Python Iterators

Iterators are everywhere in Python. They are elegantly implemented within `for` loops, comprehensions, generators etc. but hidden in plain sight. Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, Python **iterator object** must implement two special methods, `__iter__()` and `__next__()`, collectively called the **iterator protocol**.

An object is called **iterable** if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables. The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.

Iterating Through an Iterator in Python

We use the `next()` function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise `StopIteration`. Following is an example.

```
>>> # define a list
>>> my_list = [4, 7, 0, 3]

>>> # get an iterator using iter() on that list
>>> my_iter = iter(my_list)
>>> my_iter
<list_iterator object at 0x00000000031AD9B0>

>>> # iterate through it using next()
>>> next(my_iter)
4
>>> next(my_iter)
7

>>> # next(obj) is same as obj.__next__()
>>> my_iter.__next__()
0
>>> my_iter.__next__()
3

>>> next(my_iter)
Traceback (most recent call last):
...
```

```
StopIteration
>>> # no more items left
```

A more elegant way of automatically iterating is by using the `for` loop. Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

```
>>> for element in my_list:
...     print(element)
...
4
7
0
3
```

How for Loop Actually Works

As we see in the above example, the `for` loop was able to iterate automatically through the list. In fact the `for` loop can iterate over any iterable. Let's take a closer look at how the `for` loop is actually implemented in Python.

```
for element in iterable:
    # do something with element
```

Is actually implemented as.

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

So internally, the `for` loop creates an iterator object, `iter_obj` by calling `iter()` on the iterable. Ironically, this

`for` loop is actually an infinite `while` loop. Inside the loop, it calls `next()` to get the next element and executes the body of the `for` loop with this value. After all the items exhaust, `StopIteration` is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

Building Your Own Iterator in Python

Building an iterator from scratch is easy in Python. We just have to implement the methods `__iter__()` and `__next__()`. The `__iter__()` method returns the iterator object itself. If required, some initialization can be performed. The `__next__()` method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise `StopIteration`.

Here, we show an example that will give us next power of 2 in each iteration. Power exponent starts from zero up to a user set number.

```
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
```

Now we can create an iterator and iterate through it as follows.

```
>>> a = PowTwo(4)
>>> i = iter(a)
>>> next(i)
1
>>> next(i)
2
```

```
>>> next(i)
4
>>> next(i)
8
>>> next(i)
16
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

We can also use a `for` loop to iterate over our iterator class.

```
>>> for i in PowTwo(5):
...     print(i)
...
1
2
4
8
16
32
```

Python Infinite Iterators

It is not necessary that the item in an iterator object has to exhaust. There can be infinite iterators (which never ends). We must be careful when handling such iterator.

Here is a simple example to demonstrate infinite iterators. The built-in function `iter()` can be called with two arguments where the first argument must be a callable object (function) and second is the sentinel. The iterator calls this function until the returned value is equal to the sentinel.

```
>>> int()
0

>>> inf = iter(int,1)
>>> next(inf)
0
>>> next(inf)
0
```

We can see that the `int()` function always returns 0. So passing it as `iter(int,1)` will return an iterator that calls `int()` until the returned value equals 1. This never happens and we get an infinite iterator.

We can also built our own infinite iterators. The following iterator will, theoretically, return all the odd numbers.

```
class InfIter:
    """Infinite iterator to return all
        odd numbers"""

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

    def __next__(self):
        num = self.num
        self.num += 2
        return num
```

A sample run would be as follows.

```
>>> a = iter(InfIter())
>>> next(a)
1
>>> next(a)
3
>>> next(a)
5
>>> next(a)
7
```

And so on...

Be careful to include a terminating condition, when iterating over these type of infinite iterators. The advantage of using iterators is that they save resources. Like shown above, we could get all the odd numbers without storing the entire number system in memory. We can have infinite items (theoretically) in finite memory. Iterator also makes a code look cool.