

Python.

Списки. Продолжение.

Как было сказано на прошлом уроке, мы прошли далеко не все методы работы со списками. Продолжим их изучение.

Метод `extend()`

Метод `append()` добавляет элемент в конец списка, метод же `extend()` расширяет список добавляя к нему элементы другого списка, указанного в качестве параметра внутри скобок.

```
a = ["Иван", "Петр", "Василий", "Максим"]
b = ["Архип", "Денис"]
a.extend(b)
print(a)
```

Получим

```
['Иван', 'Петр', 'Василий', 'Максим', 'Архип', 'Денис']
```

Метод `insert()`

Если необходимо добавить элемент в список в заданной позиции, то используют метод `insert()`. В него передаются два параметра: номер позиции, в который необходимо вставить элемент и, собственно, само значение вставляемого элемента.

```
a = ["Иван", "Петр", "Василий", "Максим"]
a.insert(0, "Архип")
print(a)
a.insert(2, "Денис")
print(a)
```

Получим

```
['Архип', 'Иван', 'Петр', 'Василий', 'Максим']
['Архип', 'Иван', 'Денис', 'Петр', 'Василий', 'Максим']
```

При указании индекса больше длины списка, ошибки не происходит, а элемент просто добавляется в конец списка. И соответственно, если отрицательный индекс выйдет за начало строки, то элемент добавляется в начало строки.

Метод `pop()`

Разберем подробнее этот метод. Как было сказано ранее метод без параметров возвращает значение последнего элемента из списка и при этом удаляет его. Если в качестве параметра указать целое число, то будет извлечен элемент с указанным индексом.

```
a = ["Иван", "Петр", "Василий", "Максим"]
s=a.pop(1)
print(s)
print(a)
```

Получим

```
Петр
['Иван', 'Василий', 'Максим']
```

В качестве параметра можно указывать отрицательные числа, тогда отсчет будет идти с конца списка.

```
a = ["Иван", "Петр", "Василий", "Максим"]
s=a.pop(-2)
print(s)
print(a)
```

Получим

Василий
['Иван', 'Петр', 'Максим']

Если значение параметра будет больше номера последнего элемента списка, то произойдет ошибка.

Если нет необходимости получать значение элемента, а просто нужно удалить элемент по номеру индекса можно воспользоваться оператором **del**.

```
a = ["Иван", "Петр", "Василий", "Максим"]
del a[2]
print(a)
```

Получим

['Иван', 'Петр', 'Максим']

Обратите внимание на синтаксис оператора **del**. Скобки не используются.

Оператор позволяет удалять не только один элемент, но и срезы.

```
a = [2, 12, 23, 1, 17, 156]
del a[2:5]
print(a)
```

Получим

[2, 12, 156]

В результате выполнения следующей программы

```
a = [2, 12, 23, 1, 17, 156]
del a[::2]
print(a)
```

Получим

[12, 1, 156]

Метод **remove()**

Метод удаляет элемент из списка не по номеру, а по его значению. Если элементов с таким значением несколько то удаляется первый из них.

```
a = ["Иван", "Петр", "Архип", "Василий", "Архип", "Максим"]
a.remove("Архип")
print(a)
```

Получим

['Иван', 'Петр', 'Василий', 'Архип', 'Максим']

Если элемента с указанным значением нет в списке, то произойдет ошибка. Поэтому, прежде чем удалять элемент, желательно убедиться, что он есть в списке.

Оператор принадлежности **in**

Оператор **in** позволяет проверить, содержит ли список некоторый элемент.

```
a = ["Иван", "Петр", "Архип", "Василий", "Максим"]
if "Архип" in a:
    print('Архип в списке')
else:
    print('Архипа нет в списке')
```

Метод **index()**

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

```
a = ["Иван", "Петр", "Архип", "Василий", "Максим"]
pos = a.index("Архип")
print(pos)
```

Получим

2

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

В результате выполнения следующей программы

```
a = ["Иван", "Петр", "Архип", "Василий", "Максим"]
pos = a.index("Денис")
print(pos)
```

произойдет ошибка

ValueError: 'Денис' is not in list

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

```
a = ["Иван", "Петр", "Архип", "Василий", "Максим"]
if "Денис" in a:
    pos = a.index("Денис")
    print(pos)
else:
    print('Такого значения нет в списке')
```

Метод **count()**

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

Таким образом, в метод передается один параметр/

Если значение в списке не найдено, то метод возвращает 0

Следующий программный код:

```
a = ["Иван", "Архип", "Петр", "Архип", "Василий", "Максим"]
print(a.count("Архип"))
print(a.count("Василий"))
print(a.count("Денис"))
```

выведет

```
2  
1  
0
```

Метод join()

В прошлый раз мы познакомились с методом **split()**, который строку разбивал на слова и помещал их в список. Метод **join()** делает обратное. Он собирает строку из элементов списка, используя в качестве разделителя строку, к которой применяется метод.

```
a = ['День', 'сегодня', 'выдался', 'теплый']  
s = " ".join(a)  
print(s)
```

Получим

День сегодня выдался теплый

В качестве разделителей можно указывать не только пробел, но и любой другой набор символов, в том числе и пустую строку. В последнем случае все слова из списка склеятся без пробелов и других разделителей.

```
a = ['День', 'сегодня', 'выдался', 'теплый']  
print("-".join(a))  
print("+".join(a))  
print("&&&".join(a))  
print(" ура ".join(a))  
print("<>".join(a))  
print("")join(a))
```

Получим

День-сегодня-выдался-теплый

День+сегодня+выдался+теплый

День&&&сегодня&&&выдался&&&теплый

День ура сегодня ура выдался ура теплый

День<>сегодня<>выдался<>теплый

Деньсегоднявыдалсятеплый

Списочные выражения

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

Например, чтобы заполнить массив числами от 0 до 9 можно написать следующий код

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

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 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 считывать список чисел, то необходимо добавить преобразование типов:

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

Обратите внимание, был использован символ **_** (знак нижнего подчеркивания) в качестве имени переменной цикла, поскольку она не используется.

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]
```

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)
```