PES UNIVERSITY

Department of Computer Science and Engineering (UG Studies)

PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Iterators in Python

Introduction

Iterator simply is an object that can be iterated upon. It allows programmers to access or traverse through all the elements of the collection without any deeper understanding of its structure.

Python iterators implement iterator protocol which consists of two special methods `__iter__()` and `__next__()`.

The `__iter__()` method returns an iterator object whereas **`__next__()` method returns the next element from the sequence.**

How for loops actually work?

Let's take a list and iterate through it.

```
x = ['Hey','there','Python','programmers']
for i in x:
   print(i)
```

Output:

```
Hey
there
Python
programmers
```

That's the simple program we already have learned to code. But have you ever tried and dig deeper into the underlying mechanism behind such iteration.

So basically, **the process of the for loop going through each element is called** iteration and **the object x through which the for loop is iterating is called** iterable**.**

What actually is happening here?

Well, behind the scenes actually the loop is using a built-in function called **`__iter__()`** to go through all the elements one by one and the **`__next__()`** function is used for the next element in the collection.

Let's use a Python built-in function `dir()` to find out all the associated attributes of the iterable x.

```
x = ['Hey','there','Python','programmers']

print(dir(x))
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

There is the `__iter__()` method working behind the scene for the iteration. But `__iter__()`(callable as *iter(container_object)*) alone cannot iterate through all items as we need to move to next item for iteration. So `__next__()` (callable as *next(iterator_object)*) function is used for that.

So, here is how things actually work behind the iteration in for loop or any iterable in Python.

```
obj = iter(x) #using iter function for x

print(next(obj)) #Iteration 1 using next function
'Hey'

print(next(obj)) #Iteration 2
'there'

print(next(obj)) #Iteration 3
'Python'

print(next(obj)) #Iteration 4
'programmers'

print(next(obj)) #Iteration 5
Traceback (most recent call last):
  ...
StopIteration
```

Note: `obj` is the iterator returned by the function `__iter__()`.
So that's the action behind the for loop in Python where special functions `__iter__()` and `__next__()` are internally called for iteration.

Notice in 5th iteration the `__next__()` function raises an exception called StopIteration because there is no any item left to iterate through. Hence the for loop ends there.

We can summarize above process in following points.

- iterable x has function __iter__() as we saw using dir()
- __iter__() functions returns an iterator object called obj
- using iterator obj and __next__() function we traverse through all the items in the list
- once there are no items left to iterate through, the function __next__() raises an exception StopIteration and the iteration ends there.

Now that we have known about iterators in Python, let's learn how to create our own Python iterator.

Creating our own Iterator in Python

Building our own Iterator is nothing different than what we explained above. We use the same __iter__() and __next__() functions.

But this time we will define these special functions inside a class as we need.

Example to create our own Python Iterator

Here is an example to build our own iterator to display odd number from 1 to the max number supplied as the argument.

```
class OddNum:
  """Class to implement iterator protocol"""

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

  def __iter__(self):
    self.x = 1
    return self

  def __next__(self):
    if self.x <= self.num:
      odd_num = self.x
      self.x += 2
      return odd_num
    else:
      raise StopIteration
```

Now we can use directly use for loop or use __iter__() and __next__().

Using for loop

```
obj = OddNum(10)
for num in obj:
    print(num)
```

```
1
3
5
7
9
```

The for statement will call `iter(obj)`. This call is changed to `OddNum.__iter__(obj)`.

Using __iter__() and __next__()

```
>>> obj = OddNum(10)
>>> i = iter(obj)       # i = OddNum.__iter__(obj)
>>> next(i)             # OddNum.__next__(i)
1
>>> next(i)
3
```

```
>>> next(i)
5
>>> next(i)
7
>>> next(i)
9
>>> next(i)
Traceback (most recent call last):
 ...
StopIteration
```

Creating our own infinite iterator

An infinite iterator never stops itself because we don’t set limit in it and it does not raise an exception.

In infinite iterator, the user can impose a condition to stop the infinite iteration later in the program as per need using break statement and others.

Example of infinite iterator

Let’s again take the example we used above but this time we won’t set a max limit to display odd numbers. Instead, we will use a break condition to exit out of iteration in for loop.

```
class OddNum:
  """Class to implement iterator protocol"""

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

  def __iter__(self):
    self.x = 1
    return self

  def __next__(self):
    odd_num = self.x
    self.x += 2
    return odd_num

for i in OddNum():
  if i < 16:
    print (i)
  else:
    break
```

As you can see in above program, we didn’t set a limit in `__next__()` function, instead, we used a condition in for loop later to prevent from infinite iterations and jump out of it.

So, the output is

```
1
3
5
7
9
11
```

```
13
15
```

Some more examples:

In an examination hall, there could be students writing different examinations of different subjects – say the number of subjects is 4. How should the invigilator distribute the papers? Should he distribute in the order in which the students sit? Should he distribute in the order of subjects? Can we have more than one invigilator?

The class room has number of students – it is a container or a collection. The invigilator has to visit each student. He is iterating through the students in the class. He is not necessarily .art of the class. He is an iterator. It is possible to have a class with multiple invigilators. A container can have multiple iterators. Observe the fact that walking though a container depends on a container, but is not .art of the container normally. This concept is a very important concept.

The two terms I want you to master from this course are **interface** and **implementation**. We have observed that a for statement can walk through a container provided it is iterable. What does that mean?

The container class (like list) should support a function __iter__ (callable as iter(container_object)) which returns an object of a class – the object is called an iterator. This class should support __next__(callable as next(iterator_object)).

These should functions are interfaces. We can implement any way we want to support walking though a container logically. For example, we may visit only elements in odd position or elements satisfying a boolean condition – like elements greater than 100.

Let us look at a couple of examples.

This is from the file 3_iter.py

```
class MyContainer:
      def __init__(self, mylist):
            self.mylist = mylist

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

      def __next__(self):
            self.i += 1
            if self.i <= len(self.mylist):
                  return self.mylist[self.i - 1]
            else:
                  raise StopIteration

a = [ 'apple', 'banana', 'carrot', 'date', 'fish' ]
c = MyContainer(a)
for w in c :
      print(w)
```

Observe:

```
c = MyContainer(a)
```

Creates an object of MyContainer whose attribute mylist refers to the list a.

The `for` statement will call `iter(c)`. This call is changed to `MyContainer.__iter__(c)`.

This __iter__ function adds a position attribute i to the object. Then it returns the MyContainer object itself as the iterator object.

Then the for statement keeps calling next on this iterable object. This __next__ function has the logic to return the next element from the list and update the position and also raise the exception stop iteration when the end of the list is reached.

But there is one catch in this implementation. Let us examine the following code from the same file.

We create two iterator objects it1 and it2 from the same container object a. All these refer to the same object. Even though we have two iterator objects, both share the same location index. Calling next on it also affects the position index on it2.

```
a = [ 'apple', 'banana', 'carrot', 'date', 'fish' ]
c = MyContainer(a)
it1 = iter(c)
it2 = iter(c)
print(next(it1)) # apple
print(next(it2)) # expect apple, we will get banana
```

Let us look at the next example which avoid this. We do that by having another class and making the position index part of the iterator object and not the container object.

The code is from the file : 4_iter.py

```
class MyIterator:
      def __init__(self, c):
            self.c = c
            self.i = 0

      def __next__(self):
            self.i += 1
            if self.i <= len(self.c.mylist):
                  return self.c.mylist[self.i - 1]
      else:
            raise StopIteration

class MyContainer:
      def __init__(self, mylist):
            self.mylist = mylist

      def __iter__(self):
            return MyIterator(self)
```

```
a = [ 'apple', 'banana', 'carrot', 'date', 'fish' ]
c = MyContainer(a)
for w in c :
      print(w)

a = [ 'apple', 'banana', 'carrot', 'date', 'fish' ]
c = MyContainer(a)
it1 = iter(c)
it2 = iter(c)
print(next(it1)) # apple
print(next(it2)) # we will get apple as expected
```

Observe the changes.
iter function in MyContainer class returns an object of MyIterator class.
The iterator object holds a reference to the MyContainer object and also holds the position index.

```
it1 = iter(c)
it2 = iter(c)
```

These are two different objects having their own position index.
So, `next(it1)` does not affect `next(it2).`
Thats all about generators and iterators as of now.

References:

1. 19_gen_iterator.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://www.w3schools.com/python/
3. https://docs.python.org/
4. https://www.geeksforgeeks.org/ generators-in-python/
5. http://www.trytoprogram.com/python-programming/python-iterators/
6. http://www.trytoprogram.com/python-programming/python-generators