



DaneSoul 2 февраля 2017 в 13:39

Python: коллекции, часть 4/4: Все о выражениях-генераторах, генераторах списков, множеств и словарей

Python, Программирование

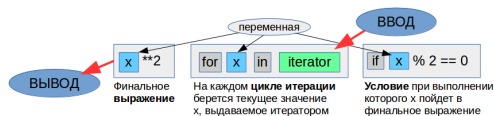
Tutorial

Часть 1

Часть 2

Часть 3

Часть 4

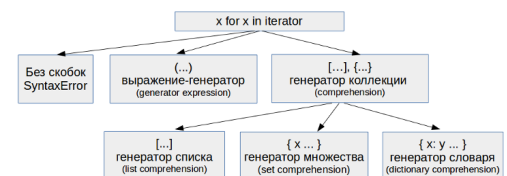


Заключительная часть моего цикла, посещенного работе с коллекциями. Данная статья самостоятельная, может изучаться и без предварительного изучения предыдущих.

Эта статья глубже и детальней предыдущих и поэтому может быть интересна **не только новичкам, но и достаточно опытным Python-разработчикам**.

Будут рассмотрены: выражения-генераторы, генераторы списка, словаря и множества, вложенные генераторы (5 вариантов), работа с `enumerate()`, `range()`.

А также: классификация и терминология, синтаксис, аналоги в виде циклов и примеры применения.



Я постарался рассмотреть **тонкости и нюансы**, которые освещаются далеко не во всех книгах и курсах, и, в том числе, отсутствуют в уже опубликованных на Habrahabr статьях на эту тему.

Оглавление:

1. Определения и классификация.
2. Синтаксис.
3. Аналоги в виде цикла `for` и в виде функций.
4. Выражения-генераторы.
5. Генерация стандартных коллекций.
6. Периодичность и частичный перебор.
7. Вложенные циклы и генераторы.
8. Использование `range()`.
9. Приложение 1. Дополнительные примеры.
10. Приложение 2. Ссылки по теме.

1. Определения и классификация

1.1 Что и зачем

- Генераторы выражений предназначены для компактного и удобного способа генерации коллекций элементов, а также преобразования одного типа коллекций в другой.
- В процессе генерации или преобразования возможно применение условий и модификация элементов.
- Генераторы выражений являются синтаксическим сахаром и не решают задач, которые нельзя было бы решить без их использования.

1.2 Преимущества использования генераторов выражений

- Более короткий и удобный синтаксис, чем генерация в обычном цикле.

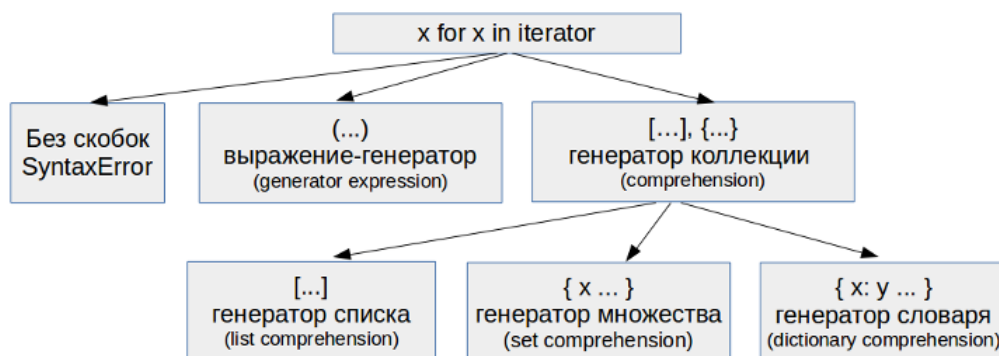
- Более понятный и читаемый синтаксис чем функциональный аналог сочетающий одновременное применение функций `map()`, `filter()` и `lambda`.
- В целом: быстрее набирать, легче читать, особенно когда подобных операций много в коде.

1.3 Классификация и особенности

Сразу скажу, что существует некоторая терминологическая путаница в русских названиях того, о чем мы будем говорить.

В данной статье используются следующие обозначения:

- **выражение-генератор** (generator expression) — выражение в круглых скобках которое выдает создает на каждой итерации новый элемент по правилам.
- **генератор коллекции** — обобщенное название для генератора списка (list comprehension), генератора словаря (dictionary comprehension) и генератора множества (set comprehension).

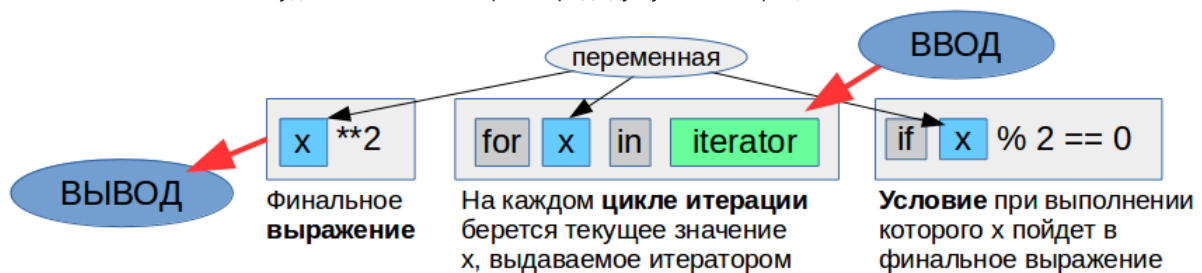


В отдельных местах, чтобы избежать нагромождения терминов, будет использоваться термин «генератор» без дополнительных уточнений.

2. Синтаксис

Для начала приведем иллюстрацию общего синтаксиса выражения-генератора.

Важно: этот синтаксис одинаков и для выражения-генератора и для всех трех типов генераторов коллекций, разница заключается, в каких скобках он будет заключен (смотрите предыдущую иллюстрацию).



Общие принципы важные для понимания:

- Ввод — это итератор — это может быть функция-генератор, выражение-генератор, коллекция — любой объект поддерживающий итерацию по нему.
- Условие — это фильтр при выполнении которого элемент пойдет в финальное выражение, если элемент ему не удовлетворяет, он будет пропущен.
- Финальное выражение — преобразование каждого выбранного элемента перед его выводом или просто вывод без изменений.

2.1 Базовый синтаксис

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5] # Пусть у нас есть исходный список
list_b = [x for x in list_a]      # Создадим новый список используя генератор списка
```

```
print(list_b)                # [-2, -1, 0, 1, 2, 3, 4, 5]
print(list_a is list_b)      # False - это разные объекты!
```

По сути, ничего интересного тут не произошло, мы просто получили копию списка. Делать такие копии или просто перегонять коллекции из типа в тип с помощью генераторов особого смысла нет — это можно сделать значительно проще применив соответствующие методы или функции создания коллекций (рассматривались в первой статье цикла).

Мощь генераторов выражений заключается в том, что мы можем задавать условия для включения элемента в новую коллекцию и можем делать преобразование текущего элемента с помощью выражения или функции перед его выводом (включением в новую коллекцию).

2.2 Добавляем условие для фильтрации

Важно: Условие проверяется на каждой итерации, и только элементы ему удовлетворяющие идут в обработку в выражении.

Добавим в предыдущий пример условие — брать только четные элементы.

```
# if x % 2 == 0 - остаток от деления на 2 равен нулю - число четное
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x for x in list_a if x % 2 == 0]
print(list_b)    # [-2, 0, 2, 4]
```

Мы можем использовать **несколько условий, комбинируя их логическими операторами**:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x for x in list_a if x % 2 == 0 and x > 0]
# берем те x, которые одновременно четные и больше нуля
print(list_b)    # [2, 4]
```

2.3 Добавляем обработку элемента в выражении

Мы можем вставлять не сам текущий элемент, прошедший фильтр, а результат вычисления выражения с ним или результат его обработки функцией.

Важно: Выражение выполняется независимо на каждой итерации, обрабатывая каждый элемент индивидуально.

Например, можем посчитать квадраты значений каждого элемента:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x**2 for x in list_a]
print(list_b)    # [4, 1, 0, 1, 4, 9, 16, 25]
```

Или посчитать длины строк с помощью функции `len()`

```
list_a = ['a', 'abc', 'abcde']
list_b = [len(x) for x in list_a]
print(list_b)    # [1, 3, 5]
```

2.4 Ветвление выражения

Обратите внимание: Мы можем использовать (начиная с Python 2.5) в выражении конструкцию **if-else для ветвления финального выражения**.

В таком случае:

- Условия ветвления пишутся не после, а перед итератором.

- В данном случае if-else это не фильтр перед выполнением выражения, а ветвление самого выражения, то есть переменная уже прошла фильтр, но в зависимости от условия может быть обработана по-разному!

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x if x < 0 else x**2 for x in list_a]
# Если x-отрицательное - берем x, в остальных случаях - берем квадрат x
print(list_b)  # [-2, -1, 0, 1, 4, 9, 16, 25]
```

Никто не запрещает **комбинировать фильтрацию и ветвление**:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_b = [x**3 if x < 0 else x**2 for x in list_a if x % 2 == 0]
# вначале фильтр пропускает в выражение только четные значения
# после этого ветвление в выражении для отрицательных возводит в куб, а для остальных в квадрат
print(list_b)  # [-8, 0, 4, 16]
```

Этот же пример в виде цикла

2.5 Улучшаем читаемость

Не забываем, что в Python синтаксис позволяет использовать переносы строк внутри скобок. Используя эту возможность, можно сделать синтаксис генераторов выражений более легким для чтения:

```
numbers = range(10)

# Before
squared_evens = [n ** 2 for n in numbers if n % 2 == 0]

# After
squared_evens = [
    n ** 2
    for n in numbers
    if n % 2 == 0
]
```

3. Аналоги в виде цикла for и в виде функций

Как уже говорилось выше, задачи решаемые с помощью генераторов выражений можно решить и без них. Приведем другие подходы, которые могут быть использованы для решения тех же задач.

Для примера возьмем простую задачу — сделаем из списка чисел список квадратов четных чисел и решим ее с помощью трех разных подходов:

3.1 Решение с помощью генератора списка

```
numbers = range(10)
squared_evens = [n ** 2 for n in numbers if n % 2 == 0]
print(squared_evens)  # [0, 4, 16, 36, 64]
```

3.2. Решение с помощью цикла for

Важно: Каждый генератор выражений можно переписать в виде цикла for, но не каждый цикл for можно представить в виде такого выражения.

```
numbers = range(10)
squared_evens = []
```

```
for n in numbers:
    if n % 2 == 0:
        squared_evens.append(n ** 2)
print(squared_evens)  # [0, 4, 16, 36, 64]
```

В целом, для очень сложных и комплексных задач, решение в виде цикла может быть понятней и проще в поддержке и доработке. Для более простых задач, синтаксис выражения-генератора будет компактней и легче в чтении.

3.3. Решение с помощью функций.

Для начала, замечу, что выражение генераторы и генераторы коллекций — это тоже функциональный стиль, но более новый и предпочтительный.

Можно применять и более старые функциональные подходы для решения тех же задач, комбинируя `map()`, `lambda` и `filter()`.

```
numbers = range(10)
squared_evens = map(lambda n: n ** 2, filter(lambda n: n % 2 == 0, numbers))
print(squared_evens)          # <map object at 0x7f661e5dba20>
print(list(squared_evens))    # [0, 4, 16, 36, 64]
# Примечание: в Python 2 в переменной squared_evens окажется сразу список, а в Python 3 «map object», который мы превращаем в список с помощью list()
```

Несмотря на то, что подобный пример вполне рабочий, читается он тяжело и использование синтаксиса генераторов выражений будет более предпочтительным и понятным.

4. Выражения-генераторы

Выражения-генераторы (generator expressions) доступны, начиная с Python 2.4. Основное их отличие от генераторов коллекций в том, что они **выдают элемент по-одному, не загружая в память сразу всю коллекцию**.

UPD: Еще раз обратите внимание на этот момент: если мы создаем большую структуру данных без использования генератора, то она загружается в память целиком, соответственно, это увеличивает **расход памяти** Вашим приложением, а в крайних случаях памяти может просто не хватить и Ваше приложение **«упадет» с MemoryError**. В случае использования выражения-генератора, такого не происходит, так как элементы создаются по-одному, в момент обращения.

Пример выражения-генератора:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_gen = (i for i in list_a)  # выражение-генератор
print(next(my_gen))          # -2 - получаем очередной элемент генератора
print(next(my_gen))          # -1 - получаем очередной элемент генератора
```

Особенности выражений-генераторов

1. Генератор **нельзя писать без скобок** — это синтаксическая ошибка.

```
# my_gen = i for i in list_a      # SyntaxError: invalid syntax
```

2. При передаче в функцию дополнительные **скобки** необязательны

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_sum = sum(i for i in list_a)
# my_sum = sum((i for i in list_a))  # так тоже можно
print(my_sum)  # 12
```

3. Нельзя получить длину функцией `len()`

```
# my_len = len(i for i in list_a)  # TypeError: object of type 'generator' has no len()
```

4. Нельзя распечатать элементы функцией **print()**

```
print(my_gen)    # <generator object <genexpr> at 0x7f162db32af0>
```

5. Обратите внимание, что после прохождения по выражению-генератору оно **остаётся пустым!**

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_gen = (i for i in list_a)
print(sum(my_gen)) # 12
print(sum(my_gen)) # 0
```

6. Выражение-генератор может быть **бесконечным**.

```
import itertools
inf_gen = (x for x in itertools.count()) # бесконечный генератор от 0 to бесконечности!
```

Будьте осторожны в работе с такими генераторами, так как при не правильном использовании «эффект» будет как от бесконечного цикла.

7. К выражению-генератору **не применимы срезы!**

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_gen = (i for i in list_a)
my_gen_sliced = my_gen[1:3]
# TypeError: 'generator' object is not subscriptable
```

8. Из генератора легко получить **нужную коллекцию**. Это подробно рассматривается в следующей главе.

5. Генерация стандартных коллекций

5.1 Создание коллекций из выражения-генератора

Создание коллекций из выражения-генератора с помощью функций `list()`, `tuple()`, `set()`, `frozenset()`

Примечание: Так можно создать и неизменное множество и кортеж, так как неизменными они станут уже после генерации.

Внимание: Для строки такой способ не работает! Синтаксис создания генератора словаря таким образом имеет свои особенности, он рассмотрен в следующем под-разделе.

1. Передачей готового выражения-генератора присвоенной переменной в функцию создания коллекции.

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_gen = (i for i in list_a) # выражение-генератор
my_list = list(my_gen)
print(my_list)              # [-2, -1, 0, 1, 2, 3, 4, 5]
```

2. Написание выражения-генератора сразу внутри скобок вызываемой функции создания коллекции.

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_list = list(i for i in list_a)
print(my_list)              # [-2, -1, 0, 1, 2, 3, 4, 5]
```

5.2 Специальный синтаксис генераторов коллекций

В отличие от выражения-генератора, которое выдает значение по-одному, не загружая всю коллекцию в память, при использовании генераторов коллекций, коллекция генерируется сразу целиком.

Соответственно, вместо особенности выражений-генераторов перечисленных выше, такая коллекция будет обладать всеми стандартными свойствами характерными для коллекции данного типа.

Обратите внимание, что для генерации множества и словаря используются одинаковые скобки, разница в том, что у словаря указывается двойной элемент ключ: значение.

1. Генератор списка (list comprehension)

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_list = [i for i in list_a]
print(my_list)           # [-2, -1, 0, 1, 2, 3, 4, 5]
```

Не пишите круглые скобки в квадратных!

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_list = [(i for i in list_a)]
print(my_list)           # [<generator object <genexpr> at 0x7fb81103bf68>]
```

2. Генератор множества (set comprehension)

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
my_set = {i for i in list_a}
print(my_set)            # {0, 1, 2, 3, 4, 5, -1, -2} - порядок случаен
```

3. Генератор словаря (dictionary comprehension)

переворачивание словаря

```
dict_abc = {'a': 1, 'b': 2, 'c': 3, 'd': 3}
dict_123 = {v: k for k, v in dict_abc.items()}
print(dict_123)          # {1: 'a', 2: 'b', 3: 'd'}
                        # Обратите внимание, мы потеряли "c"! Так как значения были одинаковы,
                        # то когда они стали ключами, только последнее значение сохранилось.
```

Словарь из списка:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
dict_a = {x: x**2 for x in list_a}
print(dict_a)            # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, -2: 4, -1: 1, 5: 25}
```

Важно! Такой синтаксис создания словаря работает только в фигурных скобках, **выражение-генератор** так создать нельзя, для этого используется немного другой синтаксис (благодарю @longclaps за подсказку в комментариях):

```
# dict_gen = (x: x**2 for x in list_a)          # SyntaxError: invalid syntax
dict_gen = ((x, x ** 2) for x in list_a)        # Корректный вариант генератора-выражения для словаря
# dict_a = dict(x: x**2 for x in list_a)        # SyntaxError: invalid syntax
dict_a = dict((x, x ** 2) for x in list_a)      # Корректный вариант синтаксиса от @Longclaps
```

5.3 Генерация строк

Для создания строки вместо синтаксиса выражений-генераторов используется метод строки `.join()`, которому в качестве аргументов можно передать выражение генератор.

Обратите внимание: элементы коллекции для объединения в строку должны быть строками!

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
# используем генератор прямо в .join() одновременно приводя элементы к строковому типу
my_str = ''.join(str(x) for x in list_a)
print(my_str) # -2-1012345
```

6. Периодичность и частичный перебор

6.1 Работа с enumerate()

Иногда в условиях задачи в условии-фильтре нужна не проверка значения текущего элемента, а проверка на определенную периодичность, то есть, например, нужно брать каждый третий элемент.

Для подобных задач можно использовать функцию `enumerate()`, задающую счетчик при обходе итератора в цикле:

```
for i, x in enumerate(iterable)
```

здесь `x` — текущий элемент `i` — его порядковый номер, начиная с нуля

Проиллюстрируем работу с индексами:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_d = [(i, x) for i, x in enumerate(list_a)]
print(list_d) # [(0, -2), (1, -1), (2, 0), (3, 1), (4, 2), (5, 3), (6, 4), (7, 5)]
```

Теперь попробуем решить реальную задачу — выберем в генераторе списка каждый третий элемент из исходного списка:

```
list_a = [-2, -1, 0, 1, 2, 3, 4, 5]
list_e = [x for i, x in enumerate(list_a, 1) if i % 3 == 0]
print(list_e) # [0, 3]
```

Важные особенности работы функции enumerate():

- Возможны два варианта вызова функции `enumerate()`:
 - `enumerate(iterator)` без второго параметра считает с 0.
 - `enumerate(iterator, start)` — начинает считать с значения `start`. Удобно, например, если нам надо считать с 1, а не 0.
- `enumerate()` возвращает **кортеж** из порядкового номера и значения текущего элемента итератора. Кортеж в выражении-генераторе результате можно получить двумя способами:
 - `(i, j) for i, j in enumerate(iterator)` — скобки в первой паре нужны!
 - `pair for pair in enumerate(mylist)` — мы работаем сразу с парой
- Индексы считаются для **всех обработанных элементов**, без учета прошли они в дальнейшем условии или нет!

```
first_ten_even = [(i, x) for i, x in enumerate(range(10)) if x % 2 == 0]
print(first_ten_even) # [(0, 0), (2, 2), (4, 4), (6, 6), (8, 8)]
```

4. Функция `enumerate()` не обращается к каким-то внутренним атрибутам коллекции, а просто реализует **счетчик обработанных элементов**, поэтому ничего не мешает ее использовать для неупорядоченных коллекций не имеющих индексации.

5. Если мы ограничиваем количество элементов включенных в результат по `enumerate()` счетчику (например `if i < 10`), то итератор будет все равно **обработан целиком**, что в случае огромной коллекции будет очень ресурс-затратно. Решение этой проблемы рассматривается ниже в под-разделе «Перебор части итерируемого».

6.2 Перебор части итерируемого.

Иногда бывает задача из очень большой коллекции или даже бесконечного генератора получить выборку первых нескольких элементов, удовлетворяющих условию.

Если мы используем обычное генераторное выражение с условием ограничением по `enumerate()` индексу или срезу полученной результирующей коллекции, то нам в любом случае придется пройти всю огромную коллекцию и потратить на это уйму компьютерных ресурсов.

Выходом может быть использование функции **`islice()`** из пакета **`itertools`**.

```
import itertools
first_ten = (itertools.islice((x for x in range(1000000000) if x % 2 == 0), 10))
print(list(first_ten)) # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Для сомневающихся: проверяем время выполнения

7. Вложенные циклы и генераторы

Рассмотрим более комплексные варианты, когда у нас циклы или сами выражения-генераторы являются вложенными. Тут возможны несколько вариантов, со своими особенностями и сферой применения, чтобы не возникало путаницы, рассмотрим их по-отдельности, а после приведем общую схему.

7.1 Вложенные циклы

В результате генерации получаем **одномерную структуру**.

Важно! При работе с вложенными циклами внутри генератора выражений порядок следования инструкций `for in` будет такой же (слева-направо), как и в аналогичном решении без генератора, только на циклах (сверху-вниз)! Тоже справедливо и при более глубоких уровнях вложенности.

7.1.1 Вложенные циклы `for` где циклы идут по независимым итераторам

Общий синтаксис: **`[expression for x in iter1 for y in iter2]`**

Применение: генерируем одномерную структуру, используя данные из двух итераторов.

Например, создадим словарь, используя кортежи координат как ключи, заполнив для начала его значения нулями.

```
rows = 1, 2, 3
cols = 'a', 'b'
my_dict = {(col, row): 0 for row in rows for col in cols}
print(my_dict) # {'a', 1): 0, ('b', 2): 0, ('b', 3): 0, ('b', 1): 0, ('a', 3): 0, ('a', 2): 0}
```

Дальше можем задавать новые значения или получать их

Тоже можно сделать и с **дополнительными условиями-фильтрами** в каждом цикле:

```
rows = 1, 2, 3, -4, -5
cols = 'a', 'b', 'abc'
# Для наглядности разнесем на несколько строк
my_dict = {
    (col, row): 0 # каждый элемент состоит из ключа-кортежа и нулевого значения
    for row in rows if row > 0 # Только положительные значения
    for col in cols if len(col) == 1 # Только односимвольные
}
print(my_dict) # {'a', 1): 0, ('b', 2): 0, ('b', 3): 0, ('b', 1): 0, ('a', 3): 0, ('a', 2): 0}
```

Эта же задача решенная с помощью цикла

7.1.2 Вложенные циклы for где внутренний цикл идет по результату внешнего цикла

Общий синтаксис: *[expression for x in iterator for y in x]*.

Применение: Стандартный подход, когда нам надо обходить двумерную структуру данных, превращая ее в «плоскую» одномерную. В данном случае, мы во внешнем цикле проходим по строкам, а во внутреннем по элементам каждой строки нашей двумерной структуры.

Допустим у нас есть двумерная матрица — список списков. И мы желаем преобразовать ее в плоский одномерный список.

```
matrix = [[0, 1, 2, 3],
          [10, 11, 12, 13],
          [20, 21, 22, 23]]

# Решение с помощью генератора списка:
flattened = [n for row in matrix for n in row]
print(flattened) # [0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23]
```

Таже задача, решенная с помощью вложенных циклов

UPD:Изящные решения из комментариев

7.2 Вложенные генераторы

Вложенными могут быть не только циклы for внутри выражения-генератора, но и сами генераторы. Такой подход применяется когда нам надо **строить двумерную структуру**.

Важно!: В отличии от примеров выше с вложенными циклами, для вложенных генераторов, вначале обрабатывается внешний генератор, потом внутренний, то есть порядок идет справа-налево.

Ниже рассмотрим два варианта подобного использования.

7.2.1 — Вложенный генератор внутри генератора — двумерная из двух одномерных

Общий синтаксис: *[[expression for y in iter2] for x in iter1]*

Применение: генерируем двумерную структуру, используя данные из двух одномерных итераторов.

Для примера создадим матрицу из 5 столбцов и 3 строк и заполним ее нулями:

```
w, h = 5, 3 # зададим ширину и высоту матрицы
matrix = [[0 for x in range(w)] for y in range(h)]
print(matrix) # [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Создание этой же матрицы двумя вложенными циклами - обратите внимание на порядок вложения

Примечание: После создания можем работать с матрицей как с обычным двумерным массивом

7.2.2 — Вложенный генератор внутри генератора — двумерная из двумерной

Общий синтаксис: `[[expression for y in x] for x in iterator]`

Применение: Обходим двумерную структуру данных, сохраняя результат в другую двумерную структуру.

Возьмем матрицу:

```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Возведем каждый элемент матрицы в квадрат:

```
squared = [[cell**2 for cell in row] for row in matrix]
print(squared) # [[1, 4, 9, 16], [25, 36, 49, 64], [81, 100, 121, 144]]
```

Эта же операция в виде вложенных циклов

Обобщим все вышеперечисленные варианты в одной схеме (полный размер по клику):



7.3 — Генератор итерирующийся по генератору

Так как любой генератор может использоваться как итератор в цикле for, это так же можно использовать и для создания генератора по генератору.

При этом синтаксически это может записываться в два выражения или объединяться во вложенный генератор.

Проиллюстрирую и такую возможность.

Допустим у нас есть два таких генератора списков:

```
list_a = [x for x in range(-2, 4)] # Так сделано для дальнейшего примера синтаксиса,
                                   # конечно в подобной задаче достаточно только range(-2, 4)
list_b = [x**2 for x in list_a]
```

Тоже самое можно записать и в одно выражение, подставив вместо list_a его генератор списка:

```
list_c = [x**2 for x in [x for x in range(-2, 4)]]
print(list_c) # [4, 1, 0, 1, 4, 9]
```

UPD от @longclaps: Преимущество от комбинирования генераторов на примере сложной функции $f(x) = u(v(x))$

```
list_c = [t + t ** 2 for t in (x ** 3 + x ** 4 for x in range(-2, 4))]
```

8. Использование range()

Говоря о способах генерации коллекций, нельзя обойти вниманием простую и очень удобную функцию range(), которая предназначена для создания арифметических последовательностей.

Особенности функции range():

- Наиболее часто функция range() **применяется** для запуска цикла for нужное количество раз. Например, смотрите генерацию матрицы в примерах выше.
- В Python 3 range() возвращает **генератор**, который при каждом к нему обращении выдает очередной элемент.
- Используемые **параметры** аналогичны таковым в срезах (кроме первого примера с одним параметром):
 - range(stop) — в данном случае с 0 до stop-1;
 - range(start, stop) — Аналогично примеру выше, но можно задать начало отличное от нуля, можно и отрицательное;
 - range(start, stop, step) — Добавляем параметр шага, который может быть отрицательным, тогда перебор в обратном порядке.
- В **Python 2** были 2 функции:
 - range(...) которая аналогична выражению list(range(...)) в Python 3 — то есть она выдавала не итератор, а сразу готовый список. То есть все проблемы возможной **нехватки памяти**, описанные в разделе 4 актуальны, и использовать ее в Python 2 надо очень аккуратно!
 - xrange(...) — которая работала аналогично range(...) в Python 3 и из 3 версии была исключена.

Примеры использования:

```
print(list(range(5)))      # [0, 1, 2, 3, 4]
print(list(range(-2, 5)))  # [-2, -1, 0, 1, 2, 3, 4]
print(list(range(5, -2, -2))) # [5, 3, 1, -1]
```

9. Приложение 1. Дополнительные примеры

9.1 Последовательный проход по нескольким спискам

```
import itertools
l1 = [1,2,3]
l2 = [10,20,30]
result = [l*2 for l in itertools.chain(l1, l2)]
print(result)  # [2, 4, 6, 20, 40, 60]
```

9.2 Транспозиция матрицы

(Преобразование матрицы, когда строки меняются местами со столбцами).

Возьмем матрицу.

```
matrix = [[1, 2, 3, 4],
          [5, 6, 7, 8],
          [9, 10, 11, 12]]
```

Сделаем ее транспозицию с помощью генератора выражений:

```
transposed = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
print(transposed)  # [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Эта же транспозиция матрицы в виде цикла

И немного черной магии от @longclaps

9.3 Задача выбора только рабочих дней

```
# Формируем список дней от 1 до 31 с которым будем работать
days = [d for d in range(1, 32)]

# Делим список дней на недели
weeks = [days[i:i+7] for i in range(0, len(days), 7)]
print(weeks)  # [[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14], [15, 16, 17, 18, 19, 20, 21], [22, 23, 24, 25, 26, 27, 28], [29, 30, 31]]

# Выбираем в каждой неделе только первые 5 рабочих дней, отбрасывая остальные
work_weeks = [week[0:5] for week in weeks]
print(work_weeks)  # [[1, 2, 3, 4, 5], [8, 9, 10, 11, 12], [15, 16, 17, 18, 19], [22, 23, 24, 25, 26], [29, 30, 31]]

# Если нужно одним списком дней - можно объединить
wdays = [item for sublist in work_weeks for item in sublist]
print(wdays)  # [1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 15, 16, 17, 18, 19, 22, 23, 24, 25, 26, 29, 30, 31]
```

Можно убрать выходные еще более изящно, используя только индексы

10. Приложение 2. Ссылки по теме

1. Хорошая англоязычная статья с детальным объяснением что такое генераторы и итераторы

Иллюстрация из статьи:

2. Если у Вас есть сложности с пониманием логики работы с генераторными выражениями, посмотрите интересную англоязычную статью, где проводятся аналогии между генераторными выражениями и работой с SQL и таблицами Excel.

Например так:

3. UPD от @fireSparrow: Существует расширение Python — PythonQL, позволяющее работать с базами данных в стиле генераторов коллекций.
4. Иллюстрированная статья на английском, довольно наглядно показывает синтаксис генераторных выражений.
5. Если требуются дополнительные примеры по теме вложенных генераторных выражений (статья на английском).

Часть 1	Часть 2	Часть 3	Часть 4
---------	---------	---------	---------

Приглашаю к обсуждению:

- Если я где-то допустил неточность или не учёл что-то важное — пишите в комментариях, важные комментарии будут позже добавлены в статью с указанием вашего авторства.
- Если какие-то моменты не понятны и требуется уточнение — пишите ваши вопросы в комментариях — или я или другие читатели дадут ответ, а дельные вопросы с ответами будут позже добавлены в статью.

Теги: python, программирование, коллекция, структуры данных, генератор, выражение-генератор, comprehension, generator, generator expression, словарь, множество, список