# fact - список значений фактического товарооборота
# procent - список значений % выполнения плана
#
k = input("Количество предприятий: ")
name = []
plan = []
fact = []

for i in range(int(k)):
    name1 = input("Название: ")
    name.append(name1)
    plan1 = input("План: ")
    plan.append(plan1)
    fact1 = input("Факт: ")
    fact.append(fact1)

procent = map(lambda x, y: x * 100 / y, fact, plan)
fakt_y = zip(name, procent)
plan_y = zip(plan, name)
plan_y.sort()

print(16 * "=")
print("Процент выполнения плана каждым предприятием:")

nedo = 0
for i in range(k):
    s1 = fakt_y[i][0]
    s2 = fakt_y[i][1]
    if s2 < 100:
        nedo = nedo + 1

    print(s1, ": ", s2)
print("Количество предприятий, не выполнивших план: ", nedo)
print("Наибольший плановый товарооборот: ", max(plan))

print("Предприятия по возрастанию плана:")

for i in range(k):
    s1 = plan_y[i][1]
    s2 = plan_y[i][0]
    print(s1, ": ", s2)
```

Двоичный (бинарный) поиск элемента в массиве.

Пользователь вводит число. Сообщить, есть ли оно в массиве, элементы которого расположены по возрастанию значений. Если есть – в каком месте находится? При решении задачи использовать бинарный (двоичный) поиск. Оформить его в виде отдельной функции.

Алгоритм поиска

Рассмотрим один из самых распространенных и популярных алгоритмов поиска – алгоритм бинарного поиска. Следует учесть, что он применяется только для уже упорядоченных массивов.

Допустим, есть массив чисел [1, 3, 7, 8, 15, 16, 19, 20, 30, 32]. Требуется определить, есть ли в нем элемент со значением 8.

Находится середина массива – как частное от целочисленного деления количества элементов массива на 2. В данном случае получаем 5.

Сравниваем элемент с пятым индексом с искомым элементом, то есть сравниваем числа 16 и 8, если индексация идет с нуля. Они не равны друг другу, и 16 больше, чем 8.

Находим середину левой части массива от пятого по индексу элемента. Для этого складываем индекс первого элемента и предыдущий от прежней середины. Если индексация идет с нуля, то получим $0+4 = 4$. Далее делим нацело на 2, получаем 2 – это новая середина.

Сравниваем второй по индексу элемент (число 7) и искомый (число 8). Они не равны, и 7 меньше, чем 8.

Значит, рассматриваем отрезок с третьего индекса до четверного. Его середина $(3+4)$, деленная нацело на 2, равна 3.

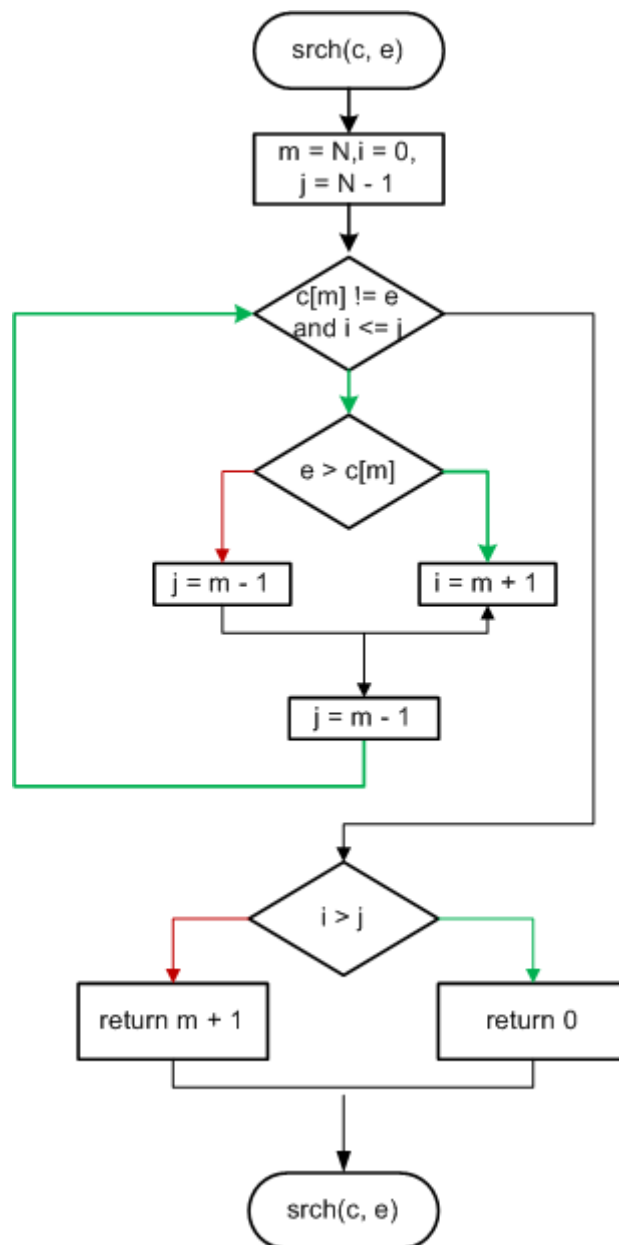
Сравниваем третий по индексу элемент с числом 8. Они равны. Значит, искомый элемент есть в массиве и находится под третьим индексом.

Теперь рассмотрим ситуацию, когда элемента нет. Допустим, требуется найти число 25 в том же массиве.

- Первая середина – число 16, что меньше 25;
- Вторая середина – $(6+10) / 2 = 8$. $30 > 25$;
- Третья середина – $(6+7) / 2 = 6$ (деление нацело). $19 < 25$;
- Четвертая середина – $(7+7) / 2 = 7$. $20 < 25$.

Левая граница массива становится больше на 1, чем правая. Это сигнал о том, что искомого элемента в массиве нет.

Блок-схема:



Программная реализация:

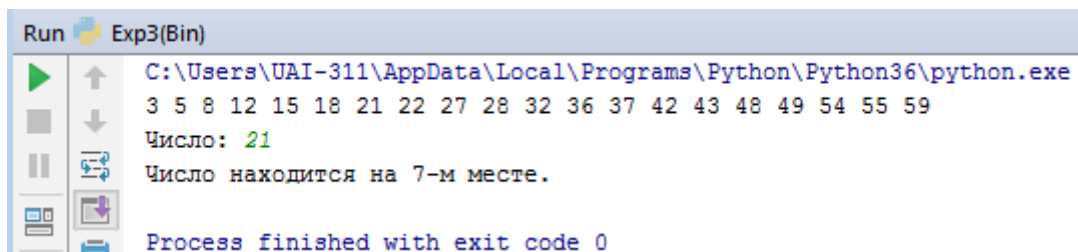
```
N = 20

def srch(c, e):
    m = N // 2
    i = 0
    j = N - 1
    while c[m] != e and i <= j:
        if e > c[m]:
            i = m + 1
        else:
            j = m - 1
        m = (i + j) // 2
    if i > j:
        return 0
    else:
        return m + 1

from random import random

p = 1
q = 4
a = [0] * N
for i in range(N):
    a[i] = int(random() * (q - p)) + p
    p += 3
    q += 3
    print(a[i], end=' ')
print()
e = int(input('Число: '))
i = srch(a, e)
if i == 0:
    print('Такого числа нет.')
else:
    print(f'Число находится на {i}-м месте.')
```

Результат работы программы:



Выводы по итогам урока:

1. Если необходимо произвести однотипные действия с данными одного вида, количество которых конечно и сохранность не важна, то следует использовать массив;

2. Если имеется постоянно изменяемый объем данных, и необходимо периодически производить над ними какие-либо вычисления, то лучше использовать список;
3. Если перед нами стоит задача получить общий анализ данных, которые изменять не нужно, рекомендован кортеж.
4. Если мы располагаем большим объемом неструктурированных данных, объединенных определенными общими свойствами, то стоит использовать множества. Они позволят определить группы (совокупности) данных из общего объема.
5. Все представленные структуры можно так или иначе представить в виде списка – он лежит в основе всего.

Практическое задание

1. В диапазоне натуральных чисел от 2 до 99 определить, сколько из них кратны каждому из чисел в диапазоне от 2 до 9.
2. Во втором массиве сохранить индексы четных элементов первого массива. Например, если дан массив со значениями 8, 3, 15, 6, 4, 2, то во второй массив надо заполнить значениями 1, 4, 5, 6 (или 0, 3, 4, 5 - если индексация начинается с нуля), т.к. именно в этих позициях первого массива стоят четные числа.
3. В массиве случайных целых чисел поменять местами минимальный и максимальный элементы.
4. Определить, какое число в массиве встречается чаще всего.
5. В массиве найти максимальный отрицательный элемент. Вывести на экран его значение и позицию (индекс) в массиве.
6. В одномерном массиве найти сумму элементов, находящихся между минимальным и максимальным элементами. Сами минимальный и максимальный элементы в сумму не включать.
7. В одномерном массиве целых чисел определить два наименьших элемента. Они могут быть как равны между собой (оба являться минимальными), так и различаться.
8. Матрица 5x4 заполняется вводом с клавиатуры кроме последних элементов строк. Программа должна вычислять сумму введенных элементов каждой строки и записывать ее в последнюю ячейку строки. В конце следует вывести полученную матрицу.
9. Найти максимальный элемент среди минимальных элементов столбцов матрицы.

Дополнительные материалы

Ссылка на проекты с практическими примерами:

1. <https://tproger.ru/tag/algorithms/>

Используемая литература

1. <https://www.python.org>
2. Марк Лутц. Изучаем Python, 4-е издание.