Ітератори

При роботі зі списком, можна переглянути його елементи один за одним. Перегляд одного елементу називається ітерацією:

```
>>> lst1 = [1, 2, 3]
>>> for i in lst1:
...     print(i)
1
2
3
```

В даному випадку, lst1 це ітерований об’єкт. При створенні списку за допомогою спискових виразів, цей список також буде ітерованим об’єктом.

```
>>> lst2 = [x*x for x in range(3)]
>>> for i in lst2:
...     print(i)
0
1
4
```

Все, до чого можна застосувати for … in … : є ітерованими об’єктами: lists, strings, files, dict, set…

Тип даних, який можна ітерувати - це такий тип, який може повертати свої елементи по одному.

Об’єкт, який має метод __iter__() вважається таким що ітерується і може надавати ітератор.

Ітератор - це об’єкт, який має метод __next__(), який при кожному виклику повертає черговий елемент і викликає виняток StopIteration після закінчення всіх елементів.

Для того щоб це все побачити можна скористайтися функціями dir() та type()

```
>>> lst1 = [2,3,1,4,5]
>>> dir(lst1)
>>> lst1_iter = iter(lst1)
>>> dir(lst1_iter)
>>> type(lst1_iter)
<class 'list_iterator'>
```

Вбудована функція iter() дозволяє отримати ітератор але її можна використовувати двома різними способами. Якщо її застосовувати до послідовностей або колекцій даних то вона повертає ітератор для заданого об’єкту або викликає виняток TypeErrror, якщо об’єкт не ітерується. Цей спосіб часто використовується для роботи з нестандартними типами колекцій.

```
lst_iter = iter([1,2,3,4])
type(lst_iter)
<class 'list_iterator'>
```

В другому способі використання функції iter() їй передається об’єкт (функція або метод) і спеціальне значення (сторож). Функція або метод будуть викликатися на кожній ітерації, а значення цієї функції, якщо воно не дорівнює спеціальному значенню, буде повернуто програмі з якої здійснювався виклик; в іншому випадку буде викликано виняток StopIteration.

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

При використанні циклу for item in iterable, інтерпретатор Python викликає функцію iter(iterable), щоб отримати ітератор. Після цього на кожній ітерації здійснюється виклик методу __next__() ітератора, для того щоб отримати черговий елемент, а коли виникає виняток StopIteration, виконання циклу завершується .

Черговий елемент ітератора можна отримати через виклик вбудованої функції next().

```
>>> product = 1
>>> i = iter([1, 2, 4, 8])
>>> while True:
```

```
        try:
            product *= next(i)
        except StopIteration:
            break

>>> print(product)
64
```

```
>>> product = 1
>>> for i in [1, 2, 4, 8]:
        product *= i

>>> print(product)
64
```

Будь який (скінченний) ітерований об’єкт може бути перетворений в кортеж за допомогою функції tuple() або в список – за допомогою функції list().

Ітерації зручно використовувати, тому що, можна прочитати, переглянути, обробити дані багато разів, але ці дані зберігаються в пам’яті і це призводить до зайвих витрат ресурсів.

Генератори

Генератори це ті ж самі ітератори, але дані можна проітерувати лише один раз. Це тому, що вони не зберігаються в пам’яті, а поступово генеруються (на льоту):

```
>>> gen1 = (x*x for x in range(3))
>>> for i in gen1:
...     print(i)
0
1
4
>>> for i in gen1:
...     print(i)

>>>
```

При створенні виразів генераторів використовують такий же синтаксис, як і для створення спискових виразів, але замість [ ]. дужок використовуються : ( ) Виконати for i in gen1 вдруге не можна, оскільки генератор може бути використаний лише раз. В наведеному прикладі він обчислює 0, тоді видаляє його з пам’яті і обчислює 1, видаляє його, обчислює 4 і також його видаляє, один за одним.

Синтаксис в загальному можна описати наступним чином:

- (expression for item in iterable)
- (expression for item in iterable if condition)

Yield

Yield це ключове слово, що використовується подібно до return, однак замість функції це буде функція - генератор.

```
>>> def create_generator():
...     lst1 = range(3)
...     for i in lst1:
...         yield i*i
...
>>> gen2 = create_generator() # створюємо генератор
>>> print(gen2) # gen2 - об'єкт!
<generator object create_generator at 0xb7555c34>
>>> for i in gen2:
...      print(i)
0
1
4
```

Доцільність використання цього прикладу не надто велика, але він чудово демонструє зручність використання генератора, коли необхідно лише прочитати дані, які повертає функція.

Функція-генератор, або метод-генератор - це функція, або метод, яка містить вираз yield. При звертанні до функції-генератора буде повернуто ітератор.

Щоб опанувати yield, потрібно зрозуміти (запам'ятати), що, коли відбувається виклик такої функції, код, написаний в тілі функції не виконується. Функція лише повертає об'єкт генератора.

Код буде викликатися (виконуватися) кожного разу, коли for буде звертатися до генератора. Значення з ітератора добуваються по одному, за допомогою методу __next__()

При кожному виклику методу __next__() він повертає результат обчислення виразу yield. (Якщо вираз відсутній то повертає значення None.) При завершенні виконання функції-генератора або виконання інструкції return, викликається виняток StopIteration.

Коли for перший раз викликає об'єкт генератора з функції, вона виконає код функції від початку і до слова yield - тоді поверне перше значення ітерації. При кожному наступному виклику буде відбуватися ще одна ітерація циклу і буде повертатися наступне значення. І так поки не закінчаться значення.

Генератор вважається порожнім, якщо при виконанні коду не зустрічається yield. Причиною може бути кінець циклу, або не виконання умови if … else

Приклад еквівалентних фрагментів програмного коду, які демонструють, як простий цикл for ... in, з yield, можна перетворити в генератор:

```
def items_in_key_order(d):
    for key in sorted(d):
        yield key, d[key]
```

```
def items_in_key_order(d):
    return ((key, d[key]) for key in sorted(d))
```

Приклади генераторів

```
def build_counter(num): # генерує значення по одному за раз
    print('build_counter started')
    while True:
        yield num # функція виконується до yield і повертає значення
        print('incrementing num')
        num = num + 1
>>> counter = build_counter(5) # виклик функції не виводить нічого на екран
>>> type(counter) # функція повертає генератор
<class 'generator'>
>>> counter
<generator object build_counter at 0x127CD490> # генератор це об'єкт
>>> next(counter)
build_counter started
5
>>> next(counter) # наступний виклик функція працює з того місця де зупинилась
incrementing num
6
>>> help(next) # Return the next item
```

```
def letter_range(a, z):
        rez = []
        while ord(a) < ord(z):
                rez.append(a)
                a = chr(ord(a) + 1)
        return rez
```

```
def letter_range(a, z):
        while ord(a) < ord(z):
            yield a
            a = chr(ord(a) + 1)
```