

PYTHON

Лекция 3

ПЛАН ЛЕКЦИИ

- Индексы, вырезки
- Основные строковые методы
- Коллекции и итерирование

НА ЧЕМ ОСТАНОВИЛИСЬ

- У строк можно брать элемент по индексу
- Индексы бывают неотрицательными (от начала строки)
- Индексы бывают отрицательными (от конца, -1 - последний символ)
- Можно брать отрезки строки

ПРОДОЛЖАЕМ

- Отрезки (slice) имеют размер шага
- По умолчанию - 1
- Можно задать через дополнительное двоеточие
- Пример `'hello, world'[2:10:2] == 'lo o'`

ОТРИЦАТЕЛЬНЫЕ ВЫРЕЗКИ

- Можно указать первый параметр больше, чем второй
- Само по себе это даст пустую строку
- Но если указать отрицательный шаг, то будет проход справа налево
- Пример: `print('0123456789'[5:2])` # ''
- Пример: `print('0123456789'[5:2:-1])` # '543'

ОБЩАЯ ЛОГИКА ВЫРЕЗОК

- [start:finish:step]
- Проверяем step. Если 0, бросаем исключение
- Устанавливаем счетчик index в start
- Перед каждой итерацией проверяем условие

ОБЩАЯ ЛОГИКА ВЫРЕЗОК

- Если $step > 0$, то условие $index < finish$
- Если $step < 0$, то условие $index > finish$
- Если условие выполнено, то кладем значение по индексу $index$ в результат
- Если нет - завершаем
- $index += step$, и повторяем проверку

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

- Если нет `step`, то `step = 1`
- Если есть только одно двоеточие, то отсутствующим считается `step`
- Если `step` положительный, то по умолчанию `start = 0`, `finish = len(s)`
- Если `step` отрицательный, то по умолчанию `start = len(s) - 1`, `finish = -1`

ПРИМЕРЫ

```
1 s = '0123456789'
2 print(s[1:7])      # '123456'
3 print(s[1:7:])     # '123456'
4 print(s[1:7:1])    # '123456'
5 print(s[1:7:2])    # '135'
6 print(s[1:8:2])    # '1357'
7 print(s[1:9:2])    # '1357'
8 print(s[1::2])     # '13579'
9 print(s[1::])      # '123456789'
10 print(s[1:])       # '123456789'
11 print(s[:5])       # '01234'
12 print(s[:5:])      # '01234'
13 print(s[:8:3])     # '036'
```

ПРИМЕРЫ

```
1 s = '0123456789'
2 print(s[7:7])      # ''
3 print(s[7:7:])     # ''
4 print(s[7:7:1])    # ''
5 print(s[7:7:2])    # ''
6 print(s[8:1:2])    # ''
7 print(s[1:9:-2])   # ''
8 print(s[8:4:-1])   # '8765'
9 print(s[8:4:])     # ''
10 print(s[8:4])      # ''
11 print(s[5::-1])    # '543210'
12 print(s[5:0:-1])   # '54321'
13 print(s[8:4:-2])   # '86'
14 print(s[8:3:-2])   # '864'
```

ЕЩЕ НЕ ВСЕ

- Если `start` отрицателен, то он меняется на `len(s) + start`
- Если `finish` отрицателен - аналогично
- Если `step` положительный, а `start` все равно отрицательный, то `start = 0`
- Если `step` отрицательный, а `start >= len(s)`, то `start = len(s) - 1`
- С `finish` попроще - промахи по строке игнорируем

ВАЖНЫЕ СЛЕДСТВИЯ

- Есть `s = '0123456789'`
- И есть `s[2::2] == '2468'`
- Сместим начало: `s[3::2] = '3579'`
- Возьмем `s[-12::2] == '02468'`
- Сместим начало: `s[-11::2] = '02468'`

ОСОБО ИНТЕРЕСНЫЕ СЛУЧАИ

- Если строка пустая, то при любом slice-е выйдет пустая строка
- Если только шаг не 0
- Пусть `s = '0123456789'`
- `print(s[5::-1])` напечатает `'543210'`
- Какое число поставить между двоеточиями, чтобы получить тот же результат ?

МЕТОДЫ STR

- Несколько десятков, очень разношерстных
- Много legacy и экзотики
- Разберем важные

STRIP/LSTRIP/RSTRIP

- Отрезает пробелы, переводы строк и т.п.
- С обоих концов/с левого/с правого
- Один из часто используемых при чтении из файла
- Питоновские стандартные методы читают строки вместе с переводом строки
- И его хочется обрезать

НЕУДАЧНЫЙ КОД

```
1 # cat-like utility
2 # python3 cat.py filename
3
4 import sys
5
6 with open(sys.argv[1], 'r') as fin:
7     for line in fin:
8         print(line)
```


ТАК ПОЛУЧШЕ

```
1 # cat-like utility
2 # python3 cat.py filename
3
4 import sys
5
6 with open(sys.argv[1], 'r') as fin:
7     for line in fin:
8         print(line.rstrip())
9 # потеряем висящие пробелы заодно
```

ИЛИ ТАК

```
1 # cat-like utility
2 # python3 cat.py filename
3
4 import sys
5
6 with open(sys.argv[1], 'r') as fin:
7     for line in map(str.rstrip, fin):
8         print(line)    # потеряем висящие пробелы заодно
```

МОЖНО РАССМОТРЕТЬ

```
1 # cat-like utility
2 # python3 cat.py filename
3
4 import sys
5
6 with open(sys.argv[1], 'r') as fin:
7     for line in fin:
8         print(line.rstrip('\n'))
9 # надо присмотреться к Windows, Mac
10 # и кроссплатформенным вариантам
```

ПРЕФИКСЫ, СУФФИКСЫ

- Часто проверяем, начинается ли строка с чего-то
- Можно через вырезку, но громоздко
- И надо высчитывать длину
- Лучше так: `s.startswith('>>>')`

ПРЕФИКСЫ, СУФФИКСЫ

- `endswith` - аналогично
- Можно указать кортеж префиксов - "хоть с чего-то"
- Кортеж принципиален, список не пойдет

ПРИМЕР

```
1 s = 'banana'
2 print(s.startswith('b')) # True
3 print(s.startswith('ba')) # True
4 print(s.startswith('banana')) # True
5 print(s.startswith('band')) # False
6 print(s.startswith(('band', 'bana'))) # True
7 print('').startswith(('band', 'bana')) # False
8 print(s.startswith(())) # False
9 print('').startswith(())) # False
10 print('').startswith('') # True
```

ПРЕФИКСЫ, СУФФИКСЫ

- Есть второй параметр start (необязательный)
- И еще один обязательный параметр end
- end без start указать нельзя
- "With optional start, test string beginning at that position. With optional end, stop comparing string at that position."

ПРЕФИКСЫ, СУФФИКСЫ

- Документация не до конца ясна
- Эмпирически похоже на то, что

```
s1.startswith(s2, start)
```

равносильно

```
s1[start:].startswith(s2)
```


ПРЕФИКСЫ, СУФФИКСЫ

- А

```
s1.startswith(s2, start, end)
```

равносильно

```
s1[start:end].startswith(s2)
```

- И аналогично для endswith

ПРЕФИКСЫ, СУФФИКСЫ

- Но есть нюансы
- Например

```
'hello'.startswith('', 10)
```

вернет

False

ПРЕФИКСЫ, СУФФИКСЫ

- Хотя

```
'hello'[10:].startswith('')
```

вернет

True

- Предположительно, там для экзотичных ситуаций стоит отдельная проверка и всегда возвращается False

COUNT

- Считаем количество непересекающихся вхождений первого параметра

- `'abracadabra'.count('a')`

вернет 5

- `'aaaaa'.count('aa')`

вернет 2

COUNT

- А что вернет

```
'12345'.count('')
```

? (Исключение тут не бросится)

- Сделайте ваше предположение

COUNT

- И тоже есть start и end
- При непустом первом в экзотичных ситуациях предсказуемо возвращается 0
- Но есть нюанс с пустой строкой
- `''.count('')`

вернет 1

COUNT

- `'hello'[2:2].count()`

вернет 1

- `'hello'[3:2].count()`

вернет 1

COUNT

- `'hello'.count(' ', 2, 2)`

вернет 1

- `'hello'.count(' ', 3, 2)`

вернет 0 !!!

АНТИПАТТЕРН С COUNT

- Задача типа "посчитать согласные буквы в слове"
- Есть соблазн для каждой согласной вызвать count и сложить результаты
- Но будет много проходов по строке

FIND

- `find` ищет подстроку и возвращает индекс
- Или -1, если не найден
- Тоже есть `start` и `end`
- "Return the lowest index in the string where substring `sub` is found within the slice `s[start:end]`"

ПРИМЕР

```
1 s='0123456789'
2
3 print(s.find('0')) # 0
4 print(s.find('')) # 0
5 print(s.find('345')) # 3
6 print(s.find('0', 1)) # -1
7 print(s.find('567', 2)) # 5
8 print(s.find('567', 2, 7)) # -1
9
10 print('').find('')
11 # 0 - хотя 0 здесь не "lowest index in the string"
12 print('').find('0') # 0
13 print('').find('0', 1) # -1
```

БЛИЗКИЕ К FIND

- `rfind` ищет правое вхождение
- `index` - как `find`, только бросает исключение там, где `index` возвращает -1
- `rindex` - как `index`, только справа
- `Или` - как `rfind`, только бросает исключение, если не найдено

КАК ВЫБРАТЬ

- find/rfind - если хочется сразу проверить успешность и отреагировать
- И/или если вариант "не нашли" - вариант нормы
- index/rindex - если в норме искомая подстрока обязана присутствовать
- И нам это даже проверять не хочется
- Но не хотим, чтобы в случае некорректных данных это было замечено

SPLIT

- Разбивка строки по стандартным разделителям
- Разбор простых форматов данных
- Базовая форма - `line.split()`
- Разделяет строку по пробельным последовательностям

SPLIT

- В пробельные символы входят табуляция, перевод строки и т.п.
- Он отбросит пробельные последовательности справа и слева
- Над пустой строкой вернет пустой список

ПРИМЕР

```
1 print('').split() # []
2 print(' '.split() # []
3 print(' \t \n'.split()) # []
4 print('hello'.split() # ['hello']
5 print('hello world'.split()) # ['hello', 'world']
6 print('hello world\n'.split()) # ['hello', 'world']
7 print('    hello    world\n'.split()) # ['hello', 'world']
```


SPLIT

- `split` можно вызвать с одним параметром
- Это просто текстовый разделитель
- На пустую строку как разделитель бросается исключение
- Если хотим превратить строку в список символов - `list(s)`

SPLIT

- Непустая строка - чаще всего односимвольная
- Но логика одинакова для любых непустых
- Двигается слева направо и ищет вхождения разделителя
- То, что оказалось слева - идет в результат

SPLIT

- Позиции "под" разделителем отбрасываются
- Если дошли до конца, то все слева - тоже идет в результат
- Если слева оказывается пустая строка, она тоже идет в результат
- Всегда, с краев - тоже

SPLIT

- Для жестко фиксированных форматов
- Могут быть контр-интуитивные результаты, особенно для пробелов
- `''.split(' ') # ['']`
- `'ab cd'.split(' ') # ['ab', '', 'cd']`
- `'ab::cd'.split(':') # ['ab', '', 'cd']`

SPLIT

- Иногда нужен неполный split
- Сначала идут строго форматированные столбцы
- А после последнего разделителя - свободный текст
- Типичные примеры: grep, log-файлы

SPLIT

- Вторым параметром можно указать лимит на столбцы
- По умолчанию -1 - отсутствие лимита
- 0 - строку оборачиваем в массив (странный вариант)
- 1 - делим по первому разделителю
- 2 - делим по первому двум

ПРИМЕР

```
1 print('step 1: add colon (":") to the end of string'
2     .split(':', 1))
3 # ['step 1', ' add colon (":") to the end of string']
4 print('step 1: add colon (":") to the end of string'
5     .split(':'))
6 # ['step 1', ' add colon ("', ':") to the end of string']
7 print('12:30:23|ApplicationRunner|DEBUG: value of v: |'
8     ' qwerty|'.split('|', 3))
9 # ['12:30:23', 'ApplicationRunner', 'DEBUG',
10 #  ' value of v: ' | qwerty|]
```

ПРИМЕР

```
1 print('12:30:23|ApplicationRunner|DEBUG: value of v: |'  
2       ' qwerty|'.split('|', 4))  
3 # ['12:30:23', 'ApplicationRunner', 'DEBUG',  
4   ' value of v', ' qwerty|']  
5 print('12:30:23|ApplicationRunner|DEBUG: value of v: |'  
6       ' qwerty|'.split('|', 10))  
7 # ['12:30:23', 'ApplicationRunner', 'DEBUG',  
8   ' value of v', ' qwerty', '']
```


SPLIT

- Еще есть `rstrip` - откусить кусочки с конца
- В целом `rstrip` удобен для тепличных данных
- Или с явными гарантиями на формат
- Лучше пользоваться проверенными инструментами (`pandas` для `csv`)
- Можно регулярками - они мощнее

JOIN

- "Обратная" к split
- Оригинальный подход: метод вызывается над разделителем
- `':'.join(["vasya", '23.03.1996'])`
- Годится любая последовательность
- Элементы обязаны быть строками

JOIN

- Не только для формирования того, что прочитает `split`
- Частое использование - формирование динамической строки
- `".join(many_small_pieces)`
- Избегаем квадратичной сложности

