title: "Iterators and Generators in PYTHON" author: "El Hadij Matar NDIAYE & Sandra Yawoa ATOPIA" date: " `r Sys.Date()` " output: html_document editor_options:

chunk_output_type: inline

ITERATOR AND GENERATOR IN PYTHON

Python is a clean and powerful programming language. You can use it to create websites, analyze data, write utilities, and create many other types of software. In this article, we are not going to talk about python as such but rather introduce you to the powerful tools that the programming language offers us, namely iterators and generators.

What are the Iterators and generators in python?

Iterators are primarily used to iterate through other objects or convert them to iterators using the iter() method. Generators are commonly used in loops to create an iterator by returning all the data without impacting the loop's iteration. Iterator makes use of the iter() and next() methods. The generator uses the yield keyword.

Iterator

What is iterator ?

An iterator can be understood as a kind of cursor that moves through a sequence of objects. Iterating over an object in programming consists of traversing this object attribute by attribute in order to access their value. It is implicitly used each time we manipulate data collections such as: lists, tuples or even strings. Usually we use a `for loop` to loop through a collection. But, what is actually happening is that the loop is calling an `iter()` function on that iterable object.

First let's see an example of iteration with the for loop.

```
 {python}
myList = [x for x in range(1, 21)]
for x in myList :
    if x %2==0 :
        print(x, ' est un nombre pair !')
    else :
        print(x, ' est un nombre impair !')
```

In this example we go through the list of numbers between 1 and 20 and we do a little check to determine if the number is even or odd.

How to use iterator ?

We will now see how to use the `iter()` method that the for loop uses under the hood.

But before that, let's see **why to use an iterator**.

The iterator brings a higher level of abstraction, we add an extra layer of code to perform an action. Moreover, the iterator is an inexpensive object in terms of memory usage.

Let's see the `iter()` and `next()` functions:

The `iter()` function allows you to obtain an iterator from an object. Concretely, this function calls the `__iter__()` method of the object passed as a parameter.

The `next()` function expects an iterator as a parameter and returns the next element. If the iterator is already positioned on the last element, this function throws an exception of type StopIteration. Concretely, this function calls the `__next__()` method of the iterator passed as a parameter.

Let's find an example to illustrate this

```
 {python}
myIter = iter(range(5))

print(next(myIter))
# affiche 0
print(next(myIter))
# affiche 1
print(next(myIter))
# affiche 2
print(next(myIter))
# affiche 3
print(next(myIter))
# affiche 4
print(next(myIter))
# génère une erreur StopIteration car l'itérateur est déjà positionné sur le dernier élément donc il n
```

What is the difference between iterable and iterator in Python?

Iterable is a type of object that can be iterated over. Iterators are used to iterate through iterable objects using the `__next__()` function. Iterators have a `__next__()` function that returns the object's next item. It is important to note that every iterator is also iterable, but not every iterable is an iterator.

A list comprehension returns an iterable. It means that you can iterate over the result of a list comprehension again and again.

However, a generator expression returns an iterator, specifically a lazy iterator. It becomes exhausted when you complete iterating over it.

Iterator made with a Class

Building an iterator from scratch in Python is an easy thing in Python. All you have to do is implement the `__iter__()` and `__next__()` methods.

The `__iter__()` method returns the iterator object itself. If necessary, an initialization can be performed.

The `__next__()` method will return the next element in the sequence. Once the end is reached, on subsequent calls it should execute the StopIteration.

Example

Here we show an example that will give us the next power of 2 at each iteration. Exponents start from zero up to a user-defined number.

```
{python}
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max=0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration


# creating an object
numbers = PowTwo(3)

# creation of an iterable from the object
i = iter(numbers)

# Using next to get the next iterator element
print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

The itertools module

Itertools is a module in Python for iterating over data structures that can be iterated over using a for loop. These data structures are also called iterable objects. This module serves as a fast and memory-efficient tool for training the algebra of iterators alone or in combination.

Why to use this module ?

This module integrates functions for efficient use of computer resources. The use of this module also tends to improve the readability and maintainability of the code.

We have three categories in the itertools module :

- infinite iterators : loops that continue indefinitely,
- Iterators ending on the shortest input sequence : to do preprocessing,
- combinatorial iterator : facilitates calculations with combinations

Example of some methods of the itertools module :

1. the group function:

   The group() function can be found in the Recipes section of the itertools documentation. The recipes are a great source of inspiration for using itertools to your advantage.

example :

```
 {python}
import itertools as it


# here we define the grouper function
def grouper(inputs, n, fillvalue = None):
    iters = [iter(inputs)] * n
    return it.zip_longest(*iters, fillvalue = fillvalue)


alpha = ['s', 'a', 'n', 'd', 'r', 'a', 'a',
        'n', 'd', 'm', 'a', 't', 'a', 'r']
print(list(grouper(alpha, 3)))
```

2. Brute Force Scenario

Brute force is a simple method of solving a problem that relies on pure computing power and trying all possibilities rather than advanced techniques to improve efficiency. There are various itertools Brute force functions such as:

- combinations()
- combinations with replacement()
- permutations()

combinations : The itertools.combinations() function takes two arguments : - an iterable input and a positive integer n - and iterates over the tuples of all n-element combinations in the inputs.

example :

```
 {pyhon}
# Python code to demonstrate combinations
import itertools as it


print(list(it.combinations([1, 2], 2)))
```

combinaisons_with_replacement() : combinations_with_replacement() works exactly like combinations(), accepting an iterable input and a positive integer n, and returns an iterator over n-tuples of elements from the inputs. The difference is that combinations_with_replacement() allows elements to be repeated in the tuples it returns.

example :

```
 {python}
# Python code to demonstrate combinations_with_replacement
import itertools as it


print(list(it.combinations_with_replacement([1, 2], 2)))
```

The output of the first example is : [(1, 2)], and the output of the second one is : [(1, 1), (1, 2), (2, 2)]

Here, we can clearly see the difference between the two. this difference is in the fact that the second allows repetition in the tuple

permutations : a permutation is a collection or combination of objects in a set where the order or arrangement of the chosen objects matters. permutations() accepts a single iterable and produces all possible permutations (rearrangements) of its elements.

example :

```
 {python}
# Python code to demonstrate permutations
import itertools as it

print(list(it.permutations(['g', 'o', 'a', 't'])))
```

The output is :

```
 {python}
[('g', 'o', 'a', 't'), ('g', 'o', 't', 'a'), ('g', 'a', 'o', 't'), ('g', 'a', 't', 'o'), ('g', 't', 'o
```

Generator

What is a generator ?

Creating iterators can sometimes be quite complicated to do. The usefulness of generators is therefore to allow us to create iterators more easily. A generator can be seen as a function that behaves similarly to iterators. It generates the elements you loop over : it's an on-demand iterable.

How to get a generator ?

Any function that uses `yield` instead of `return` is a generator function or just a generator.

`Yield` allows the generator to pause but its execution context will be preserved, which is what will allow us to iterate. Each time a generator is called with the `next()` function, the generator will resume its work until it encounters `yield` again.

When defining a generator, the `iter()` and `next()` methods are automatically created. Also, generators automatically throw a `StopIteration` exception when they complete their execution.

Take the following example

```
{python}
def squareSeqNumber(n):
    for i in range(1, n+1):
        yield i*i

if __name__ == "__main__" :
    a = squareSeqNumber(5)

    print("Les carrés des nombres sont : ")
    print(next(a))
    # affiche 1
    print(next(a))
    # affiche 4
    print(next(a))
    # affiche 9
    print(next(a))
    # affiche 16
    print(next(a))
    # affiche 25
    print(next(a))
    #génère une erreur StopIteration car l'itérateur est déjà positionné sur le dernier élément donc i
```

What is the purpose of the yield statement?

A `yield` statement is similar to a `return` statement in its simplest form, except that instead of halting execution of the function and returning, yield instead delivers a value to the code looping over the generator and stops execution of the generator function.

Generator expression

A generator expression is an expression that returns a generator object.

Basically, a generator function is a function that contains a yield statement and returns a generator object.

For example the following defines a generator function:

```
{python}
def squares(length):
    for n in range(length):
        yield n ** 2
```

The `squares` generator function returns a generator object that produces square numbers of integers from `0` to `length - 1`.

Because a generator object is an iterator, you can use a for loop to iterate over its elements:

```
{python}
for square in squares(5):
    print(square)
```

A generator expression provides you with a more simple way to return a generator object.

The following example defines a generator expression that returns square numbers of integers from 0 to 4:

```
 {python}
squares = (n** 2 for n in range(5))
```

As you can see, instead of using a function to define a generator function, you can use a generator expression.

A generator expression is like a list comprehension in terms of syntax. For example, a generator expression also supports complex syntaxes including:

- if statements
- Multiple nested loops
- Nested comprehensions

However, a generator expression uses the parentheses () instead of square brackets []

Generator expressions vs list comprehensions

The following shows how to use the list comprehension to generate square numbers from 0 to 4:

```


{python}
square_list = [n** 2 for n in range(5)]
```

And this defines a square number generator:

```
 {python}
square_generator = (n** 2 for n in range(5))
```

1. ***Syntax:***

In terms of syntax, a generator expression uses square brackets [] while a list comprehension uses parentheses ().

2. ***Memory utilization:***

A list comprehension returns a list while a generator expression returns a generator object.

It means that a list comprehension returns a complete list of elements upfront. However, a generator expression returns a list of elements, one at a time, based on request.

A list comprehension is eager while a generator expression is lazy.

In other words, a list comprehension creates all elements right away and loads all of them into the memory.

Conversely, a generator expression creates a single element based on request. It loads only one single element to the memory.

3. ***Iterable vs iterator:***

A list comprehension returns an iterable. It means that you can iterate over the result of a list comprehension again and again.

However, a generator expression returns an iterator, specifically a lazy iterator. It becomes exhausted when you complete iterating over it.

Generator made with a class

There's a lot of work involved in building an iterator in Python. We need to implement a class with `__iter__()` and a `__next__()` method, keep track of internal states, and rerun StopIteration when there are no values to return.

It's both long and counter-intuitive. The generator comes to the rescue in such situations.

Python generators are a simple way to create iterators. All the work we mentioned above is automatically handled by generators in Python.

Simply put, a generator is a function that returns an object (iterator) that we can iterate over (one value at a time).

Example :

The iterator example program made with a class was long and confusing. Now let's do the same using a generator function

```
{python}
def PowTwoGen(max=0):
    n = 0
    while n < max:
        yield 2 ** n
        n +=
```

Since the generators automatically keep track of the details, the implementation was concise and much cleaner.

Difference between Iterators and generators

ITERATOR	GENERATOR
HeaderClass is used to implement an iterator	Function is used to implement a generator.
Local Variables aren't used here	All the local variables before the yield function are stored
Iterator uses iter() and next() functions	Generator uses yield keyword
Every iterator is not a generator	Every generator is an iterator
Iterators are the objects that use the next() method to get the next value of the sequence.	A generator is a function that produces or yields a sequence of values using a yield statement
Iterators in python are less memory efficient	Generators in Python are more memory efficient
Iterators are used mostly to iterate or convert other objects to an iterator using iter() function.	Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop

Example of use of iterators and generators in the data science domain

People working in machine learning and AI rely heavily on iterators and generators in Python. They are a basic concept of the language and are used a lot in machine learning packages such as scikit-learn or Keras. Understanding how they work and why Pythonistas use them gives you two advantages : - You can use them more efficiently and effectively. - You can understand the issues related to them faster. This understanding will save you a lot of time both writing code and debugging it.

Examples

Here we will try to illustrate its use through two example implementations of scikit-learn

1. Example 1: KFold

One of the main challenges in machine learning is to avoid overfitting. Many algorithms have hyper-parameters or modifications to solve this problem, but the most important concept is cross-validation.

Let's see K-Folds cross-validation. The idea is to split a dataset into folds and use each fold once to test the model that was trained on the remaining folds. For example, if you use three folds, the model trains on two of them and uses one for testing.

The reason for using a generator here is that you don't have to remember all the clue combinations from the start. You just need to know which clues should be tested in the respective round. The rest will be used for training.

Let's take a look at the source code (lines 357-434) of scikit-learn for KFold to see how the generator was implemented:

```
 {pyhton}
class KFold(...):
    def __init__(...):
        ...
    def _iter_test_indices(...):
        ...
        current = 0
        for fold_size in fold_sizes:
            start, stop = current, current + fold_size
            yield indices[ start:stop]
            courant = stop
```

There is a `yield` statement near the end of the class definition, indicating a generator function. The method name `_iter_test_indices` suggests that the generator only produces the indices for the test bend. As described earlier, the training will be done on all the other indices, so there is no need to calculate them explicitly.

The first time the generator is called, it starts with index 0 (current=0) and adds the ply size to get the last index of that ply. He then gives these clues. Next time it looks up its local variables and sets current to the last iteration's stop value before executing another round of the for loop.

2. Example 2: GradientBoostingClassifier

The basic idea behind gradient boosting is to combine enough weak learners to get a strong model. Unlike bagging approaches that train many learners in parallel, such as random forest, boosting works iteratively. He trains a weak learner, assesses his main weaknesses and tries to compensate for them in the next round.

The iterative character of this algorithm calls the generators. Let's check the source code of GradientBoostingClassifier, specifically line 2151.

```
 {python}
def staged_predict_proba(...):
    ...
    for score in self._staged_decision_function(X):
          yield self.loss_._score_to_proba(score)
```

A gradient boosting step is a step in the iterative process. This generator monitors the error on the test set at each step. The reason for using a generator here is that you don't have to define how many steps there are in a particular instance of this model.

CONCLUSION

As we have just seen, the operation of iterators and generators is elementary and not at all complex.

On the other hand, if a beginner can very well do without knowing them, it is not the case for experienced developers.

Indeed, this Python concept will allow them to deeply optimize their code and will be essential in some cases.

On the module side, itertools is a great classic whose name and possibilities you should at least know. Especially since in some algorithms, it will quickly prove to be indispensable. Not to mention that, as we saw a little while ago, the code becomes even more readable.