Python Iterators

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but are 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, a 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 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:

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 the StopIteration Exception. Following is an example.

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

# get an iterator using iter()
my_iter = iter(my_list)

# iterate through it using next()

# Output: 4
print(next(my_iter))

# Output: 7
print(next(my_iter))

# next(obj) is same as obj.__next__()

# Output: 0
print(my_iter.__next__())

# Output: 3
print(my_iter.__next__())

# This will raise error, no items left
next(my_iter)
```

Output:

```
4
7
0
3
Traceback (most recent call last):
  File "/home/nirajan/PycharmProjects/PythonExamples/IteratingThroughAnIterator.py", line 24, in <module>
    next(my_iter)
StopIteration
```

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
```

Working of for loop for Iterators:

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 Custom Iterators:

Building an iterator from scratch is easy in Python. We just have to implement the __iter__() and the __next__() methods.

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 the next power of 2 in each iteration. Power exponent starts from zero up to a user set number.

For Example:

```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


# create an object
numbers = PowTwo(3)
```

```python
# create an object
numbers = PowTwo(3)

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

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

Output:

```
1
2
4
8
Traceback (most recent call last):
  File "/home/nirajan/PycharmProjects/PythonExamples/BuildingCustomGenerators.py", line 32, in <module>
    print(next(i))
  File "/home/nirajan/PycharmProjects/PythonExamples/BuildingCustomGenerators.py", line 18, in __next__
    raise StopIteration
StopIteration
```

Python Infinite Iterators:

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

Here is a simple example to demonstrate infinite iterators.

```
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 = iter(InfIter())
print(next(a))
print(next(a))
print(next(a))
print(next(a))
```

Output:

```
1
3
5
7
```

And so on...

Be careful to include a terminating condition, when iterating over these types 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.