ЯЗЫК PYTHON

Лекция 4. Списки, строки

Списки

- Часто нужно обрабатывать не какой-то один показатель
- А несколько, но как одно целое
- Например, оценки студентов на экзамене
- Чтобы посчитать средний балл

Списки

- Пусть в одной группе 25 студентов, а в другой 19
- Мы не хотим заводить сначала 25 переменных, а потом 19
- И считать среднее как-то так
  - (stidnet_1 + student_2 + ... + student_25) / 25
- Хочется как-то по-другому

Списки

- Есть специальный тип данных - список
- Набор элементов
- Любого типа
- И его можно задать в программе как литерал

Пример со списком

```
1 data = [12, 43, 11, 0, 34]
2 empty = []
3 print(data)
4 print(empty)
```

Заполним список

```
numbers = []
while True:
    s = input()
    if not s:
        break
    numbers.append(int(s))
    print(numbers)
```

Операции со списком

- Определить длину списка - функция len
- Часто хочется получить элемент списка по номеру
  - После списка - номер в квадратных скобках
  - Номер элемента называем индексом
  - Любые индексы в Python начинаются с нуля

Пример

```
1 numbers = [123, 234, -321, 456]
2 print(len(numbers))  # 4
3 print(len([]))  # 0
4 print(numbers[0])  # 123
5 print(numbers[1])  # 234
6 print(numbers[2])  # -321
7 print(numbers[3])  # 456
```

Индексирование с конца

- Иногда хочется взять последний элемент
- Или предпоследний
- То есть по номеру от конца
- Можно это выразить так
  - data[len(data) - 1] # последний
  - data[len(data) - 2] # предпоследний

Индексирование с конца

- Но это немножко громоздко
- Есть более компактная форма
- Можно убрать вызов len()
- И использовать отрицательный индекс
  - data[-1] # последний
  - data[-2] # предпоследний

Пример

```
1 numbers = [123, 234, -321, 456]
2 print(numbers[-1])  # 456
3 print(numbers[-2])  # -321
4 print(numbers[-3])  # 234
5 print(numbers[-4])  # 123
```

Оборотная сторона

- Это удобно
- Но немного коварно
- Допустим, мы хотим напечатать разницу двух соседних элементов
- Пройдем циклом по списку

Для начала - просто напечатаем все элементы

```
1 numbers = [123, 234, -321, 456]
2 i = 0
3 while i < len(numbers):
4     print(numbers[i])
5     i += 1
```

А теперь - разницу между соседними

```
1 numbers = [123, 234, -321, 456]
2 i = 0
3 while i < len(numbers):
4     print(numbers[i] - numbers[i - 1])
5     i += 1
```

Видно проблему ?

Тихая проблема

- Проблема не только в том, что логика некорретна
- А в том, что программа не сломалась
- И у нас меньше шансов заметить проблему
- Это плата за удобство отрицательного индекса

Отрезки

- Иногда хочется взять кусочек списка
- Например, первые три элемента
- Или третий и второй с конца - как список из двух элементов
- Для этого есть своя конструкция
- Очень похожая на индекс

Отрезки

- Конструкция называется slice
- Тоже оформляется квадратными скобками после списка
- Только там два элемента
- Разделенные двоеточием
- Или меньше, чем два - но все равно с двоеточием

Отрезки

Зачем такая ассиметрия

- В ней есть немало удобств
- Например, легко определить длину отрезка
- Просто вычесть из правого число левое
- И легко разделить список части
- Например, так: [0:k] и [k:len(data)]

Сокращенные отрезки

- Но вот так писать: [k:len(data)]
- Совсем необязательно
- Как и вот так: [0:k]
- Если отрезок начинается в начале списка
- Или заканчивается в конце
- Можно там ничего не писать
- Но двоеточие нужно оставить

Пример с отрезками

```
1 data = [12, 43, 11, 0, 34, 45, 23]
2 print(data[1:4]) # [43, 11, 0]
3 print(data[:4]) # [12, 43, 11, 0]
4 print(data[1:]) # [43, 11, 0, 34, 45, 23]
5 print(data[:]) # копия исходного списка
```

Отрезки с шагом

- Иногда хочется взять все четные
- Или с шагом 5
- Для этого используется второе двоеточие
- Если его нет, то шаг - 1

Отрезки с шагом

- Пусть есть отрезок data[a:b:c]
- В результат попадает data[a], data[a + c], data[a + 2 * c]
- И так далее
- Пока индекс строго меньше, чем b

Пример с шагами

```
1 data = [12, 43, 11, 0, 34, 45, 23]
2 print(data[1:4:2]) # [43, 0]
3 print(data[1:5:2]) # [43, 0]
4 print(data[::2]) # [12, 11, 34, 23] - четные позиция
5 print(data[1::2]) # [43, 0, 45] - нечетные позиции
```

Отрицательные значения в параметрах отрезка

- Отрицательные начало и/или конец - как в индексе
- Отсчет от конца
- Одна граница может быть положительной, другая - отрицательной
- Типичный пример: data[1:-1] # кроме начального и конечного

Отрицательные значения в параметрах отрезка

- Отрицательный шаг - движение в обратную сторону
- Отсчет от конца
- С границами - все так же
- Левая входит, правая не входит
- Особый случай - шаг 0
- Это ошибка
- Даже на пустом отрезке

Пример с шагами

```
1 data = [12, 43, 11, 0, 34, 45, 23]
2 print(data[1:-1:2]) # [43, 0, 45]
3 print(data[-1:1:-2]) # [23, 34, 11] - это не просто перевер
4 print(data[2:2:1]) # []
5 print(data[2:2:-1]) # []
6 print(data[2:2:0]) # ошибка
```

Знак шага и границы отрезка

- При положительном шаге (включая ситуацию по умолчанию)
- Если левая граница больше или равна правой
- То результат всегда будет пустым
- А иначе - хочется сказать, что будет точно непустым
- Но это не всегда так
- Нужно еще, чтобы левая граница попала в пределы списка

Знак шага и границы отрезка

- При отрицательном шаге
- Если левая граница меньше или равна правой
- То результат всегда будет пустым
- А иначе - все как и для положительного шага

Еще пример

```
1 data = [12, 43, 11, 0, 34, 45, 23]
2 print(data[1:1]) #  []
3 print(data[2:2:1]) # []
4 print(data[2:2:-1]) # []
5 print(data[2:3:2]) # [11]
6 print(data[2:3:-1]) # []
7 print(data[3:2:2]) # []
8 print(data[3:2:-1]) # [0]
```

Отдельный интересный случай

- Отрезок с двумя двоеточиями
- С отрицательным шагом
- И отсутствующим началом и концом
- data[::-1] или data[::-5]
- data[::-1] - это перевернутый список
- data[::-5] - начинаем с последнего и идем с шагом 5

Пример на переворачивание

```
1 data = [12, 43, 11, 0, 34, 45, 23]
2 print(data[::-1]) #  [23, 45, 34, 0, 11, 43, 12]
3 print(data[::-2]) #  [23, 34, 11, 12]
4 print(data[::-5]) #  [23, 43]
```

Изменение содержимого списка

- Индексированное значение можно поставить слева от присваивания
- Тогда значение соответствующего элемента изменится
- Важно, чтобы это был индекс существующего элемента
- Иначе - ошибка

Пример с присваиванием

```
1 data = [12, 43, 11, 0, 34, 45, 23]
2 data[3] = 123
3 print(data[2:4]) #  [11, 123]
4 data[-1] = 0
5 print(data[-3:]) #  [34, 45, 0]
```

Тонкий момент

- Список - объект в памяти
- Переменная (data в примерах) хранит адрес этого объекта
- Присваивание копирует адрес, но не создает нового объекта
- То есть две переменных могут "смотреть" на один объект
- В частности, на список
- И изменения, сделанные "через" одну переменную будут видны "через" другую

Пример на тонкий момент

```
1 data = [12, 43, 11]
2 print(data) #  [12, 43, 11]
3 data2 = data
4 print(data2) #  [12, 43, 11]
5 data[1] = 0
6 print(data) #  [12, 0, 11]
7 print(data2) #  [12, 0, 11]
8 print(data is data2) # True
9 print(data == data2) # True
```

Решение и еще один тонкий момент

- Можно применить отрезок без обоих параметров
- Вот так: data[:]
- Но есть еще один момент
- Элементами списка могут быть списки
- Иногда это полезно
- Но при использовании отрезка скопируется только список первого уровня
- А его элементы так и останутся адресами одних и тех же объектов

Пример исправления тонкого момента

```
 1 data = [12, 43, 11]
 2 print(data) #  [12, 43, 11]
 3 data2 = data[:]
 4 print(data == data2) # True
 5 print(data2) #  [12, 43, 11]
 6 data[1] = 0
 7 print(data) #  [12, 0, 11]
 8 print(data2) #  [12, 43, 11]
 9 print(data is data2) # False
10 print(data == data2) # False
```

Пример на еще один тонкий момент

```
 1 data = [[12, 43], [11, 22]]
 2 print(data) # [[12, 43], [11, 22]]
 3 data2 = data[:]
 4 print(data2) # [[12, 43], [11, 22]]
 5 data[1] = [33, 44]
 6 print(data) #  [[12, 43], [33, 44]]
 7 print(data2) # [[12, 43], [11, 22]]
 8 data[0][0] = 55
 9 print(data) #  [[55, 43], [33, 44]]
10 print(data2) # [[55, 43], [11, 22]]
11
12
13 print(data is data2) # False
14 print(data[0] is data2[0]) # True
```

Присваивать список можно и отрезкам

- Возможны два случая
- Первый: длины отрезков слева и справа от знака присваивания совпадают
- Тогда происходит поэлементное присваивание
- И отрезок может содержать шаг
- Например, можно заменить элементы с пятого по восьмой
- Или все нечетные

Присваивать список можно и отрезкам

- Возможны два случая
- Второй: длины отрезков слева и справа от знака присваивания НЕ совпадают
- Тогда происходит замена отрезка другим
- И список НЕ может содержать шаг
- Если справа пустой список - получаем удаление отрезка
- Если слева пустой отрезок - получаем вставку элементов
- Важно: всегда остается тот же объект-список

Необычные варианты

- При индексации все строго
- Любой индекс пустого списка - это ошибка
- У списка длины 1 корректный индекс - 0 или -1, остальное - ошибка
- У списка длины 2 корректные индексы - только 0, 1, -1, -2, остальное - ошибка
- И так далее

Необычные варианты

- Отрезки гораздо мягче
- Если начальный индекс оказался правее самого правого элемента
- А шаг - положительный
- То получим пустой отрезок

Пример на присваивание отрезкам

```
1 data = [12, 43, 11, 45, 34]
2 print(data) # [12, 43, 11, 45, 34]
3 data[1:3] = [5, 6]
4 print(data) # [12, 5, 6, 45, 34]
5 data[::2] = [7, 8, 9]
6 print(data) # [7, 5, 8, 45, 9]
7 data[3::-2] = [10, 11]
8 print(data) # [7, 11, 8, 10, 9]
```

Разные длины слева и справа

```
 1 data = [12, 43, 11, 45, 34]
 2 print(data) # [12, 43, 11, 45, 34]
 3 data[1:3] = [5]
 4 print(data) # [12, 5, 45, 34]
 5 data[1:3] = [7, 8, 9]
 6 print(data) # [12, 7, 8, 9, 34]
 7 data[1:3] = []
 8 print(data) # [12, 9, 34]
 9 data[1:1] = [1, 2, 3]
10 print(data) # [12, 1, 2, 3, 9, 34]
```

Сложение и умножение

- Два списка можно сложить
- Получится новый список
- Сначала элементы первого
- Потом - второго
- Можно умножить на число
- Содержимое старого списка повторится ровно столько раз

Пример на сложение и умножение

```
1 data = [12, 43]
2 print(data + [1, 2]) # [12, 43, 1, 2]
3 print(data * 0) # []
4 print(data * 1) # [12, 43]
5 print(data * 2) # [12, 43, 12, 43]
6 print(3 * (data + [1, 2])[1:3]) # [43, 1, 43, 1, 43, 1]
```

Строки

- Строковая константа - в кавычках
- В одинарных или двойных
- Лучше не смешивать без надобности
- Если хочется сделать одинарную кавычку частью строковой константы
- Проще всего завернуть константу в двойные кавычки
- А если хочется такое сделать с двойной - завернуть в одинарные

Строки

- А если и те, и другие
- Можно использовать \
- Поставить ее перед кавычкой
- И она потеряет смысл завершения строки
- А если \ надо сделать частью строки - пишем его два раза: \\

Пример на строковые константы

```
1 print('hello') # hello
2 print("hello") # hello
3 print('Роман "Война и мир"') # Роман "Война и мир"
4 print("Роман 'Война и мир'") # Роман 'Война и мир'
5 print("Кавычки бывают одинарные(') и двойные(\")") # Кавычки
6 print("Символ \\ обрабатывается по-особому") # Символ \ обра
```

Длинные строки

- Строковая константа должна умещаться на экранную строку
- А если не умещается
-
- Поставить \ перед переводом строки
- И она потеряет смысл завершения строки
- Или использовать две строковые константы подряд (именно константы)
- Или использовать три кавычки (двойных или одинарных)

Пример на длинные строковые константы

```
 1 s = 'hello, \
 2 world'
 3 print(s) # hello, world
 4 s = 'hello, ' \
 5     'world'
 6 print(s) # hello, world
 7 s = '''Длинный текст
 8 Ну очень длинный
 9 '''
10 print(s)
```

Списки и строки

- Общее: и то, и другое - нумеруемые наборы элементов
- Все сказанное про индексы и отрезки для списков - верно и для строк
- В Python нет понятия символа на отдельного типа данных
- Поэтому при взятии по индексу получает строку (одноэлементную)
- Отличие: строки неизменяемы
- Присваивание индексу и отрезкам - не работает для строк