# Методичка по **Python 3** (v1.31)

R314 & TL

15 декабря 2017 г.

# Оглавление

# Глава 1

# Ввод, вывод и переменные

## 1.1 Простейшая программа

```
1 print ('Hello, World!')
```

Данный код выводит на экран строку “Hello, World!”.
Функция `print()` выводит на экран свой аргумент (то, что дано в скобках).
Код на Питоне сохраняется в файле с расширением `.py` и запускается из консоли одной из следующих команд:

```
1 > python program.py
2 Hello, World!
```

```
1 > py -3 program.py
2 Hello, World!
```

## 1.2 Условные операторы и циклы

### 1.2.1 Условный оператор

Алгоритмическая конструкция “если *условие*, то *делай действия*, иначе *делай другие действия*” описывается в Питоне с помощью `if-else`. Условие в условном операторе пишется в скобках, и в случае, если условие — истина (или не ноль), выполняется действие, указанное сразу после оператора. Опционально можно после действия для `if` добавить команду `else`, в таком случае если условие в условном операторе — ложь (или ноль), выполнится действие, указанное сразу после `else`.
В качестве условия можно использовать несколько выражений, объединённых логическими операторами. Оператор `and` (логическое “И”) даёт истину только тогда, когда применяется к двум выражениям, также дающим истину. Оператор `or` (логическое “ИЛИ”) даёт ложь только тогда, когда применяется к двум выражениям, также дающим ложь. Оператор `not` (логическое “НЕ”) меняет истину на ложь, и наоборот.

Пример: проверить, что число $a$ входит в диапазон значений [5, 10]:

```
if(a >= 5 and a <= 10):
    print("YES")
else:
    print("NO")
```

```
if(a < 5 or a > 10):
    print("NO")
else:
    print("YES")
```

### 1.2.2 Цикл с предусловием

Если команда или команды в условном операторе должны выполняться не один раз, а много раз, пока условие возвращает истину, примеряют оператор `while`. Проверив условие на истинность, оператор выполнит команду после `while`, а затем снова проверит условие, и снова выполнит команду, и так далее до тех пор, пока условие не станет ложным.
Пусть надо вывести числа то 1 до 10 (см. следующую секцию Форматированный вывод):

```
i = 1
while(i <= 10):
    print(i, end = ' ')
    i += 1
```

### 1.2.3 Перебор элементов множества

Цикл, в каждой итерации которого переменная ссылается на следующий элемент множества, пока они не закончатся. Для перебора последовательностей чисел можно создавать множества с помощью функции `range()` (см. документацию); некоторые множества символов содержатся в библиотеке `string` (см. документацию).
С помощью `for` код из предыдущего примера можно записать в две строки:

```
for i in range(1, 11):
    print(i, end = ' ')
```

### 1.2.4 Прерывание цикла

Особые команды `break` и `continue` позволяют прервать ход выполнения цикла изнутри цикла. Команда `break` моментально завершает выполняющийся цикл (только один!), а команда `continue` прерывает текущую итерацию и переходит к следующей.
Эти операции полезны, когда условием выполнения цикла нельзя полностью описать всю логику алгоритма. Они применяются, например, когда неизвестно, в какой момент внутри цикла выполнится нужное условие.

Пусть надо возводить в квадрат вводимые числа до тех пор, пока на ввод не будет дан ноль:

```
while(1 == 1):
    a = int(input())
    if(a == 0):
        break
    print(a*a)
```

### 1.2.5 “Иначе” после циклов!

Оператор `else` можно использовать не только после `if`, но и после `while` и `for`. Команды в блоке `else` будут выполняться в случае завершения цикла по условию или при завершении обхода элементов множества, но **не будут** в случае завершения цикла по команде `break`.

## 1.3 Форматированный вывод

### 1.3.1 Строки с подстановкой

При выводе функцией `print()` строки можно подставлять в неё значения. В том месте, где в выводимую строку нужно вставить нужное значение, пишется оператор `%` и так называемый спецификатор (для целых чисел это буква `d`). После самой строки ставится знак `%`, а потом в скобках через запятую перечисляются все подставляемые значения **в том же порядке, в котором они должны быть подставлены в строку**.

```
print ('%d + %d = %d' % (3, 5, 3+5))
print ('And %d * %d = %d' % (3, 5, 3*5))
```

Выполнение программы

```
> python program.py
3 + 5 = 8
And 3 * 5 = 15
```

Список некоторых спецификаторов для различных типов данных:

| | |
|---|---|
| `d, i` | Знаковое целое десятичное число |
| `o` | Знаковое целое восьмеричное число |
| `x, X` | Знаковое шестнадцатеричное число |
| `f, F` | Вещественное число |
| `c` | Символьная переменная |
| `s` | Строка |

Пример использования различных спецификаторов:

```
print ('My name is %s.\nI am %i and love the number %12.4f' %
                                          ('Ruslan', 22, 3.1415926))
print ('My name is %s.\nI am %i and love the number %2.10f' %
                                          ('Ruslan', 22, 3.1415926))
```

(Замечание: на самом деле переносов строки между % и перечислением переменных быть не должно. Они добавлены, чтобы код вмещался в страницу)

Выполнение программы:

```
> python program.py
My name is Ruslan.
I am 22 and love the number       3.1416
My name is Ruslan.
I am 22 and love the number 3.1415926000
```

Отдельное внимание стоит уделить выводу вещественных чисел. В этом примере между `%` и спецификатором `f` стоят дополнительные параметры. Число до точки — это минимальное количество символов, которое должно занимать выводимое число целиком. “Лишние” символы забиваются пробелами. Число после точки — это количество знаков после запятой, которое должно быть у выводимого числа. Этот параметр не просто обрезает число, а округляет его по всем правилам (поэтому в выводе число `3.1416`, а не `3.1415`).

Эти параметры необязательны для использования, в таком случае спецификатор будет выглядеть просто как `%f`. Если же используется только один из параметров, нужно обязательно ставить точку: `%.2f`, `%5.f`.

Вместо известных значений в качестве параметров в функцию `print()` можно подставлять переменные:

```
a = 3
b = 5
print ('%d + %d = %d' % (a, b, a+b))
print ('And %d * %d = %d' % (a, b, a*b))
```

Выполнение программы:

```
> python program.py
3 + 5 = 8
And 3 * 5 = 15
```

### 1.3.2 Перечисление аргументов

У функции `print()` есть ещё один “режим работы” — вывод перечисленных через запятую аргументов с определённым разделителем `sep` и концом вывода `end` (строковые переменные). По умолчанию (если не задан) `sep` — пробел, а `end` — перевод строки.

```
print (1, 2, 3, 4, 5)
print (1, 2, 3, 4, 5, sep = '~', end = ' ^___^')
```

Выполнение программы:

```
> python program.py
1 2 3 4 5
1~2~3~4~5 ^___^
```

## 1.4 Ввод

### 1.4.1 “Сырой” ввод

Считывание ввода в Питоне осуществляется функцией `input()`. Эта функция возвращает **все** символы из ввода до ближайшего перевода строки.

```
name = input()
print ("Hello", name)
```

Выполнение программы:

```
> python program.py
Tanya
Hello, Tanya
```

В этом примере после запуска программы она ожидает, пока пользователь не введёт что-то (может быть, и ничего) и не нажмёт клавишу `Enter` (строка 2 в листинге выполнения программы). После этого программа переходит к выполнению следующей команды.

### 1.4.2 Парсинг

Функция `input()` считывает весь ввод до перевода строки в одну строку, поэтому если какие-нибудь значения должны вводиться через пробел или любой другой разделитель, ввод приходится сначала разобрать (распарсить). Применяемый к строке (а `input()` возвращает строку) метод `split()` позволяет поделить строку на части в соответствии с заданным разделителем. Если разделитель не задан, он по умолчанию считается пробелом.

Функция `split()` возвращает структуру данных **список**.

```
name1, name2 = input().split()
print ("Hello, ", name1, ", I'm ", name2, sep = '')
```

Выполнение программы:

```
> python program.py
Ruslan Tanya
Hello, Ruslan, I'm Tanya
```

Если нам известно, сколько именно строк через разделитель введёт пользователь, можно считывать их как в примере выше. Можно также просто записывать результат в список, а потом выводить его сожержимое с помощью оператора цикла `for`, который перебирает все значения данного списка `a`, подставляя их по очереди в переменную `x` (списки и переменные можно называть как угодно).

```
a = input().split()
for x in a:
    print(x)
```

Выполнение программы:

```
> python program.py
Leonardo Donatello Raphael Michaelangelo
Leonardo
Donatello
Raphael
Michaelangelo
```

### 1.4.3 Файловый ввод-вывод

Давайте рассмотрим ещё пару примеров ввода и вывода с добавлением осмысленной манипуляции данными — сортировки. Встроенная функция сортировки `sort()` упорядочивает элементы списка в порядке возрастания, а в случае строк — в лексикографическом порядке (как в словаре).


```
a = map(int, input().split())
a.sort()
for x in a:
    print(x, end = ' ')
```

Выполнение программы:

```
> python program.py
Leonardo Donatello Raphael Michaelangelo
Donatello Leonardo Michaelangelo Raphael
```

В следующем примере мы будем проделывать то же, что и в предыдущем, до тех пор, пока пользователь не введёт пустую строку, то есть пока длина `len(a)` списка слов `a` не равна нулю (кто определит, какую непустую строку можно ввести, чтобы программа завершилась, имеет право немедленно потребовать с нас конфетку).
Чтобы в примере не было путаницы в консоли между тем, что ввёл пользователь, и тем, что вывела программа, сделаем что-то новенькое: файловый ввод и вывод. Есть несколько способов сделать так, чтобы программа работала с файлами.

### Перенаправление потоков

Простейший из них — перенаправление потоков ввода и вывода в консоли. Вместо того, чтобы менять что-то в коде, мы немного по-другому вызываем программу.

```
def obrabotka():
    a = input().split()
    a.sort()
    for x in a:
        print(x, end = ' ')
    print()
    return len(a) != 0

while (obrabotka()):
    continue
```

Перед выполнением прокомментируем код. В нём мы создали функцию, которая обрабатывает одну строку. В Python для создания функций надо написать ключевое слово `def`, далее имя функции, параметры в круглых скобках и двоеточие. Далее с отступом в один таб идёт тело функции. Ключевое слово `return` указывает на возвращаемое значение.
Кроме этого, мы использовали цикл `while`, синтаксис которого вплоне логичен. Слово `continue` по сути указывает на переход к очередному витку цикла (итерации). Его следует писать в одном из двух случаев:

- принудительный переход на следующий виток цикла (дальнейший код в цикле не будет выполняться на текущем витке);
- пустое тело цикла.

Теперь перейдём к запуску. Сначала надо создать файл, в котором будет содержаться весь будущий ввод:

input.txt

```
Leonardo Donatello Raphael Michaelangelo
Kraang Shredder Bebop Rocksteady


```

Потом программа вызывается с дополнительным ключом перенаправления потока ввода-вывода в конце, чтобы ввод в программу поступал из файла:

Выполнение программы:

```
> python program.py < input.txt
Donatello Leonardo Michaelangelo Raphael
Bebop Kraang Rocksteady Shredder
```

Можно также перенаправить вывод в файл:

Выполнение программы:

```
> python program.py < input.txt > output.txt
```

Чтобы считывать до конца файла, можно считывать строчки функцией `input()`, пока её размер отличен от нуля. Есть и другие способы это сделать.

### Чтение из файла в коде

Рассмотрим код, который выполняет то же самое, что и код в предыдущем примере, если использовать работу с файлами в самом коде.
Используем для этого функцию открытия потоков файлов `open()` и возможности библиотеки `sys`, в которой можно перенаправить стандартный ввод из файла и стандартный вывод в файл.
Функция `open()` открывает файл для чтения или записи. Первый аргумент — имя файла, второй — модификатор, указывающий, для чего открывать файл (`'w'` для write, то есть записи, `'r'` для read, то есть чтения).
Кроме этого, новый код от старого ничем не отличается:

```
import sys
sys.stdin = open("input.txt", "r")
sys.stdout = open("output.txt", "w")

def obrabotka():
    a = input().split()
    a.sort()
    for x in a:
        print(x, end = ' ')
    print()
    return len(a) != 0

while (obrabotka()):
    continue
```

input.txt

```
Leonardo Donatello Raphael Michaelangelo
Kraang Shredder Bebop Rocksteady
```

Выполнение программы:

```
> python program.py
```

После запуска программы в файле `output.txt` будет результат.

output.txt

```
Donatello Leonardo Michaelangelo Raphael
Bebop Kraang Rocksteady Shredder
```

## 1.5 Ввод чисел

Так как функция `input()` возвращает строку, а не число, строковое значение необходимо преобразовать в числовое перед тем, как производить числовые манипуляции со значениями. Приведение к типу данных в Питоне осуществляется с помощью ряда функций, которые имеют наглядные имена: `int()` для приведения в целочисленный тип данных, `float()` — в вещественный, `str()` — в строковый.

```
a, b = input().split(' ')
a = int(a)
b = int(b)
res = a + b
print (res)
```

Выполнение программы:

```
> python program.py
3 5
8
```

Для того, чтобы не приводить значения каждой переменной к другому типу данных вручную, используют функцию `map()`, которая применяет заданную функцию к списку значений, который в нашем случае будет результатом применения `split()` к вводу `input()`. Вывод у этого кода будет такой же, но сам код занимает меньше строк.

```
a, b = map(int, input().split(' '))
res = a + b
print (res)
```

Рассмотрим пример использования функции `map()` со своей функцией. В нём пользователь вводит число в восьмеричной системе счисления, а программа должна перевести его в шестнадцатеричную и вывести.
Функция `int8()` упрощает использование функции `int()`, которая в данном случае используется для того, чтобы перевести считанные данные в восьмеричное число (для этого у функции `int` добавляется второй параметр — система счисления вводимого числа).

```
def int8(x):
    return int(x, 8)

a = map(int8, input().split())
for x in a:
    print('Octal number %o is hexadecimal number %X' % (x, x))
```

Выполнение программы:

```
> python program.py
77
Octal number 77 is hexadecimal number 3F
```

# Глава 2

# Иллюстрации алгоритмизации

## 2.1 Математика и операторы

| Оператор | Описание |
|---:|---|
| `()` | Скобки (группировка выражения) (`a * (b + c)`) |
| `**` | Возведение в степень (`2**3` → `8`) |
| `~` | Побитовое отрицание |
| `-x` | Отрицательное значение |
| `*, /, //, %` | Умножение, деление, деление нацело, остаток от деления |
| `+, -` | Сложение, вычитание |
| `<<, >>` | Побитовый сдвиг (`x = 00000001, x<<2` = 00000100) |
| `&` | Побитовое И (`x = 0110, y = 1100, x & y = 0100`) |
| `^` | Побитовое Искл. ИЛИ (`x = 0110, y = 1100, x^y = 1010`) |
| `\|` | Побитовое ИЛИ (`x = 0110, y = 1100, x \| y = 1110`) |
| `in, not in` | Включение (`x = [1, 5, 7, 8], 7 in x` → `true`) |
| `is, is not` | Идентификация (`5 is 7` → `false`) |
| `<, <=, >, >=, <>, !=, ==` | Сравнение |
| `not x` | Логическое отрицание |
| `and` | Логическое И |
| `or` | Логическое ИЛИ |

Всем известно, что арифметические операции имеют свойство под называнием “приоритет”. Например, в выражении $a + b * c$ сначала надо посчитать произведение $b$ и $c$, а

затем уже прибавить результат к $a$.
В этой таблице операторы перечислены в порядке убывания приоритета, за исключением операторов включения, идентификации и сравнения — все они на самом деле имеют одинаковый приоритет.

## 2.2 Сумма N чисел

Мы не знаем, сколько чисел введёт пользователь, но всё равно можем посчитать их сумму. Сумма чисел хранится в переменной `res`, которая должна обязательно быть равной нулю до начала суммирования. Во-первых, так мы обозначаем, что такая переменная существует в нашей программе, а во-вторых, присваивая ей значение ноль, мы уверены, что результат подсчёта не будет испорчен неизвестным значением.

```
a = list(map(int, input().split(' ')))
res = 0
for x in a:
    res = res + x
print(res)
```

Выполнение программы:

```
> python program.py
3 5 10 5 4
27
```

В Питоне есть встроенная функция для подсчёта суммы чисел в списке:

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

## 2.3 N!

Алгоритмы вроде подсчёта значений сумм или произведений правильных последовательностей можно считать двумя способами: рекурсивно и итеративно. Если говорить кратко, рекурсивный метод подразумевает, что в функции, которая возвращает ответ, вызывается эта же самая функция от других параметров, чтобы подсчитать необходимые промежуточные значения, а в итеративном методе весь подсчёт выполняются последовательными командами в одном цикле.
Для иллюстрации примера ниже приведено два кода, которые считают и выводят факториал введённого числа. Факториал числа N ($N!$) — это математическая функция, которая для любого неотрицательного целого числа N равна произведению всех чисел от 1 до этого числа включительно. Факториал числа 0 принято считать единицей.

```python
def fact(n):
    if(n == 0):
        return 1
    return n * fact(n-1)

a = int(input())
print(fact(a))
```

```python
def fact(n):
    res = 1
    for i in range(1, n+1):
        res *= i
    return res

a = int(input())
print(fact(a))
```

Выполнение программы:

```
> python program.py
5
120
```

Точно так же, как в суммировании, переменной `res` здесь заранее присвоено значение, но так как здесь считается произведение чисел, а не сумма, переменная `res` равна единице.

В Питоне есть встроенная функция для подсчёта факториала числа:

```python
import math
a = int(input())
print(math.factorial(a))
```

## 2.4 Фибоначчи

Последовательность Фибоначчи — это последовательность чисел, заданная следующей рекуррентной формулой:

$$a_N = \begin{cases} 0, & N = 0 \\ 1, & N = 1 \\ a_{N-1} + a_{N-2}, & \text{иначе} \end{cases}$$

Числа Фибоначчи также можно находить итеративным и рекурсивным методом, но просто рекурсия в этом случае будет выполнять огромное количество повторяющихся операций. Чтобы избежать этого, используют запоминание значений. В примере ниже

список `was` хранит в себе значения ранее подсчитанных чисел Фибоначчи, и при запросе на число Фибоначчи с номером `n` он либо использует записанное в n-м элементе списка значение, либо считает его, если оно ещё не было подсчитано (если `was[n] == -1`, значит, это значение не подсчитано, потому что число Фибоначчи не может быть отрицательным).

Также стоит заметить, что для того, чтобы создать список какой-либо фиксированной длины, заполненный определёнными одинаковыми значениями, используется приём конкатенации (склеивания) множества списков.

```
was = [-1] * 501
def fib(n):
    if(was[n] != -1):
        return was[n]
    if(n == 0):
        was[n] = 0
    elif(n == 1):
        was[n] = 1
    else:
        was[n] = fib(n-1) + fib(n-2)
    return was[n]

n = int(input())
print(fib(n))
```

В питоне глубина рекурсии — количество последовательных рекурсивных вызовов функции — ограничена внутренней переменной. Увеличить глубину рекурсии можно вручную командой `sys.setrecursionlimit()`, но если сделать её значение слишком большим и создать рекурсию слишком большой глубины, программа заполнит всю доступную ей память и аварийно завершится.

```
import sys
sys.setrecursionlimit(10001)
was = [-1] * 10001
def fib(n):
    if(was[n] != -1):
        return was[n]
    if(n == 0):
        was[n] = 0
    elif(n == 1):
        was[n] = 1
    else:
        was[n] = fib(n-1) + fib(n-2)
    return was[n]

n = int(input())
print(fib(n))
```

Обратите внимание на условную конструкцию `if - elif - else`. Она соотвествует утверждениям в русском языке "если *условие*" – "иначе, если *условие*" – "иначе", то есть действие внутри `elif` выполнится, только если **не выполнилось** условие у `if` и всех вышестоящих `elif` и **выполнилось** условие у `elif`.
Предыдущие примеры кода также годятся для того, чтобы посчитать и вывести все числа Фибоначчи до n-го включительно, но если требуется просто найти n-е число Фибоначчи, можно просто следовать рекуррентной формуле в итеративном виде.
Для этого в коде ниже используется специальная функция `range()`. При вызове `range()` с двумя целочисленными параметрами, скажем, `a` и `b`, где `a` не превышает `b`, она вернёт список, в котором в возрастающем порядке будут записаны все числа от `a` до `b`. При вызове с одним параметром `b` она вернёт список с числами от `0` до `b`.
Функция `range()` чаще всего используется в циклах `for` для того, чтобы переменная-счётчик менялась определённым образом в определённых рамках.

```
1  def fib(n):
2      a = 0
3      b = 1
4      for i in range(0, n-1):
5          a, b = b, a+b
6      return b
7
8  n = int(input())
9  print(fib(n))
10
```

Заметьте интересный приём в присваивании, который используется здесь, чтобы сократить количество строк кода и уменьшить его сложность: кортежу (двум переменным через запятую, по сути - список) присваивается кортеж, что значит, первому элементу левого кортежа присваивается значение первого элемента правого кортежа, второму элементу — значение второго и так далее. Самое интересное в этом то, что присваивание происходит следующим образом: сначала подсчитываются значения правого кортежа, а только затем они присваиваются элементам левого, поэтому если мы в присваивании меняем значение переменной (в примере выше — обеих, `a` и `b`), то им присваиваются значения, вычисленные с учётом старых значений этих переменных. Например, если написать строчку `a, b = b, a`, они поменяются значениями, но если написать сначала `a = b`, а потом `b = a`, то при выполнении первого действия значение `a` будет навсегда потеряно, и в результате у нас будут две переменные со старыми значениями `b`.

## 2.5 Тяжёлый, медленный Питон

Для программирования Питон — в первую очередь очень удобный язык, который не требует кучи лишнего кода. К сожалению, за свою лаконичность Питон платит скоростью исполнения и занимаемой памятью. Об этом следует помнить в олимпиадном программировании, когда на счету каждая миллисекунда и каждый байт.

# Глава 3

# Структуры данных

## 3.1 Кортежи и списки

И то, и другое является структурой для хранения *последовательности* значений, но их не следует путать между собой.
Списки объявляются с помощью квадратных скобок `[a, b]`, любая последовательность к списку приводится функцией `list()`. Как правило, их используют для хранения однотипных данных (но это не обязательно). К элементам списка можно обращаться по индексу, и работать с ними, как с обычными переменными.
Кортежи объявляются с помощью круглых скобок `(a, b)`, любая последовательность к кортежу приводится функцией `tuple()`. Как правило, кортежи используют для хранения различных параметров (вероятно, различных типов данных). К элементам кортежа можно обращаться по индексу, но менять таким образом значения в кортеже нельзя.
В остальном, различия между списком и кортежем очень размыты: к ним применяются одни и те же функции `len()` (количество элементов), `cmp()` (проверка двух списков/кортежей на равенство), `max()` и `min()`. Но для списков также реализованы методы, такие как `append()`, `count()`, `extend()` и прочие. Подробнее о списках и кортежах можно прочитать в документации.

## 3.2 Стек

Структура данных, позволяющая осуществлять следующие операции с множеством:

- Добавление элемента в множество;
- Получение последнего добавленного элемента;
- Удаление последнего добавленного элемента.

В реальном мире отличным примером стека является стопка (книг, например): мы можем добавлять или удалять элементы только сверху, и видим только верхний элемент. Стек вызовов тоже назван так не случайно.

В Python можно в качестве стека использовать список: метод `append(a)` добавляет элемент `a` в список, метод `pop()` удаляет последний элемент.
Рассмотрим следующую задачу: в стеке добавляются (запрос `+` число) и удаляются (запрос `-`) числа, а также даются запросы (`=`) на вывод минимального числа на всём множестве. Ввод заканчивается пустой строкой.
Вместо того, чтобы каждый раз искать минимум, стоит заметить две вещи: новое число в стеке либо не изменит минимум, если оно больше или равно ему, либо изменит, если оно меньше минимума; при удалении числа, какие бы изменения оно не привносило, они уходят вместе с ним, и положение откатывается к более старой версии.
Таким образом, наряду со стеком чисел можно создать стек минимумов, с которым будут происходить те же операции, что и со стеком чисел, только для каждого добавленного числа в стек минимумов будет добавляться текущий минимум: если число больше предыдущего минимума, то будет добавлен старый минимум, иначе — новое число. При удалении числа удалится и соотвествующий ему минимум.
При этом, хранение стека чисел становится лишним.
Вот так это решение выглядит в коде:

```
st = []
while(1 == 1):
    inf = input()
    if(len(inf)<1):
        break
    if(len(inf)>1):
        inf, n = inf.split()
        if(len(st)):
            n = min(int(n), st[-1])
        else:
            n = int(n)
        st.append(n)
    elif(inf=='-'):
        st.pop()
    else:
        print(st[-1])
```

## 3.3 Очередь

Структура данных, позволяющая осуществлять следующие операции с множеством:

- Добавление элемента в множество;
- Получение первого добавленного элемента;
- Удаление первого добавленного элемента.

Отличный живой пример очереди в реальном мире — очередь. Элементы встают в один конец множества и ждут, когда они станут первыми, чтобы уйти из него.

Теоретически, очередь также можно реализовать через списки, но удаление элементов из начала списка (или добавление элемента в начало списка) — это слишком долгая операция по сравнению с добавлением (и удалением) элементов из конца. Поэтому в Питоне в качестве очереди используют встроенную структуру данных дек из библиотеки `collections`.

## 3.4 Дек (очередь о двух концах)

Структура данных, позволяющая осуществлять следующие операции с множеством:

- Добавление элемента в “начало” или “конец” множества;
- Получение одного из двух “крайних” элементов;
- Удаление одного из двух “крайних” элементов.

Работает подобно очереди, но добавлять и удалять элементы можно с обоих концов. Стандартная имплементация дека в Питоне содержится в библиотеке `collections`. Использует стандартные методы `append(a)` и `appendleft(a)` для добавления элемента `a` в дек с конца или с начала (а также `extend()` и `extendleft` для добавления последовательности элементов, причём последний метод “переворачивает” добавляемую последовательность), и `pop()` и `popleft()` для удаления крайних элементов. Ещё больше методов и применений дека есть в документации.
Вот пример применения дека в качестве очереди в простой задаче про эмуляцию очереди с запросами `in` $n$ на добавления числа $n$ в очередь, `out` на вывод и удаление первого числа из очереди, и `end` на прекращение ввода:

```
from collections import deque
queue = deque()
while(1 == 1):
    query = input()
    if(query == 'end'):
        break
    if(query == 'out'):
        print(queue.popleft())
    else:
        query, n = query.split()
        queue.append(n)
```

## 3.5 Сет (множество уникальных элементов)

Структура данных в Python, которая поддерживает операции добавления, удаления и поиска элемента во множестве. Сет хранит только уникальные элементы, то есть в сете не может быть двух одинаковых элементов. При попытке добавить в сет неуникальный элемент ничего не изменится.

Сеты в Питоне являются аналогом структуры данных из C++ `unordered_set`, так как оба хранят элементы неупорядоченно и используют для быстрой адресации данных хэши.
Некоторые методы сетов:

- `add(a)` — добавление элемента `a` во множество;
- `remove(a)`, `discard(a)` — удаляет элемент из множества; во втором случае, в отличие от первого, происходит также проверка на вхождение элемента в сет перед удалением;
- `first` и `second` — обращение к ключу и значению элемента, соответственно (так как каждый элемент мапа — это пара, обращение к нему происходит как в паре).
- К сетам применимы операторы проверки на вхождение во множество `in`, `not in`, а также операции со множествами, такие как объединение, пересечение и их производные (см. документацию).

Для создания сета используется конструктор `set()`, а в случае создания непустых сетов также можно использовать фигурные скобки `{}`.
Пример задачи для использования сетов: на ввод подаются строки, и для каждой строки надо сказать, была ли она дана ранее или нет:

```
s = set()
while(1 == 1):
    string = input()
    if(len(string) < 1):
        break
    if(string in s):
        print('YES')
    else:
        print('NO')
        s.add(string)
```

Ещё одна простейшая задача — использование свойства сета для нахождения количества уникальных элементов в последовательности:

```
l = list(map(int, input().split()))
s = set(l)
print(len(s))
```

## 3.6 Словарь (ассоциативный массив)

Структура данных в Питоне, которая позволяет ассоциировать некоторый *ключ* с некоторым *значением*, и обращаться к значению по ключу, как по индексу массива.
Словарь в Питоне можно создать с помощью конструктора `dict()` или фигурных скобок `{}`.

Некоторые функции/методы/операторы для работы со словарями:

- `[a]` — обращение к элементу с ключом `a`, но **не добавляет значения по умолчанию, если такого элемента не существовало**;
- `in`, `not it` — проверяет ключ на вхождение во множество;
- `del d[a]` — удаляет элемент с ключом `a` из множества `d`;
- `keys()`, `values()` — возвращают последовательности (**но не списки!**) всех ключей и всех значений в словаре, соответственно.

(См. документацию)

Вспомним, что сортировка подсчётом заключается в подсчёте количества вхождений элементов в множество. Как правило, она применяется в случаях, когда при большом количестве элементов количество *различных* элементов мало. Если диапазон значений невелик, можно использовать для хранения количеств элементов массив, но если значения могут быть какие угодно, но гарантированно, что различных среди них мало, задачу о сортировке такого множества чисел можно решить с помощью словаря.

```
1  n = list(map(int, input().split()))
2  d = {}
3  for x in n:
4      if(x not in d):
5          d[x] = 1
6      else:
7          d[x] += 1
8  for x in d:
9      for i in range(d[x]):
10         print(x, end = ' ')
```