Функции-генераторы.

Инструкция yield. Методы next(), iter(), send()

https://www.bestprog.net/ru/2021/12/03/python-generator-functions-yield-statement-next-iter-send-methods-ru/

1. Конструкции, воспроизводящие результаты по требованию. Особенности. Понятие отложенной операции

Язык программирования Python имеет средства, позволяющие получать результаты по требованию. К этим средствам относятся:

функции-генераторы;

выражения-генераторы.

Функции-генераторы позволяют генерировать последовательность значений по требованию. Эти функции отличаются от обычных функций тем, что они, вернув значение, могут продолжить свою работу с того места, где они были остановлены. Как известно, обычные функции, вернув значение, прекращают свою работу.

2. Понятие протокола итераций. Методы __next__() и iter(). Поддержка протокола итераций функциями-генераторами

С помощью функций-генераторов или других механизмов (например, циклов for) можно производить обход последовательностей или значений генераторов. Для этого используется так называемый итерационный протокол (протокол итераций). Если протокол не поддерживается, то выполнение цикла for считается неудачным и эта инструкция выполняет операцию индексирования последовательности.

Согласно этому протоколу все итерированные объекты определяют метод __next__(). Этот метод предназначен для возвращения следующего элемента в итерации. Если итерации завершаются, то метод генерирует исключение StopIteration. Для каждого итерированного объекта (строка, список и т.д.) можно получить доступ к итератору этого объекта с помощью вызова встроенной функции iter().

Например, для заданной строки s можно вызвать функцию iter(), чтобы получить итератор следующим образом

```
# Функции-генераторы
# Функция iter()

# 1. Задана строка - это есть итерированный объект
s = "abcde axsadlk sdw"

# 2. Получить итератор объекта s
iterObj = iter(s)

# 3. Использовать итератор для обхода строки s
count_a = 0
for c in iterObj:
    if c=='a':
        count_a = count_a+1

print("count of 'a' = ", count_a) # count of 'a' = 3
```

Так же как цикл for, функции-генераторы поддерживают протокол итераций. Если вызывается функция-генератор, то она возвращает являющийся генератором объект. Этот объект-генератор содержит метод __next__(), создаваемый автоматически. Наличие метода __next__() позволяет продолжить выполнение итераций на следующих шагах.

3. Отличия между обычными функциями и функциями-генераторами. Замораживание состояния в функциях-генераторах

Для обычных функций и функций-генераторов можно выделить следующие общие и отличительные черты:

Оба вида функций создаются с помощью инструкции def.

Обычные функции возвращают значение оператора return. Функции-генераторы возвращают значение оператора yield.
Обычные функции не поддерживают протокол итераций. Функции-генераторы поддерживают этот протокол.
При возврате значения оператором return обычные функции прекращают свою работу.
При возврате значения операторам yield, функции-генераторы автоматически приостанавливают и возобновляют свое выполнение, сохраняя информацию, необходимую для генерирования значений.
Функции-генераторы поставляют значение. Обычные функции возвращают значение.
Наиболее важной особенностью функций-генераторов является то, что они могут приостанавливать свою работу и автоматически сохранять информацию о своем состоянии. Это состояние определяет всю локальную область видимости функции со всеми локальными переменными. При возобновлении работы функции-генератора ранее сохраненное состояние становится доступным для получения следующего сгенерированного значения.
Следовательно, процесс приостановки выполнения функции и сохранения информации о состоянии локальных переменных или параметров называется замораживанием состояния.
С точки зрения использования памяти, функции-генераторы позволяют избегать необходимости выполнять всю работу сразу. Это свойство усиливается, когда список результатов имеет значительный объем или вычисление каждого нового значения занимает длительное время. В этом случае использование функций-генераторов дает более эффективное распределение времени, необходимое для создания всей последовательности значений.

4. Функции-генераторы. Особенности реализации. Инструкция yield. Общая форма
Чтобы функция могла генерировать последовательность значений и замораживать свое состояние, нужно, чтобы эта функция поддерживала протокол итераций. В этом случае функция должна возвращать результат инструкцией yield вместо инструкции return.
Формирование значений, возвращаемых из функции, может производиться одним из двух известных операторов цикла: for или while.
Если значения в теле функции-генератора формируются с помощью цикла for, то примерно общая форма такой функции следующая:

```
def FuncName(list_of_parameters):
    ...
    for value in iterObj:
        ...
        yield value
```

здесь
FuncName – имя функции;
list_of_parameters – список параметров, которые получает функция;
iterObj – итерированный объект. Этот объект может создаваться стандартными средствами, например методом range();
value – элемент из множества значений, сформированных в итерированном объекте iterObj.
Если функция-генератор использует цикл while для генерирования значений, то общая форма такой функции следующая

```
def FuncName(list_of_parameters):
    ...
    while condition:
        ...
        yield return_value
```

здесь
FuncName – имя функции;
list_of_parameters – список параметров, которые получает функция;
condition – условие выполнения цикла while;
return_value – значение, возвращаемое функцией.

5. Примеры функций-генераторов

5.1. Функция, возводящая число в степень 3. Использование цикла for

В примере приводится функция-генератор, возвращающая число в степени 3. Возврат результата происходит с помощью конструкции yield. Такая функция поддерживает протокол итераций.

```
# Функции-генераторы

# Функция, которая возводит число в степень 3.
# Результат возвращается конструкцией yield
def Power3(value):
    for i in range(value):
        yield i*i*i

for i in Power3(10):
    print(i)
```

В цикле for происходит вызов функции Power3(). При каждой итерации в цикле for функция возвращает следующее значение. Это означает, что функция автоматически приостанавливает свою работу и возобновляет свое выполнение на следующей итерации цикла.

Результат выполнения программы

```
0
1
8
27
64
125
216
343
512
729
```

Если вместо инструкции yield в функции использовать инструкцию return то это значит что функция будет обычной и не будет поддерживать протокол итераций. В этом случае, при запуске приложения, будет возникать ошибка типа TypeError

```
TypeError: 'int' object is not iterable
```

5.2. Функция, формирующая последовательность значений с помощью цикла while

Ниже приводится простая функция, формирующая последовательность значений 0, 0.2, 0.4, ..., value

Значения изменяются с шагом 0.2.

```
# Функция-генератор, формирующая числа 0, 0.2, 0.4, ..., value.
# Функция использует цикл while для формирования значений.
def GetFloatValues(value):
    t = 0
    while t<value:
        t = t + 0.2
        yield t

# Использование функции-генератора во внешнем коде
for i in GetFloatValues(2):
    print(i)
```

Результат выполнения программы

```
0.2
0.4
0.6000000000000001
0.8
```

1.0
1.2
1.4
1.5999999999999999
1.7999999999999998
1.9999999999999998
2.1999999999999997

5.3. Функция, генерирующая случайные числа в указанном диапазоне
В примере демонстрируется использование функции-генератора GetRandomValues(), формирующей случайные числа с помощью метода random.randint() из модуля random.
Функция получает параметры:
n – количество случайных чисел, которые нужно сгенерировать;
a, b – соответственно верхняя и нижняя границы диапазона генерируемых чисел.

```
# Функции-генераторы

# Подключить модуль random
import random

# 1. Объявить Функцию-генератор, которая формирует n случайных чисел
# с помощью цикла while в указанном диапазоне [a; b]
def GetRandomValues(a, b, n):
    for t in range(n):
        yield random.randint(a, b)

# 2. Использование функции-генератора во внешнем коде
# 2.1. Ввод входных данных
n = int(input("n = "))
a = int(input("a = "))
b = int(input("b = "))

# 2.2. Сформировать список из n случайных чисел.
L = []
for i in GetRandomValues(a, b, n):
    L = L + [i]

# 2.3. Вывести список
print("L = ", L)
```

Тестовый пример
n = 8
a = 20
b = 40
L = [34, 22, 25, 33, 22, 27, 29, 34]

6. Расширенный протокол функций-генераторов. Передача значения в функцию-генератор. Метод send(). Пример
Кроме метода next() в протокол функций-генераторов добавлен метод send(). Этот метод обеспечивает так называемый расширенный протокол и используется для взаимодействия между функцией-генератором и вызывающей программой. Поддержка расширенного протокола введена начиная с версии 2.5.
При помощи метода send() можно передать значение выражения yield. В этом случае выражение yield размещается в правой части оператора присваивания. Общий вид выражения yield может быть следующим
value = yield return_value
здесь
return_value – значение, возвращаемое оператором yield;

value – новое значение, которое передается инструкции yield. Фактически, инструкция yield является выражением, возвращающим элемент, передаваемый методу send().
В вызывающем коде перед тем как передать значение функции генератору, нужно выполнить следующие три шага:
1. Задать имя, которое будет связано с функцией-генератором, например
FN = FuncName(*list_of_parameters*)
здесь
FuncName – имя функции-генератора;
list_of_parameters – список параметров, которые передаются в функцию-генератор;
FN – имя, которое связано с функцией-генератором.
2. Запустить генератор вызовом метода next()
next(FN)
3. Передать значение в генератор путем вызова метода send()
FN.send(value)
здесь
value – значение, передаваемое в функцию-генератор. Это значение в функции-генераторе может обрабатываться некоторым образом.

7. Пример, демонстрирующий использование метода send() для управления последовательностью получаемых значений из функции-генератора
Условие задачи. Реализовать функцию-генератор, поставляющую *n* случайных чисел. Значение каждого числа находится в пределах от 1 до 10 включительно. Если число размещается на позиции кратной числу 3, то функция-генератор должна возвращать 0. Считать, что позиции чисел нумеруются с 1.
Решение.

```
# Функции-генераторы. Метод send().

import random

# Функция-генератор, которая использует расширенный протокол.
# Эта функция поставляет случайные числа в диапазоне [1, 10].
# Количество чисел равно n.
def FnGen(n):
    for t in range(n):
        # Получить случайное значение
        value = yield random.randint(1, 10)

        # Если в эту функцию передано значение -1, то вернуть 0
        if value==-1:
            yield 20

# 1. Указать количество случайных чисел
n = int(input("n = "))

# 2. Запустить генератор
V = FnGen(n+1)
next(V) # Получить некоторое число - это есть запуск генератора

# 3. Получить n случайных чисел за исключением чисел,
# которые лежат на позициях, кратных числу 3: (3, 6, 9, ...).
# В эти позиции вписать число 0.

# 4. Инициализировать результирующий список чисел
L = []

# 5. Цикл формирования списка
```

```
i = 1
while i<=n:
    if i%3==0:
        item = V.send(-1) # если позиция кратна 3, то передать -1
    else:
        item = next(V)
    L = L + [item]
    i=i+1

# 6. Вывести результат
print(L)
```

Тестовый пример

n = 20

[10, 2, 20, 8, 6, 20, 4, 9, 20, 5, 6, 20, 4, 9, 20, 4, 1, 20, 8, 7]