



Итераторы, генераторы

Python 3.3

Во многих современных языках программирования используют такие сущности как итераторы.

Основное их назначение - это упрощение навигации по элементам объекта, который, как правило, представляет собой некоторую коллекцию (список, словарь и т.п.).

Определения

- ▶ **Итерируемый объект** - это объект, который позволяет поочередно обойти свои элементы и может быть преобразован к итератору.

- ▶ *Итератор* - это объект, который поддерживает функцию `next()` для перехода к следующему элементу коллекции.

пример

Когда вы создаёте список (list) вы можете считывать его элементы по одному — это называется итерацией.

```
lst = [1, 2, 3]
```

```
for i in lst:
```

```
    print(i)
```

```
1
```

```
2
```

```
3
```

Lst — итерируемый объект (iterable)

for

Основное место использования итераторов - это цикл `for`.

Если вы перебираете элементы в некотором списке или символы в строке с помощью цикла `for`, то, фактически, это означает, что при каждой итерации цикла происходит обращение к итератору, содержащемуся в строке/списке, с требованием выдать следующий элемент, если элементов в объекте больше нет, то итератор генерирует исключение, обрабатываемое в рамках цикла `for` незаметно для пользователя.

Итерируемые объекты достаточно удобны потому что вы можете считывать из них столько данных, сколько вам необходимо, но при этом вы храните все значения последовательности в памяти и это не всегда приемлемо, особенно если вы имеете достаточно большие последовательности.

iter() и next()

Объекты, элементы которых можно перебирать в цикле `for`, содержат в себе объект итератор, для того, чтобы его получить необходимо использовать функцию `iter()`, а для извлечения следующего элемента из итератора - функцию `next()`.

пример

```
num_list = [1, 2, 3, 4, 5]
```

```
for i in num_list:
```

```
    print(i)
```

1

2

3

4

5


```
itr = iter(num_list)
```

```
print(next(itr))
```

```
1
```

```
print(next(itr))
```

```
2
```

```
print(next(itr))
```

```
3
```

```
print(next(itr))
```

```
4
```

```
print(next(itr))
```

```
5
```

```
print(next(itr))
```

```
Traceback (most recent call last):
```

```
File "<pyshell#12>", line 1, in <module>
```

```
print(next(itr))
```

Как видно из примера вызов функции *next(itr)* каждый раз возвращает следующий элемент из списка, а когда эти элементы заканчиваются, генерируется исключение *StopIteration*.

Генератор

Генератор - это итератор, элементы которого можно перебирать (итерировать) только один раз.

Любая функция в Python, в теле которой встречается ключевое слово *yield*, называется *генераторной функцией* — при вызове она возвращает *объект-генератор*.

Вместо ключевого слова `return` в генераторе используется `yield`.

yield

При первом исполнении кода тела функции код будет выполнен с начала и до первого встретившегося оператора **yield**. После этого будет возвращено первое значение и выполнение тела функции опять приостановлено.

Запрос следующего значения у генератора во время итерации заставит код тела функции выполняться дальше (с предыдущего **yield**'а), пока не встретится следующий **yield**. Генератор считается «закончившимся» в случае если при очередном исполнении кода тела функции не было встречено ни одного оператора **yield**.

Функции-генераторы - это функции, которые возвращают значение и затем могут продолжить работу с того места, где они остановились в предыдущий раз.

В результате генераторы позволяют нам генерировать последовательности значений постепенно, не создавая всю последовательность одновременно в памяти.

Во многих отношениях, функция-генератор выглядит очень похоже на обычную функцию. Основное отличие в том, что когда эта функция компилируется, она становится объектом, который поддерживает протокол итераций.

Это значит, что когда такая функция вызывается в Вашем коде, она не просто возвращает значение и завершает работу.

Вместо этого, функция-генератор ставит своё выполнение на паузу, и возобновляет выполнение с последней точки генерации значений.

Основное преимущество такого подхода в том, что вместо необходимости сразу вычислить всю серию значений, генератор генерирует одно значение и ставит выполнение на паузу, ожидая дальнейших инструкций.

Такая особенность работы называется **state suspension**.

range()

Например, функция `range()` не создает весь список в памяти от начала до конца.

Вместо этого она просто хранит последнее значение и размер шага, и постепенно возвращает значения.

В итоге список генерируется постепенно без необходимости создания одного большого списка в памяти.

Обычно генераторы используются в циклах. На каждой итерации цикла используется только очередное значение из генератора

пример

Функция, которая возводит числа в куб

```
def create_cubes(n):  
    result = []  
    for x in range (n):  
        result.append(x**3)  
    return result
```

здесь мы храним в памяти весь список

Функция-генератор, которая возводит числа в куб

```
def gencubes(n):  
    for x in range(n):  
        yield x**3
```

Здесь каждый раз получаем лишь одно значение, всю последовательность одновременно в списке не храним, используем память более эффективно. Особенно заметно при работе с Big Data

Чтобы получить результат в виде списка используем `list(gencubes(10))`

аналогично

```
for x in gencubes(10):
```

```
    print(x)
```

```
0
```

```
1
```

```
8
```

```
27
```

```
64
```

```
125
```

```
216
```

```
343
```

```
512
```

```
729
```

не хранит в памяти список, каждый раз
выводит лишь одно значение

функция для получения чисел Фибоначчи

```
def genfibon(n):  
    """  
    Generate a fibonnaci sequence up to n  
    """  
    a = 1  
    b = 1  
    for i in range(n):  
        yield a  
        a,b = b,a+b
```

a - очередное число
b - предыдущее число
yield возвращает очередное
значение

пример

```
for num in genfibon(10):  
    print(num)
```

```
1  
1  
2  
3  
5  
8  
13  
21  
34  
55
```

обычная функция

храним в памяти весь список

```
def fibon(n):
```

```
    a = 1
```

```
    b = 1
```

```
    output = []
```

```
    for i in range(n):
```

```
        output.append(a)
```

```
        a,b = b,a+b
```

```
    return output
```

Если мы укажем больше значение n (например 100000), вторая функция будет хранить каждое из результирующих значений, хотя в нашем случае нам только нужен предыдущий результат, чтобы вычислить следующее значение

Выражение - генератор

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, так же как и генераторы коллекций являются синтаксическим сахаром и не решают задач, которые нельзя было бы решить без их использования.

Преимущества использования генераторов выражений

- ▶ Более короткий и удобный синтаксис, чем генерация в обычном цикле.
- ▶ Более понятный и читаемый синтаксис
- ▶ Быстрее набирать, легче читать, особенно когда подобных операций много в коде.

классификация

- ▶ выражение-генератор (generator expression) — выражение в круглых скобках которое выдает 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)

List comprehensions

Генераторы списков предназначены для удобной обработки списков, к которой можно отнести и создание новых списков, и модификацию существующих.

Генератор списков

Для создания списка, заполненного одинаковыми элементами, можно использовать оператор повторения списка, например:

```
A = [0] * n
```

Общий вид генератора следующий:

```
[выражение for переменная in список]
```

[выражение for переменная in список]

где

переменная – идентификатор некоторой переменной,

список – список значений, который принимает данная переменная (как правило, полученный при помощи функции `range`),

выражение – некоторое выражение, которым будут заполнены элементы списка, как правило, зависящее от использованной в генераторе переменной.

Вот несколько примеров использования генераторов.

Создать список, состоящий из `n` нулей

```
A = [0 for i in range(n)]
```

Генераторы списков

Создать список, заполненный квадратами целых чисел можно так:

```
A = [i ** 2 for i in range(n)]
```

Если нужно заполнить список квадратами чисел от 1 до n, то можно изменить параметры функции `range` на `range(1, n + 1)`:

```
A = [i ** 2 for i in range(1, n + 1)]
```

Генератор списков

Вот так можно получить список, заполненный случайными числами от 1 до 9 (используя функцию `randint` из модуля `random`):

(про работу с модулями подробности позже)

```
A = [randint(1, 9) for i in range(n)]
```

А в этом примере список будет состоять из строк, считанных со стандартного ввода: сначала нужно ввести число элементов списка (это значение будет использовано в качестве аргумента функции `range`), потом — заданное количество строк:

```
A = [input() for i in range(int(input()))]
```

Генератор списков

```
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]
```

```
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]
```

Генератор списков

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

Можно использовать условия:

```
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]
```

Генератор списков

```
>>> c = [c * 3 for c in 'list']
```

```
>>> c
```

```
['lll', 'iii', 'sss', 'ttt']
```

```
-----
```

```
>>> c = [c * 3 for c in 'list' if c != 'i']
```

```
>>> c
```

```
['lll', 'sss', 'ttt']
```

```
-----
```

```
>>> c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']
```

```
>>> c
```

```
['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```


сравнение

```
numbs = [1, 2, 3, 4, 5]
```

```
result = []
```

```
for x in numbs:
```

```
if x > 3:
```

```
y = x * x
```

```
result.append(y)
```

```
numbs = [1, 2, 3, 4, 5]
```

```
result = [x * x for x in numbs if x > 3]
```

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
```

Генератор словаря (dictionary comprehension) - переворачиваем словарь

```
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.

Генератор словаря

```
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}
```

```
dict_gen = ((x, x ** 2) for x in list_a)
```

генератор-выражения для словаря

Выражение-генератор

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.

Если мы создаем большую структуру данных без использования генератора, то она загружается в память целиком, соответственно, это увеличивает расход памяти приложением, а в крайних случаях памяти может просто не хватить.

В случае использования выражения-генератора, такого не происходит, так как элементы создаются по-одному, в момент обращения.

СИНТАКСИС

```
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
```

Выражение-генератор

```
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
```

Выражение-генератор

```
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
```

Обратите внимание, что после прохождения по выражению-генератору оно остается пустым!

Практика

Создать генератор списка из исходного

- 1) берет только четные значения, отрицательные возводит в куб, остальные в квадрат
- 2) считает длину строк для списка из строк
- 3) список квадратов четных чисел
- 4) только положительные, кратные 5, отрицательные заменить на 0
- 5) из строки - только гласные буквы
- 6) Создать генератор словаря, значение равно квадрат ключа

Практика

7) из [1,2,3,4,5,6,7] получить {1: 1, 3: 27, 5: 125, 7: 343}

8) из [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7] получить {2, 4, 6}

9) получить список [0, 10, 20, 30, 40, 50, 60, 70, 80, 90] без исходного

10) написать функцию-генератор с yield, которая может перебирать числа, делящиеся на 7, в диапазоне от 0 до n.

11) функция генератор, выводит четные числа, разделенные запятыми от 0 до n