Другие методы списков и строк

Аннотация

В материалах урока приводятся таблицы с почти полным перечнем методов списков и строк, которые можно использовать как справочный материал. Рассматривается неявное приведение объектов к булеву типу. Приводятся примеры цепочек вызова методов. Появление метода pop позволяет познакомиться с понятием стека.
Этот урок отличается от прочих: большую часть его материала не нужно запоминать, но можно использовать как справочный материал. С этой же целью вводятся функции dir и help.

1.Как пользоваться таблицами

Мы уже знакомы с некоторыми функциями для работы со списками, строками и множествами. Мы знаем также методы для работы с элементами множеств (`add`, `remove`, `discard` и т. д.), метод `append`, который позволяет добавлять элементы в список, метод `split` для разбиения строки на список «слов» и метод `join` для «сшивки» списка строк в одну. Однако мы не знаем, как делать со списками многие другие вещи: добавлять элементы в произвольное место, удалять элементы и т. д. Такие задачи решаются с помощью других функций и методов.

Далее приводится перечень основных методов списков и строк, а также функций и специальных синтаксических конструкций для работы с ними, включая уже знакомые нам. Если при написании программы вы забудете, как называется тот или иной метод или какие у него аргументы, смело заглядывайте в материалы этого урока,

в документацию, пользуйтесь поиском в Интернете или описанными здесь функциями `dir` и `help`.

Важно!

Главное — усвоить основные возможности методов для стандартных типов строки и списка, а детали (например, порядок аргументов в методе `insert` или название именованного аргумента в методе `split`) всегда можно посмотреть здесь или в документации.

В таблице предполагается, что `a` и `a2` — списки, `x` — элемент списка, `s` и `s2` — строки, `c` — символ (строка длины 1), `n` — индекс элемента списка или строки, `start`, `stop`, `step` — границы среза (т. е. тоже индексы) и шаг среза, `k` — натуральное число.

Для методов и функций также даны примеры выражений, которые будут истинными True и демонстрируют действие этого метода. Например, выражения `5 in [2, 3, 5]` и `'abc'.isalpha()` равны True. Их можно подставить в оператор `if` (и соответствующий блок будет выполнен) или в функцию `print` (и тогда она выведет True).

2.Методы списков

Операция	Описание	Пример
`x in a`	Проверка, что `x` содержится в `a`	`5 in [2, 3, 5]`
`x not in a`	Проверка, что `x` не содержится в `a` То же, что и `not (x in a)`	`5 not in [2, 3, 6]`
`a + a2`	Конкатенация списков, то есть новый список, в котором сначала идут все элементы `a`, а затем все элементы `a2`	`[2, 4] + [5, 3] == [2, 4, 5, 3]`
`a * k`	Список `a`, повторенный `k` раз	`[2, 3] * 3 == [2, 3, 2, 3, 2, 3]`

`a[n]`	n-й элемент списка, отрицательные n — для отсчета с конца	`[2, 3, 7][0] == 2` `[2, 3, 7][-1] == 7`
`a[start:stop:step]`	Срез списка	`[2, 3, 7][:2] == [2, 3]`
`len(a)`	Длина списка	`len([2, 3, 7]) == 3`
`max(a)`	Максимальный элемент списка	`max([2, 3, 7]) == 7`
`min(a)`	Минимальный элемент списка	`min([2, 3, 7]) == 2`
`sum(a)`	Сумма элементов списка	`sum([2, 3, 7]) == 12`
`a.index(x)`	Индекс первого вхождения x в a (вызовет ошибку, если x `not in` a, то есть если x отсутствует в a)	`[2, 3, 7].index(7) == 2`
`a.count(x)`	Количество вхождений x в a	`[2, 7, 3, 7].count(7) == 2`

`a.append(x)`	Добавить x в конец a	`a = [2, 3, 7]`
		`a.append(8)`
		`a == [2, 3, 7, 8]`

`a.extend(a2)`	Добавить элементы коллекции a2 в конец a	`a = [2, 3, 7]`
		`a.extend([8, 4, 5])`
		`a == [2, 3, 7, 8, 4, 5]`

`del a[n]`	Удалить n-й элемент списка	`a = [2, 3, 7]`
		`del a[1]`
		`a == [2, 7]`

`del a[start:stop:step]`	Удалить из a все элементы, попавшие в срез	`a = [2, 3, 7]`
		`del a[:2]`
		`a == [7]`

`a.clear()`	Удалить из a все элементы (то же, что `del a[:]`)	`a.clear()`
`a.copy()`	Копия a (то же, что и полный срез `a[:]`)	`b = a.copy()`
`a += a2` `a *= k`	Заменить содержимое списка на `a + a2` и `a * k` соответственно	

`a.insert(n, x)`	Вставить x в a на позицию n, подвинув последующую часть дальше	`a = [2, 3, 7]`
		`a.insert(0, 8)`
		`a == [8, 2, 3, 7]`

`a.pop(n)`	Получить n-й элемент списка и одновременно удалить его из списка. Вызов метода без аргументов равносилен удалению последнего элемента:
	`a.pop() == a.pop(-1)`
		`a = [2, 3, 7]`
		`a.pop(1) == 3`
		`a == [2, 7]`

`a.remove(x)`	Удалить первое вхождение x в a, в случае `x not in a` — ошибка	`a = [2, 3, 7]`
		`a.remove(3)`
		`a == [2, 7]`

`a.reverse()`	Изменить порядок элементов в a на обратный (перевернуть список)	`a = [2, 3, 7]`
		`a.reverse()`
		`a == [7, 3, 2]`

`a.sort()`	Отсортировать список по возрастанию	`a = [3, 2, 7]`
		`a.sort()`
		`a == [2, 3, 7]`

`a.sort(reverse=True)`	Отсортировать список по убыванию	`a = [3, 2, 7]`
		`a.sort(reverse = True)`
		`a == [7, 3, 2]`

`bool(a)`	Один из способов проверить список на пустоту (возвращает `True`, если список непустой, и `False` в противном случае)	

3.Методы строк

Операция	**Описание**	**Пример**
`s2 in s`	Проверка, что подстрока `s2` содержится в `s`	`'m' in 'team'`
`s2 not in s`	Проверка, что подстрока `s2` не содержится в `s` то же, что `not (s2 in s)`	`'I' not in 'team'`
`s + s2`	Конкатенация (склейка) строк, то есть строка, в которой сначала идут все символы из `s`, а затем все символы из `s2`	`'tea' + 'm' == 'team'`
`s * k`	Строка `s`, повторенная `k` раз	`'ha' * 3 == 'hahaha'`
`s[n]`	`n`-й элемент строки, отрицательные `n` — для отсчета с конца	`'team'[2] == 'a'` `'team'[-1] == 'm'`
`s[start:stop:step]`	Срез строки	`'mama'[:2] == 'ma'`

`len(s)`	Длина строки	`len('abracadabra') == 11`

`s.find(s2)`
`s.rfind(s2)`
	Индекс начала первого или последнего вхождения подстроки `s2` в `s` (вернет -1, если `s2 not in s`)	`s = 'abracadabra'`
		`s.find('ab') == 0`
		`s.rfind('ab') == 7`
		`s.find('x') == -1`

`s.count(s2)`	Количество неперекрывающихся вхождений `s2` в `s`	`'abracadabra'.count('a') == 5`
`s.startswith(s2)` `s.endswith(s2)`	Проверка, что `s` начинается с `s2` или оканчивается на `s2`	`'abracadabra'.startswith('abra')`
`s += s2` `s *= k`	Заменить содержимое строки на `s + s2` и `s * k` соответственно	

`s.isdigit()`
`s.isalpha()`
`s.isalnum()`
	Проверка, что в строке `s` все символы — цифры,
	буквы (включая кириллические),
	цифры или буквы соответственно
		`'100'.isdigit()`
		`'abc'.isalpha()`
		`'E315'.isalnum()`

`s.islower()` `s.isupper()`	Проверка, что в строке `s` не встречаются большие буквы, маленькие буквы. Обратите внимание, что для обеих этих функций знаки препинания и цифры дают True	`'hello!'.islower()` `'123PYTHON'.isupper()`

`s.lower()` `s.upper()`	Строка s, в которой все буквы (включая кириллические) приведены к верхнему или нижнему регистру, т. е. заменены на строчные (маленькие) или заглавные (большие)	`'П р и в е т!'.lower() == 'п р и в е т!'` `'П р и в е т!'.upper() == 'П Р И В Е Т!'`
`s.capitalize()`	Строка s, в которой первая буква — заглавная	`'п р и в е т'.capitalize() == 'П р и в е т'`

`s.lstrip()`
`s.rstrip()`
`s.strip()`
	Строка s, у которой удалены символы пустого пространства (пробелы, табуляции) в начале,
	в конце или с обеих сторон
		`' П р и в е т! '.strip() == 'П р и в е т!'`

`s.ljust(k, c)` `s.rjust(k, c)`	Добавляет справа или слева нужное количество символов c, чтобы длина s достигла k	`'П р и в е т'.ljust(8, '!') == 'П р и в е т!!'`
`s.join(a)`	Склеивает строки из списка a через символ s	`'+'.join(['В а с я', 'М а ш а']) == 'В а с я+М а ш а'`
`s.split(s2)`	Список всех слов строки s (подстрок, разделенных строками s2)	`'Р а з  д в а т р и!'.split('а') == ['Р', 'з  д в', ' т р и!']`

`s.replace(s2, s3)`	Строка `s`, в которой все неперекрывающиеся вхождения `s2` заменены на `s3` Есть необязательный третий параметр, с помощью которого можно указать, сколько раз производить замену	`'Раз два три!'.replace('а', 'я') =='Ряз двя три!'` `'Раз два три!'.replace('а', 'я', 1) == 'Ряз два три!'`
`list(s)`	Список символов из строки строки `s`	`list('Привет') == ['П', 'р', 'и', 'в', 'е', 'т']`
`bool(s)`	Проверка, что строка не пустая (возвращает `True`, если не пустая, и `False` в противном случае)	
`int(s)` `float(s)`	Если в строке `s` записано целое (дробное) число, получить это число, иначе — ошибка	`int('25') == 25`
`str(x)`	Представить любой объект `x` в виде строки	`str(25) == '25'`

Важно!

Обратите внимание: никакие методы строк, включая `s.replace(...)`, не изменяют саму строку `s`. Все они лишь возвращают измененную строку, в отличие от большинства методов списков.

`a.sort()`, например, ничего не возвращает, а изменяет сам список `a`.

4.Функции dir и help

Для получения информации о списке методов любого объекта (в том числе тех, о которых вы вряд ли хотели узнать) в Python существует специальная функция dir. Например, `dir([])` вернет все методы списков, и оператор `print(dir([]))` выведет длинный список, оканчивающийся так: `'index', 'insert', 'pop', 'remove', 'reverse', 'sort'`.

Если же нам нужно узнать справочную информацию по конкретному методу или типу данных, для этого существует функция help. `help([])` выведет на экран много информации, большая часть которой пока лишняя. А вот `help([].insert)` выведет на экран краткую справку именно по методу `insert`, подсказывая, что первый аргумент этого метода — индекс, а второй — тот объект, который нужно вставить в список на этот индекс.

Важно!

Заметьте: при вызове справки по методу `help([].insert)` после `insert` нет скобок (...) — ведь мы не хотим вызвать этот метод, чтобы вставить что-то в какой-то список. Функции `help` в качестве аргумента передается сам метод `insert`.

5.Форматированный вывод. f-строки

Этот блок урока не имеет отношения к методам списков и строк, но позволит нам познакомиться с возможностями форматированного вывода на экран переменных и выражений.

Начиная с версии 3.6 в Python появился новый тип строк — f-строки, которые улучшают читаемость кода и работают быстрее других способов форматирования.

f-строки

f-строки, которые буквально означают formatted string, задаются с помощью литерала f перед кавычками:

```
>>> "обычная строка"
```

```
>>> f"f-строка"
```

f-строки делают очень простую вещь: они берут значения переменных, которые есть в текущей области видимости, и подставляют их в строку. В самой строке вам лишь нужно указать имя этой переменной в фигурных скобках.

```
name = "Аркадий"

age = 14

print(f"Меня зовут {name} Мне {age} лет.")
```

```
Меня зовут Аркадий. Мне 14 лет.
```

В качестве подставляемого значения внутри фигурных скобок может быть:

-Имя переменной (`f"Меня зовут {name}"`)
-Элемент списка (`f"Третий месяц в году - {month[2]}"`) или словаря
-Методы объектов (`f"Имя: {name.upper()}"`)
-Базовые арифметические операции (`f"({x} + {y}) / 2 = {(x + y) / 2}"`)
-И даже вызов функции (`f"13 / 7 = {round(13/7)}"`)

Кроме того, вы можете задавать форматирование для чисел, например:

-Указать необходимое количество знаков после запятой, спецификатор `f` отвечает за вывод чисел с плавающей точкой (тип `float`):

```
print(f"Число пи по Архимеду - {(22/7):.2f}")
```

-Представить результат в двоичной системе счисления, используя спецификатор `b`:

```
print(f"10 в двоичной системе счисления - {(10):b}")
```

Аналогично для шестнадцатеричной системы счисления используется спецификатор `x`, а для восьмеричной — `o`.

Подробнее о допустимых вариантах форматирования можно почитать в документации или на вот этом ресурсе.

6.Цепочки вызовов

Бывает удобно строить последовательные цепочки вызовов методов.

Например, `s.strip().lower().replace('ё', 'е')` выдаст строку `s`, в которой убраны начальные и конечные пробелы, все буквы сделаны маленькими, после чего убраны все точки над Ё. В результате этого преобразования строка ' Зелёный клён ' превратится в 'зеленый клен'.

Другой пример:

```
'мало Средне
МНОГО'.lower().split().index('средне')
```

```
1
```

В данном примере строка сначала полностью приводится к нижнему регистру (все буквы маленькие) при помощи метода `lower()`. Затем она превращается в список слов `['мало', 'средне', 'много']` при помощи `split()`. Далее метод `index` находит в этом списке слово «средне» на позиции номер 1.

7.Использование методов списков. Структура данных «Стек»

Приведем пример использования методов списков append и pop для моделирования структуры данных «Стек». Эта структура данных часто используется при решении различных задач (например, при вычислении выражений).

Представьте себе стопку сложенных футболок или любых других неодинаковых предметов, которые можно сложить в стопку. Из стопки удобно забрать самый верхний предмет — тот, который положили в нее последним. Если вы полностью разберете стопку, будете брать футболки в порядке, обратном тому, в котором их в нее клали.

Стек

Структура данных называется стек (stack, что и значит «стопка»). По-английски такую организацию данных называют **LIFO — Last In First Out — Первым Пришел Последним Ушел!**

Если пользоваться только методами append и pop без аргументов, список оказывается как раз такой стопкой.

Так, в примере ниже порядок вывода будет обратным порядку ввода:

```
stack = []

for i in range(5):

    print('К а к у ю  ф у т б о л к у  в ы  к л а д ё т е  с в е р х у  с т о п к и ?')

    stack.append(input())

while stack:  # п о к а  с т о п к а  н е  з а к о н ч и л а с ь

    print('С е г о д н я  в ы  н а д е в а е т е  ф у т б о л к у', stack.pop())
```

```
К а к у ю  ф у т б о л к у  в ы  к л а д ё т е  с в е р х у  с т о п к и ?
б е л у ю
К а к у ю  ф у т б о л к у  в ы  к л а д ё т е  с в е р х у  с т о п к и ?
к р а с н у ю
К а к у ю  ф у т б о л к у  в ы  к л а д ё т е  с в е р х у  с т о п к и ?
с и н ю ю
К а к у ю  ф у т б о л к у  в ы  к л а д ё т е  с в е р х у  с т о п к и ?
з е л е н у ю
К а к у ю  ф у т б о л к у  в ы  к л а д ё т е  с в е р х у  с т о п к и ?
в е с е л у ю
С е г о д н я  в ы  н а д е в а е т е  ф у т б о л к у  в е с е л у ю
С е г о д н я  в ы  н а д е в а е т е  ф у т б о л к у  з е л е н у ю
С е г о д н я  в ы  н а д е в а е т е  ф у т б о л к у  с и н ю ю
С е г о д н я  в ы  н а д е в а е т е  ф у т б о л к у  к р а с н у ю
С е г о д н я  в ы  н а д е в а е т е  ф у т б о л к у  б е л у ю
```

Здесь использовалась конструкция `while stack:`. В условии оператора `if` или `while` любой объект интерпретируется как `bool`: либо как True, либо как False. В случае списков и строк в False превращаются только `[]` и `''` соответственно (а в случае чисел — только ноль).

Иными словами, можно было бы написать `while bool(stack):` или `while stack != []:` и получить тот же самый результат, но самый короткий и общепринятый вариант — просто `while stack:`.