Python - Iterator

Here, you will learn about the iterator function `iter()` in Python.

Iterators are implicitly used whenever we deal with collections of data types such as list, tuple or string (they are quite fittingly called iterables). The usual method to traverse a collection is using the for loop, as shown below.

Example:

```
myList = [1, 2, 3, 4]
for item in myList:
    print(item)
```

Result:

```
1
2
3
4
```

In the above example, the for loop iterates over a list object- myList, and prints each individual element.

When we use a for loop to traverse any iterable object, internally it uses the `iter()` method, same as below.

```
def traverse(iterable):
    it=iter(iterable)
    while True:
        try:
            item=next(it)
            print (item)
        except StopIteration:
            break
```

iter()

Instead of using the for loop as shown above, we can use the iterator function `iter()`. An iterator is an object which represents a data stream. It returns one element at a time. Python's built-in method `iter()` receives an iterable and returns an iterator object. The iterator object uses the `__next__()` method. Every time it is called, the next element in the iterator stream is returned. When there are no more elements available, StopIteration error is encountered.

```
>>> L1=[1, 2, 3]
>>>it=iter(L1)
>>>it.__next__()
1
>>>it.__next__()
2
```

```
>>>it.__next__()
3
>>>it.__next__()
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
it.__next__()
StopIteration
```

In the above example, list `L1` is specified in the `iter(L1)` method, which will return an iterator object for `L1`. Now, we can traverse a list of items using the `__next__()` method. So, `it.__next__()` will return the first item, and calling the `__next__()` method again will return the second item from `L1` and so on. It will throw the StopIteration error when calling the `__next__()` method after receiving the last item.

next()

Calling the `it.__next__()` method every time is tedious. The built-in function `next()` accepts an iterator object as a parameter and calls the `__next__()` method internally. Hence, `it.__next__()` is the same as `next(it)`. The following example uses the `next()` method to iterate the list.

```
>>> L1=[1,2,3]
>>>it=iter(L1)
>>>next(it)
>>>1
>>>next(it)
>>>2
>>>next(it)
>>>3
>>>next(it)
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
it.__next__()
StopIteration
```

If the argument is not iterable, e.g. number, boolean, etc., TypeError is encountered.

```
>>>iter(100)
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
iter(100)
TypeError: 'int' object is not iterable
```

The limitation of an iterator function is that it raises an exception when there is no next item.