Генераторы

№ урока: 6 **Курс:** Python Essential

Средства обучения: PyCharm

Обзор, цель и назначение урока

После завершения урока обучающиеся будут иметь представление о механизме генераторов, научатся создавать генераторы и простейшие сопрограммы.

Изучив материал данного занятия, учащийся сможет:

- Создавать и использовать генераторы
- Использовать генераторы-выражения
- Использовать подгенераторы
- Иметь представление о yield-выражениях

Содержание урока

1. Что такое генераторы?
2. Создания генератора
3. Различия между функцией генератора и нормальной функцией
4. Что такое списковые включения?
5. Выражение генератора Python

Резюме

Что такое генераторы?
Чтобы создать итератор в Python иногда приходится приложить немало усилий: определить методы __iter__() и __next__() методы, сохранить информацию о внутреннем состоянии, выбрасывать ошибку StopIteration, когда элементы контейнера закончились.
Это одновременно длительно и утомительно. Генераторы созданы, чтобы решить подобные ситуации.
Генераторы - это простой путь создания итераторов. Вся работа по созданию итератора выполняться автоматически генераторами.
Проще говоря, генераторы - это функция возвращающая объект (итератор), который мы можем итерировать.

Создания генератора
Функции-генераторы похожи на обычные функции в Python. Вместо оператора return в генераторах используется yield.
Если функция возвращает значение с помощью yield, Python автоматически создает не обычную функцию, а генератор.
Разница return и yield в том, что, оператор return полностью завершает функцию, а оператор yield приостанавливает функцию, сохраняя все ее состояния, а затем продолжает оттуда при последующих вызовах.
Различия между функцией генератора и нормальной функцией

1. **Функция генератора** содержит один или несколько операторов yield.
2. При вызове **генератор** возвращает объект (итератор), но не начинает выполнение немедленно.
3. Такие методы, как __iter__() и __next__(), реализуются автоматически. Мы можем перебирать элементы, используя next().
4. Как только возвращается значение с помощью yield, функция приостанавливается, и управление передается вызывающей стороне.
5. Локальные переменные и их состояния запоминаются между вызовами.

CyberBionic systematics
CyberBionic Systematics ®
19 Eugene Sverstyuk Str., 5 floor
Kyiv, Ukraine
Tel. 0 800 337 146
E-mail: edu@cbsystematics.com
Site: www.edu.cbsystematics.com | itvdn.com

6. Наконец, когда функция завершается, StopIteration автоматически вызывается при последующих вызовах.

Вот пример, иллюстрирующий все вышеизложенное. У нас есть функция генератора с именем fibonacci_generator() и операторам yield.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

Запустим пример в интерактивной консоли:

```
>>> # Получаем объект, но не начинаем выполнение сразу.
>>> a = my_gen()

>>> # Мы можем перебирать элементы, используя next ().
>>> next(a)
This is printed first
1
>>> # Когда функция возвразает значение, функция приостанавливается, и управление передается
вызывающей стороне.

>>> # Локальные переменные и их состояния запоминаются между вызовами.
>>> next(a)
This is printed second
2

>>> next(a)
This is printed at last
3

>>> # Когда функция завершается, StopIteration автоматически вызывается при последующих вызовах.
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

В приведенном выше примере следует отметить одну интересную вещь: значение переменной n запоминается между каждым вызовом.

В отличие от обычных функций, локальные переменные не уничтожаются при выполнении функции. Более того, объект-генератор может быть повторен только один раз.

Чтобы перезапустить процесс, нам нужно создать еще один объект-генератор, используя запись a = my_gen().

И последнее, что следует отметить, это то, что мы можем использовать генераторы напрямую с циклами for.

Это связано с тем, что цикл for принимает итератор и выполняет итерацию по нему с помощью функции next(). Он автоматически завершается, когда вызывается StopIteration.

Рассмотрим более сложный пример использования итератора:

```
def fibonacci_generator(count):
    value_1, value_2 = 0, 1
```


CyberBionic Systematics ®
19 Eugene Sverstyuk Str., 5 floor
Kyiv, Ukraine

Tel. 0 800 337 146
E-mail: edu@cbsystematics.com
Site: www.edu.cbsystematics.com | itvdn.com

```
    for _ in range(count):
        value_1, value_2 = value_2, value_1 + value_2
        yield value_1

fibonacci_iterator = fibonacci_generator(10)

for number in fibonacci_iterator:
    print(number)
```

Данный генератор возвращает по одному элементу ряда Фибоначчи за раз, при этом состояние цикла for в функции замораживается и восстанавливается на следующей итерации.

Что такое списковые включения?

Списковые включения - это элегантный и лаконичный способ создания нового списка из существующего. Списковое включение состоит из выражения, за которым следует оператор for в квадратных скобках.
Вот пример создания списка, в котором каждый элемент имеет степень увеличения 2.

```
pow2 = [2 ** x for x in range(10)]
print(pow2)

# [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Эквивалентный код без спискового включения:

```
pow2 = []
for x in range(10):
   pow2.append(2 ** x)
```

Списковое включение может дополнительно содержать операторы for или if. Необязательный оператор if может отфильтровывать элементы для нового списка. Вот несколько примеров:

```
>>> pow2 = [2 ** x for x in range(10) if x > 5]
>>> pow2
[64, 128, 256, 512]
>>> odd = [x for x in range(20) if x % 2 == 1]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> [x+y for x in ['Python ','C '] for y in ['Language','Programming']]
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

Выражение генератора Python

Простые генераторы можно легко создавать "на лету" с помощью **выражений генератора**. Это упрощает создание генераторов.
Синтаксис выражения генератора аналогичен синтаксису списковому включению. Но квадратные скобки заменены круглыми скобками.
Основное различие между пониманием списка и выражением генератора состоит в том, что списковое включение создает весь список сразу, в то время как выражение генератора создает по одному элементу за раз.
У генераторов ленивое исполнение (производство предметов только по запросу). По этой причине выражение генератора намного эффективнее с точки зрения памяти, чем эквивалентное списковое включение.

```
# Создание списка
my_list = [1, 3, 6, 10]

# возвести каждый елемент в квадрат, используя списковое включение
list_ = [x**2 for x in my_list]

# подобный механихм с генераторами:
# выражения генератора заключены в круглые скобки ()
generator = (x**2 for x in my_list)

print(list_)
```


CyberBionic Systematics ®
19 Eugene Sverstyuk Str., 5 floor
Kyiv, Ukraine

Tel. 0 800 337 146
E-mail: edu@cbsystematics.com
Site: www.edu.cbsystematics.com | itvdn.com

```
print(generator)

# [1, 9, 36, 100]
# <generator object <genexpr> at 0x7f5d4eb4bf50>
```

Выше видно, что выражение генератора не сразу дало требуемый результат. Вместо этого он вернул объект-генератор, который производит элементы только по запросу.
Вот как мы можем начать получать предметы из генератора:

```
my_list = [1, 3, 6, 10]

a = (x**2 for x in my_list)
print(next(a)) # 1

print(next(a)) # 9

print(next(a)) # 36

print(next(a)) # 100

next(a)

# Traceback (most recent call last):
#   File "<string>", line 15, in <module>
# StopIteration
```

Выражения генератора могут использоваться как аргументы функции. При таком использовании круглые скобки можно опустить:

```
>>> sum(x**2 for x in my_list)
146

>>> max(x**2 for x in my_list)
100
```

Есть несколько причин, по которым генераторы являются выгодной реализацией:

1. Легко реализовать и читать код
2. Эффективная работа с памятью
3. Создание бесконечной последовательности

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. Создание вложенных генераторов

```
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
        yield num**2

    print(sum(square(fibonacci_numbers(10))))
```

Закрепление материала

- Что такое генератор?
- Что такое выражение-генератор?

CyberBionic Systematics ®
19 Eugene Sverstyuk Str., 5 floor
Kyiv, Ukraine

Tel. 0 800 337 146
E-mail: edu@cbsystematics.com
Site: www.edu.cbsystematics.com | itvdn.com

- Что является результатом yield-выражения?
- Какие методы есть у генераторов?
- Что такое списковые включения?
- В каких случаях лучше использовать генераторы?

Дополнительное задание

Задание
Напишите функцию-генератор для получения n первых простых чисел.

Самостоятельная деятельность учащегося

Задание 1
Напишите генератор, который возвращает элементы заданного списка в обратном порядке (аналог reversed).

Задание 2
Выведите из списка чисел список квадратов четных чисел. Используйте 2 варианта решения: генератор и цикл

Рекомендуемые ресурсы

Документация Python
https://docs.python.org/3/tutorial/classes.html#generators
https://docs.python.org/3/tutorial/classes.html#generator-expressions
https://docs.python.org/3/reference/expressions.html#generator-expressions
https://docs.python.org/3/reference/expressions.html#yield-expressions
https://docs.python.org/3/reference/expressions.html#generator-iterator-methods
https://docs.python.org/3/reference/expressions.html#examples
https://www.python.org/dev/peps/pep-0255/
https://www.python.org/dev/peps/pep-0342/
https://www.python.org/dev/peps/pep-0380/

Статьи в Википедии о ключевых понятиях, рассмотренных на этом уроке
https://en.wikipedia.org/wiki/Generator_(computer_programming)
https://ru.wikipedia.org/wiki/Сопрограмма

CyberBionic Systematics ®
19 Eugene Sverstyuk Str., 5 floor
Kyiv, Ukraine

Tel. 0 800 337 146
E-mail: edu@cbsystematics.com
Site: www.edu.cbsystematics.com | itvdn.com