Lists

Списки. Вводная часть.

Об авторе и курсе.

Информация собрана из курсов Stepik python для начинающих и python для продвинутых. Пройдены тесты.

Автор документа - Латыпов А.Ф.

Структура данных (data structure) — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.

Список представляет собой последовательность элементов, пронумерованных от 0, как символы в строке.

Создание списка

Чтобы создать список, нужно перечислить его элементы через запятую в квадратных скобках:

```
numbers = [2, 4, 6, 8, 10]
languages = ['Python', 'C#', 'C++', 'Java']
```

Список numbers состоит из 5 элементов, и каждый из них — целое число.

```
numbers[0] == 2;
numbers[1] == 4;
numbers[2] == 6;
numbers[3] == 8;
numbers[4] == 10.
```

Список languages состоит из 4 элементов, каждый из которых — строка.

```
languages[0] == 'Python';
languages[1] == 'C#';
languages[2] == 'C++';
languages[3] == 'Java'.
```

Значения, заключенные в квадратные скобки и отделенные запятыми, называются элементами списка.

Список может содержать значения разных типов данных:

```
info = ['Timur', 1992, 61.5]
```

Список info содержит строковое значение, целое число и число с плавающей точкой.

```
info[0] == 'Timur';
info[1] == 1992;
info[2] == 61.5
```

Обычно элементы списка содержат данные одного типа, и на практике редко

приходится создавать списки, содержащие элементы разных типов данных.

Пустой список

Создать пустой список можно двумя способами:

Использовать пустые квадратные скобки [];
Использовать встроенную функцию, которая называется list.

Следующие две строки кода создают пустой список:

```
mylist = []  # пустой список
mylist = list()  # тоже пустой список
```

Вывод списка

Для вывода всего списка можно применить функцию print():

```
numbers = [2, 4, 6, 8, 10]
languages = ['Python', 'C#', 'C++', 'Java']
print(numbers)
print(languages)
```

Функция print() выводит на экран элементы списка, в квадратных скобках, разделенные запятыми:

```
[2, 4, 6, 8, 10]
['Python', 'C#', 'C++', 'Java']
```

Обратите внимание, что вывод списка содержит квадратные скобки. Позже мы научимся выводить элементы списка в более удобном виде с помощью циклов или с помощью распаковки.

Встроенная функция list

Python имеет встроенную функцию list(), которая помимо создания пустого списка может преобразовывать некоторые типы объектов в списки.

Например, мы знаем, что функция range() создает последовательность целых чисел в заданном диапазоне. Для преобразования этой последовательности в список, мы пишем следующий код:

```
numbers = list(range(5))
```

Во время исполнения этого кода происходит следующее:

Вызывается функция range(), в которую в качестве аргумента передается число 5;
Эта функция возвращает последовательность чисел 0, 1, 2, 3, 4;
Последовательность чисел 0, 1, 2, 3, 4 передается в качестве аргумента в функцию list();
Функция list() возвращает список [0, 1, 2, 3, 4];
Список [0, 1, 2, 3, 4] присваивается переменной numbers.

Вот еще один пример:

```
even_numbers = list(range(0, 10, 2))  # список содержит четные числа 0, 2, 4, 6, 8
odd_numbers = list(range(1, 10, 2))  # список содержит нечетные числа 1, 3, 5, 7, 9
```

Точно так же с помощью функции list() мы можем создать список из символов строки. Для преобразования строки в список мы пишем следующий код:

```
s = 'abcde'
chars = list(s)  # список содержит символы 'a', 'b', 'c', 'd', 'e'
```

Во время исполнения этого кода происходит следующее:

Вызывается функция list(), в которую в качестве аргумента передается строка 'abcde';
Функция list() возвращает список ['a', 'b', 'c', 'd', 'e'];
Список ['a', 'b', 'c', 'd', 'e'] присваивается переменной chars.

Примечания

Примечание 1. Как уже было сказано, списки в Python аналогичны массивам в других языках программирования. Однако разница между списками и массивами все же существует. Элементы массива всегда имеют одинаковый тип данных и располагаются в памяти компьютера непрерывным блоком, а элементы списка могут быть разбросаны по памяти как угодно и могут иметь разный тип данных.

Примечание 2. Обратите внимание, при выводе содержимого списка с помощью функции print(), все строковые элементы списка обрамляются одинарными кавычками. Если требуется осуществить вывод в двойных кавычках, нужно самостоятельно писать код вывода.

Списки. Основы работы (len, min, max, срезы, индексы)

1. Встроенные функции len(), sum(), min(), max()
2. Оператор принадлежности in
3. Индексация и срезы
4. Конкатенация и умножение на число
5. Отличие списков от строк

Работа со списками очень сильно напоминает работу со строками, поскольку и списки, и строки содержат отдельные элементы. Однако элементы списка могут иметь произвольный тип, а элементами строк всегда являются символы. Многое из того, что мы делали со строками, доступно и при работе со списками.

Функция len()

Длиной списка называется количество его элементов. Для того, чтобы посчитать длину списка мы используем встроенную функцию len() (от слова length – длина).

Следующий программный код:

```
numbers = [2, 4, 6, 8, 10]
languages = ['Python', 'C#', 'C++', 'Java']
print(len(numbers))      # выводим длину списка numbers
print(len(languages))    # выводим длину списка languages

print(len(['apple', 'banana', 'cherry']))   # выводим длину списка, состоящего из 3 элементов
```

Оператор принадлежности in

Оператор in позволяет проверить, содержит ли список некоторый элемент.
Рассмотрим следующий код:

```
numbers = [2, 4, 6, 8, 10]
if 2 in numbers:
    print('Список numbers содержит число 2')
else:
    print('Список numbers не содержит число 2')
```

Такой код проверяет, содержит ли список numbers число 2 и выводит соответствующий текст:

```
Список numbers содержит число 2
```

Мы можем использовать оператор in вместе с логическим оператором not. Например

```
numbers = [2, 4, 6, 8, 10]
if 0 not in numbers:
    print('Список numbers не содержит нулей')
```

Индексация

При работе со строками мы использовали индексацию, то есть обращение к конкретному символу строки по его индексу. Аналогично, можно индексировать и списки.

Для индексации списков в Python используются квадратные скобки [], в которых указывается индекс (номер) нужного элемента в списке:

Пусть numbers = [2, 4, 6, 8, 10].

Таблица ниже показывает, как работает индексация:

Выражение	Результат	Пояснение
numbers[0]	2	первый элемент списка
numbers[1]	4	второй элемент списка
numbers[2]	6	третий элемент списка
numbers[3]	8	четвертый элемент списка
numbers[4]	10	пятый элемент списка

Обратите внимание: первый элемент списка numbers[0], а не numbers[1]. Программисты же всё считают с нуля.

Так же, как и в строках, для нумерации с конца разрешены отрицательные индексы.

Выражение	Результат	Пояснение
numbers[-1]	10	пятый элемент списка
numbers[-2]	8	четвертый элемент списка
numbers[-3]	6	третий элемент списка
numbers[-4]	4	второй элемент списка
numbers[-5]	2	первый элемент списка

Как и в строках, попытка обратиться к элементу списка по несуществующему индексу:

```
print(numbers[17])
```

вызовет ошибку:

```
IndexError: index out of range
```

Срезы

Рассмотрим список numbers = [2, 4, 6, 8, 10].

С помощью среза мы можем получить несколько элементов списка, создав диапазон индексов, разделенных двоеточием numbers[x:y].

Следующий программный код:

```
print(numbers[1:3])
print(numbers[2:5])
```

выводит:

```
[4, 6]
[6, 8, 10]
```

При построении среза numbers[x:y] первое число – это то место, где начинается срез (включительно), а второе – это место, где заканчивается срез (невключительно). Разрезая списки, мы создаем новые списки, по сути, подсписки исходного.

При использовании срезов со списками мы также можем опускать второй параметр в срезе numbers[x:] (но поставить двоеточие), тогда срез берется до конца списка. Аналогично если опустить первый параметр numbers[:y], то можно взять срез от начала списка.

Срез numbers[:] возвращает копию исходного списка.

Как и в строках, мы можем использовать отрицательные индексы в срезах списков.
Использование срезов для изменения элементов в заданном диапазоне

Для изменения целого диапазона элементов списка можно использовать срезы. Например, если мы хотим перевести на русский язык названия фруктов 'banana', 'cherry', 'kiwi', то это можно сделать с помощью среза.

Следующий программный код:

```
fruits = ['apple', 'apricot', 'banana', 'cherry', 'kiwi', 'lemon', 'mango']
fruits[2:5] = ['банан', 'вишня', 'киви']
print(fruits)
```

выводит:

```
['apple', 'apricot', 'банан', 'вишня', 'киви', 'lemon', 'mango']
```

Операция конкатенации + и умножения на число *

Мы можем применять операторы + и * для списков подобно тому, как мы это делали со строками.

Следующий программный код:

```
print([1, 2, 3, 4] + [5, 6, 7, 8])
print([7, 8] * 3)
print([0] * 10)
```

выводит:

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

Для генерации списков, состоящих строго из повторяющихся элементов, умножение на число — самый короткий и правильный метод.

Мы также можем использовать расширенные операторы += и *= при работе со списками.

Следующий программный код:

```
a = [1, 2, 3, 4]
b = [7, 8]
a += b   # добавляем к списку a список b
b *= 5   # повторяем список b 5 раз
print(a)
print(b)
```

выводит:

```
[1, 2, 3, 4, 7, 8]
[7, 8, 7, 8, 7, 8, 7, 8, 7, 8]
```

Встроенные функции sum(), min(), max()

Встроенная функция sum() принимает в качестве параметра список чисел и вычисляет сумму его элементов.

Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print('Сумма всех элементов списка =', sum(numbers))
```

выводит:

```
Сумма всех элементов списка = 55
```

Встроенные функции min() и max() принимают в качестве параметра список и находят минимальный и максимальный элементы соответственно.

Следующий программный код:

```
numbers = [3, 4, 10, 3333, 12, -7, -5, 4]
print('Минимальный элемент =', min(numbers))
print('Максимальный элемент =', max(numbers))
```

выводит:

```
Минимальный элемент = -7
Максимальный элемент = 3333
```

Отличие списков от строк

Несмотря на всю схожесть списков и строк, есть одно очень важное отличие: строки — неизменяемые объекты, а списки – изменяемые.

Следующий программный код:

```
s = 'abcdefg'
s[1] = 'x'    # пытаемся изменить 2 символ (по индексу 1) строки
```

приводит к ошибке:

```
object does not support item assignment
```

Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7]
numbers[1] = 101     # изменяем 2 элемент (по индексу 1) списка
print(numbers)
```

выводит:

```
[1, 101, 3, 4, 5, 6, 7]
```

Запомни: изменять отдельные символы строк нельзя, однако можно изменять отдельные элементы списков. Для этого используем индексатор и оператор присваивания.

Списки. Методы (добавление, удаление)

Добавление элементов. Метод append()

Мы научились создавать статические списки, то есть списки, элементы которых известны на этапе создания. Следующий шаг – научиться добавлять элементы в уже существующие списки.

Для добавления нового элемента в конец списка используется метод append().

Следующий программный код:

```
numbers = [1, 1, 2, 3, 5, 8, 13]  # создаем список
numbers.append(21)  # добавляем число 21 в конец списка
numbers.append(34)  # добавляем число 34 в конец списка
print(numbers)
```

выведет:

```
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Обратите внимание, для того чтобы использовать метод append(), нужно, чтобы список был создан (при этом он может быть пустым).

Следующий программный код:

```
numbers = []  # создаем пустой список
numbers.append(1)
numbers.append(2)
numbers.append(3)
print(numbers)
```

выведет:

```
[1, 2, 3]
```

Важно: мы не можем использовать индексаторы для установки значений элементов списка, если список пустой. Следующий программный код:

```
numbers = []  # создаем пустой список
numbers[0] = 1
numbers[1] = 2
numbers[2] = 3
print(numbers)
```

приводит к ошибке:

```
IndexError: list assignment index out of range
```

Метод extend()

Можно также расширить список другим списком путем вызова метода **extend()**.

Следующий программный код:

```
numbers = [0, 2, 4, 6, 8, 10]
odds = [1, 3, 5, 7]
numbers.extend(odds)
print(numbers)
```

выведет:

```
[0, 2, 4, 6, 8, 10, 1, 3, 5, 7]
```

Метод extend() как бы расширяет один список, добавляя к нему элементы другого списка.

Отличие между методами append() и extend() проявляется при добавлении строки к списку.

Следующий программный код:

```
words1 = ['iq option', 'stepik', 'beegeek']
words2 = ['iq option', 'stepik', 'beegeek']
words1.append('python')
words2.extend('python')

print(words1)
print(words2)
```

выведет:

```
['iq option', 'stepik', 'beegeek', 'python']
['iq option', 'stepik', 'beegeek', 'p', 'y', 't', 'h', 'o', 'n']
```

Метод append() добавляет строку 'python' целиком к списку, а метод extend() разбивает строку 'python' на символы 'p', 'y', 't', 'h', 'o', 'n' и их добавляет в качестве элементов списка.

Удаление элементов

С помощью оператора del можно удалять элементы списка по определенному индексу.

Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del numbers[5]    # удаляем элемент имеющий индекс 5
print(numbers)
```

выведет:

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

Элемент под указанным индексом удаляется, а список перестраивается.

Обратите внимание на синтаксис удаления, так как он отличается от обычного вызова метода. При удалении элементов не надо передавать аргумент внутри круглых скобок.

Оператор del работает и со срезами: мы можем удалить целый диапазон элементов списка.

Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del numbers[2:7]    # удаляем элементы с 2 по 6 включительно
print(numbers)
```

выведет:

```
[1, 2, 8, 9]
```

Мы можем удалить все элементы на четных позициях (0, 2, 4, ...) исходного списка.

Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del numbers[::2]
print(numbers)
```

выведет:

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

Списки. Вывод элементов списка

При выводе содержимого списка с помощью функции print() вывод элементов осуществляется в квадратных скобках, причем все элементы разделены запятой. Такой вывод не всегда удобен и предпочтителен, поэтому нужно уметь выводить элементы списка нужным нам способом.

Вывод с помощью цикла for

Для вывода элементов списка **каждого на отдельной строке** можно использовать следующий код:

Вариант 1. Если нужны индексы элементов:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

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

Мы передаем в функцию range() длину списка len(numbers). В нашем случае длина списка numbers, равна 11. Таким образом вызов функции range(len(numbers)) имеет вид range(11) и переменная цикла i последовательно перебирает все значения от 0 до 10. Это означает, что выражение numbers[i] последовательно вернет все элементы списка numbers. Такой способ итерации списка удобен, когда нам нужен не только сам элемент numbers[i], но и его индекс i.

Вариант 2. Если индексы не нужны:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in numbers:
    print(num)
```

Этот цикл пройдет по списку numbers, придавая переменной цикла num значение каждого элемента списка (!) в отличие от предыдущего цикла, в котором переменная цикла «бегала» по индексам списка.

Если требуется выводить элементы списка на одной строке, через пробел, то мы можем использовать необязательный параметр end функции print():

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for num in numbers:
    print(num, end=' ')
```

Вывод с помощью распаковки списка

В Python есть удобный способ вывода элементов списка без использования цикла for.

Вариант 1. Вывод элементов списка через один символ пробела:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(*numbers)
```

Такой код выведет:

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

Вариант 2. Вывод элементов списка, каждого на отдельной строке

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(*numbers, sep='\n')
```

Такой код выведет:

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

Поскольку строки содержат символы, подобно тому, как списки содержат элементы, то мы можем использовать распаковку строк точно так же, как и распаковку списков.

Такой код:

```
s = 'Python'
print(*s)
print()
print(*s, sep='\n')
```

выведет:

```
P y t h o n

P
y
t
h
o
n
```

Списки. Строковые методы

В предыдущем модуле мы детально изучили основные строковые методы, однако обошли стороной два важных: split() и join(), имеющих отношение к спискам. Они как бы противоположны по смыслу: метод split() разбивает строку по произвольному разделителю на список слов, а метод join() собирает строку из списка слов через заданный разделитель.

Метод split()

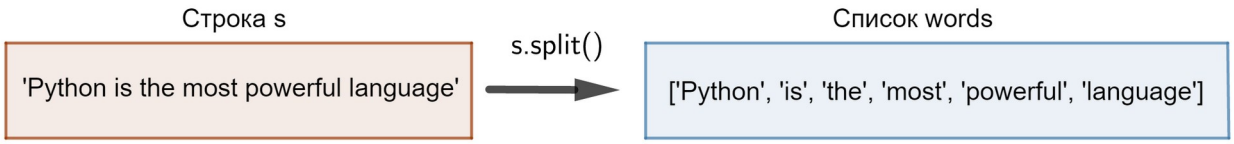
Метод split() разбивает строку на слова, используя в качестве разделителя последовательность пробельных символов.

Следующий программный код:

```
s = 'Python is the most powerful language'
words = s.split()
print(words)
```

выведет

```
['Python', 'is', 'the', 'most', 'powerful', 'language']
```



Таким образом, вызов метода split() разбивает строку на слова и возвращает список, содержащий все слова.

Рассмотрим следующий программный код:

```
numbers = input().split()
```

Если при запуске этой программы ввести строку 1 2 3 4 5, то список numbers будет следующим ['1', '2', '3', '4', '5']. Обратите внимание, что список будет состоять из строк, а не из чисел. Если требуется получить именно список чисел, то затем нужно элементы списка по одному преобразовать в числа:

```
numbers = input().split()
for i in range(len(numbers)):
    numbers[i] = int(numbers[i])
```

Необязательный параметр

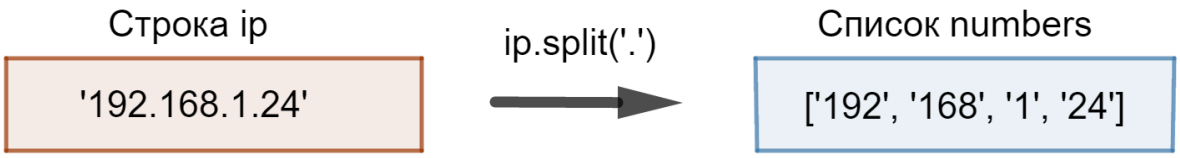
У метода split() есть необязательный параметр, который определяет, какой набор символов будет использоваться в качестве разделителя между элементами списка. Например, вызов метода split('.') вернет список, полученный разделением исходной строки по символу '.'

Следующий программный код:

```
ip = '192.168.1.24'
numbers = ip.split('.')  # указываем явно разделитель
print(numbers)
```

выведет список:

```
['192', '168', '1', '24']
```



Метод join()

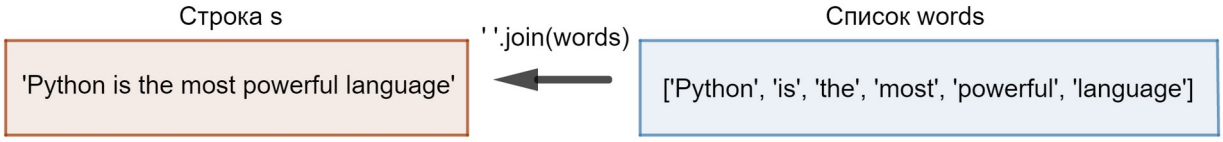
Метод join() собирает строку из элементов списка, используя в качестве разделителя строку, к которой применяется метод.

Следующий программный код:

```
words = ['Python', 'is', 'the', 'most', 'powerful', 'language']
s = ' '.join(words)
print(s)
```

выведет:

```
Python is the most powerful language
```



Обратите внимание, все слова разделены одним пробелом, поскольку метод join() вызывался на строке, состоящей из одного символа пробела ' '.

Рассмотрим еще пару примеров:

```
words = ['Мы', 'учим', 'язык', 'Python']
print('*'.join(words))
print('-'.join(words))
print('?'.join(words))
print('!'.join(words))
print('*****'.join(words))
print('abc'.join(words))
print('123'.join(words))
```

Результатом выполнения такого кода будет:

```
Мы*учим*язык*Python
Мы-учим-язык-Python
Мы?учим?язык?Python
```

```
Мы!учим!язык!Python
Мы*****учим*****язык*****Python
МыabcучимabcязыкabcPython
Мы123учим123язык123Python
```

Запомни: Строковый метод split() служит для преобразования строки в список, а метод join() — для преобразования списка в строку.

Примечания

Примечание 1. Существует большая разница между результатами вызова методов s.split() и s.split(' '). Разница в поведении проявляется, когда строка содержит несколько пробелов между словами.

Следующий программный код:

```
s = 'Python   is   the  most  powerful  language'
words1 = s.split()
words2 = s.split(' ')
print(words1)
print(words2)
```

выведет списки:

```
['Python', 'is', 'the', 'most', 'powerful', 'language']
['Python', '', '', '', 'is', '', '', 'the', '', 'most', '', 'powerful', '', 'language']
```

Примечание 2. Методы split() и join() являются строковыми методами. Следующий код приводит к ошибке:

```
print([1, 2].split())
print([1, 2].join([3, 4, 5]))
```

Примечание 3. Строковый метод join() работает только со списком строк. Следующий код приводит к ошибке:

```
numbers = [1, 2, 3, 4]  # список чисел
s = '*'.join(numbers)
print(s)
```

Списки. Методы списков.

Мы уже познакомились с двумя списочными методами append() и extend(). Первый добавляет в конец списка один новый элемент, а второй расширяет список другим списком. К спискам в Python применимы и другие удобные методы, с которыми мы познакомимся в этом уроке.

Метод insert()

Метод insert() позволяет вставлять значение в список в заданной позиции. В него передается два аргумента:

index: индекс, задающий место вставки значения;
value: значение, которое требуется вставить.

Когда значение вставляется в список, список расширяется в размере, чтобы разместить новое значение. Значение, которое ранее находилось в заданной индексной позиции, и все элементы после него сдвигаются на одну позицию к концу списка.

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']
print(names)
names.insert(0, 'Anders')
print(names)
names.insert(3, 'Josef')
print(names)
```

выведет:

```
['Gvido', 'Roman' , 'Timur']
['Anders', 'Gvido', 'Roman' , 'Timur']
['Anders', 'Gvido', 'Roman' , 'Josef', 'Timur']
```

При указании недопустимого индекса во время выполнения программы ошибки не происходит. Если задан индекс за пределами конца списка, то значение будет добавлено в конец списка. Если применен отрицательный индекс, который указывает на недопустимую позицию, то значение будет вставлено в начало списка.

Метод index()

Метод index() возвращает индекс первого элемента, значение которого равняется переданному в метод значению. Таким образом, в метод передается один параметр:

value: значение, индекс которого требуется найти.

Если элемент в списке не найден, то во время выполнения происходит ошибка.

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']
position = names.index('Timur')
print(position)
```

выведет:

```
2
```

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']
position = names.index('Anders')
print(position)
```

приводит к ошибке:

```
ValueError: 'Anders' is not in list
```

Чтобы избежать таких ошибок, можно использовать метод index() вместе с оператором принадлежности in:

```
names = ['Gvido', 'Roman' , 'Timur']
if 'Anders' in names:
    position = names.index('Anders')
    print(position)
else:
    print('Такого значения нет в списке')
```

Метод remove()

Метод remove() удаляет первый элемент, значение которого равняется переданному в метод значению. В метод передается один параметр:

value: значение, которое требуется удалить.

Метод уменьшает размер списка на один элемент. Все элементы после удаленного элемента смещаются на одну позицию к началу списка. Если элемент в списке не найден, то во время выполнения происходит ошибка.

Следующий программный код:

```
food = ['Рис', 'Курица', 'Рыба', 'Брокколи', 'Рис']
print(food)
food.remove('Рис')
print(food)
```

выведет:

```
['Рис', 'Курица', 'Рыба', 'Брокколи', 'Рис']
['Курица', 'Рыба', 'Брокколи', 'Рис']
```

Важно: метод remove() удаляет только первый элемент с указанным значением. Все последующие его вхождения остаются в списке. Чтобы удалить все вхождения, нужно использовать цикл while в связке с оператором принадлежности in и методом remove().

Метод pop()

Метод pop() удаляет элемент по указанному индексу и возвращает его. В метод pop() передается один необязательный аргумент:

index: индекс элемента, который требуется удалить.

Если индекс не указан, то метод удаляет и возвращает последний элемент списка. Если список пуст или указан индекс за пределами диапазона, то во время выполнения происходит ошибка.

Следующий программный код:

```
names = ['Gvido', 'Roman', 'Timur']
item = names.pop(1)
print(item)
print(names)
```

выведет:

```
Roman
['Gvido', 'Timur']
```

Метод count()

Метод count() возвращает количество элементов в списке, значения которых равны переданному в метод параметру.

Таким образом, в метод передается один параметр:

value: значение, количество вхождений которого нужно посчитать.

Если значение в списке не найдено, то метод возвращает 0.

Следующий программный код:

```
names = ['Timur', 'Gvido', 'Roman', 'Timur', 'Anders', 'Timur']
cnt1 = names.count('Timur')
cnt2 = names.count('Gvido')
cnt3 = names.count('Josef')
print(cnt1)
print(cnt2)
print(cnt3)
```

выведет:

```
3
1
0
```

Метод reverse()

Метод reverse() инвертирует порядок следования значений в списке, то есть меняет его на противоположный.

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']
names.reverse()
print(names)
```

выведет:

```
['Timur', 'Roman', 'Gvido']
```

Существует большая разница между вызовом метода names.reverse() и использованием среза names[::-1]. Метод reverse() меняет порядок элементов на обратный в текущем списке, а срез создает копию списка, в котором элементы следуют в обратном порядке.

Метод clear()

Метод clear() удаляет все элементы из списка.

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']
names.clear()
print(names)
```

выведет:

```
[]
```

Метод copy()

Метод copy() создает поверхностную копию списка.

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']
names_copy = names.copy()            # создаем поверхностную копию списка names
print(names)
print(names_copy)
```

выведет:

```
['Gvido', 'Roman', 'Timur']
['Gvido', 'Roman', 'Timur']
```

Аналогичного результата можно достичь с помощью срезов или функции list():

```
names = ['Gvido', 'Roman' , 'Timur']
names_copy1 = list(names)          # создаем поверхностную копию с помощью функции list()
names_copy2 = names[:]             # создаем поверхностную копию с помощью среза от начала
до конца
```

Подробнее про поверхностные копии мы поговорим в курсе для продвинутых.

Примечания

Примечание. Существует большая разница в работе строковых и списочных методов. Строковые методы не изменяют содержимого объекта, к которому они применяются, а возвращают новое значение. Списочные методы (кроме методов index(), count(), copy()), напротив, меняют содержимое объекта, к которому применяются.

Например, когда мы применяем метод lower() к строке original_string, то создаётся новая строка lowercase_string, которая не имеет ничего общего со строкой original_string. Более точно будет сказать, что строковый метод вернул новую строку, не изменяя старую:

```
original_string = "HELLO WORLD"
lowercase_string = original_string.lower()  # тут возвращается новая, модифицированная строка

print(original_string)   # HELLO WORLD
print(lowercase_string)  # hello world
```

Как мы видим, сама строка original_string не изменилась. Мы просто использовали её для создания строки lowercase_string через метод lower(). А lowercase_string действительно создана на основе строки original_string, где каждый символ в нижнем регистре.

Совершенно иная ситуация возникает при работе со списками. Например, при добавлении элемента в список через метод append() мы изменяем наш список, не создавая новый. Именно поэтому использование метода append() ничего не возвращает.

```
original_list = [1, 2, 4, 7]
modified_list = original_list.append(11)  # тут ничего не возвращается

print(original_list)  # [1, 2, 4, 7, 11]
print(modified_list)  # None
```

Как мы видим, original_list изменился. Элемент 11 был добавлен в конец. А вот наша попытка присвоить списку modified_list метод append() не увенчалась успехом. Был возвращён объект None (подробнее этот объект будет рассмотрен в курсе для продвинутых). Вкратце, объект None используется для обозначения отсутствия значения или буквально "ничего", т.е. пустоты.

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]
```

Примечания

Примечание 1. С помощью метода sort() можно сортировать списки содержащие не только числа, но и строки. В таком случае элементы списка сортируются в соответствии с лексикографическим порядком.

Следующий программный код:

```
a = ['бета', 'альфа', 'дельта', 'гамма']
a.sort()
print('Отсортированный список:', a)
```

выведет:

```
Отсортированный список: ['альфа', 'бета', 'гамма', 'дельта']
```

Примечание 2. Метод sort() использует алгоритм Timsort.

Списки. Списочные выражения

Аннотация. Списочные выражения. Создание списков без явного использования циклов и вызова списочного метода append().

Создание списков

Для того, чтобы создать список состоящий из 10 нулей мы можем использовать следующий код:

```
zeros = []
for i in range(10):
    zeros.append(0)
```

В Python, однако есть более простой и компактный способ для создания такого списка. Мы можем использовать оператор умножения списка на число:

```
zeros = [0] * 10
```

Для создания списков, заполненных по более сложным правилам нам приходится явно использовать цикл for.

Например, для создания списка целых чисел от 0 до 99, мы вынуждены писать такой код:

```
numbers = []
for i in range(10):
    numbers.append(i)
```

Такой код хоть и не является сложным, однако достаточно громоздок.

Списочные выражения

В Python есть механизм для создания списков из неповторяющихся элементов. Такой механизм называется — списочное выражение (list comprehension).

Предыдущий код можно записать следующим образом:

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

Общий вид списочного выражения следующий:

[выражение for переменная in последовательность]

где переменная — имя некоторой переменной, последовательность — последовательность значений, которые она принимает (список, строка или объект, полученный при помощи функции range), выражение — некоторое выражение, как правило, зависящее от использованной в списочном выражении переменной, которым будут заполнены элементы списка.

Примеры использования списочных выражений

1. Создать список, заполненный 10 нулями можно и при помощи списочного выражения:

```
zeros = [0 for i in range(10)]
```

2. Создать список, заполненный квадратами целых чисел от 0 до 9 можно так:

```
squares = [i ** 2 for i in range(10)]
```

3. Создать список, заполненный кубами целых чисел от 10 до 20 можно так:

```
cubes = [i ** 3 for i in range(10, 21)]
```

4. Создать список, заполненный символами строки:

```
chars = [c for c in 'abcdefg']
print(chars)
```

Считывание входных данных

При решении многих задач из предыдущих уроков мы считывали начальные данные (строки, числа) и заполняли ими список. С помощью списочных выражений процесс заполнения списка можно заметно сократить.

Например, если сначала вводится число n – количество строк, а затем сами строки, то создать список можно так:

```
n = int(input())
lines = [input() for _ in range(n)]
```

Можно опустить описание переменной n:

```
lines = [input() for _ in range(int(input()))]
```

Если требуется считать список чисел, то необходимо добавить преобразование типов:

```
numbers = [int(input()) for _ in range(int(input()))]
```

Обратите внимание, мы используем символ _ в качестве имени переменной цикла, поскольку она не используется.

Списочные выражения часто используются для инициализации списков. В Python не принято создавать пустые списки, а затем заполнять их значениями, если можно этого избежать.

Условия в списочном выражении

В списочных выражениях можно использовать условный оператор. Например, если требуется создать список четных чисел от 0 до 20, то мы можем написать такой код:

```
evens = [i for i in range(21) if i % 2 == 0]
```

Важно: для того, чтобы получить список, состоящий из четных чисел, лучше использовать функцию range(0, 21, 2). Предыдущий пример приведен для демонстрации возможности использования условий в списочных выражениях.

Вложенные циклы

В списочном выражении можно использовать вложенные циклы.

Следующий программный код:

```
numbers = [i * j for i in range(1, 5) for j in range(2)]
print(numbers)
```

выведет список:

```
[0, 1, 0, 2, 0, 3, 0, 4]
```

Такой код равнозначен следующему:

```
numbers = []
for i in range(1, 5):
    for j in range(2):
        numbers.append(i * j)
print(numbers)
```

Подводя итог

Пусть word = 'Hello', numbers = [1, 14, 5, 9, 12], words = ['one', 'two', 'three', 'four', 'five', 'six'].

Списочное выражение	Результирующий список
`[0 for i in range(10)]`	`[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`
`[i ** 2 for i in range(1, 8)]`	`[1, 4, 9, 16, 25, 36, 49]`
`[i * 10 for i in numbers]`	`[10, 140, 50, 90, 120]`
`[c * 2 for c in word]`	`['HH', 'ee', 'll', 'll', 'oo']`
`[m[0] for m in words]`	`['o', 't', 't', 'f', 'f', 's']`
`[i for i in numbers if i < 10]`	`[1, 5, 9]`
`[m[0] for m in words if len(m) == 3]`	`['o', 't', 's']`

Списки. Сортировка списков.

Аннотация. Задачи и способы (алгоритмы) сортировки списков.

Задача сортировки

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.**

Основные алгоритмы сортировки

Медленные:

Пузырьковая сортировка (Bubble sort);
Сортировка выбором (Selection sort);
Сортировка простыми вставками (Insertion sort).

Быстрые:

Сортировка Шелла (Shell sort);
Быстрая сортировка (Quick sort);
Сортировка слиянием (Merge sort);
Пирамидальная сортировка (Heap sort);
Сортировка TimSort (используется в Java и Python).

Большинство алгоритмов сортировки, в частности, указанные выше, основаны на сравнении двух элементов списка. Существуют однако алгоритмы не основанные на сравнениях. Такие алгоритмы как правило используют наперед заданные условия относительно элементов списка. Например, элементами списка являются натуральные или целые числа в некотором диапазоне, элементами являются строки и т.д.

К алгоритмам не основанным на сравнениях можно отнести следующие:

Сортировка подсчетом (Counting sort);

Блочная сортировка (Bucket sort);
Поразрядная сортировка (Radix sort).

В рамках курса мы рассмотрим несложные алгоритмы пузырьковой сортировки, сортировки выбором и сортировки простыми вставками.

Примечания

Примечание 1. Подробнее об алгоритмах сортировки можно почитать тут

Примечание 2. Мы называем некоторые алгоритмы сортировки медленными, поскольку они тратят много времени на сортировку больших списков. Например, если список содержит порядка миллиона элементов, то такие алгоритмы тратят часы, а то и дни на выполнение сортировки, в то время как быстрые алгоритмы справляются с задачей за секунды.

Примечание 3. Наглядную работу алгоритмов сортировки на разных входных данных можно посмотреть тут.

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 nn – 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.

Алгоритм пузырьковой сортировки считается учебным и практически не применяется вне учебной литературы, а на практике применяются более эффективные.

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

Первый проход:

[5, 1, 4, 2, 8] → [1, 5, 4, 2, 8]: меняем местами первый и второй элементы, так как 5>1;
[1, 5, 4, 2, 8] → [1, 4, 5, 2, 8]: меняем местами второй и третий элементы, так как 5>4;
[1, 4, 5, 2, 8] → [1, 4, 2, 5, 8]: меняем местами третий и четвертый элементы, так как 5>2;
[1, 4, 2, 5, 8] → [1, 4, 2, 5, 8]: не меняем четвертый и пятый элементы местами, так как 5<8;

Самый большой элемент встал («всплыл») на свое место.

Второй проход:

[1, 4, 2, 5, 8] → [1, 4, 2, 5, 8]: не меняем первый и второй элементы местами, так как 1<4;
[1, 4, 2, 5, 8] → [1, 2, 4, 5, 8]: меняем местами второй и третий элементы, так как 4>2;
[1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]: не меняем местами третий и четвертый элементы, так как 4 <5;

Второй по величине элемент встал («всплыл») на свое место.

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

Третий проход:

[1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]: не меняем первый и второй элементы местами, так как 1<2;
[1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]: не меняем второй и третий элементы местами, так как 2 <4;

Третий по величине элемент встал («всплыл») на свое место. (на котором и был)

Четвертый проход:

[1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]:

Четвертый по величине элемент встал («всплыл») на свое место.

Теперь список отсортирован и алгоритм может быть завершен.

Реализация алгоритма

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 - 1 - i):
        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]

Оптимизация алгоритма

Алгоритм пузырьковой сортировки можно немного ускорить. Если на одном из очередных проходов окажется, что обмены больше не нужны, то это означает, что все элементы списка находятся на своих местах, то есть список отсортирован. Для реализации такого ускорения нужно воспользоваться сигнальной меткой, то есть флажком и оператором прерывания break.

Списки. Вложенные списки. ч.1

Введение

Как мы уже знаем, список представляет собой упорядоченную последовательность элементов, индексы которых пронумерованы от 0. Элементами списка могут быть любые типы данных – числа, строки, булевы значения и т.д. Например,

```
numbers = [10, 3]
constants = [3.1415, 2.71828, 1.1415]
countries = ['Russia', 'Armenia', 'Argentina']
flags = [True, False]
```

Список numbers состоит из 2-х элементов, и каждый из них — целое число:

numbers[0] == 10;
numbers[1] == 3.

Список constants состоит из 3-х элементов, каждый из которых — вещественное число:

constants[0] == 3.1415;
constants[1] == 2.71828;
constants[2] == 1.1415

Список countries состоит из 3-х элементов, каждый из которых — строка:

countries[0] == 'Russia';
countries[1] == 'Armenia';
countries[2] == 'Argentina'

Список flags состоит из 2-х элементов, и каждый из них — булево значение:

flags[0] == True;
flags[1] == False.

Мы также говорили, что элементы списка не обязательно должны иметь одинаковый тип данных. Список может содержать значения разных типов данных:

```
info = ['Timur', 1992, 72.5]
```

Вложенные списки

Оказывается, элементами списков могут быть другие списки и в реальной разработке такая конструкция оказывается очень полезной. Такие списки называются вложенными списками.

Создание вложенного списка

Работа с вложенными списками принципиально ничем не отличается от работы со списками, например, чисел или строк. Чтобы создать вложенный список, мы также перечисляем элементы через запятую в квадратных скобках:

```
my_list = [[0], [1, 2], [3, 4, 5]]
```

Переменная my_list ссылается на список, состоящий из других списков (с вложенными списками).

Поскольку глубина вложенности списка my_list равна двум, то такой список обычно называют двумерным списком. На практике, как правило, мы работаем с двумерными списками, реже – с трехмерными.

Рассмотрим программный код:

```
my_list = [[0], [1, 2], [3, 4, 5]]
print(my_list)
print(my_list[0])
print(my_list[1])
print(my_list[2])
print(len(my_list))
```

Результатом работы такого кода будет:

```
[[0], [1, 2], [3, 4, 5]]
[0]
[1, 2]
[3, 4, 5]
3
```

Индексация

При работе с одномерными списками мы использовали индексацию, то есть обращение к конкретному элементу по его индексу. Аналогично можно индексировать и вложенные списки:

```
my_list = ['Python', [10, 20, 30], ['Beegeek', 'Stepik']]
print(my_list[0])
print(my_list[1])
print(my_list[2])
```

Результатом работы такого кода будет:

```
Python
[10, 20, 30]
['Beegeek', 'Stepik']
```

Так как элементы списка my_list – строка и списки, их также можно индексировать.

Рассмотрим программный код:

```
my_list = ['Python', [10, 20, 30], ['Beegeek', 'Stepik!']]
rint(my_list[0][2])       # индексирование строки 'Python'
print(my_list[1][1])      # индексирование списка [10, 20, 30]
print(my_list[2][-1])     # индексирование списка ['Beegeek', 'Stepik!']
```

```
print(my_list[2][-1][-1])  # индексирование строки 'Stepik!'
```

Результатом работы такого кода будет:

```
t
20
Stepik!
!
```

Попытка обратиться к элементу списка по несуществующему индексу:

```
print(my_list[3])  # у списка my_list индексы от 0 до 2
```

вызовет ошибку:

```
IndexError: index out of range
```

Функции len(), max(), min()

В прошлом курсе мы рассматривали встроенные функции max(), min(), len(), полезные и при работе с вложенными списками (обработке вложенных списков).

Функция len()

Рассмотрим программный код:

```
my_list = [[0], [1, 2], [3, 4, 5], [], [10, 20, 30]]
print(len(my_list))
```

Результатом работы такого кода будет:

```
5
```

Обратите внимание, встроенная функция len() возвращает количество элементов (число 5) списка my_list, а не общее количество элементов во всех списках (число 9).

Если требуется посчитать общее количество элементов во вложенном списке my_list, мы можем использовать цикл for в связке с функцией len():

```
total = 0
my_list = [[0], [1, 2], [3, 4, 5], [], [10, 20, 30]]
for li in my_list:
    total += len(li)
print(total)
```

Результатом работы такого кода будет:

```
9
```

Переменная li последовательно принимает все значения элементов списка my_list, то есть является сама по себе списком, поэтому мы можем вызывать функцию len() с переданным аргументом li.

Функции min() и max()

Функции min() и max() могут работать и со списками. Если этим функциям передается несколько списков, то целиком возвращается один из переданных списков. При этом сравнение происходит поэлементно: сначала сравниваются первые элементы списков. Если они не равны, то функция min() вернет тот список, первый элемент которого меньше, max() – наоборот. Если первые элементы равны, то будут сравниваться вторые и т. д.

Рассмотрим программный код:

```
list1 = [1, 7, 12, 0, 9, 100]
list2 = [1, 7, 90]
list3 = [1, 10]
print(min(list1, list2, list3))
print(max(list1, list2, list3))
```

Результатом работы такого кода будет:

```
[1, 7, 12, 0, 9, 100]
[1, 10]
```

Функции min() и max() также можно использовать при работе с вложенными списками. Рассмотрим программный код:

```
list1 = [[1, 7, 12, 0, 9, 100], [1, 7, 90], [1, 10]]
list2 = [['a', 'b'], ['a'], ['d', 'p', 'q']]
print(min(list1))
print(max(list1))
print(min(list2))
print(max(list2))
```

Результатом работы такого кода будет:

```
[1, 7, 12, 0, 9, 100]
[1, 10]
['a']
['d', 'p', 'q']
```

Обратите внимание – элементы вложенных списков в этой ситуации должны быть сравнимы.

Таким образом, следующий код:

```
my_list = [[1, 7, 12, 0, 9, 100], ['a', 'b']]
print(min(my_list))
print(max(my_list))
```

приведет к возникновению ошибки:

TypeError: '<' not supported between instances of 'str' and 'int'

Примечания

Примечание 1. Независимо от вложенности списков, нам нужно помнить по возможности все списочные методы:

метод append() добавляет новый элемент в конец списка;
метод extend() расширяет один список другим списком;
метод insert() вставляет значение в список в заданной позиции;
метод index() возвращает индекс первого элемента, значение которого равняется переданному в метод значению;
метод remove() удаляет первый элемент, значение которого равняется переданному в метод значению;
метод pop() удаляет элемент по указанному индексу и возвращает его;
метод count() возвращает количество элементов в списке, значения которых равны переданному в метод значению;
метод reverse() инвертирует порядок следования значений в списке, то есть меняет его на противоположный;
метод copy() создает поверхностную копию списка.;
метод clear() удаляет все элементы из списка;
оператор del позволяет удалять элементы списка по определенному индексу.

Примечание 2. Методы строк, работающие со списками:

метод split() разбивает строку на слова, используя в качестве разделителя последовательность пробельных символов, символ табуляции (\t) или символ новой строки (\n).

метод join() собирает строку из элементов списка, используя в качестве разделителя строку, к которой применяется метод.

Примечание 3. Язык Python не ограничивает нас в уровнях вложенности: элементами списка могут быть списки, их элементами могут быть другие списки, элементами которых в свою очередь могут быть другие списки...

Списки. Вложенные списки. ч 2.

Создание вложенных списков

Для создания вложенного списка можно использовать литеральную форму записи – перечисление элементов через запятую в квадратных скобках:

```
my_list = [[0], [1, 2], [3, 4, 5]]
```

Иногда нужно создать вложенный список, заполненный по определенному правилу – шаблону. Например, список длиной n, содержащий списки длиной m, каждый из которых заполнен нулями.

Рассмотрим несколько способов решения задачи.

Способ 1. Создадим пустой список, потом n раз добавим в него новый элемент – список длины m, составленный из нулей:

```
n, m = int(input()), int(input())    # считываем значения n и m
my_list = []

for _ in range(n):
    my_list.append([0] * m)

print(my_list)
```

Если ввести значения n = 3, m = 5, то результатом работы такого кода будет:

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Если передать значения n = 5, m = 3, то результатом работы такого кода будет:

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Способ 2. Сначала создадим список из n элементов (для начала просто из n нулей). Затем сделаем каждый элемент списка ссылкой на другой список из m элементов, заполненный нулями:

```
n, m = int(input()), int(input())  # считываем значения n и m
my_list = [0] * n

for i in range(n):
    my_list[i] = [0] * m

print(my_list)
```

Способ 3. Можно использовать генератор списка: создадим список из n элементов, каждый из которых будет списком, состоящих из m нулей:

```
n, m = int(input()), int(input())  # считываем значения n и m
my_list = [[0] * m for _ in range(n)]
print(my_list)
```

В этом случае каждый элемент создается независимо от остальных (заново конструируется вложенный список [0] * m для заполнения очередного элемента списка).

Обратите внимание, что очевидное решение, использующее операцию умножения списка на число (операция повторения), оказывается неверным:

```
n, m = int(input()), int(input())  # считываем значения n и m
my_list = [[0] * m ] * n
print(my_list)
```

В этом легко убедиться, если присвоить элементу my_list[0][0] любое значение, например, 17, а затем вывести список на печать:

```
n, m = int(input()), int(input())
my_list = [[0] * m ] * n
my_list[0][0] = 17
print(my_list)
```

Если ввести значения n = 5, m = 3, то результатом работы такого кода будет:

```
[[17, 0, 0], [17, 0, 0], [17, 0, 0], [17, 0, 0], [17, 0, 0]]
```

То есть, изменив значение элемента списка my_list[0][0], мы также изменили значения элементов my_list[1][0], my_list[2][0], my_list[3][0], my_list[4][0].

Причина такого поведения кроется в самой природе списков (тип list). В Python списки – ссылочный тип данных. Конструкция [0] * m возвращает **ссылку** на список из m нулей. Повторение этого элемента создает список из n ссылок на один и тот же список.

Вложенный список нельзя создать при помощи операции повторения (умножения списка на число). Для корректного создания вложенного списка мы используем способы 1–3, отдавая предпочтение способу 3.

Считывание вложенных списков

Если элементы списка вводятся через клавиатуру (каждая строка на отдельной строке, всего n строк, числа в строке разделяются пробелами), для ввода списка можно использовать следующий код:

```
n = 4                                  # количество строк (элементов)
my_list = []

for _ in range(n):
    elem = [int(i) for i in input().split()]  # создаем список из элементов строки
    my_list.append(elem)
```

В этом примере мы используем списочный метод append(), передавая ему в качестве аргумента другой список. Так у нас получается список списков.

В результате, если на вход программе подаются строки:

```
2 4
6 7 8 9
1 3
```

5 6 5 4 3 1

то в переменной my_list будет храниться список:

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

Не забывайте, что метод split() возвращает **список строк, а не чисел**. Поэтому мы предварительно сконвертировали строку в число, с помощью вызова функции int().

Также следует помнить отличие работы списочных методов append() и extend().

Следующий код:

```
n = 4
my_list = []

for _ in range(n):
    elem = [int(i) for i in input().split()]
    my_list.extend(elem)
```

создает одномерный **(!)** список, а не вложенный. В переменной my_list будет храниться список:

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

Перебор и вывод элементов вложенного списка

Как мы уже знаем, для доступа к элементу списка указывают индекс этого элемента в квадратных скобках. В случае двумерных вложенных списков надо указать два индекса (каждый в отдельных квадратных скобках), в случае трехмерного списка — три индекса и т. д.

Рассмотрим программный код:

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

print(my_list[0][0])
print(my_list[1][2])
print(my_list[2][1])
```

Результатом работы такого кода будет:

```
1
6
8
```

Когда нужно перебрать все элементы вложенного списка (например, чтобы вывести их на экран), обычно используются **вложенные циклы**.

Рассмотрим программный код:

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

for i in range(len(my_list)):
    for j in range(len(my_list[i])):
```

```
        print(my_list[i][j], end=' ')  # используем необязательный параметр end
    print()                            # перенос на новую строку
```

Результатом работы такого кода будет:

```
1 2 3
4 5 6
7 8 9
```

Вызов функции print() с пустыми параметрами нужен для того, чтобы переносить вывод на новую строку, после того как будет распечатан очередной элемент (список) вложенного списка.

В предыдущем примере мы перебирали **индексы элементов**, а можно сразу перебирать сами элементы вложенного списка:

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

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

Результатом работы такого кода будет:

```
1 2 3
4 5 6
7 8 9
```

Перебор элементов вложенного списка по индексам дает нам больше гибкости для вывода данных. Например, поменяв порядок переменных i и j, мы получаем иной тип вывода:

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

for i in range(len(my_list)):
    for j in range(len(my_list[i])):
        print(my_list[j][i], end=' ')  # выводим my_list[j][i] вместо my_list[i][j]
    print()
```

Результатом работы такого кода будет:

```
1 4 7
2 5 8
3 6 9
```

Обработка вложенных списков

Для обработки элементов вложенного списка, так же как и для вывода его элементов на экран, как правило, используются **вложенные циклы**.

Используем вложенный цикл для подсчета суммы всех чисел в списке:

```
my_list = [[1, 9, 8, 7, 4], [7, 3, 4], [2, 1]]
total = 0
for i in range(len(my_list)):
```

```
    for j in range(len(my_list[i])):
        total += my_list[i][j]

print(total)
```

Или то же самое с циклом не по индексу, а по значениям:

```
my_list = [[1, 9, 8, 7, 4], [7, 3, 4], [2, 1]]

total = 0
for row in my_list:
    for elem in row:
        total += elem

print(total)
```

Таким образом, можно обработать элементы вложенного списка практически в любом языке программирования. В Python, однако, можно упростить код, если использовать встроенную функцию sum(), которая принимает список чисел и возвращает его сумму. Подсчет суммы с помощью функции sum() выглядит так:

```
my_list = [[1, 9, 8, 7, 4], [7, 3, 4], [2, 1]]

total = 0
for row in my_list:  # в один цикл
    total += sum(row)
print(total)
```