# Python.

## Списки. Сортировка.

### Копирование списков.

Давайте рассмотрим такой пример.

```
a = [1, 2, 3, 4, 5, 6]
b = a
for i in range(len(b)):
    b[i] = 2 * b[i]
print(b)
print(a)
```

Получим

[2, 4, 6, 8, 10, 12]
[2, 4, 6, 8, 10, 12]

Как можно видеть, изменился не только список ***b***, но и список ***a***. В чем дело? Оказывается, если присвоить один список другому, не происходит копирования списков, а тот же список просто получает второе имя.

Что делать, если все-таки необходимо получить копию списка. Один из способов – это создать новый список, и в цикле поэлементно присвоить значения одного списка другому. Однако это далеко не самый эффективный способ. Гораздо удобнее воспользоваться методом copy(). Исправим предыдущий пример

```
a = [1, 2, 3, 4, 5, 6]
b = a.copy()
for i in range(len(b)):
    b[i] = 2 * b[i]
print(b)
print(a)
```

Получим

[2, 4, 6, 8, 10, 12]
[1, 2, 3, 4, 5, 6]

В данном случае список ***b*** стал копией списка ***a***. И изменение списка ***b*** не затронуло список ***a***. Так же срезы возвращают копии списка. Можно было написать и так: ***b = a[:]***

### Функции высшего порядка

В языке Python существуют функции, которые принимают или/и возвращают другие функции. Такие функции называются **функциями высшего порядка.** Рассмотрим работу некоторых из них.

### Встроенная функция map()

Встроенная функция map() имеет сигнатуру map(func, *iterables).

В качестве параметра func указывается функция, которой будет передаваться текущий элемент последовательности. Внутри функции func необходимо вернуть новое значение. Для примера прибавим к каждому элементу списка число 7

Если запустить следующую программу

```
def increase(num):
    return num + 7

numbers = [1, 2, 3, 4, 5, 6]
new_numbers = map(increase, numbers)
print(new_numbers)
```

то будет напечатано

<map object at 0x...>

То есть вместо списка выводится некий специальный объект. Такой объект похож на список тем, что его можно перебирать циклом for, то есть итерировать. Такие объекты в Python называют **итераторами**.

Следующая программа

```
def increase(num):
    return num + 7

numbers = [1, 2, 3, 4, 5, 6]
new_numbers = map(increase, numbers)
for num in new_numbers: # итерируем циклом for
    print(num)
```

выведет

8
9
10
11
12
13

Чтобы получить из итератора список, нужно воспользоваться функцией list():

```
new_numbers = list(map(increase, numbers))
```

Функция map() возвращает объект, поддерживающий итерации, а не список. Чтобы получить из него список, необходимо результат передать в функцию list().

Очень удобно использовать функцию map(), например, когда необходимо считать числа из вводимой строки, если они записаны через пробел.

```
a = list(map(int, input().split()))
print(a)
```

Если при выполнении этой программы ввести с клавиатуры следующую строку

5 23 7 45 12

То будет напечатано

[5, 23, 7, 45, 12]

Как видим, метод ***split()*** разбил введенную строку на отдельные слова, функция ***map()*** перевела каждый элемент в число, а функция ***list()*** собрала все в список.

### Встроенная функция filter()

Встроенная функция filter() имеет сигнатуру filter(func, iterable).

В качестве параметра func указывается ссылка на функцию, которой будет передаваться текущий элемент последовательности. Внутри функции func необходимо

вернуть значение True или False. Для примера, удалим все отрицательные значения из списка.

```
def func(elem):
    return elem >= 0

numbers = [-1, 2, -3, 4, 0, -20, 10]
positive_numbers = list(filter(func, numbers)) # преобразуем итератор в список
print(positive_numbers)
```

Получим

[2, 4, 0, 10]

Обратите внимание: функция filter() как и функция map() возвращает не список, а специальный объект, который называется итератором. Итераторы можно перебрать с помощью цикла for, либо преобразовать в список.

## Сортировка списков

Задача сортировки списка заключается в перестановке его элементов так, чтобы они были упорядочены по возрастанию или убыванию. Это одна из основных задач программирования. Мы сталкиваемся с ней очень часто: при записи фамилий учеников в классном журнале, при подведении итогов соревнований и т.д.

## Алгоритмы сортировки

**Алгоритм сортировки** — это алгоритм упорядочивания элементов в списке. Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

- время — основной параметр, характеризующий быстродействие алгоритма;
- память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных.

Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к **сортировкам на месте**.

Рассмотрим некоторые алгоритмы сортировки.

## Сортировка пузырьком

Алгоритм сортировки пузырьком состоит из повторяющихся проходов по сортируемому списку. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по списку повторяются n-1 раз, где n – длина списка. При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент списка ставится на свое место в конце списка рядом с предыдущим «наибольшим элементом».

Наибольший элемент каждый раз «всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма.

Рассмотрим работу алгоритма на примере сортировки списка `a = [5, 1, 4, 2, 8]` по возрастанию.

**Первый проход:**

1. `[5, 1, 4, 2, 8]` → `[1, 5, 4, 2, 8]`: меняем местами первый и второй элементы, так как 5>1;
2. `[1, 5, 4, 2, 8]` → `[1, 4, 5, 2, 8]`: меняем местами второй и третий элементы, так как 5>4;
3. `[1, 4, 5, 2, 8]` → `[1, 4, 2, 5, 8]`: меняем местами третий и четвертый элементы, так как 5>2;
4. `[1, 4, 2, 5, 8]` → `[1, 4, 2, 5, 8]`: не меняем четвертый и пятый элементы местами, так как 5<8;
5. Самый большой элемент встал («всплыл») на свое место.

**Второй проход:**

1. `[1, 4, 2, 5, 8]` → `[1, 4, 2, 5, 8]`: не меняем первый и второй элементы местами, так как 1<4;
2. `[1, 4, 2, 5, 8]` → `[1, 2, 4, 5, 8]`: меняем местами второй и третий элементы, так как 4>2;
3. `[1, 2, 4, 5, 8]` → `[1, 2, 4, 5, 8]`: не меняем местами третий и четвертый элементы, так как 4 <5;
4. Второй по величине элемент встал («всплыл») на свое место.

Теперь список полностью отсортирован, но алгоритму это неизвестно и он работает дальше.

**Третий проход:**

1. `[1, 2, 4, 5, 8]` → `[1, 2, 4, 5, 8]`: не меняем первый и второй элементы местами, так как 1<2;
2. `[1, 2, 4, 5, 8]` → `[1, 2, 4, 5, 8]`: не меняем второй и третий элементы местами, так как 2 <4;
3. Третий по величине элемент встал («всплыл») на свое место. (на котором и был)

**Четвертый проход:**

1. `[1, 2, 4, 5, 8]` → `[1, 2, 4, 5, 8]`:
2. Четвертый по величине элемент встал («всплыл») на свое место.

Теперь список отсортирован и алгоритм может быть завершен.

Пусть требуется отсортировать по возрастанию список чисел: a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99].

Следующий программный код реализует алгоритм пузырьковой сортировки:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99]
n = len(a)

for i in range(n - 1):
    for j in range(n - i - 1):
        if a[j] > a[j + 1]:       # если порядок элементов пары неправильный
            a[j], a[j + 1] = a[j + 1], a[j]  # меняем элементы пары местами
print('Отсортированный список:', a)
```

Результатом выполнения такого кода будет:

```
Отсортированный список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]
```

## Сортировка простым выбором

Сортировка выбором улучшает пузырьковую сортировку, совершая всего **один обмен за каждый проход по списку**. Для этого алгоритм ищет максимальный элемент и помещает его на соответствующую позицию. Как и для пузырьковой сортировки, после первого прохода самый большой элемент находится на правильном месте. После второго прохода на своё место становится следующий максимальный элемент. Проходы по списку повторяются $n-1$ раз, где $n$ – длина списка, поскольку последний из них автоматически оказывается на своем месте.

Рассмотрим работу алгоритма на примере сортировки списка `a = [5, 1, 8, 2, 4]` по возрастанию.

**Первый проход:**

Находим максимальный элемент `8` в неотсортированной части списка и меняем его с последним элементом списка:

`[5, 1, 4, 2, 8]`.

**Второй проход:**

Находим максимальный элемент `5` в неотсортированной части списка и меняем его с предпоследним элементом списка:

`[2, 1, 4, 5, 8]`.

**Третий проход:**

Находим максимальный элемент `4` в неотсортированной части списка и меняем его с пред-предпоследним элементом списка:

`[2, 1, 4, 5, 8]`.

**Четвертый проход:**

Находим максимальный элемент `2` в неотсортированной части списка и меняем его с вторым элементом списка:

`[1, 2, 4, 5, 8]`.

Теперь список отсортирован и алгоритм может быть завершен.

Вместо максимального элемента можно искать минимальный. Так же можно размещать найденные элементы не с конца списка, а с начала.

Следующий программный код реализует алгоритм сортировки методом простого выбора:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99]

def findmin(x, start):
    mn = start
    for i in range(start + 1, len(x)):
        if x[i] < x[mn]:
            mn = i
    return mn

n = len(a)
for i in range(n - 1):
    k=findmin(a,i) # находим номер минимального эл-та, начиная с текущего
```

```
    a[i], a[k] = a[k], a[i]  # меняем минимальный с текущим эл-том
print('Отсортированный список:', a)
```

### Метод sort()

В Python списки имеют встроенный метод sort(), который сортирует элементы списка по возрастанию или убыванию.

Следующий программный код:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99]
a.sort()
print('Отсортированный список:', a)
```

выведет:

```
Отсортированный список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]
```

По умолчанию метод sort() сортирует список по возрастанию. Если требуется отсортировать список по убыванию, необходимо явно указать параметр reverse = True.

Следующий программный код:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99]
a.sort(reverse=True)  # сортируем по убыванию
print('Отсортированный список:', a)
```

выведет:

```
Отсортированный список: [1000, 99, 45, 34, 12, 9, 8, 7, 6, 1, 0, -2, -3, -67]
```