Подробнее о строках

Строка считывается со стандартного ввода функцией `input()`. Напомним, что для двух строк определена операция сложения (конкатенации), также определена операция умножения строки на число.

Строка состоит из последовательности символов. Узнать количество символов (длину строки) можно при помощи функции `len`:

```
>>> S = 'Hello'
>>> print(len(S))
5
```

Срезы (slices)

Срез (slice) --- извлечение из данной строки одного символа или некоторого фрагмента подстроки или подпоследовательности.

Есть три формы срезов. Самая простая форма среза: взятие одного символа строки по индексу (номеру в строке), `S[i]` --- это срез, состоящий из одного символа, который имеет номер `i` (нумерация начинается с 0). То есть если `S='Hello'`, то `S[0]=='H'`, `S[1]=='e'`, `S[2]=='l'`, `S[3]=='l'`, `S[4]=='o'`.

Если указать отрицательное значение индекса, то номер будет отсчитываться с конца, начиная с номера `-1`. То есть `S[-1]=='o'`, `S[-2]=='l'`, `S[-3]=='l'`, `S[-4]=='e'`, `S[-5]=='H'`.

Если же номер символа в срезе строки `S` больше либо равен `len(S)`, или меньше, чем `-len(S)`, то при обращении к этому символу строки произойдёт ошибка `IndexError: string index out of range`.

Срез с двумя параметрами: `S[a:b]` возвращает подстроку из `b-a` символов, начиная с символа с индексом `a`, то есть до символа с индексом `b`, не включая его. Например, `S[1:4]=='ell'`, то же самое получится если написать `S[-4:-1]`. Можно использовать как положительные, так и отрицательные индексы в одном срезе, например, `S[1:-1]` --- это строка без первого и последнего символа (срез начинается с символа с индексом 1 и заканчивается индексом -1, не включая его).

Если опустить второй параметр (но поставить двоеточие), то срез берётся до конца строки. Например, чтобы удалить из строки первый символ (его индекс равен 0, то есть взять срез, начиная с символа с индексом 1), то можно взять срез `S[1:]`, аналогично если опустить первый параметр, то срез берётся от начала строки. То есть удалить из строки последний символ можно при помощи среза `S[:-1]`. Срез `S[:]` совпадает с самой строкой `S`.

Если задать срез с тремя параметрами `S[a:b:d]`, то третий параметр задаёт шаг, как в случае с функцией `range`, то есть будут взяты символы с индексами `a`, `a+d`, `a+2*d` и т.д. При задании значения третьего параметра,

равному 2, в срез попадёт каждый второй символ, а если взять значение среза, равное `-1`, то символы будут идти в обратном порядке.

Методы

Метод --- это функция, применяемая к объекту, в данном случае --- к строке. Метод вызывается в виде `Имя_объекта.Имя_метода(параметры)`. Например, `S.find("e")` --- это применение к строке `S` метода `find` с одним параметром `"e"`.

Метод `find` находит в данной строке (к которой применяется метод) данную подстроку (которая передаётся в качестве параметра). Функция возвращает индекс первого вхождения искомой подстроки. Если же подстрока не найдена, то метод возвращает значение -1. Например:

```
>>> S = 'Hello'
>>> print(S.find('e'))
1
>>> print(S.find('ll'))
2
>>> print(S.find('L'))
-1
```

Аналогично, метод `rfind` возвращает индекс последнего вхождения данной строки («поиск справа»).

```
>>> S = 'Hello'
>>> print(S.find('l'))
2
>>> print(S.rfind('l'))
3
```

Если вызвать метод `find` с тремя параметрами `S.find(T, a, b)`, то поиск будет осуществляться в срезе `S[a:b]`. Если указать только два параметра `S.find(T, a)`, то поиск будет осуществляться в срезе `S[a:]`, то есть начиная с символа с индексом `a` и до конца строки. Метод `S.find(T, a, b)` возвращает индекс в строке `S`, а не индекс относительно начала среза.

Метод `replace` заменяет все вхождения одной строки на другую. Формат: `S.replace(old, new)` --- заменить в строке `S` все вхождения подстроки `old` на подстроку `new`. Пример:

```
>>> 'Hello'.replace('l', 'L')
'HeLLo'
```

Если методу `replace` задать ещё один параметр: `S.replace(old, new, count)`, то заменены будут не все вхождения, а только не больше, чем первые `count` из них.

```
>>> 'Abrakadabra'.replace('a', 'A', 2)
'AbrAkAdabra'
```
Метод `count` подсчитывает количество вхождений одной строки в другую строку. Простейшая форма вызова `S.count(T)` возвращает число вхождений строки `T` внутри строки `S`. При этом подсчитываются только непересекающиеся вхождения, например:

```
>>> 'Abracadabra'.count('a')
4
>>> ('a' * 100000).count('aa')
50000
```
При указании трёх параметров `S.count(T, a, b)`, будет выполнен подсчёт числа вхождений строки `T` в срез `S[a:b]`.

Массивы

Большинство программ работает не с отдельными переменными, а с набором переменных. Например, программа может обрабатывать информацию об учащихся класса, считывая список учащихся с клавиатуры или из файла, при этом изменение количества учащихся в классе не должно требовать модификации исходного кода программы.

Для хранения таких данных можно использовать структуру данных, называемую в Питоне список (в большинстве же языков программирования используется другой термин --- «массив», разницу мы обсудим позже). Список представляет собой последовательность элементов, пронумерованных от 0. Список можно задать перечислением элементов в квадратных скобках, например, список можно задать так:

```
primes = [2, 3, 5, 7, 11, 13]
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo',
'Violet']
```
В списке `primes` --- 6 элементов типа int Список `rainbow` состоит из 7 элементов, каждый из которых является строкой.

Также как и символы строки, элементы списка можно индексировать отрицательными числами с конца, например, `primes[-1] == 13`, `primes[-6] == 2.`

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

Рассмотрим несколько способов создания и считывания списков. Пустой, т.е.

не имеющий элементов список, можно создать следующим образом:

```
A = []
```

Для добавления элементов в конец списка используется метод `append`. Если программа получает на вход количество элементов в списке `n`, а потом `n` элементов списка по одному в отдельной строке, то организовать считывание списка можно так:

```
A = []
for i in range(int(input())):
    A.append(int(input())
```

В этом примере создается пустой список, далее считывается количество элементов в списке, затем по одному считываются элементы списка и добавляются в его конец.

Для списков целиком определены следующие операции: конкатенация списков (добавление одного списка в конец другого) и повторение списков (умножение списка на число). Например:

```
A = [1, 2, 3]
B = [4, 5]
C = A + B
D = B * 3
```

В результате список `C` будет равен `[1, 2, 3, 4, 5]`, а список `D` будет равен `[4, 5, 4, 5, 4, 5]`. Это позволяет по-другому организовать процесс считывания списков: сначала считать размер списка и создать список из нужного числа элементов, затем организовать цикл по переменной `i` начиная с числа 0 и внутри цикла считывается `i`-й элемент списка:

```
A = [0] * int(input())
for i in range(len(A)):
    A[i] = int(input())
```

Вывести элементы списка `A` можно одной инструкцией `print(A)`, при этом будут выведены квадратные скобки вокруг элементов списка и запятые между элементами списка. Такой вывод неудобен, чаще требуется просто вывести все элементы списка в одну строку или по одному элементу в строке. Приведем два примера, также отличающиеся организацией цикла:

```
for i in range(len(A)):
    print(A[i])
```

Здесь в цикле меняется индекс элемента `i`, затем выводится элемент списка с индексом `i`.

```
for elem in A:
    print(elem, end = ' ')
```

В этом примере элементы списка выводятся в одну строку, разделенные

пробелом, при этом в цикле меняется не индекс элемента списка, а само значение переменной. Например, в цикле `for elem in ['red', 'green', 'blue']` переменная `elem` будет последовательно принимать значения 'red', 'green', 'blue'.

Внутри одного списка могут быть любые объекты (и даже вперемешку), поэтому такая конструкция как список списков вполне осмысленна (аналог двумерного массива). Обращаться к элементам внутри такого списка нужно так `A[i][j]` , где `j` - индекс внутри внутреннего списка, `i` - индекс внутри внешнего списка.

Обратите внимание на следующую вещь:

```
A = [[0] * 10]*10 # вроде бы это обычный список списков 10x10 состоящий из 0
A[0][0] = 1 # меняем элемент с индексом 0 в списке с индексом 0
print(A) # печатаем A
```

Что вывела программа? Как можно это объяснить? Вспомните про ссылочную модель данных.

Методы split и join

Выше мы рассмотрели пример считывания списка, когда каждый элемент расположен на отдельной строке. Иногда бывает удобно задать все элементы списка при помощи одной строки. В такой случае используется метод `split` , определённый для строк:

```
A = input().split()
```

Если при запуске этой программы ввести строку 1 2 3, то список `A` будет равен `['1', '2', '3']` . Обратите внимание, что список будет состоять из строк, а не из чисел. Используя функции языка map и list можно в одну строку считать последовательность чисел в список:

```
A = list(map(int, input().split()))
```

Здесь мы считываем строку (функция input(), разделяем ее на список чисел (записанных как строки) функцией split(), затем функцией map применяем функцию int ко всем элементам списка, получившегося из input().split(), и затем результат исполнения функции map переводим в списочный тип функцией list().

У метода `split` есть необязательный параметр, который определяет, какая строка будет использоваться в качестве разделителя между элементами списка. Например, вызов метода `split('.')` для строки вернет список, полученный разрезанием этой строки по символам '.'.

Используя «обратные» методы можно вывести список при помощи однострочной команды. Для этого используется метод строки `join`. У этого метода один параметр: список строк. В результате создаётся строка, полученная соединением элементов списка (которые переданы в качестве параметра) в одну строку, при этом между элементами списка вставляется разделитель, равный той строке, к которой применяется метод. Например, программа

```
A = ['red', 'green', 'blue']
print(' '.join(A))
print(''.join(A))
print('***'.join(A))
```

выведет строки `red green blue`, `redgreenblue` и `red***green***blue`. Обратите внимание, что `join` является методом **строки**, а не списка.

Если же список состоит из чисел, то придется использовать еще и функцию map. То есть вывести элементы списка чисел, разделяя их пробелами, можно так:

```
print(' '.join(map(str, A)))
```

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

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

Получится 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]`.

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

Получится 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.

Если **не непрерывному** срезу (то есть срезу с шагом `k`, отличному от 1), присвоить новое значение, то количество элементов в старом и новом срезе обязательно должно совпадать, в противном случае произойдет ошибка `ValueError`.

Обратите внимание, `A[i]` --- это **элемент** списка, а не срез!

(1) Генерация массива из нулей

In [ ]:
```
n = int(input())
a = [0] * n # [0, 0, 0, 0, 0]

for x in range(n):
  a[x] = input()

print(a)
```

```
3
1
4
5
[1, 4, 5]
```

(2) Создание массива из строки

In [ ]:
```
s = input().split(';')

for i in range(len(s)):
  s[i] = int(s[i])

print(s)
```

```
1;4
[1, 4]
```

(3) Функция map

In [ ]:
```
a = []
a = list()
```

In [ ]:
```
s = ['2', '3']
s = list(map(float, s))

print(s)
```

```
[2.0, 3.0]
```

In [ ]:
```
type(a)
```

Out[ ]:
```
list
```

(4) Добавление элементов

In [ ]:
```
s.append(4)
print(s)
```

```
[2.0, 3.0, 4]
```

(5) Удаление элемента (с конца)

In [ ]:
```
s.pop()
print(s)
```

```
[2.0, 3.0]
```

Если хотим извлечь конкретный элемент, то в () ставим его номер.

In [ ]:
```
s.pop(1)
```

Out[ ]:
```
3.0
```

In [ ]:
```
s
```

Out[ ]:
```
[2.0]
```

(6) Добавление элемента на определённое место

a.insert(position, element)

In [7]:
```
a = [4, 6, 7]
a.insert(1, 5)

print(a)
```

```
[4, 5, 6, 7]
```

In [ ]:
```
a.insert(3, 7)

print(a)
```

```
[4, 5, 6, 7, 7]
```

In [ ]:
```
a.reverse()

print(a)
```

```
[7, 7, 6, 5, 4]
```

In [8]:
```
a.sort()

print(a)
```

```
[4, 5, 6, 7]
```

(7)*** Более сложный способ заполнить/сгенерировать массив => списковое включение/List Comprehensions

In [ ]:
```
a = [i for i in range(10)]

print(a)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [14]:
```
a = [i**i for i in range(1, 10)]

print(a)
```

```
[1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489]
```

In [11]:
```
a = [i for i in range(10) if i % 2 == 0]

print(a)
```

```
[0, 2, 4, 6, 8]
```

In [ ]:
```
squares = list(map(lambda x: x**2, range(10)))

print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(8) Срезы

In [ ]:
```
a = [4, 5, 7, 6, 3]

print(a[1:3])
print(a[::-1])
```

```
[5, 7]
[3, 6, 7, 5, 4]
```

(9)** срезы для строк

In [ ]:
```
s = 'Hello, world!'

print(s[1::2])
print(s[::-1])
print(s[-4])
```

```
el,wrd
!dlrow ,olleH
r
```

Пример: поиск суммы чётных положительных элементов в массиве

In [ ]:
```
n = int(input())
a = [int(input()) for _ in range(n)]
s = 0

for e in a:
  if e > 0 and e % 2 == 0:
    s += e
print(s)
```

```
2
1
3
0
```

(8) Можно пройтись и по объектам массива, и по номерам

In [ ]:
```
a = 'psppspsppsps'

for i, e in enumerate(a):
  print(i, e)
```

```
0 p
1 s
2 p
3 p
4 s
5 p
6 s
7 p
8 p
9 s
10 p
11 s
```

In [ ]:
```
a = 'psppspsppsps'
b = '546474848484'

for x,y in zip(a,b):
  print(x, y)
```

```
p 5
s 4
p 6
p 4
s 7
p 4
s 8
p 4
p 8
s 4
p 8
s 4
```

Summary:

Операция	Действие
x in A	Проверить, содержится ли элемент в списке. Возвращает True или False.
x not in A	То же самое, что not(x in A).
min(A)/max(A)	Наименьший/наибольший элемент списка. Элементы списка могут быть числами или строками, для строк сравнение элементов проводится в лексикографическом порядке.
sum(A)	Сумма элементов списка, элементы обязательно должны быть числами.
A.index(x)	Индекс первого вхождения элемента x в список, при его отсутствии генерирует исключение ValueError.
A.count(x)	Количество вхождений элемента x в список.
A.append(x)	Добавить в конец списка A элемент x.
A.insert(i, x)	Вставить в список A элемент x на позицию с индексом i. Элементы списка A, которые до вставки имели индексы i и больше сдвигаются вправо.
A.extend(B)	Добавить в конец списка A содержимое списка B.
A.pop()	Удалить из списка последний элемент, возвращается значение удаленного элемента.
A.pop(i)	Удалить из списка элемент с индексом i, возвращается значение удаленного элемента. Все элементы, стоящие правее удаленного, сдвигаются влево.

Пасхалка №1! тест: https://sky.pro/media/tests/test-na-znanie-python/?ysclid=mey4hwjp30801748245

In [ ]:

Задание №1.1 Калькулятор с пользовательским вводом

На вход поступают 2 числа и знак арифметической операции, выведете ответ, исходя из этих данные. Пример: Операции: -, +, *, /

```
4
5
-
```

Ответ:

```
-1
```

Задание №1.2

Заполните массив числами Фибоначчи

Задание №1.3

Дана последовательность и число X, нужно найти его положение в массиве

Задание №1.4 Вспомним функции

Разложите число на множители:

```
def factorize(x):
  ...
256

1 256
2 128
4 64
8 32
16 16
```

Форматирование строк

Часто возникает необходимость выводить строки, содержание которых зависит от некоторых переменных в коде. Например:

```
a = int(input())
b = int(input())
c = a + b
print('sum of', a,' and', b, ' =', c)
```

Такая запись становится неудобной и нечитаемой, когда у нас становится много переменных с длинными наименованиями. Для решения этой проблемы существует **форматирование строк**. В python есть несколько способов форматирования строк, которые немного отличаются функционалом и синтаксисом.

```
str_1 = "Name: %s, email: %s, phone: %s" % (name, email, phone)
str_2 = "Name: {}, email: {}, phone: {}".format(name, email, phone)
```
В обоих случаях мы подставляем значения переменных name, email, phone в нужные места в строке. Несложно заметить, что здесь тоже возникают проблемы с читаемостью: необходимо сопоставлять порядок перечисления аргументов форматирования `name, email, phone` с позициями `%s` или `{}`.

Начиная с python 3.6 форматирование можно производить используя **f-строки** -- наиболее функциональный и удобный способ на данный момент. Таким способом мы можем записать наш пример так:

```
f_str = f"Name: {name}, email: {email}, phone: {phone}"
```
Очень удобно и читаемо. Но на этом функционал не заканчивается, в фигурных скобках можно вычислять новые значения (и даже вызывать методы):

```
num = 7
print(f'{num} squared is {num * num}')
>>> '7 squared is 49'
```
В форматированных строках можно указать количество десятичных знаков после запятой, количество символов, выделенных для вывода значения, а также выровнять значение по левому или правому краю. Общий синтаксис выглядит так:

```
f'{value:{width}.{precision}}'
```
Значение, двоеточие, затем ширина строки в фигурных скобках, точка, требуемая точность в фигурных скобках. Пара примеров:

```
pi = 3.14159265
print(f'{pi:.2f}') #выводим pi с двумя знаками после запятой
>>> '3.14'
print(f'{5:>5}₽')  # ширина 5 символов, выравниванием вправо >
>>> '   5₽'
```
Момент катарсиса: форматированные строки могут работать с генераторами списков!

```
list_a = ['a', 'b', 'c', 'd']
list_b = [f"{x + x}" for x in list_a]
print(list_b)
>>> "['aa', 'bb', 'cc', 'dd']"
```
Конечно, подобные операции можно реализовать и без перечисленных фишек. Однако использование удобных инструментов языка повышает наглядность решений и тем самым ускоряет процесс решения задач.

Работа с файлами

Конечно, в реальности нам часто приходится работать с файлами. Прежде, чем работать с файлом, его надо открыть. С этим замечательно справится встроенная функция open:

```
f = open('text.txt', 'r')
```

У функции open много параметров, они указаны в статье "Встроенные функции", нам пока важны 3 аргумента: первый, это имя файла. Путь к файлу может быть относительным (то есть определенным относительно исполняемого файла) или абсолютным. Второй аргумент, это режим, в котором мы будем открывать файл.

Режим Обозначение

- 'r' открытие на чтение (является значением по умолчанию).
- 'w' открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.
- 'a' открытие на дозапись, информация добавляется в конец файла.
- 't' открытие в текстовом режиме (является значением по умолчанию).
- '+' открытие на чтение и запись. Режимы могут быть объединены, то есть, к примеру, 'rb' - чтение в двоичном режиме. По умолчанию режим равен 'rt'.

И последний аргумент, encoding, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку. Пока нам это не нужно.

Чтение из файла

Открыли мы файл, а теперь мы хотим прочитать из него информацию. Для этого есть несколько способов, но большого интереса заслуживают лишь два из них.

Первый - метод read, читающий весь файл целиком, если был вызван без аргументов, и n символов, если был вызван с аргументом (целым числом n).

```
f = open('text.txt')
f.read()
'Hello world!\nThe end.\n\n'
```

Ещё один способ сделать это - прочитать файл построчно, воспользовавшись циклом for:

```
f = open('text.txt')
for line in f:
    print(line)
'Hello world!\n'
'\n'
'The end.\n'
'\n'
```

Запись в файл

Запись в файл осуществляется с помощью метода write:

```
f = open('text.txt', 'w')
l = ['1', '2', '3', '4', '5']
for index in l:
     f.write(index + '\n')
```

Кстати, метод write сам по себе возвращает число записанных символов.

После окончания работы с файлом его обязательно нужно закрыть с помощью метода close:

```
f.close()
```

Задание №2.1 Калькулятор в файл

В файле input.txt на первой строке перечислены числа (через пробел), на второй строке написан символ арифметической операции (+, -, *), которую необходимо выполнить с ними. Необходимо вывести в файл output.txt результат выполнения арифметической операции.

Задание №2.2 Решето Эратосфена

Есть натуральное число N, хотим найти все простые числа, которые не превышают N.

Однопроходные алгоритмы

Упражнение 4. Перестановка

Переставьте соседние элементы в списке. Задача решается в три строки.

Ввод	Вывод
1 2 3 4 5	2 1 4 3 5

Упражнение 5. Циклический сдвиг

Выполните циклический сдвиг элементов списка вправо. Решите задачу в две строки.

Ввод	Вывод
1 2 3 4 5	5 1 2 3 4

Упражнение 6. Уникальные элементы

Выведите элементы, которые встречаются в списке только один раз. Элементы нужно выводить в том порядке, в котором они встречаются в списке.

Ввод	Вывод
1 2 2 3 3 3	1

В этой задаче нельзя модицифицировать список, использовать вспомогательные списки, строки, срезы.

Упражнение 7. Самое частое число

Определите, какое число в этом списке встречается чаще всего. Если таких чисел несколько, выведите любое из них.

Ввод	Вывод
1 2 3 2 3 3	3

В этой задаче также нельзя модицифицировать список, использовать вспомогательные списки, строки, срезы.

Упражнение 8. Медиана списка

В списке — нечетное число элементов, при этом все элементы различны. Найдите медиану списка: элемент, который стоял бы ровно посередине списка, если список отсортировать.

При решении этой задачи нельзя модифицировать данный список (в том числе и сортировать его), использовать вспомогательные списки.

Программа получает на вход нечетное число N, в следующей строке заданы N элементов списка через пробел.

Программа должна вывести единственное число — значение медианного

элемента в списке.

Ввод	Вывод
5 3 1 2 5 4	3

In [ ]: