PYTHON

Лекция 6

ПЛАН ЛЕКЦИИ

- Строки: завершение
- Списки

SPLIT

- Иногда нужен неполный split
- Сначала идут строго форматированные столбцы
- А после последнего разделителя - свободный текст
- Типичный пример: 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',
 4 #  ' add colon (":") to the end of string',
 5 # ]
 6
 7 print('step 1: add colon (":") to the end of string'
 8       .split(':'))
 9 # ['step 1',
10 #   ' add colon ("',
11 #   '") to the end of string'
12 # ]
```

ПРИМЕР

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

SPLIT

- Еще есть rstrip - откусить кусочки с конца
- В целом strip 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
- Альтернатива - io.StringIO

СПИСКИ

- Последовательность ссылок на объекты
- Нумерованная
- Материализованная
- Изменяемая

СОЗДАНИЕ

- Литерал - элементы через запятую
- В квадратных скобках
- Через операции: вырезки, сложение, умножение на число
- Функция list

ПРИМЕР

```
1 data_1 = []
2 data_2 = [123]
3 data_3 = ['vasya', 'dima', 'kolya']
4 data_4 = data_2 + data_3
5 print(data_4)
6 data_4 = data_4 * 3
7 print(data_4)
8 data_5 = [data_1, data_2]
9 print(data_5)
```

МНОГОМЕРНОСТЬ

- Явочным порядком
- Можно создать список списков
- Точнее - список ссылок на списки
- Встроенного механизма в языке нет
- Только в библиотеках (numpy)

ФУНКЦИЯ LIST

```
1 print(list('hello'))
2 print(list(enumerate('hello')))
3 print(enumerate('hello')) # для сравнения
4 print(list(range(10)))
5 # print(list(range(1000000000000)))  - так не надо
```

ОПАСНОСТЬ

```
1 # Наивная инициализация двумерного списка 5 x 6
2 data = [[None] * 6] * 5
3 print(data)
4 data[0][1] = 10
5 print(data)  # !!!!!
```

ОПАСНОСТЬ

```
1 # Правильная, но громоздкая инициализация
2 # двумерного списка 5 x 6
3 data = [None] * 5
4 for i, _ in enumerate(data):
5     data[i] = [None] * 6
```

ВАЖНЫЕ МЕТОДЫ

- append - добавляет элемент в список
- extend - расширяет список другим списком
- Эффективные способы расширять список
- += - тоже (для списка)

ВСПОМНИМ СТРОКИ

- += для строк - не "extend"
- Возможны оптимизации
- Я бы не полагался

POP

- Без параметров - убирает элемент из конца. Недорого
- Комбинация append(v)/pop() - дает стек
- С параметром - удаляет элемент из середины
- В среднем - дорого

INSERT/REMOVE/CLEAR

- insert - вставить элемент в позицию. Дорого
- remove - удалить первый с данным значением. Дорого
- Если не нашли - исключение
- clear - очистка текущего списка

ОПАСНОСТЬ

```
 1 print('123'.index('23'))  # 1
 2 #print(list('123').index(list('23'))) # исключение
 3 #print(list('123').index('23')) # исключение
 4 #print(list('123').index(list('2'))) # исключение
 5 print(list('123').index('2')) # 1
 6
 7 print('123'.index('', 2))  # 2
 8 print('123'.index('', 3))  # 3
 9 #print('123'.index('', 4))  # исключение
10
11 #print(list('123').index([], 2))  # исключение
12 #print(list('123').index([], 3))  # исключение
13 #print(list('123').index([], 4))  # исключение
```

SORT

- Сортирует in-place
- Важно, чтобы сравнения были определены
- Можно задать функцию-ключ
- Например, str.lower - если сортируем строчки без учета регистра

ОПАСНОСТЬ

```python
import random

data = ['123', 'qwerty', 'Hello', 'abc', 'ABC']
data.sort()
print(data)
# ['123', 'ABC', 'Hello', 'abc', 'qwerty']
data.sort(key=str.lower)
print(data)
# ['123', 'ABC', 'abc', 'Hello', 'qwerty']

# ............
```

ОПАСНОСТЬ

```
1 # ..............
2
3 data.sort(key=lambda v: random.random())
4 print(data)
5 # плохой ключ - непредсказуемый порядок
6 data.sort(key=lambda v: random.random())
7 print(data)
8 # для перемешивания есть random.shuffle - он эффективнее
```

ПОДЕТАЛЬНЕЕ

ЧТО ЛУЧШЕ

- Ключ обычно делегируется к логике сравнения базового тпа
- Компаратор требует доказательства полного порядка
- И в нем легко сажаются баги
- И в простых случаях ключ выглядит предпочтительнее

НО НЕ ВСЕ ТАК ПРОСТО

- Не всегда можно получить ключ в виде простого типа или строки
- Пример: рациональные числа
- Или это может быть неудобно/ контринтуитивно
- Пример: обратный алфавитный порядок не в первом поле составного ключа
- Или модификации алфавитного порядка

НО НЕ ВСЕ ТАК ПРОСТО

- Ключ может быть дорогим сравнительно с компаратором
- Пример: длинные строковые вырезки vs прямое сравнение
- А еще компаратор помогает наблюдать за сортировкой

НО НЕ ВСЕ ТАК ПРОСТО

- Но есть и минус: вызывается O(nlog(n)) раз
- С другой стороны: ключ для экономии вызовов потребует O(n) памяти
- И будет вызван O(n) раз

КАК В PYTHON

- Python2 использовал компараторы
- Python3 перешел на ключи
- Из ключа легко сделать компаратор
- Из компаратора ключ - сходу неочевидно

КАК В PYTHON

- Но Python это умеет
- functools.cmp_to_key
- Переносит логику сравнения в ключ

ПРИМЕРНЫЕ ИНСТРУКЦИИ

- Если есть понятный ключ и издержки не жмут - через ключ как есть
- Если ключ не придумывается - через компаратор и cmp_to_key
- (Понимая, компаратор все равно вызовется на каждом сравнении)
- Если издержки на ключ дают большую константу по памяти - пробуем через компаратор и cmp_to_key

ПРОЧЕЕ

- copy() - неглубокое копирование
- reverse() - переворачивание in-place
- count(x) - поэлементный подсчет, без start, end

СПИСОЧНЫЕ ВЫРАЖЕНИЯ

- "List comprehension"
- Короткая запись преобразования коллекции
- Базовая идея: многие задачи можно выразить в виде цепочки типовых преобразований
- Базис: фильтрация, отображение, свертка

ФИЛЬТРАЦИЯ

- Пройдя по коллекции, отбросим ненужное
- Ненужность определим по значению функции вызванной над элементом
- Например, по списку пользователей отберем тех, кто зарегестрирован в этом году

ОТОБРАЖЕНИЕ

- Для каждого элемента определим новое значение
- Логика определения нового значения задается функцией-параметром
- Пройдем по коллекции, породив новую коллекцию, преобразовав каждый элемент
- Например по списку пользователей получим пару (id, количество постов)

СВЕРТКА

- Возьмем начальное значение
- Поместим в условную переменную-аккумулятор
- Будем перебирать элементы коллекции, используя функцию с двумя параметрами
- На каждом элементе передадим в функцию аккумулятор и очередной элемент

СВЕРТКА

- Возвращенное функцией значение сделаем новым аккумулятором
- Результат свертки - финальное значение аккумулятора
- Примеры: count, sum, max, min
- Более сложные: группировка, zip

СПИСОЧНОЕ ВЫРЕЖЕНИЕ

- Списочное выражение позволяет компактно записать отображение с фильтрацией
- Описывает проход по коллекции
- И для каждого элемента позволяет задать отображение и фильтр

ПРИМЕР

```
print([len(v) for v  in input().split()])
print([v[::-1] for v  in input().split() if len(v) > 5])
print(
  [v for v  in input().split() if v and v[0] in 'aeouiy']
)
```

ИДИОМА

- Хороший стиль - писать цепочки преобразований
- Плохой стиль - выписывать цикл, в котором делается append под if
- Особенно если не делается больше ничего

ДВУМЕРНЫЙ МАССИВ

```
1 data = [[None] * 5 for _ in range(6)]
2 # Инициализация здорового человека
```