Генератори

№ уроку: 7 **Курс:** Python Essential

Засоби навчання: PyCharm

Огляд, ціль та призначення уроку

Після завершення уроку учні матимуть уявлення про механізм генераторів, навчаться створювати генератори та найпростіші співпрограми.

Вивчивши матеріал даного заняття, учень зможе:

- Створювати та використовувати генератори
- Використовувати генератори-вирази
- Використовувати підгенератори
- Мати уявлення про yield-вирази

Зміст уроку

1. Що таке генератори?
2. Створення генератора
3. Відмінності між функцією генератора та нормальною функцією
4. Що таке спискові включення?
5. Вираз генератора Python

Резюме

Що таке генератори?
Щоб створити ітератор в Python, іноді доводиться докласти чимало зусиль: визначити методи __iter__() та __next_(), що зберігають інформацію про внутрішній стан, викидати помилку StopIteration, коли елементи контейнера закінчилися.
Це водночас довго та втомлююче. Генератори створені, щоб вирішити такі ситуації.
Генератори – це простий шлях створення ітераторів. Вся робота зі створення ітератора виконуватиметься автоматично генераторами.
Простіше кажучи, генератори – це функція, що повертає об'єкт (ітератор), який ми можемо ітерувати.

Створення генератора
Функції-генератори схожі на звичайні функції у Python. Замість оператора return у генераторах використовується yield.
Якщо функція повертає значення за допомогою yield, Python автоматично створює не звичайну функцію, а генератор.
Різниця return та yield в тому, що оператор return повністю завершує функцію, а оператор yield призупиняє функцію, зберігаючи всі її стани, а потім продовжує звідти при наступних викликах.

Відмінності між функцією генератора та нормальною функцією

1. **Функція генератора** містить один або кілька операторів yield.
2. При виклику **генератор** повертає об'єкт (ітератор), але не починається виконання негайно.
3. Такі методи, як __iter__() та __next_() реалізуються автоматично. Ми можемо перебирати елементи за допомогою next().
4. Як тільки повертається значення за допомогою yield, функція призупиняється, і керування передається стороні, що викликає.
5. Локальні змінні та їх стани запам'ятовуються між викликами.
6. Нарешті, коли функція завершується, StopIteration автоматично викликається під час наступних викликів.

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

Ось приклад, що ілюструє все вищевикладене. У нас є функція генератора з ім'ям fibonacci_generator() та 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
```

Запустимо приклад в інтерактивній консолі:

```
>>> # Отримуємо об’єкт, але не починаємо виконання одразу.
>>> a = my_gen()

>>> # Ми можемо перебирати елементи, використовуючи next ().
>>> next(a)
This is printed first
1
>>> # Коли функція повертає значення, функція призупиняється, і керування передається викликаючій стороні

>>> # Локальні змінні та їх стани запам’ятовуються між викликами.
>>> next(a)
This is printed second
2

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

>>> # Коли функція завершується, StopIteration автоматично викликається під час наступних викликів
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

У наведеному вище прикладі слід зазначити одну цікаву річ: значення змінної n запам'ятовується між кожним викликом.
На відміну від звичайних функцій, локальні змінні не знищуються при виконанні функції. Більш того, об'єкт-генератор може бути повторений лише один раз.
Щоб перезапустити процес, потрібно створити ще один об'єкт-генератор, використовуючи запис a = my_gen().
І останнє, що слід зазначити, це те, що ми можемо використовувати генератори напряму з циклами for.
Це пов'язано з тим, що цикл for приймає ітератор і виконує ітерацію по ньому за допомогою функції next().
Він автоматично завершується, коли викликається StopIteration.
Розглянемо складніший приклад використання ітератора:

```
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)
```

Даний генератор повертає по одному елементу ряду Фібоначчі за раз, при цьому стан циклу for у функції заморожується і відновлюється на наступній ітерації.

Що таке спискові включення?

Спискові включення – це елегантний та лаконічний спосіб створення нового списку з існуючого. Спискове включення складається з виразу, за яким слідує оператор for у квадратних дужках.

Ось приклад створення списку, у якому кожен елемент має ступінь збільшення 2.

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

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

Еквівалентний код без спискового включення:

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

Спискове включення може додатково містити оператори for або if. Необов'язковий оператор if може відфільтровувати елементи для нового списку. Ось кілька прикладів:

```
>>> 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']
```

Вираз генератора Python

Прості генератори можна легко створювати "на льоту" за допомогою **виразів генератора**. Це спрощує створення генераторів.

Синтаксис виразу генератора аналогічний до синтаксису спискового включення. Але квадратні дужки замінені круглими дужками.

Основна різниця між розумінням списку і виразом генератора у тому, що спискове включення створює весь список відразу, тоді як вираз генератора створює за одним елементу за раз.

У генераторів ліниве виконання (виробництво предметів лише за запитом). З цієї причини вираз генератора набагато ефективніший з точки зору пам'яті, ніж еквівалентне спискове включення.

```
# Створення списку
my_list = [1, 3, 6, 10]

# звести кожен елемент у квадрат, використовуючи спискове включення
list_ = [x**2 for x in my_list]

# подібний механізм із генераторами:
# вирази генератора заключені в круглі дужки ()
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>
```

Вище видно, що вираз генератора не відразу дав необхідний результат. Натомість він повернув об'єкт-генератор, який виробляє елементи лише за запитом.
Ось як ми можемо почати отримувати предмети із 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
```

Вирази генератора можна використовувати як аргументи функції. При такому використанні круглі дужки можна опустити:

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

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

Є кілька причин, через які генератори є вигідною реалізацією:

1. Легко реалізувати та читати код
2. Ефективна робота з пам'яттю
3. Створення нескінченної послідовності

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

4. Створення вкладених генераторів

```
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))))
```

Закріплення матеріалу

- Що таке генератор?
- Що таке вираз-генератор?
- Що є результатом yield-виразу?

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

- Які методи є у генераторів?
- Що таке спискові включення?
- У яких випадках краще використати генератори?

Додаткове завдання

Завдання
Напишіть функцію-генератор для отримання n перших простих чисел.

Самостійна діяльність учня

Завдання 1
Напишіть генератор, який повертає елементи заданого списку у зворотному порядку (аналог reversed).

Завдання 2
Виведіть із списку чисел список квадратів парних чисел. Використовуйте 2 варіанти рішення: генератор та цикл

Рекомендовані ресурси

Документація 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/

Статті у Вікіпедії про ключові поняття, розглянуті на цьому уроці
https://en.wikipedia.org/wiki/Generator_(computer_programming)
https://uk.wikipedia.org/wiki/Співпрограма

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