1 Introduction

An iterator is a special type of object that generalizes the notion of a looping variable. When passed as an argument to a function that knows what to do with it, the iterator supplies a sequence of values. The iterator also maintains information about its state, in particular its current index. The iterators package includes a number of functions for creating iterators, the simplest of which is \texttt{iter}, which takes virtually any R object and turns it into an iterator object. The simplest function that operates on iterators is the \texttt{nextElem} function, which when given an iterator, returns the next value of the iterator. For example, here we create an iterator object from the sequence 1 to 10, and then use \texttt{nextElem} to iterate through the values:

\begin{verbatim}
> library(iterators)
> i1 <- iter(1:10)
> nextElem(i1)

[1] 1

> nextElem(i1)

[1] 2
\end{verbatim}

You can create iterators from matrices and data frames, using the \texttt{by} argument to specify whether to iterate by row or column:

\begin{verbatim}
> istate <- iter(state.x77, by='row')
> nextElem(istate)
\end{verbatim}
Using The iterators Package

<table>
<thead>
<tr>
<th>Population</th>
<th>Income</th>
<th>Illiteracy</th>
<th>Life Exp</th>
<th>Murder</th>
<th>HS Grad</th>
<th>Frost</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>3615</td>
<td>3624</td>
<td>2.1</td>
<td>69.05</td>
<td>15.1</td>
<td>41.3</td>
<td>20</td>
</tr>
<tr>
<td>Alaska</td>
<td>365</td>
<td>6315</td>
<td>1.5</td>
<td>69.31</td>
<td>11.3</td>
<td>66.7</td>
<td>152</td>
</tr>
</tbody>
</table>

Iterators can also be created from functions, in which case the iterator can be an endless source of values:

```r
> ifun <- iter(function() sample(0:9, 4, replace=TRUE))
> nextElem(ifun)

[1] 4 3 8 6

> nextElem(ifun)

[1] 8 9 7 7
```

For practical applications, iterators can be paired with `foreach` to obtain parallel results quite easily:

```r
> library(foreach)

foreach: simple, scalable parallel programming from Revolution Analytics
Use Revolution R for scalability, fault tolerance and more.
http://www.revolutionanalytics.com

> x <- matrix(rnorm(1e+06), ncol = 10000)
> itx <- iter(x, by = "row")
> foreach(i = itx, .combine = c) %dopar% mean(i)

```

```r
[1] -0.0069652059 0.0161112989 0.0080068074 -0.0120020610 0.0017168149
[6] 0.0139835943 -0.0078172106 -0.0024762273 -0.0031558268 -0.0072662893
[11] -0.0055142639 0.0015717907 -0.0100842965 -0.0123601527 0.0136420084
[16] -0.0242922105 -0.0126416949 -0.00552951152 0.0216329326 -0.0262476648
[21] 0.0041937609 0.0121253368 -0.0110165729 0.0044267635 0.0080241894
[26] 0.0042995539 -0.0102826632 0.0051185628 -0.0013970812 -0.0172380786
```
2 Some Special Iterators

The notion of an iterator is new to R, but should be familiar to users of languages such as Python. The *iterators* package includes a number of special functions that generate iterators for some common scenarios. For example, the `irnorm` function creates an iterator for which each value is drawn from a specified random normal distribution:

```r
> library(iterators)
> itrn <- irnorm(10)
> nextElem(itrn)
```

```
[1]  1.674270235  0.006717347  0.498024218 -0.075201538 -1.292718480
[6] -1.222374316  2.152976791 -0.067293668 -0.300861921 -0.410451137
```

```r
> nextElem(itrn)
```

```
[1]  1.30820241  1.31483944  1.35874472 -1.76713320  1.37676515 -1.48627013
[7]  1.32807836  1.85137340  0.02946489 -0.39687072
```

Similarly, the `irunif`, `irbinom`, and `irpois` functions create iterators which draw their values from uniform, binomial, and Poisson distributions, respectively.

We can then use these functions just as we used `irnorm`:

```r
> itru <- irunif(10)
> nextElem(itru)
```
Using The `iterators` Package

```
[1] 0.09693224 0.87218070 0.16036355 0.08101483 0.24296013 0.46463357
[7] 0.87178980 0.99886539 0.65196785 0.05526307

> nextElem(itru)

[1] 0.2284789 0.4671382 0.7638112 0.4783986 0.3798136 0.3836742 0.4676569
[8] 0.2972599 0.9200900 0.5763258

The `icount` function returns an iterator that counts starting from one:

> it <- icount(3)
> nextElem(it)

[1] 1

> nextElem(it)

[1] 2

> nextElem(it)

[1] 3
```