eebb

ЯЗЫК PYTHON

Лекция 11. Генераторы. Comprehensions

List comprehension

- Иногда называют "списочные выражения"
- Мотивация: часто есть некоторая регулярность в заполнении списка
- Частный случай регулярности - range
- Но не все можно свести к range

Примеры регулярности

- Список квадратов чисел
- Список значений функции от 0, 1, 2, ...
- Или от элементов другого списка
- Список разностей между соседними значениями списка
- Оставляем в списке значения, удовлетворяющие критерию, выбрасываем остальные

Пример на квадраты чисел

```
1 squares = []
2 for i in range(100):
3     squares.append(i * i)
4 print(squares)
```

Пример на список значений

```
1 import math
2
3 values = []
4 for i in range(100):
5     values.append(math.sin(i))
6 print(values)
```

Пример на функцию от значений

```
1 import math
2 source = [1, 10, 2, 2.9, 1.2]
3 values = []
4 for v in source:
5     values.append(math.sin(v))
6 print(values)
```

Пример на разности

```
1 source = [1, 10, 2, 2.9, 1.2]
2 deltas = []
3 for v1, v2 in zip(source, source[1:]):
4     deltas.append(v2 - v1)
5 print(deltas)
```

Пример на фильтрацию

```
1 import math
2 source = list(range(100))
3 values = []
4 for v in source:
5     if math.sin(v) <= 0:
6         values.append(v)
7 print(values)
```

Что нас не устраивает

- Все примеры работают
- Хочется выражать мысль покомпактнее
- Чтобы видеть логику крупным планом
- И мыслить в терминах преобразований коллекций

Списочное выражение

- Снаружи квадратные скобки
- Внутри цикл, только не с начала
- А в начале - тело цикла
- Тело цикла - выражение
- Его значение на каждой итерации добавляем в результат
- Без явного вызова append

Пример на квадраты чисел

```
1 squares = [i * i for i in range(100)]
2 print(squares)
```

Пример на список значений

```
1 import math
2 values = [math.sin(i) for i in range(100)]
3 print(values)
```

Пример на функцию от значений

```
1 import math
2 source = [1, 10, 2, 2.9, 1.2]
3 values = [math.sin(v) for v in source]
4 print(values)
```

Пример на разности

```
1 source = [1, 10, 2, 2.9, 1.2]
2 deltas = [v2 - v1 for v1, v2 in zip(source, source[1:])]
3 print(deltas)
```

Осталась фильтрация

- Того, что есть, не хватает для фильтрации
- Мы подразумеваем добавление на каждой итерации
- Но есть синтаксис и под это
- if можно указать после for

Пример на фильтрацию

```
1 import math
2 source = list(range(100))
3 values = [v for v in source if math.sin(v) <= 0]
4 print(values)
```

Важная особенность

- В обычном цикле for переменная цикла создается в текущем контексте
- И остается после цикла
- А списочное выражение создает свой контекст
- И все определенное в нем уничтожается по завершении
- Что создает более чистую и предсказуемую среду выполнения

Пример на корректное создание двумерного списка

```
1 list_2d = [[None] * 50  for _ in range(100)]
2 print(list_2d)
3
4 # Некорректный вариант
5 list_2d = [[None] * 50] * 100
```

Аналогичные конструкции

- Есть конструкция для множеств
- Все так же, только с фигурными скобками
- Порождает множество
- Есть конструкция для словарей
- Перед for - пара, разделенная двоеточием

Пример на множество

```
1 text = 'qndcqwcqnkjnqjbcjoiocqoqhcbjwhbhjcqcqjhcqbcqjcbqjhcb
2 vowels = {c for c in text if c in 'aeiouy'}
```

Пример на словарь

```
1 text = 'wdqdf ifwdfdw   ffqw fwfwq'
2 starts = {index: c2 for index, (c1, c2) in enumerate(zip(tex
3 print(starts)
```

Генераторы

- Есть еще кортежы
- И хочется перенести идею на них
- Чтобы были круглые скобки и порождались кортежи
- Конструкция с круглыми скобками есть, но порождает она НЕ кортежи

Мотивирующий пример

- Нужно найти первые 100 чисел Фибоначчи, сумма цифр которых делится на 5
- Или более жизненный вариант
- Есть файл размером в 500Gb
- Там построчно хранятся записи с полями, разделенными табуляциями
- Найти первые 100 записей, в которых пятое поле - число, не превосходящее 10000

В чем проблема

- И ту, и другую задачу можно решить циклом
- Но в них есть что-то обшее
- И там, и там есть коллекция значений
- И мы хотим ее профильтровать
- И потом взять 100 значений

В чем проблема

- Мы можем написать функцию
- Которая принимает коллекцию и условие
- Условие - виде функции
- И возвращает отфильтрованную коллекцию

В чем проблема

- Мы можем написать другую функцию
- Которая принимает коллекцию и количество
- И возвращает первые n элементов
- Но мы не можем им передать все числа Фибоначчи
- Или 500 Gb данных

В чем проблема

- А если бы данных было 10Gb
- А у нас 16Gb памяти
- Все влезет, но очень может быть, что не пригодится
- Хотим декомпозировать вычисления
- И не терять в эффективности работы

В чем решение

- Реализуем механизм ленивых коллекций
- Чтобы можно было перебирать их по запросу
- И делать над ними преобразования
- Например, применять функцию к каждому элементу
- Или фильтровать
- Но делать это тоже "лениво"

Ленивый генератор

- Синтаксически выглядит как функция
- Одно отличие - наличие ключевого слова yield
- Обычно в том месте, где бывает return
- И при нем может быть какое-то значение
- Наличие такого слова меняет смысл вызова

Ленивый генератор

- Вызов такой функции не приводит к выполнению кода
- Пока еще
- Вместо этого создается специальный объект
- У которого есть метод __next__

Ленивый генератор

- Первый вызов __next__ приводит к началу исполнения кода функции
- До первого yield
- Когда встретили yield, исполнение приостанавливается
- Но контекст исполнения не уничтожается
- Значение при yield возвращается как результат __next__

Ленивый генератор

- Потом можно снова вызвать __next__
- Тогда переходим в приостановленный контекст
- И продолжаем работать до следующего yield
- Или до выхода из функции
- Выход приводит к исключению в точке вызова __next__

Ленивый генератор

- Функцию можем вызвать несколько раз
- Каждый раз получим новый объект
- Со своим отдельным состоянием
- Над каждым можем вызывать свой __next__
- Вместо метода __next__ часто используют функцию next

Пример

```
 1 def hello():
 2     print('started to work')
 3     return "hello"
 4
 5 def hello_lazy():
 6     print('started to work')
 7     yield "hello"
 8     print('continue to work')
 9
10
11 print('call hello')
12 s = hello()
13 print(s)
14 print('call hello_lazy')
15 gen = hello_lazy()
```

Пример поинтереснее

```
 1 def lazy_lines(name):
 2     with open(name, 'rt') as f:
 3         for line in f:
 4             yield line
 5
 6 gen1 = lazy_lines('data.txt')
 7 print(next(gen1))
 8 print(next(gen1))
 9 gen2 = lazy_lines('data.txt')
10 print(next(gen2))
11 print(next(gen1))
12 print(next(gen2))
```

Пример бесконечного генератора

```
 1 def lazy_fibo():
 2     prev, curr = 0, 1
 3     while True:
 4         yield prev
 5         prev, curr = curr, prev + curr
 6
 7 gen = lazy_fibo()
 8 for _ in range(100):
 9     print(next(gen))
10 for _ in range(100):
11     print(next(gen))
```

Производные генераторы

- Можно передать параметром другой генератор
- В простейшем случае вызывать next и передавать результат в yield
- А можно преобразовывать каждое значение
- Например, превращать строку в число
- А если не превращается, то пропускать

Слабое место

- Мы можем преобразовывать генераторы
- И это похоже на списочные выражения
- Только здесь отложенные вычисления
- И принципиально мы могли бы лениво преобразовывать список
- Но у него нет метода __next__

Слабое место

- И метода __next__ там быть не может
- Потому что список - не ленивая структура
- Но для конкретного списка мы можем создать генератор
- Который будет выдавать элемент за элементом

Решение

- Для конструкций, которые сами не являются генераторами
- Но для которых можно создать генератор, перебирающий их элементы
- Заведем отдельный метод __iter__ (и функцию iter)
- И пусть от возвращает генератор

Решение

- И для унификации у генераторов тоже есть такой метод
- Он возвращает сам генератор
- В итоге мы можем создавать генераторы, принимающие параметером объект с методом __iter__
- Вызывать внутри iter
- И перебирать элементы полученного генератора

Пример ленивого отображения

```
 1 def lazy_hello():
 2     yield "hello"
 3
 4 def lazy_map(data, f):
 5     gen = iter(data)
 6     while True:
 7         yield f(next(gen))
 8
 9 gen = lazy_map(lazy_hello(), len)
10 print(next(gen))
11
12 gen = lazy_map(['q', 'qwerty'], len)
13 print(next(gen))
14 print(next(gen))
```

Встроенные функции

- map - лениво отображает
- filter - лениво фильтрует
- enumerate, zip - тоже ленивые
- Ленивый генератор можно материализовать
- Например, передав в list или использовав в списочном выражении

Генераторное выражение

- Как списочное, только в круглых скобках
- Если это единственный параметр в вызове функции
- То скобки уже есть
- Вторые можно не писать
- Например: ":".join(v[1:-1] for v in data.split(":"))

Плюсы и минусы

- Плюсы - экономия времени на больших структурах
- Минусы
  - Накладные расходы (когда много преобразований на маленьких структурах)
  - Сложности с отладочной печатью