
Яндекс

**Лицей Академии Яндекса**

# Обработка коллекций. Потоковый ввод sys.stdin

# Итерируемые объекты. Почему filter и map возвращают не список

# Почему filter и map возвращают не список

**Задача**: обработать очень большое количество информации. Например, список из миллиарда чисел (он занимает не меньше 4 гигабайт памяти) для поиска суммы квадратов всех этих чисел. Есть несколько вариантов:

1. Пройтись циклом **for** и посчитать сумму руками. Это просто, но не слишком элегантно и вообще не python-style. (Этот путь даже рассматривать не будем)
2. Использовать функцию **sum** и списочное выражение
3. Использовать **map**

# Почему filter и map возвращают не список

Решение со списочным выражением использует очень много памяти, так как сначала строит список всех квадратов, а затем уже считает их сумму (можно посмотреть в диспетчере задач как съедается память во время работы программы):

```
print(sum([x ** 2 for x in range(50 * 1000 * 1000)]))
```

При определенных настройках python можно даже получить ошибку **MemoryError**, так как такое вычисление может потребовать больше памяти, чем способно выделить устройство, на котором выполняется программа

# Почему filter и map возвращают не список

Решение с фунцией map позволяет нам вычислять значение квадратов чисел «на лету» в тот момент, когда они нам нужны для подсчета суммы:

```
print(sum(map(lambda x: x ** 2, range(50 * 1000 * 1000))))
```

Таким образом мы можем избежать использование дополнительного объема оперативной памяти

# Почему filter и map возвращают не список

Упрощенно говоря, есть два типа итерируемых объектов:

1. Итераторы, которые позволяют перебирать элементы. Они не хранят все значения элементов, им нужно помнить только начало промежутка, его конец и текущий элемент.
2. Коллекции (списки, строки, словари и т.д.), которые позволяют создать итератор по своим элементам.

# Функции max/min/sorted и использование ключа сортировки

# Использование ключа сортировки

У функций вроде **min/max/sorted** есть опциональный (необязательный) параметр **key**. Параметр **key** принимает функцию, по значению которой будут сравниваться элементы.

```
words = ['мир', 'и', 'война']
print(sorted(words)) # => ['война', 'и', 'мир’]
```

Если параметр **key** не указан, то строки сортируются в лексикографическом порядке, но мы можем указать, каким образом проводить сортировку. Например по длине строки:

```
print(sorted(words, key=lambda s: len(s)))
# => ['и', 'мир', 'война']
```

# Использование ключа сортировки

Можно проводить сортировку по нескольким критериям, для этого функция для ключа сортировки должна возвращать кортеж значений. Например, отсортируем список сначала по критерию, что последний символ – цифра, а затем по длине строки.

```
words = "А он сделал ход с E2 на E4".split()
print(sorted(
                words,
                key=lambda s: (0 if s[-1].isdigit() else 1, len(s))
      ))
```

# Проверка коллекций: all, any

# Проверка коллекций: all, any

Есть встроенные в python функции для проверки коллекций **all** и **any**. Первая проверяет, что все элементы переданного ей итерируемого набора значений истинны (приводятся к **True**). Вторая проверяет, что есть хотя бы один такой элемент.

# Проверка коллекций: all, any

```
a = [1, 2, 3, "1", [1, 2], True]
b = [[], 1, 2, 3, 4]
c = [None, 0, "", [], set(), {}, False]
d = [None, 0, "", [], set(), {}, False, 99]

print(all(a)) # => True
print(all(b)) # => False
print(all(c)) # => False
print(all(d)) # => False
print(any(a)) # => True
print(any(b)) # => True
print(any(c)) # => False
print(any(d)) # => True
```

# Потоковый ввод stdin

# Потоковый ввод sys.stdin

Поток ввода (sys.stdin) — это специальный итерируемый объект в программе, куда попадает весь текст, который ввёл пользователь. Потоком его называют потому, что данные хранятся там до тех пор, пока программа их не считала (например, с помощью функции input()).

sys.stdin — пример итератора, который невозможно перезапустить. Как и любой итератор, он может двигаться только вперёд. Но если для списка можно сделать второй итератор, который начнёт чтение с начала списка, то с потоком ввода такое не пройдёт. Как только данные прочитаны, они удаляются из потока ввода безвозвратно.

Но, если неизвестно, в какой момент надо прекратить ввод, то воспользоваться функцией input() не удастся. В таких случаях остаётся только работать с sys.stdin.

# Потоковый ввод sys.stdin

Чтобы работать со стандартным потоком ввода надо сначала импортировать модуль **sys** (обычно все импорты делаются в самом начале программы)

```
import sys
```

После чего мы можем воспользоваться объектом **stdin** этого модуля

```
import sys
for line in sys.stdin:
    print(line.rstrip('\n'))
```

**rstrip** – делается для того чтобы «отрезать» символ перевода строки

# Потоковый ввод sys.stdin

С помощью **sys.stdin** можно «в одну строку» прочитать весь ввод (о количестве строк которого мы ничего не знаем) в список. Реализуется это, например, так:

```
data = list(map(str.strip, sys.stdin))
```

Можно считать все строки (с сохранением символов перевода строки) в список вот таким образом:

```
data = sys.stdin.readlines()
```

А считать многострочный текст из стандартного потока ввода в текстовую переменную можно вот так:

```
str_data = sys.stdin.read()
```


Яндекс