# Массивы.

Основное предназначение современных компьютеров – обработка большого количества данных. При этом надо как-то обращаться к каждой из тысяч (или даже миллионов) ячеек с данными. Очень сложно дать каждой ячейке собственное имя и при этом не запутаться. Из этой ситуации выходят так: дают имя не ячейке, а группе ячеек, в которой каждая ячейка имеет собственный номер. Такая область памяти называется массивом (или таблицей).

**Массив** – это группа переменных одного типа, расположенных в памяти рядом (в соседних ячейках) и имеющих общее имя. Каждая ячейка в массиве имеет уникальный номер.

Для работы с массивами нужно, в первую очередь, научиться:

• *выделять память нужного размера под массив;*

• *записывать данные в нужную ячейку;*

• *читать данные из ячейки массива.*

В языке Python нет такой структуры как «массив». Вместо этого для хранения группы одно-типных13 объектов используют списки (тип данных **list**).

**Список в Python** – это набор элементов, каждый из которых имеет свой номер (*индекс*). Нумерация всегда начинается с нуля (как в Си-подобных языках), второй по счёту элемент имеет номер 1 и т.д. В отличие от обычных массивов в большинстве языков программирования список –это динамическая структура, его размер можно изменять во время выполнения программы (удалять и добавлять элементы), при этом все операции по управлению памятью берёт на себя транслятор.

Список можно создать перечислением элементов через запятую в квадратных скобках, например, так:

```
A = [1, 3, 4, 23, 5]
```

Списки можно «складывать» с помощью знака «+», например, показанный выше список можно было построить так:

```
A = [1, 3] + [4, 23] + [5]
```

Сложение одинаковых списков заменяется умножением «*». Вот так создаётся список из 10 эле-ментов, заполненный нулями:

```
A = [0]*10
```

В более сложных случаях используют *генераторы списков* – выражения, напоминающие цикл, с помощью которых заполняются элементы вновь созданного списка:

```
A = [ i for i in range(10) ]
```

Как вы знаете, цикл **for i in range(10)** перебирает все значения **i** от 0 до 9. Выражение перед словом **for** (в данном случае – **i**) – это то, что записывается в очередной элемент списка для каждого **i**. В приведённом примере список заполняется значениями, которые последовательно принимает переменная **i**, то есть получим такой список:**[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**То же самое можно получить, если использовать функцию **list** для того, чтобы создать список изданных, которые получаются с помощью функции **range**:

```
A = list ( range(10) )
```

Для заполнения списка квадратами этих чисел можно использовать такой генератор:

```
A = [ i*i for i in range(10)]
```

В конце записи генератора можно добавить условие отбора. В этом случае в список включаются лишь те из элементов, перебираемых в цикле, которые удовлетворяют этому условию. Например, следующий генератор составляет список из всех простых чисел в диапазоне от 0 до 99:

```
A = [ i for i in range(100) if isPrime(i) ]
```

Здесь **isPrime** – логическая функция, которая определяет простоту числа. Часто в тестовых и учебных программах массив заполняют случайными числами. Это тоже можно сделать с помощью генератора:

```
from random import randint
A = [ randint(20,100) for x in range(10)]
```

Здесь создается массив из 10 элементов и заполняется случайными числами из отрезка [20,100]. Для этого

используется функция **randint**, которая импортируется из модуля **random**. Длина списка (количество элементов в нём) определяется с помощью функции **len**:

```
N = len(A)
```

## Ввод и вывод массива

Далее во всех примерах мы будем считать, что в программе создан список **A**, состоящий из **N** элементов (целых чисел). В этом списке хранится массив данных, поэтому под выражением «массив» мы будем подразумевать «однотипные данные, хранящиеся в виде списка». Переменная **i** будет обозначать индекс элемента списка. Чтобы ввести значения элементов массива с клавиатуры, нужно использовать цикл:

```
for i in range(N):
  print ( "A[", i, "]=", sep = "", end = "" )
  A[i] = int( input() )
```

В этом примере перед вводом очередного элемента массива на экран выводится подсказка. На-пример, при вводе 3-го элемента будет выведено «**A[3]=**».Если никакие подсказки нам не нужны, создать массив из N элементов и ввести их значения можно с помощью генератора списка:

```
A = [ int(input()) for i in range(N) ]
```

Здесь на каждом шаге цикла строка, введённая пользователем, преобразуется в целое число спомощью функции **int**, и это число добавляется к массиву.

Возможен еще один вариант ввода, когда весь массив вводится в одной строке. В этом случае строку, полученную от функции **input**, нужно «расщепить» на части с помощью метода **split**:

```
data = input()
s = data.split()
```

или сразу

```
s = input().split()
```

Например, если ввести строку **"1 2 3 4 5"**, то после «расщепления» мы получим список

**['1', '2', '3', '4', '5']**

Это список символьных строк. Для того, чтобы построить массив (список), состоящий из целых чисел, нужно применить к каждому элементу списка функцию **int**:

```
A = [ int(x) for x in s ]
```

Вместо генератора можно было использовать функцию **map**:

```
A = list( map(int, s) )
```

Такая запись означает «применить функцию **int** ко всем элементам списка **s** и составить из полученных чисел новый список (объект типа **list**)».

Теперь поговорим о выводе массива на экран. Самый простой способ – вывести список как один объект:

```
print ( A )
```

После вывода каждого элемента ставится пробел, иначе все значения сольются в одну строку.

Удобно записывать такой цикл несколько иначе:

```
for x in A:
  print ( x, end = " " )
```

Здесь не используется переменная-индекс **i**, а просто перебираются все элементы списка: на каждом шаге в переменную **x** заносится значение очередного элемента массива (в порядке возрастания индексов).

Более быстрый способ – построить одну символьную строку, содержащую все элементы массива, и сразу вывести её на экран:

```
print ( " ".join( [str(x) for x in A] ) )
```

Функция **join** (англ. *join* – объединить) объединяет символьные строки, используя указанный пе-ред точкой разделитель, в данном случае – пробел. Запись **str(x)** означает «символьная запись **x**». Таким образом, элементы массива записываются через пробел в одну символьную строку, и эта строка затем выводится на экран с помощью функции **print**.

## Перебор элементов

Перебор элементов массива состоит в том, что мы в цикле просматриваем все элементы списка и, если нужно, выполняем с каждым из них некоторую операцию. Переменная цикла изменяется от **0** до **N-1**, где **N** – количество элементов массива, то есть в диапазоне **range(N)**:

```
for i in range(N):
  A[i] += 1
```

в этом примере все элементы массива **A** увеличиваются на 1.

Если массив изменять не нужно, для перебора его элементов удобнее всего использовать такой цикл:

```
for x in A:
  ...
```

Здесь вместо многоточия можно добавлять операторы, работающие с копией элемент, записанной в переменную **x**. Обратите внимание, что изменение переменной **x** в теле цикла не приведёт к изменению соответствующего элемента массива **A**.

Заметим, что для первой задачи (увеличить все элементы массива на 1) есть красивое решение в стиле Python, использующее генератор списка, который построит новый массив:

```
A = [ x+1 for x in A ]
```

Здесь в цикле перебираются все элементы исходного массива, и в новый список они попадают после увеличения на 1. Во многих задачах требуется найти в массиве все элементы, удовлетворяющие заданному условию, и как-то их обработать. Простейшая из таких задач – подсчёт нужных элементов. Для решения этой задачи нужно ввести переменную-счётчик, начальное значение которой равно нулю. Далее в цикле просматриваем все элементы массива. Если для очередного элемента выполняется заданное условие, то увеличиваем счётчик на 1. На псевдокоде этот алгоритм выглядит так:

```
счётчик = 0
for x in A:
  if условие выполняется для x:
    счётчик += 1
```

Предположим, что в массиве **A** записаны данные о росте игроков баскетбольной команды. Найдем количество игроков, рост которых больше 180 см, но меньше 190 см. В следующей программе используется переменная-счётчик **count**:

```
count = 0
for x in A:
  if 180 < x and x < 190:
    count += 1
```

Теперь усложним задачу: требуется найти средний рост этих игроков. Для этого нужно дополнительно в отдельной переменной складывать все нужные значения, а после завершения цикла разделить эту сумму на количество. Начальное значение переменной **Sum**, в которой накапливается сумма, тоже должно быть равно нулю.

```
count = 0
sum = 0
for x in A:
  if 180 < x and x < 190:
    count += 1
    sum += x
print ( sum/count )
```

Суммирование элементов массива – это очень распространённая операция, поэтому длясуммирования элементов списка в Python существует встроенная функция **sum**:

```
print ( sum(A) )
```

С её помощью можно решить предыдущую задачу более элегантно, в стиле языка Python: сначала выделить в дополнительный список все нужные элементы[14], а затем поделить их сумму на количество (длину списка).

Для построения нового списка будем использовать генератор:

```
B = [ x for x in A if 180 < x and x < 190 ]
```

Условие отбора заключено в рамку. Таким образом, мы отбираем в список **B** те элементы из списка A, которые удовлетворяют этому условию. Теперь для вывода среднего роста выбранных игроков остается разделить сумму элементов нового списка на их количество:

```
print ( sum(B) / len(B) )
```

# Алгоритмы обработки массивов

## Поиск в массиве

Требуется найти в массиве элемент, равный значению переменной **X**, или сообщить, что его там нет. Алгоритм решения сводится к просмотру всех элементов массива с первого до последнего. Как только найден элемент, равный **X**, нужно выйти из цикла и вывести результат. Напрашивается такой алгоритм:

```
i = 0
while A[i] != X:
  i += 1
print ( "A[", i, "]=", X, sep = "" )
```

Он хорошо работает, если нужный элемент в массиве есть, однако приведет к ошибке, если такого элемента нет – получится зацикливание и выход за границы массива. Поэтому в условие нужно добавить еще одно ограничение: **i< N**. Если после окончания цикла это условие нарушено, значит поиск был неудачным – элемента нет:

```
i = 0
while i < N and A[i] != X:
  i += 1
if i < N:
  print ( "A[", i, "]=", X, sep = "" )
else:
  print ( "Не нашли!" )
```

Отметим одну тонкость. В сложном условии **i < N** и **A[i]** != **X** первой должно проверяться именно отношение **i < N**. Если первая часть условия, соединенного с помощью операции «И», ложно, то вторая часть, как правило[15], не вычисляется – уже понятно, что всё условие ложно. Дело в том, что если **i >= N**, проверка условия **A[i]** != **X** приводит к *выходу за границы массива*, и программа завершается аварийно.

Возможен ещё один поход к решению этой задачи: используя цикл с переменной, перебрать все элементы массива и досрочно завершить цикл, если найдено требуемое значение.

```
nX = -1
for i in range ( len(A) ):
  if A[i] == X:
    nX = i
    break
if nX >= 0:
  print ( "A[", nX, "]=", X, sep = "" )
else:
  print ( "Не нашли!" )
```

Для выхода из цикла используется оператор **break**, номер найденного элемента сохраняется в переменной **nX**. Если её значение осталось равным –1 (не изменилось в ходе выполнения цикла), то в массиве нет элемента, равного **X**.

Последний пример можно упростить, используя особые возможности цикла **for** в языке Python:

```
for i in range ( len(A) ):
  if A[i] == X:
    print ( "A[", i, "]=", X, sep = "" )
    break
else:
  print ( "Не нашли!" )
```

Итак, здесь мы выводим результат сразу, как только нашли нужный элемент, а не после цикла. Слово **else** после цикла **for** начинает блок, который выполняется при нормальном завершении цикла (без применения **break**). Таким образом, сообщение «Не нашли!» будет выведено только тогда, когда условный оператор в теле цикла ни разу не сработал.

Возможен другой способ решения этой задачи, использующий метод (функцию) **index** для типа данных **list**, которая возвращает номер первого найденного элемента, равного **X**:

```
nX = A.index(X)
```

Тут проблема только в том, что эта строчка вызовет ошибку при выполнении программы, если нужного элемента в массиве нет. Поэтому нужно сначала проверить, есть ли он там (с помощью оператора **in**), а потом использовать метод **index**:

```
if X in A:
  nX = A.index(X)
  print ( "A[", nX, "]=", X, sep = "" )
else:
  print ( "Не нашли!" )
```

Запись «**if X in A**» означает «если значение **X** найдено в списке **A**».

## Максимальный элемент

Найдем в массиве максимальный элемент. Для его хранения выделим целочисленную переменную **M**. Будем в цикле просматривать все элементы массива один за другим. Если очередной элемент массива больше, чем максимальный из предыдущих (находящийся в переменной **M)**, запомним новое значение максимального элемента в **M**.

Остается решить, каково должно быть начальное значение **M**. Во-первых, можно записать туда значение, заведомо меньшее, чем любой из элементов массива. Например, если в массиве записаны натуральные числа, можно записать в **M** ноль или отрицательное число. Если содержи-мое массива неизвестно, можно сразу записать в **M** значение **A[0]**, а цикл перебора начать со второго счёту элемента, то есть, с **A[1]**:

```
M = A[0]
for i in range(1,N):
  if A[i] > M:
    M = A[i]
print ( M )
```

Вот еще один вариант:

```
M = A[0]
for x in A:
  if x > M:
    M = x
```

Он отличается тем, что мы не используем переменную-индекс, но зато дважды просматриваем элемент **A[0]** (второй раз – в цикле, где выполняется перебор *всех* элементов).Поскольку операции поиска максимального и минимального элементов нужны очень часто, в Python есть соответствующие встроенные функции **max** и **min**:

```
Ma = max ( A )
Mi = min ( A )
```

Теперь предположим, что нужно найти не только значение, но и номер максимального элемента. Казалось бы, нужно ввести еще одну переменную **nMax** для хранения номера, сначала за-писать в нее 0 (считаем элемент **A[0]** максимальным) и затем, когда нашли новый максимальный элемент, запоминать его номер в переменной **nMax**

```
M = A[0]; nMax = 0
for i in range(1,N):
  if A[i] > M:
    M = A[i]
    nMax = i
print ( "A[", nMax, "]=", M, sep = "" )
```

Однако это не самый лучший вариант. Дело в том, что по номеру элемента можно всегда определить его значение. Поэтому достаточно хранить только номер максимального элемента. Если этот номер равен **nMax**, то значение максимального элемента равно **A[nMax]**:

```
nMax = 0
for i in range(1,N):
  if A[i] > A[nMax]:
    nMax = i
print ( "A[", nMax, "]=", A[nMax], sep = "" )
```

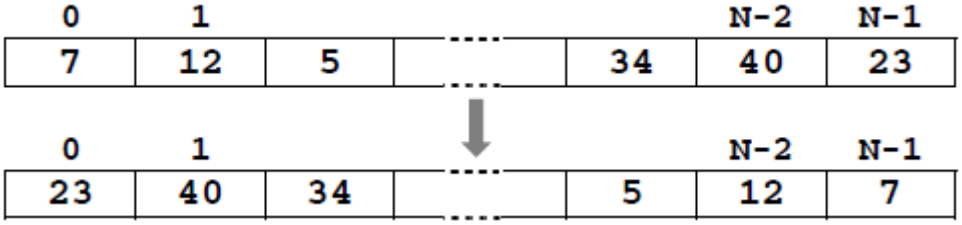
Для решения этой задачи можно использовать встроенные функции Python: сначала найти максимальный элемент, а потом его индекс с помощью функции **index**:

```
M = max(A)
nMax = A.index(M)
print ( "A[", nMax, "]=", M, sep = "" )
```

В этом случае фактически придётся выполнить два прохода по массиву. Однако такой вариант работает во много раз быстрее, чем «рукописный» цикл с одним проходом, потому что встроенные функции написаны на языке C++ и подключаются в виде готового машинного кода, а не выполняются относительно медленным интерпретатором Python.

## Реверс массива

Реверс массива – это перестановка его элементов в обратном порядке: первый элемент становится последним, а последний – первым.




Из рисунка следует, что 0-й элемент меняется местами с (**N-1**)-м, второй – с (**N-2**)-м и т.д. Сумма индексов элементов, участвующих в обмене, для всех пар равна **N-1**, поэтому элемент с номером **i** должен меняться местами с (**N-1-i**)-м элементом. Кажется, что можно написать такой цикл:

```
for i in range(N):
  поменять местами A[i] и A[N-1-i]
```

однако это неверно. Посмотрим, что получится для массива из четырёх элементов:

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| | 7 | 12 | 40 | 23 |
| A[0]↔A[3] | 23 | 12 | 40 | 7 |
| A[1]↔A[2] | 23 | 40 | 12 | 7 |
| A[2]↔A[1] | 23 | 12 | 40 | 7 |
| A[3]↔A[0] | 7 | 12 | 40 | 23 |

Как видите, массив вернулся в исходное состояние: реверс выполнен дважды. Поэтому нужно остановить цикл на середине массива:

```
for i in range(N//2):
  поменять местами A[i] и A[N-1-i]
```

Для обмена можно использовать вспомогательную переменную **c**:

```
for i in range(N//2):
  c = A[i]
  A[i] = A[N-1-i]
  A[N-1-i] = c
```

или возможности Python:

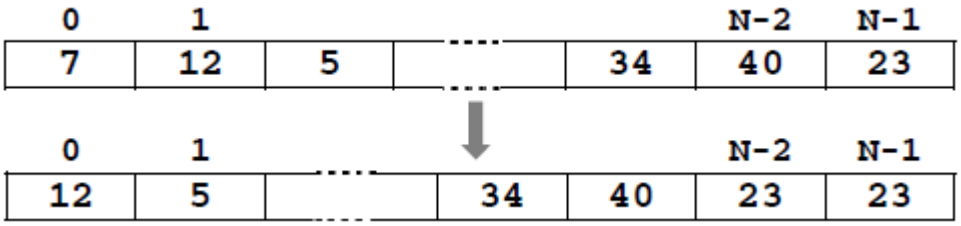
```
for i in range(N//2):
  A[i], A[N-i-1] = A[N-i-1], A[i]
```

Эта операция может быть выполнена и с помощью стандартного метода **reverse** (в переводе с англ. – реверс, обратный) типа **list**:

```
A.reverse()
```

## Сдвиг элементов массива

При удалении и вставке элементов необходимо выполнять сдвиг части или всех элементов массива в ту или другую сторону. Массив часто рисуют в виде таблицы, где первый элемент расположен слева. Поэтому сдвиг влево – это перемещение всех элементов на одну ячейку, при ко-тором **A[1]** переходит на место **A[0]**, **A[2]** – на место **A[1]** и т.д.




Последний элемент остается на своем месте, поскольку новое значение для него взять неоткуда –массив кончился. Алгоритм выглядит так:

```
for i in range(N-1):
  A[i] = A[i+1]
```

Обратите внимание, что цикл заканчивается при **i=N-2** (а не **N-1**), чтобы не было выхода за границы массива, то есть обращения к несуществующему элементу **A[N]**. При таком сдвиге первый элемент пропадает, а последний – дублируется. Можно старое значение первого элемента записать на место последнего. Такой сдвиг называется *циклическим*. Предварительно (до начала цикла) первый элемент нужно запомнить во вспомогательной переменной, а после завершения цикла записать его в последнюю ячейку массива:

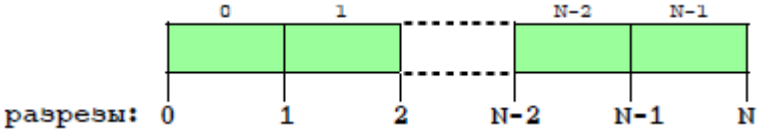
```
c = A[0]
for i in range(N-1):
  A[i] = A[i+1]
A[N-1] = c
```

Ещё проще выполнить такой сдвиг, используя встроенные возможности списков Python:

```
A = A[1:N] + [A[0]]
```

Здесь использован так называемый *срез* – выделение части массива. Срез **A[1:N]** означает «все элементы с **A[1]** до **A[N-1]**», то есть не включая элемент с последним индексом. Таким образом, этот срез «складывается» со списком, состоящим из одного элемента **A[0]**, в результате получается новый массив, составленный из «списков-слагаемых».

Чтобы такая система (исключение последнего элемента) была более понятной, можно представить, что срез массива выполняется по *разрезам* – границам между элементами:




Таким образом, срез **A[0:2]** выбирает все элементы между разрезами 0 и 2, то есть элементы **A[0]** и **A[1]**.Если срез заканчивается на последнем элементе массива, второй индекс можно не указывать:

```
A = A[1:] + [A[0]]
```

Аналогично, **A[:5]** обозначает «первые 5 элементов массива» (начальный индекс не указан). Если не указать ни начальный, ни конечный индекс, мы получим копию массива:

```
Acopy = A[:]
```

Если использованы отрицательные индексы, к ним добавляется длина массива. Например, срез **A[:-1]** выбирает все элементы, кроме последнего (он равносилен **A[:N-1]**). А вот так можно выделить из массива три последних элемента:

```
B = A[-3:]
```

Заметим, что с помощью среза можно, например, выполнить реверс массива:

```
A = A[::-1]
```

Рассмотрим подробно правую часть оператора присваивания. Число «–1» обозначает шаг выборки значений, то есть, элементы выбираются в обратном порядке. Слева от первого и второго знаков двоеточия записывают начальное и конечное значения индексов; если они не указаны, считается, что рассматривается весь массив.

## Отбор нужных элементов

Требуется отобрать все элементы массива **A**, удовлетворяющие некоторому условию, в новый массив **B**. Поскольку списки в Python могут расширяться во время работы программы, можно использовать такой алгоритм: сначала создаём пустой список, затем перебираем все элементы исходного массива и, если очередной элемент нам нужен, добавляем его в новый список:

```
B = []
for x in A:
  if x % 2 == 0:
    B.append(x)
```

Здесь для добавления элемента в конец списка использован метод **append**.
Второй вариант решения – использование генератора списка с условием.

```
B = [x for x in A if x % 2 == 0 ]
```

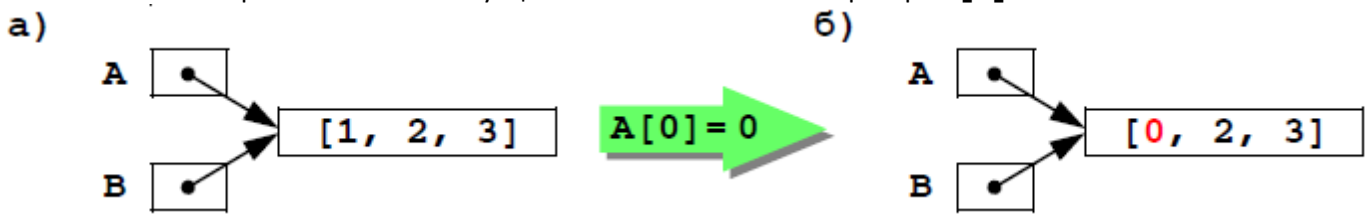
В цикле перебираются все элементы массива **A**, и только чётные из них включаются в новый массив.

## Особенности копирования списков в Python

Имя переменной в языке Python связывается с объектом в памяти: числом, списком и др. При выполнении операторов

```
A = [1, 2, 3]
B = A
```

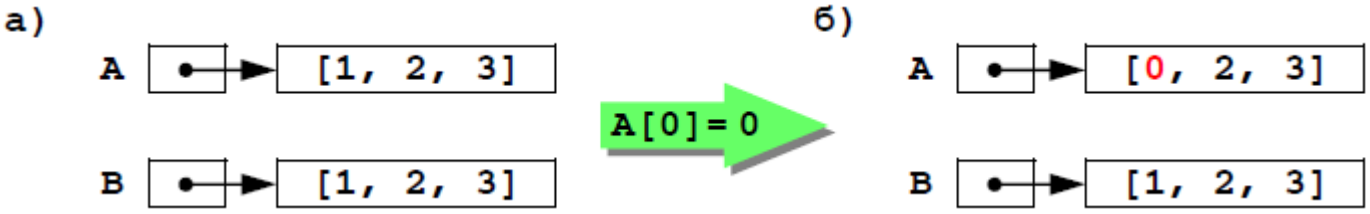
две переменные **A** и **B** будут связаны с одним и тем же списком (рис. а), поэтому при изменении одного списка будет изменяться и второй, ведь это фактически один и тот же список, к которому можно обращаться по двум разным именам. На рис. б показана ситуация после выполнения оператора **A[0] = 0**:




Эту особенность Python нужно учитывать при работе со списками. Если нам нужна именно копия списка (а не ещё одна ссылка на него), можно использовать срез, строящий копию:

```
B = A[:]
```

Теперь A и **B** – это независимые списки и изменение одного из них не меняет второй:




Вместо среза можно было использовать функцию **copy** из модуля **copy**:

```
import copy
A = [1,2,3]
B = copy.copy(A)
```

Это так называемая «поверхностная» копия – она не создаёт полную копию, если список содержит какие-то изменяемые объекты, например, другой список. Для полного копирования используется функция **deepcopy** из того же модуля:

```
import copy
A = [1,2,3]
B = copy.deepcopy(A)
```