Програмування

ТЕМА 16. ІТЕРАТОРИ ТА ГЕНЕРАТОРИ

Ітератори

Ми вже неодноразово зустрічались з об’єктами складених типів даних, які допускають перебір всіх елементів за допомогою циклу for … in … .

Ці об’єкти ми називали такими, що ітеруються.

Раніше це були об’єкти стандартних типів Python.

Але є можливість будувати і свої власні об’єкти та навіть надавати такої поведінки (здатності до перебору всіх елементів) раніше побудованим типам.

Для цього використовують ітератори.

Нехай *x* – об’єкт типу, що ітерується (це, зокрема, означає, що *x* складається з декількох елементів).

Тоді **ітератором** *y* називається об’єкт, який здатен повертати по черзі всі елементи *x* у деякому порядку та фіксувати момент завершення елементів *x*.

У такому випадку кажуть, що ітератор *y* підтримує ітераційний протокол.

Функції next() та iter()

Ітераційний протокол полягає у наступному.

- Для повернення елементів *x* ітератор *y* використовує стандартну функцію next(y).
- Коли елементи *x* завершуються, ітератор у відповідь на черговий виклик next ініціює виключення класу StopIteration.

Для того, щоб для *x* повернути об’єкт-ітератор, використовують стандартну функцію iter(x).

Після цього послідовно викликаючи next, можна отримати всі елементи *x*.

Функції next() та iter().2

Цикл for … in … також використовує ітераційний протокол.

Цей цикл рівносильний такій інструкції:

```
for a in x:     ≡        y = iter(x)
    P                    try:
                             while True
                                 a = next(y)
                                 P
                         except StopIteration:
                             pass
```

Власні класи-ітератори

Для реалізації власних ітераторів треба описати клас, який реалізує методи __iter__ та __next__.

При виклику функції iter(x) Python буде викликати метод __iter__: x.__iter__().

Як правило, цей метод повертає у якості ітератора самого себе, тобто, об’єкт цього класу.

При виклику функції next(x) Python викликає метод __next__: x.__next__().

Метод __next__ повинен повертати елементи та ініціювати виключення StopIteration, якщо елементи завершились.

Приклад

Клас-ітератор Reverse, що повертає елементи послідовності в оберненому порядку.

Генератори

Генератори – це об’єкти, які створюють послідовність поелементно та повертають по одному елементу послідовності за один крок.

Генератори схожі на ітератори.

Але якщо ітератор повертає елементи вже існуючої послідовності, то генератор цю послідовність створює.

Генератори у Python синтаксично можуть бути реалізовані як **генератори-вирази** або як г**енератори-функції**.

Генератори-вирази

Синтаксис генератора-виразу виглядає так:

(*e*(a) for a in *x* if *F*)

- де *e*(a) – вираз, що залежить від a, *x* – вираз типу, що ітерується, *F* – умова.

Python повертає значення для всіх елементів a з *x*, для яких істинною є умова *F*.

Умову *F* можна не вказувати. Тоді if *F* опускають.

Генератор-вираз схожий на спискоутворення.

Але окрім різних дужок у синтаксисі суттєвою відмінністю є те, що генератор-вираз не будує всю послідовність, а повертає її поелементно.

Генератори-функції

Генератори-функції мають такий же синтаксис, як і звичайні функції, за виключенням того, що для повернення результату замість оператора

return e

використовують

yield e

З точки зору виконання функції yield відрізняється від return тим, що не тільки повертає значення виразу e, але й запам’ятовує стан функції (місце завершення, значення всіх локальних змінних).

При наступному зверненні до генератора-функції за допомогою next її виконання починається з наступного оператора після yield.

Тобто, при багатьох зверненнях до генератора-функції управління виконанням програми перемикається від програми до генератора-функції і назад, аналогічно тому, як це відбувається при паралельному виконанні програм.

Тому генератори-функції ще називають «паралельним програмуванням для бідних».

Приклад

Отримати всі числа Фібоначчі в діапазоні від 1 до n. Використати генератор-функцію.

Застосування генераторів

Генератори не дають якихось унікальних переваг у порівнянні з іншими засобами роботи з послідовностями, окрім випадків, коли треба обробляти великі обсяги даних і коли, можливо, з великого масиву даних достатньо отримати кілька елементів.

Тому їх застосування є доцільним саме при наявності послідовностей великого розміру.

Варто зазначити, що результати раніше розглянутих функцій zip() та map() є саме об’єктами генераторами

Приклад

Побудувати всі перестановки з n заданих елементів.

Реалізація ітераторів у класах

Один із способів побудови власних ітераторів – це додавання підтримки ітераційного протоколу у існуючих класах або їх нащадках.

Інший спосіб – написання окремих класів ітераторів

Для цього необхідно реалізувати у класі методи

- __iter__ та __next__
- або навіть один метод __iter__, який повертає об’єкт, що підтримує ітераційний протокол.

Наприклад, генератор-функцію.

Приклад.

Побудувати ітератор, який повертає усі одночлени поліному у порядку спадання степенів

Резюме

Ми розглянули:

1. Ітератори, функції iter та next
2. Ітераційний протокол
3. Генератори
4. Генератори-вирази та генератори функції
5. Написання власних класів-ітераторів

Де прочитати

1. Обвінцев О.В. Інформатика та програмування. Курс на основі Python. Матеріали лекцій. – К., Основа, 2017
2. Марк Лутц, Изучаем Python, 4-е издание, 2010, Символ-Плюс
3. Python 3.4.3 documentation
4. Марк Саммерфилд, Программирование на Python 3. Подробное руководство. - Символ-Плюс, 2009.
5. Bruno R. Preiss, Data Structures and Algorithms with Object-Oriented Design Patterns in Python, 2003, http://www.brpreiss.com/books/opus7/
6. Tarek Ziadé. Expert Python Programming. - Packt Publishing, 2008.
7. David Beazley and Brian K. Jones, Python Cookbook. - O'Reilly Media, 2013.
8. http://www.python-course.eu/python3_generators.php