EPAM Python Software Engineer Training

Lesson 2: Python iterators

Contents

1 Course

1.1 Iterators

Iterator is and essential entity in Python. In general and iterator is an object implementing a method *next* which either returns something or raises a *StopIteration* error on each call to this method. An object which implements a method *__iter__* returning an iterator is called an *Iterable*. All collections in Python are iterables; a *for* loop is internally implemented as a call to an *__iter__* method followed by a sequence of calls to a *next* method in a *while* loop until a *StopIteration* error is raised. Note that for a *for* loop one should use an *Iterable*, not an *Iterator*, but the latter is created during a loop internally.

> ***Note***
>
> In Python 3.x a *next* method was renamed to *__next__* as it should have been done from the very beginning.

For many tasks it is enough to use standard Python iterators and it is rarely needed to declare some custom iterator. To create your own custom iterator it is enough to declare a class which implements a *next* method. To create a custom iterable in most cases it is enough to declare an iterator class which additionally implements an *__iter__* method as simple as returning a *self*.

1.2 Generators

In Python a generator is an iterator which additionally implements a *send* method used to pass some data into a generator. In theory it can be considered as a bidirectional iterator. It can be used to either produce data, consume data, or in duplex mode.

There are quite few standard generators in Python, as for most cases a custom behavior is required. A custom generator can be created by declaring a class which defines all of *__iter__*, *next*, and *send* methods.

However, a much easier way to achieve the same is to declare a function which uses a *yield* keyword to generate values. At each occurence of *yield* a context is switched between a function execution context and a context of a *next* or *send* method caller (in most cases *next* is implemented as *send(None)*). An argument of a *send* method is passed into a result of a *yield* statement. These peculiarities can be used to build complex relationships between a generator and its caller.

For example, a below code will print numbers `1, 3, 5, 7, 9`.

```
def gen():
    x = 0
    while True:
        x = yield x + 1

g = gen()
x = g.next()
while x < 10:
    print x
    x = g.send(x + 1)
```

1.3 Generator Expressions

A more convenient way to declare generators for simple cases is to use generator expressions. This is a syntax sugar to wrap a for loop into parenthesis (e.g. below code will return a generator yielding values `1, 2, 3`):

```
(x for x in (1, 2, 3))
```

It is possible to add conditions into generator expressions (e.g. below code will produce a generator returning odd numbers smaller than 10):

```
(x for x in xrange(10) if x % 2)
```

1.4 List Comprehensions

A list comprehension is a syntax sugar to create a list by wrapping a for loop into brackets. It is quite similar to generator expression but produces a list (e.g. below code will produce a list of even numbers smaller than 10):

```
[x for x in xrange(10) if not x % 2]
```

A list comprehension is equivalent to passing a generator expression into a list constructor. Hence, generator expressions are faster and use less memory, but a generator can be used only once to iterate over a sequence. In cases when more than one iteration is necessary - list comprehension is necessary.

1.5 Itertools

Many useful routines operating with iterators can be found in an *itertools* built-in module.

2 Tasks

2.1 Factorial

Write a non-recursive function calculating a factorial of a number using an *xrange* function.

2.2 Multiplier

Create a function returning a list of all numbers `N` smaller than input integer `M` such that `N` is a multiplier of `3` while `N + 1` is a multiplier of `5`. Use it to print all such numbers smaller than `100`.

> ***Hint***
>
> Use an *xrange* function and a list comprehension to solve this task.

2.3 Fibonacci

Write a generator which produces an infinite sequence of Fibonacci numbers. Use it to print first 100 Fibonacci numbers followed by every tenth such number between 100th and 1000th (e.g. 100th, 110th, 120th and so on).

> ***Hint***
>
> Use *enumerate* to solve the second part of a task. Think of a way to form a finite loop off an infinite generator (e.g. use some of *itertools* module functions to achieve this).

2.4 Itertools

Using *itertools* write an function returning an iterator over the following tuples of numbers:

- the first number shall be equal to the first function argument;
- the second number shall start with the second function argument and increase by the first function argument after each turn;
- the third number shall loop between the first and the third argument forth and back several times that is equal to the second argument;

For example, the following call `func(1, 2, 3)` shall have the following output:

```
1, 2, 1
1, 3, 2
1, 4, 3
1, 5, 2
1, 6, 1
1, 7, 2
```

```
1, 8, 3
1, 9, 2
1, 10, 1
```

2.5 Flatten

Write an iterator which takes an arbitrary number of iterables and flattens their output (i.e. iterates over their elements returning one element from each iterable in a loop). For example, a return of these two iterables: `A, B, C`, `D, E, F` - shall be `A, D, B, E, C, F`. An iterator shall end when all of iterables are exhausted.

2.6 Consumer

Write a generator that consumes lines of a text and prints them to standard output. Use this generator and a *flatten* function from the previous task to print contents of two different files to a screen pseudo-simultaneously.

2.7 Tail

Write a program that prints `N` last lines of a file in reverse order (just like a `tail -r` FreeBSD command). Both file name and a number of lines to print shall be passed as command-line arguments.

Think of a memory-efficient yet fast way to implement this task.