Using The `iterators` Package

Rich Calaway

richcalaway@revolution-computing.com

July 20, 2009

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 `iter`, which takes virtually any R object and turns it into an iterator object. The simplest function that operates on iterators is the `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 `nextElem` to iterate through the values:

```
> library(iterators)
> i1 <- iter(1:10)
> nextElem(i1)
```

```
[1] 1
```

```
> nextElem(i1)
```

```
[1] 2
```

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

```
> istate <- iter(state.x77, by = "row")
> nextElem(istate)
```

```
          Population Income Illiteracy Life Exp Murder HS Grad Frost   Area
Alabama         3615   3624        2.1    69.05   15.1    41.3     20 50708
```

```
> nextElem(istate)
```

```
          Population Income Illiteracy Life Exp Murder HS Grad Frost    Area
Alaska           365   6315        1.5    69.31   11.3    66.7    152 566432
```

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

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

```
[1] 1 6 5 6
```

```
> nextElem(ifun)
```

```
[1] 0 4 3 8
```

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

```
> library(foreach)
```

```
foreach: simple, scalable parallel programming from REvolution Computing
Use REvolution R for scalability, fault tolerance and more.
http://www.revolution-computing.com
```

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

```
  [1] -0.0040710604  0.0097467171  0.0022043596  0.0109891194 -0.0062254874
  [6]  0.0130028917 -0.0024575821  0.0052974462 -0.0029302729 -0.0107083931
 [11] -0.0164476128 -0.0065090243  0.0016966248 -0.0123011644 -0.0006232640
 [16]  0.0081913672  0.0030860009  0.0130874302 -0.0096216450  0.0088969712
 [21] -0.0020604662 -0.0043140207 -0.0012823852  0.0060662727 -0.0101623872
 [26] -0.0037512240  0.0016970882  0.0017170333 -0.0141240177 -0.0042973247
```

```
 [31]   0.0010162580   0.0116825920   0.0045971389  -0.0040916539   0.0053578457
 [36]   0.0034443511   0.0164102293   0.0031012621   0.0037105796   0.0096859089
 [41]   0.0120647321  -0.0008609619   0.0053445192   0.0097641619   0.0113396312
 [46]   0.0078107070   0.0139583829   0.0024135899  -0.0208531016  -0.0062693833
 [51]   0.0151801025  -0.0027044067   0.0059327559   0.0025392198   0.0073770923
 [56]  -0.0005665411   0.0248923225  -0.0154207459   0.0127998011  -0.0090775257
 [61]  -0.0077243702  -0.0175771448   0.0107690258   0.0036175829   0.0004195700
 [66]  -0.0020560297   0.0070443072   0.0098662270  -0.0002170741  -0.0170780578
 [71]   0.0034730946  -0.0017520503  -0.0100526727  -0.0162407305  -0.0012446651
 [76]  -0.0048357119   0.0234021283   0.0141282018  -0.0056361458   0.0071032796
 [81]  -0.0089421043   0.0059342475   0.0070410147  -0.0070316701   0.0065529424
 [86]   0.0288267901  -0.0097902841  -0.0058481502   0.0064289241  -0.0103422987
 [91]   0.0008049182  -0.0045195646  -0.0130103143   0.0074859619   0.0086388529
 [96]   0.0058278355   0.0125116002  -0.0013952979   0.0162009874   0.0085128135
```

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:

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

 [1]   1.0350858  -0.2464664  -2.2801741  -1.5195656  -1.1507377   0.7881754
 [7]  -0.8338092  -0.8315724   0.1029678   0.7088185

> nextElem(itrn)

 [1]  -0.8444356  -1.3008714  -0.2281682  -0.1154227   0.3875200   0.8456657
 [7]   0.6657748   1.1860130  -2.0066959   0.3022127
```

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

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

```
> itru <- irunif(10)
> nextElem(itru)
```

```
 [1]  0.12272881 0.23282231 0.88774034 0.41627943 0.80330098 0.55480407
 [7]  0.93245959 0.08326419 0.03605037 0.55380334
```

```
> nextElem(itru)
```

```
 [1]  0.31878022 0.06503089 0.11131593 0.68338828 0.87976008 0.54240576
 [7]  0.25652083 0.49187155 0.87888189 0.34929741
```

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
```