

mango-office.ru

Виртуальная АТС Расширенная

2 000 ₽

Узнать больше

РЕКЛАМА

[Справочник по языку Python3.](#) / Функция генератора в Python

[Генератор](#) - это функция, которая возвращает объект [итератора](#). Она выглядит как [обычная функция](#), за исключением того, что она содержит [выражение yield](#) для создания серии значений, которые можно использовать в [цикле for ... in](#) или которые можно извлечь по одному с помощью [функции next\(\)](#).

Когда вызывается [обычная функция](#), то она получает личное пространство имен, в котором создаются ее локальные переменные. Когда функция достигает оператора [return](#), локальные переменные уничтожаются и значение возвращается вызывающей стороне. Последующий вызов той же функции создает новое [локальное пространство имен](#) и новый набор локальных переменных. Но что, если локальные переменные не были возвращены при выходе из функции? Что если позже можно возобновить функцию с того места, где она остановилась? Это то, что обеспечивают генераторы. О них можно думать как о возобновляемых функциях.

Вот самый простой пример функции генератора:

```
def generate_ints(N):
    for i in range(N):
        yield i
```

Любая функция, содержащая [ключевое слово yield](#), является [функцией генератора](#). Ключевое слово yield обнаруживается компилятором байт-кода Python, который компилирует функцию в результате.

Когда вызывается [функция генератора](#), то она **не возвращает единственное значение**, как это делает оператор [return](#). Вместо этого она возвращает [объект генератора](#), который поддерживает протокол итератора.

При выполнении [выражения yield](#) генератор выводит значение i, аналогичное [оператору return](#). Разница между yield и оператором return заключается в том, что при достижении выхода, состояние выполнения генератора приостанавливается и локальные переменные сохраняются. При следующем вызове [метода генератора __next__\(\)](#) функция возобновит свое выполнение.

Вот пример использования генератора generate_ints():

```
>>> gen = generate_ints(3)
>>> gen
# <generator object generate_ints at ...>
>>> next(gen)
# 0
>>> next(gen)
# 1
>>> next(gen)
# 2
>>> next(gen)
# Traceback (most recent call last):
#   File "stdin", line 1, in <module>
#   File "stdin", line 2, in generate_ints
# StopIteration
```

Также можно написать для i в generate_ints(5) или a, b, c = generate_ints(3).

Внутри функции генератора возвращаемое значение вызывает [исключение StopIteration(value) из метода __next__()]. Как то

Вверх

 происходит или достигается нижняя часть функции, обработка значений завершается и генератор не может выдавать дальнейшие значения.

Можно достичь эффекта генераторов вручную, написав свой собственный класс и сохранив все локальные переменные генератора в качестве переменных экземпляра. Например вернуть [список целых чисел](#) можно, установив `self.count` в `0`, а метод `__next__()` увеличит `self.count` и вернет его. Однако для умеренно сложного генератора написание соответствующего класса может быть намного сложнее.

Набор тестов, включенный в библиотеку `Python Lib/test/test_generators.py` содержит ряд более интересных примеров. Вот один генератор, который реализует обход дерева по порядку, используя генераторы рекурсивно.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Два других примера в `test_generators.py` дают решения для проблемы N-Queens (размещение N ферзей на шахматной доске NxN, чтобы ни одна королева не угрожала другому) и "Королевский тур" - поиск маршрута, по которому король ведет к каждому квадрату шахматной доски NxN не посещая ни одного квадрата дважды.

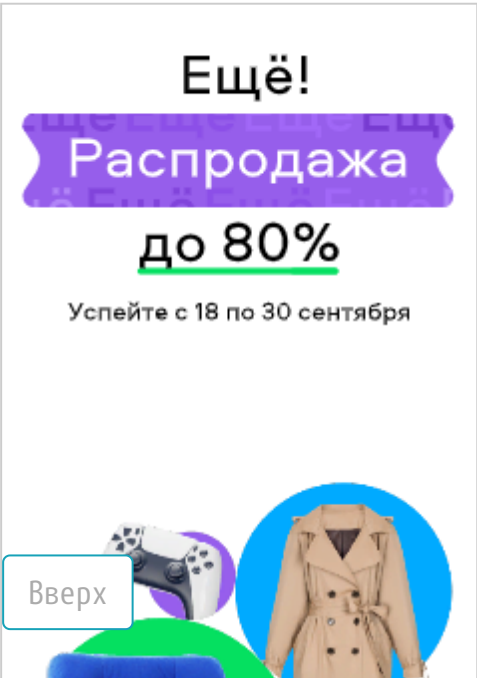
Ограничения выражений-генераторов.

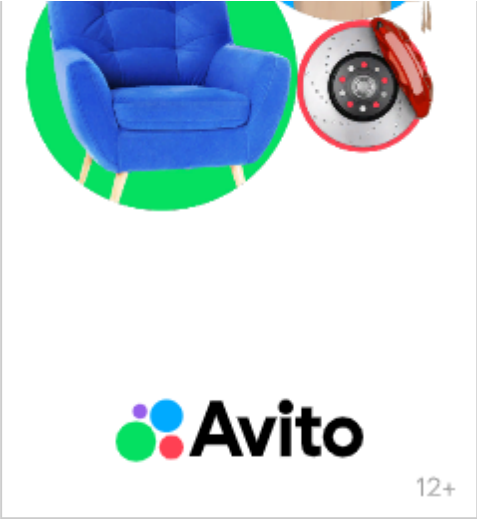
- Нельзя получить длину генератора [функцией len\(\)](#);
- Нельзя распечатать элементы генератора [функцией print\(\)](#);
- У генератора нельзя получить [элемент по индексу](#);
- К выражению-генератору нельзя применить [обычные операции среза](#) или [функцию slice\(\)](#), хотя, для этих целей, можно воспользоваться функцией `itertools.islice()` модуля `itertools`.
- После использования/итерации по генератору, он остается пустым;

Содержание раздела:

- [КРАТКИЙ ОБЗОР МАТЕРИАЛА.](#)
- [Преобразование простой функции в генератор Python](#)
- [Передача значений в генератор Python](#)
- [Использование send\(\), throw\(\) и close\(\) в генераторах Python](#)
- [Выражение yield](#)
- [Выражение yield from <expression>](#)
- [Выражение-генератор](#)
- [Обработка больших данных при помощи генераторов Python](#)

ХОЧУ ПОМОЧЬ
ПРОЕКТУ





[DOCS-Python.ru](#)[™], 2023 г.

(**Внимание!** При копировании материала ссылка на источник обязательна)

[@docs_python_ru](#)

Вверх