Iterators and Generators

Table of Contents

What are Iterators?

We have seen how the `for` loop can be used to iterate over any sequence type (such as lists, tuples and strings). Iterators were introduced in Python 2.2 to give non-sequence container objects a sequence-like interface.

The concept of iterable objects, though relatively new, has become pervasive in Python. So you can now iterate over objects such as the keys of a dictionary, lines of a file or user-defined objects which are not sequences but provide a sequence-like behavior.

To provide iteration support, the container object needs to define two methods :

1. `__iter__()` which returns an iterator object
2. `next()` which returns the next item from the container

Understanding the Iteration Protocol

An easy way to understand is to look at how the iteration protocol works with the `file` built-in type. Prior to Python 2.2, to read lines from a file you could code something like the following :

In [1]:
```
cd code
```

```
C:\python_training\notebooks\code
```

In [2]:
```
f = open('iter1.py')
while True:
    line = f.readline()
    if not line:
        break
    print line,
```

```
# iter1.py

import sys

print sys.platform
x = 2
print x**16
```

In the above example the `readline()` method reads one line at a time and returns an empty string at the end of the file which detect and break out of the while loop.

In [4]:
```
# read all lines at a time
for line in open('iter1.py').readlines():
    print line,
```

```
# iter1.py

import sys

print sys.platform
x = 2
print x**16
```

However the **best practice** today is to use the `file` object as an iterable :

In [5]:
```
for line in open('iter1.py'):
    print line,
```

```
# iter1.py

import sys

print sys.platform
x = 2
print x**16
```

To support the iteration protocol, the file object has a method next() that returns the next line each time it is called. At the end-of-file, `next()` raises a `StopIteration` exception instead of an empty string. To understand what happens experiment as follows:

In [6]:
```
# iter1.py

import sys

print sys.platform
x = 2
print x**16
```

```
win32
65536
```

In [7]:
```
f = open('iter1.py')
f.next()
```

Out[7]:
```
'# iter1.py\n'
```

In [14]:
```
f.next()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-14-c3e65e5362fb> in <module>()
----> 1 f.next()

StopIteration:
```

In [ ]:
```
f.next()
```

In [ ]:
```
f.next()
```

In [ ]:
```
f.next()
```

In [ ]:
```
f.next()
```

In [ ]:
```
f.next()
```

In [ ]:
```
f.next()
```

The above interface is called the *iteration protocol* in Python. To summarize, an object is considered as iterable if it provides the following :

- an `__iter__()` method (not shown in the above example)
- a `next()` method to advance to the next item
- raise a `StopIteration` exception at the end of the series of items

Any iterable object can be stepped over with a `for` loop or other iteration tool since all iteration tools work internally by calling `next()` on each iteration and exiting upon catching the `StopIteration` exception.

Example :

We had a first look at the initial subsequence (10%20Statements%2C%20Conditionals%20and%20Loops.ipynb#Blocks-and-Indentation) of the Fibonacci series in a previous section. Let us look at how to implement an iterator for producing the Fibonacci series :

In [1]:
```
# fibos.py - Iterator for Fibonacci series

class Fibos(object):
    def __init__(self):
        self.a = 0
        self.b = 1

    def next(self):
        self.a, self.b = self.b, self.a+self.b
        return self.a

    def __iter__(self):
        return self

if __name__ == '__main__':

    fibos = Fibos()    # Create a Fibos instance

    # now we can use fibos as an iterator
    # say, to find smallest Fibonacci number that is greater than 500
    for fn in fibos:
        if fn > 500:
            print 'Smallest Fibonacci number greater than 500 is ', fn
            break
```

```
Smallest Fibonacci number greater than 500 is  610
```

NOTE

In Python 3, the `next()` method has been renamed to `__next__()`.

Generators

This subject has been covered in the Functions and Functional Programming Notebook (04%20Functions%20and%20Functional%20Programming.ipynb#Generators-and-yield)

License

Back to Top