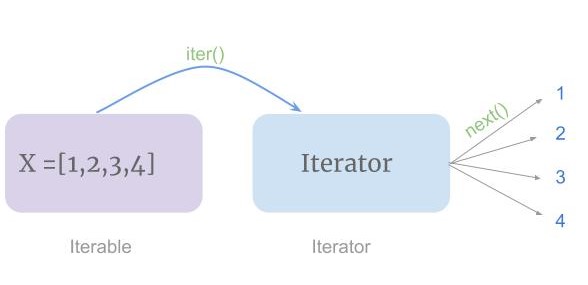
**Iterator**

**What is an iterator in Python?**

An iterator in Python is an object that contains a countable number of elements that can be iterated upon. In simpler words, we can say that Iterators are objects that allow you to traverse through all the elements of a collection and return one element at a time. More specifically, we say that an iterator is an object that implements the iterator protocol. We are going to discuss this protocol in the later section.

**Iterator and Iterable**



Iterable is an object, which one can iterate over. It generates an Iterator when passed to the iter() method. Lists, tuples, dictionaries, strings and sets are all iterable objects. They are iterable containers that you can convert into an iterator.

Note that every iterator is also an iterable, but not every iterable is an iterator. For example, a tuple is iterable, but it is not an iterator. An iterator can be created from an iterable by using the function iter(). Thus, when we pass this tuple to an iter() function, we will get an iterator. Actually, this is possible because the class of an object has a method \_\_iter\_\_, which returns an iterator.

**Which methods are defined in an iterator class in python?**

As mentioned above, an object that implements the iterator protocol is an iterator. But what is this iterator protocol? An iterator protocol is nothing but a specific class in Python which further has the \_\_iter\_\_()  and \_\_next\_\_()  methods.

**\_\_iter\_\_()**

This method returns the iterator object itself as mentioned above. This is required to allow both containers and iterators to be used with the for and in statements. This method corresponds to the tp\_iter slot of the type structure for Python objects in the Python/C API.

**\_\_next\_\_()**

This method returns the next item from the container. If there are no further items, raise the StopIteration exception. This method corresponds to the tp\_iternext slot of the type structure for Python objects in the Python/C API.

Here is a simple example, describing these methods:

# define a iterable such as a list

>>> list1 = [0, 1, 2]

# get an iterator using iter()

>>> iter1 = list1.\_\_iter\_\_()

#iertae the item using \_\_next\_\_method

>>> print(iter1.\_\_next\_\_())

0

We can use the \_\_next\_\_ method again to get the next item, in fact, we can use it as many times as we want and each time it’ll return the next item of the iterator. But when there are no elements in the iterator to return, it throws an error.

**Iterating Through an Iterator**

Now that we have an iterator object, what are the various ways by which we can traverse its elements such that we only get one item at a time? In this section, we are going to see a few ways in which we can do so, also we are going to show some of the examples for you to understand iterators better:

**Using the next() function**

We can directly use next() function or \_\_next\_\_() method to traverse the elements as shown in the examples below:

# define a iterable such as a list

>>> list1 = [1, 2, 0]

# get an iterator using iter()

>>> iter1 = iter(list1)

# iterate through it using next()

>>> print(next(iter1))

1

# next(obj) is same as obj.\_\_next\_\_()

>>> print(iter1.\_\_next\_\_())

2

>>> print(next(iter1))

0

# This will raise error, no items left

>>> next(my\_iter)

Traceback (most recent call last):

File "<pyshell#8>", line 1, in <module>

next(my\_iter)

Now let us see how we can traverse a string iterator:

# define a iterable such as a string

>>> string = "hi"

# get an iterator using iter()

>>> iter1 = iter(string)

# iterate through it using next()

>>> print(next(iter1))

h

>>> print(next(iter1))

i

# This will raise error, no items left

>>> next(my\_iter)

Traceback (most recent call last):

File "<pyshell#8>", line 1, in <module>

next(my\_iter)

**Iterating an iterator using while loop and for loop**

Since we are using the next() function repeatedly, why not use a while loop and make our work a bit easier. Also, we will use exception handling to take care of the error that we get when the iterator has no elements left. So whenever the error “StopIteration” occurs, the control goes to the except block and the break statement is executed. Here is an example:

# define an iterable such as a list

list1=[1,2,3,4,5,6,7,8,9,0]

# get an iterator using iter()

iter1=iter(list1)

# infinite loop

while True:

try:

# get the next item

print(next(iter1))

# do something with element

except StopIteration:

# if StopIteration is raised, break from loop

break

**Output:**

1  
2  
3  
4  
5  
6  
7  
8  
9  
0

There is a much easier way to do this also by using a For loop. The for loop does all this under the hood, thus, you don’t need to explicitly call the iter() and next() functions. Here are a few examples:

list1=[1,2,3,4,5,6,7,8,9,0]

for i in list1:

print(i)

Output:

1  
2  
3  
4  
5  
6  
7  
8  
9  
0

Now let us take some more example:

list2=[1,2,"hello",[9,8,7],(11,12),{'one': 'husaain'}]

for i in list2:

print(i)

Output:

1  
2  
hello  
[9, 8, 7]  
(11, 12)  
{‘one’: ‘husaain’}

string="Hello World "

for i in string:

print(i)

Output:

H  
e  
l  
l  
o    
  
W  
o  
r  
l  
d

**Create an Iterator**

Till now we have only used the inbuilt iterables such as lists or strings, but we can also build an iterator from scratch is easy in Python. We just have to implement the \_\_iter\_\_() and the \_\_next\_\_() methods.

Here is our own custom Iterator that returns an even number or 1 every time we iterate upon it:

class Evenit:

def \_\_init\_\_(self, max=0):

self.max = max

def \_\_iter\_\_(self):

self.n = 0

return self

def \_\_next\_\_(self):

if self.n <= self.max:

if self.n % 2 ==0:

result=self.n

self.n += 1

return result

else:

self.n += 1

return 1

else:

raise StopIteration

# create an object

numbers = Evenit(10)

for i in numbers:

print(i)

Output:

0  
1  
2  
1  
4  
1  
6  
1  
8  
1  
10

As we can see, this iterator returns even numbers up to 10 (because we have given the argument to the event as 10), and whenever it encounters an odd number, it just returns 1. In a similar manner, you can create your own iterator.

This brings us to the end of this article where we learned about iterators in Python and also created our own custom iterator from scratch.