ЯЗЫК PYTHON

Лекция 6. Методы строк

lower/upper

- Все обнаруженные буквы превращаются
- В строчные/заглавные
- Частое использование - для сравнения без учета регистра
- Обе строки перед сравнением переводим в общий регистр

Пример

```
1 value1 = 'Строка с РазНыми раЗМЕРАми букв'
2 value2 = 'СтроКа С разНыми РаЗмЕРаМи Букв'
3 print(value1.lower())  # 'строка с разными размерами букв'
4 print(value1.lower())  # 'строка с разными размерами букв'
5 print(value1.lower() == value2.lower())  # True
6 print(value1.upper() == value2.upper())  # True
```

is-методы

- Проверяют все символы строки
- На принадлежность определенному классу
- isalpha - буквы
- isdigit - цифры
- isalnum - буквы или цифры
- islower/isupper - строчные/заглавные
- Другие - см. документацию

Пример

```
 1 print('aerwgerj'.isalpha())  # True
 2 print('aerw gerj'.isalpha())  # False
 3 print('325325'.isdigit())  # True
 4 print('23252e'.isdigit())  # False
 5 print('23252e'.isalnum())  # True
 6 print('23252 '.isalnum())  # False
 7 print('aerwgerj'.islower())  # True
 8 print('Aerwgerj'.islower())  # False
 9 print('AERWGER'.isupper())  # True
10 print('Aerwgerj'.isupper())  # False
```

Не совсем логично, но так сделано

```
1 print(''.isalpha())  # False
2 print(''.isdigit())  # False
3 print(''.isalnum())  # False
4 print(''.islower())  # False
5 print(''.isupper())  # False
```

find/index

- Ищут подстроку в строке
- Если нашли, возвращают индекс начала совпадения
- Разница - в поведении при отсутствии вхождения
- find - возвращает -1
- index - бросает исключение

Примеры find/index

```
 1 print('hello, world'.find('hello'))  # 0
 2 print('hello, world'.find('world'))  # 7
 3 print('hello, world'.index('hello'))  # 0
 4 print('hello, world'.index('world'))  # 7
 5 print('hello, world'.find(' '))  # 6
 6 print('hello, world'.index(' '))  # 6
 7 print('hello, world'.find('l'))  # 2
 8 print('hello, world'.index('l'))  # 2
 9 print('hello, world'.index( ''))  # 0
10 print('hello, world'.index(''))  # 0
11 print(''.find(''))  # 0
12 print(''.index('')) # 0
```

Примеры find/index: ненайденное

```
1 print('hello, world'.find('hello,world'))  # -1
2 print('hello, world'.find('  '))  # -1
3 print(''.find(' '))  # -1
4 print('hello, world'.index('hello,world'))  # исключение
5 print('hello, world'.index('  '))  # исключение
6 print(''.index(' '))  # исключение
```

Проверка префикса или суффикса

- Иногда хочется проверить, начинается ли строка с определенного префикса
- Например, читаем файл
- В каждой строке - какая-то запись
- А в некоторых строках - особо важная запись
- И она начинается с '### '

Проверка префикса или суффикса

- Технически можно реализовать как комбинацию отрезков и сравнений
- Например: s[:4] == '### '
- Но это некрасиво
- И не дает увидеть общий смысл происходящего
- Поэтому есть специальный метод

startswith/endswith

- startswith - проверяет начало строка
- endswith - проверяет конец строки
- s.startswith('### ')
- s.endswith('%%')
- Так понятнее

Склеивание строки из списка

- Есть список строк
- data = ['Мама', 'мыла', 'раму']
- Хотим склеить их в строку
- Теоретически, можем в цикле склеивать через сложение

Склеивание строки из списка

- Но у такого способа проблема не только с ясностью происходящего
- Проблема еще и в скорости
- Особенно если строки маленькие
- И их при это много

Склеивание строки из списка

- Получаем слишком много копирований
- И может работать заметно медленнее
- Есть метод join
- И это метод строки, а не списка
- Вызывается над строкой-разделителем
- А параметр - список

Пример на join

```
1 print(','.join(['Мама', 'мыла', 'раму'])) # "Мама,мыла,раму"
2 print(' '.join(['Мама', 'мыла', 'раму'])) # "Мама мыла раму"
```

Разбивка строки

- Часто нужно сделать обратное действие
- Разбить строку по разделителям
- И получить список элементов
- Можно решить, используя метод find, цикл и отрезки

Разбивка строки

- Найти первое вхождение разделителя
- Отрезок до него добавить в результат
- К отрезку после него применить те же действия
- Пока не переберем все разделители

split

- Есть специальный метод split
- Два параметра - строка-разделитель и лимит разбиений
- Если лимит разбиений -1, то разбиваем, пока есть разделители
- По умолчанию лимит разбиений -1, то есть ограничений нет

Простейшие примеры на split

```
print('vasya,petya,dima'.split(',')) # ['vasya', 'petya', 'd
print('vasya,petya,dima'.split(', ')) # ['vasya,petya,dima']
print('vasya, petya,dima'.split(', ')) # ['vasya', 'petya,di
print('vasya,petya,dima'.split(',', -1)) # ['vasya', 'petya'
print('vasya,petya,dima'.split(',', 0)) # ['vasya,petya,dima
print('vasya,petya,dima'.split(',', 1)) # ['vasya', 'petya,d
print('vasya,petya,dima'.split(',', 2)) # ['vasya', 'petya',
print('vasya,petya,dima'.split(',', 3)) # ['vasya', 'petya',
```

split

- Если разделитель повторяется
- В результат входит пустая строка
- Аналогично - если строка начинается с разделителя
- Или им заканчивается
- Даже если разделитель - пробел

Примеры на split с пустыми строками

```
1 print(',vasya,petya,dima'.split(',')) # ['', 'vasya', 'petya
2 print(',vasya,petya,dima'.split(',', 1)) # ['', 'vasya,petya
3 print(' vasya, petya,dima'.split(' ')) # ['', 'vasya,', 'pet
4 print('  vasya '.split(' ')) # ['', '', 'vasya', '']
5 print(' '.split(' ')) # ['', '']
6 print('  '.split(' ')) # ['', '', '']
7 print('hello  world'.split(' ')) # ['hello', '', 'world']
```

split

- Особый случай - None в качестве разделителя
- Он же и является значением по умолчанию
- В этом случае любая последовательность пробельных символов будет разделителем
- Это - едиинственный вариант разделителя длиннее одного символа

Примеры на split с длинными пробельными разделителями

```
1 print('vasya petya dima'.split(' ')) # ['vasya', 'petya', 'd
2 print('vasya  petya   dima'.split(' ', 3)) # ['vasya', '', '
3 print('vasya  petya   dima'.split()) # ['vasya', 'petya', 'd
4 print('vasya  petya   dima'.split(None)) # ['vasya', 'petya'
5 print('vasya  petya   dima'.split(None, 2)) # ['vasya', 'pet
```

split

- Еще интересный случай - пустая строка как разделитель
- join над пустой строкой просто склеивает аргументы
- split выглядит как функция, обратная join
- Можно ожидать, что split с пустой строкой как разделителем разобьет строку по символам
- Но это не так
- Будет брошено исключение

Разбивка на две части

- Возможный частный случай для split - разбивка на 2 части
- Например, файл состоит из строк вида '<имя>: <описание>'
- И мы хотим перебрать строки файла
- И получить список пар имен и описаний
- И отдельно обработать ситуацию отсутствия двоеточия
- Напечатав предупреждение

Разбивка на две части

- Можно использовать split
- Вторым параметром передать 1
- И проверять длину полученного списка
- Если длина 1, то двоеточие не нашлось
- Будет корректно, но общий смысл затуманится деталями

partition

- Есть метод partition
- Один параметр - строка-разделитель
- Всегда возвращает набор из трех элементов
- Если разделитель не найден, то сначала идет строка до разделителя
- Потом - сам разделитель
- И потом - строка после разделителя

partition

- А если не найдет, то сначала идет вся строка
- А потом - две пустые строки
- Зачем возвращать строку-разделитель ?
- Мы же ее и так знаем
- Ради второго случая
- Чтобы проверять - нашелся разделитель или нет

partition

- Всегда возвращается три элемента
- Это располагает к групповому присваиванию
- А пустая строка в качестве разделителя запрещена
- И None - тоже
- Поэтому если разделитель найден
- Второй элемент группового присваивания будет нулевой строкой
- Иначе - пустой

Пример на partition

```
1 line = input()
2 id, found, details = line.partition(':')
3 if found:
4     print('id', id)
5     print('details', details)
6 else:
7     print('no colon found')
```

Подсчет вхождений

- Если ходим посчитать вхождения символа
- Можем пройтись циклом с индексацией и сравнениями
- И посчитать
- Если вхождения подстроки, то индексацию можно заменить на отрезки
- Но есть готовый метод - count

Подсчет вхождений

- Основной параметр - что считаем
- Как ни странно - не запрещена пустая строка
- При пустой строке получаем len(s) + 1
- Если непустая - считаем количество вхождений

count

- Если нашли вхождение строки
- То следующее начинаем искать после его окончания
- Например: '-aaaaa-'.count('aa')
- По индексу 1 нашли вхождение 'aa'
- Индекс 2 пропускаем, потому что его покрывает уже найденное вхождение
- А вот подстроку с началом в индексе 3 - уже считаем
- И больше ничего найти не получится

count

- Есть еще два параметра
- Начальный и конечный индексы
- Их использование аналогично вызову count над отрезками
- С теми же значениями в качестве границ

Пример на count

```
1 s = 'aabbaaaabbb'
2 print(s.count('')) # 12
3 print(s.count('a')) # 6
4 print(s.count('b')) # 5
5 print(s.count('c')) # 0
6 print(s.count('aa')) # 3
7 print(s.count('bb')) # 2
8 print(s.count('ab')) # 2
```

Правосторонние аналоги

- rfind/rindex - ищем подстроку справа налево
- Возвращаемый индекс найденной подстроки - неотрицательный
- Индекс начала найденной подстроки
- Если не нашли - все как в базовой версии

Пример на rfind/rindex

```
1 s = 'aabbaaaabbb'
2 print(s.rfind('b')) # 10
3 print(s.rfind('bb')) # 9
4 print(s.rfind('ab')) # 7
5 print(s.rfind('ba')) # 3
6 print(s.rfind('bbba')) # -1
```

Правосторонние аналоги

- rpartition - ищем разделитель, начиная справа
- При успехе элементы тройки идут в том же порядке, что и у partition
- То, что слева; разделитель; то, что справа
- Если не нашли - сначала две пустые строки, потом исходная строка

Пример на rpartition

```
1 s = 'aabb,aaa,abbb'
2 print(s.rpartition(',')) # ('aabb,aaa', ',', 'abbb')
3 print(s.rpartition(' ')) # ('', '', 'aabb,aaa,abbb')
4 print(s.rpartition('ab')) # ('aabb,aaa,', 'ab', 'bb')
```

Правосторонние аналоги

- rsplit - все как в split
- Только при лимите на количество
- Применяем разбиение, начиная справа
- Но получившиеся куски идут в порядке слева направо
- То есть передав результат в join, получим исходную строку

Пример на rsplit

```
1 s = 'aabb,aaa,abbb'
2 print(s.rsplit(',')) # ['aabb' , 'aaa', 'abbb']
3 print(s.rsplit(',', 1)) # ['aabb,aaa', 'abbb']
```

Удаление префикса/суффикса

- Есть методы для проверки префикса или суффикса
- А иногда хочется сначала проверить
- И если он такой, как ожидалось
- То удалить его
- Есть removeprefix/removesuffix
- Начиная с 3.9

Пример на removeprefix/removesuffix

```
1 s = 'aabb,aaa,abbb'
2 print(s.removeprefix('a')) # 'abb,aaa,abbb'
3 print(s.removeprefix('aa')) # 'bb,aaa,abbb'
4 print(s.removeprefix('aab')) # 'b,aaa,abbb'
5 print(s.removeprefix('aaa')) # 'aabb,aaa,abbb'
6 print(s.removesuffix('b')) # 'aabb,aaa,abb'
7 print(s.removesuffix('abbb')) # 'aabb,aaa,'
8 print(s.removesuffix('aab')) # 'aabb,aaa,abbb'
```

Прочие

- capitalize - первая буква переводится в верхний регистр
- Если актуально
- Остальные не изменяются
- center(width, [, fillchar]) - центрирует до ширины width
- fillchar - заполнитель, по умолчанию пробел

Прочие

- expandtabs(tabsize=8) - заменяем табуляции пробелами
- Табуляция - специальный символ
- Иногда используется для визуального подравнивания текстов
- Если редактор его понимает
- Перенос курсора на ближайшую позицию, кратную 8

Прочие

- expandtabs преобразует строку
- По подобию такого редактора
- Если табуляция в позиции, кратной 8, то заменяет на 8 пробелов
- Если на одну позицию правее, то на 7
- И так далее

Прочие

- ljust/rjust - подравнивание строки на заданную ширину
- По левой/правой границе
- swapcase - заглавные в строчные и наоборот
- title - каждое слово с заглавной
- И еще несколько совсем экзотичных

Что осталось за скобками

- encode - оптимизированное представление строки
- Часто используется для сохранения, передачи и т.п.
- format - создание строки из разных компонент
- Немного устаревший
- Форматированная строка как альтернатива
- Регулярные выражения