Iterators in R

According to Wikipedia, an iterator is “an object that enables a programmer to traverse a container”. A collection of items (stashed in a container) can be thought of as being “iterable” if there is a logical progression from one element to the next (so a list is iterable, while a set is not). An iterator is then an object for moving through the container, one item at a time.

Iterators are a fundamental part of contemporary Python programming, where they form the basis for loops, list comprehensions and generator expressions. Iterators facilitate navigation of container classes in the C++ Standard Template Library (STL). They are also to be found in C#, Java, Scala, Matlab, PHP and Ruby.

Although iterators are not part of the native R language, they are implemented in the iterators and itertools packages.

iterators Package

The iterators package is written and maintained by Revolution Analytics. It contains all of the basic iterator functionality.

```
> library(iterators)
```

Iterating over a Vector

The iter() function transforms a container into an iterator. Using it we can make an iterator from a vector.

```
> name = c("Bob", "Mary", "Jack", "Jane")
>
> iname <- iter(name)
> nextElem(iname)
[1] "Bob"
> nextElem(iname)
[1] "Mary"
> nextElem(iname)
[1] "Jack"
> nextElem(iname)
[1] "Jane"
```

Here iteration is performed manually by calling nextElem(): each call yields the next element in the vector. And

when we fall off the end of the vector, we get a StopIteration error.

```
> nextElem(iname)
Error: StopIteration
```

Iterating over Data Frames and Matrices

Similarly, we can make an iterator from a data frame.

```
> (people = data.frame(name, ages = c(17, 23, 41, 19), gender = rep(c("M", "F"), 2)))
  name ages gender
1  Bob   17      M
2 Mary   23      F
3 Jack   41      M
4 Jane   19      F
```

We can iterate over the data frame by column (the default behaviour)…

```
> ipeople <- iter(people)
> nextElem(ipeople)
[1] Bob  Mary Jack Jane
Levels: Bob Jack Jane Mary
> nextElem(ipeople)
[1] 17 23 41 19
> nextElem(ipeople)
[1] M F M F
Levels: F M
```

… or by row.

```
> ipeople <- iter(people, by = "row")
> nextElem(ipeople)
  name ages gender
1  Bob   17      M
> nextElem(ipeople)
  name ages gender
2 Mary   23      F
> nextElem(ipeople)
  name ages gender
3 Jack   41      M
> nextElem(ipeople)
  name ages gender
4 Jane   19      F
```

To my mind iterating by row is likely to be the most common application, so I am a little puzzled by the fact that it is not the default. No doubt the creators had a solid reason for this design choice though!

Not surprisingly, we can iterate through the rows and columns of a matrix as well. But, taking things one step further, we will do this in parallel using the foreach library. First we set up the requisites for multicore processing.

```
> library(foreach)
> #
```

```
> library(doMC)
> registerDoMC(cores=4)
```

Then, for illustration purposes, we create a matrix of normally distributed random numbers. Our parallel loop will use an iterator to run through the matrix, computing the mean for every row. Of course, there are other ways of accomplishing the same task, but this syntax is rather neat and intuitive.

```
> Z <- matrix(rnorm(10000), ncol = 100)
> foreach(i = iter(Z, by = "row"), .combine = c) %dopar% mean(i)
  [1]  0.0603617  0.1588855 -0.0973709  0.0624385  0.0446618 -0.1528632  0.1295854
  [8]  0.0222499 -0.0420512 -0.0202990 -0.0855573 -0.0506006 -0.0062855 -0.0621060
 [15] -0.0248356 -0.0108870  0.2863956  0.0515027  0.1393728 -0.1280088 -0.0092294
 [22]  0.0373697 -0.1844472  0.1223116  0.0094714  0.0557162 -0.0061684 -0.0404645
 [29] -0.1935638  0.0626071  0.0246498  0.0438532 -0.0254658  0.0733297 -0.1075596
 [36]  0.0659071  0.1146148  0.0684804 -0.0621941 -0.1250420  0.0303364 -0.0975498
 [43] -0.0908564 -0.2332864 -0.1978676  0.1444749  0.0202549  0.0246314  0.0871942
 [50]  0.0152622 -0.0353871 -0.0708653  0.0927205 -0.0601456 -0.0190797 -0.0397878
 [57] -0.2066360 -0.0078547 -0.1445867  0.0955066  0.0130583 -0.0656415  0.1003008
 [64]  0.0794965 -0.1044313 -0.1024588  0.0019395 -0.0390669 -0.0722961  0.0772205
 [71] -0.0137769  0.1186736  0.0714488 -0.0525488  0.1204149 -0.0682227 -0.0110207
 [78] -0.0079334  0.0589447 -0.0464116 -0.0293395  0.2112834  0.0112325  0.0907571
 [85]  0.0036712 -0.0375137  0.0817137  0.0092214  0.0068116 -0.0887101 -0.0623266
 [92] -0.1580448  0.0870227 -0.0517170  0.1160297 -0.2065726 -0.0517491 -0.0312476
 [99] -0.0363837 -0.1619014
```

Another way of slicing up a matrix uses iapply(), which is analogous to the native apply() function. First, let's create a small and orderly matrix.

```
> (Z <- matrix(1:100, ncol = 10))
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    1   11   21   31   41   51   61   71   81    91
 [2,]    2   12   22   32   42   52   62   72   82    92
 [3,]    3   13   23   33   43   53   63   73   83    93
 [4,]    4   14   24   34   44   54   64   74   84    94
 [5,]    5   15   25   35   45   55   65   75   85    95
 [6,]    6   16   26   36   46   56   66   76   86    96
 [7,]    7   17   27   37   47   57   67   77   87    97
 [8,]    8   18   28   38   48   58   68   78   88    98
 [9,]    9   19   29   39   49   59   69   79   89    99
[10,]   10   20   30   40   50   60   70   80   90   100
```

Choosing margin = 1 gives us the rows, while margin = 2 yields the columns.

```
> irow <- iapply(Z, 1)
> nextElem(irow)
 [1]  1 11 21 31 41 51 61 71 81 91
> nextElem(irow)
 [1]  2 12 22 32 42 52 62 72 82 92
> icol <- iapply(Z, 2)
> nextElem(icol)
 [1]  1  2  3  4  5  6  7  8  9 10
> nextElem(icol)
 [1] 11 12 13 14 15 16 17 18 19 20
```

Iterating through a File

An iterator to read lines from a file is extremely useful. This is certainly the iterator pattern that I have used most

often in Python. In R a file iterator is created using ireadLines().

```
> ifile <- ireadLines('/etc/passwd')
> nextElem(ifile)
[1] "root:x:0:0:root:/root:/bin/bash"
> nextElem(ifile)
[1] "daemon:x:1:1:daemon:/usr/sbin:/bin/sh"
> nextElem(ifile)
[1] "bin:x:2:2:bin:/bin:/bin/sh"
```

Each call to nextElem() returns a new line from the file. We can put this into a loop and traverse the entire file.

```
> while (TRUE) {
+   d = try(nextElem(ifile))
+   if (class(d) == "try-error") break
+   print(d)
+ }
[1] "sys:x:3:3:sys:/dev:/bin/sh"
[1] "sync:x:4:65534:sync:/bin:/bin/sync"
[1] "games:x:5:60:games:/usr/games:/bin/sh"
[1] "man:x:6:12:man:/var/cache/man:/bin/sh"
#
# content omitted for brevity...
#
[1] "ntp:x:121:129::/home/ntp:/bin/false"
[1] "nx:x:122:1010::/var/NX/nx:/etc/NX/nxserver"
Error : StopIteration
```

Hmmmm. The code for the loop is a little clunky and it ends in a rather undignified fashion. Surely there is a better way to do this? Indeed there is! More on that later.

Filtering with an Iterator

iter() allows you to specify a filter function which can be used to select particular items from a container. This function should return a Boolean (or a value which can be coerced to Boolean) indicating whether (TRUE) or not (FALSE) an item should be accepted. To illustrate this, we will pick out prime numbers from the first one hundred integers. The gmp package has an isprime() function which is perfectly suited to this job.

```
> library(gmp)
```

First we'll look at the basic functionality of isprime().

```
> isprime(13)
[1] 2
> isprime(8)
[1] 0
> isprime(2147483647)
[1] 1
```

A return value of 2 indicates that a number is definitely prime, while 0 indicates a composite number. For small numbers these are the only two options, however, for larger numbers the Miller-Rabin primality test is applied, in which case a return value of 1 indicates that a number is *probably* prime.

So, here we go, grabbing the first four primes:

```
> iprime <- iter(1:100, checkFunc = function(n) isprime(n))
> nextElem(iprime)
[1] 2
> nextElem(iprime)
[1] 3
> nextElem(iprime)
[1] 5
> nextElem(iprime)
[1] 7
```

An Iterable Version of split()

The native function split() accepts two arguments: the first is a vector and the second is a factor which dictates how the vector's elements will be divided into groups. The return value is a list of vectors corresponding to each of the groups. The isplit() function accepts the same arguments and results in an iterator which steps through each of the groups, returning a list with two fields: the key (one of the levels of the factor) and the corresponding values extracted from the vector.

```
> ipeople <- isplit(people$name, people$gender)
> #
> p = nextElem(ipeople)
> p
$value
[1] Mary Jane
Levels: Bob Jack Jane Mary

$key
$key[[1]]
[1] "F"
```

```
> p = nextElem(ipeople)
> p$key
[[1]]
[1] "M"

> p$value
[1] Bob  Jack
Levels: Bob Jack Jane Mary
```

Functions as Iterators

It is also possible to have a function packaged as an iterator. The function is called every time that the iterator is advanced.

```
> ifunc <- iter(function() sample(name, 1))
> nextElem(ifunc)
[1] "Bob"
> nextElem(ifunc)
[1] "Mary"
> nextElem(ifunc)
```

```
[1] "Mary"
> nextElem(ifunc)
[1] "Jack"
```

Here each succesive element returned by the iterator is generated by a call to an anonymous function which extracts a single sample from our list of names. I am not quite sure how I would use this facility in a practical situation (why is this superior to simply calling the function?), but I am sure that a good application will reveal itself in due course.

Rolling Your Own: A Fibonacci Iterator

An iterator to generate Fibonacci Numbers is readily implemented in Python.

```
def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a+b

for n in fibonacci():
    print(n)
```

An analogous iterator can be implemented in R. Perhaps the syntax is not quite as succinct, but it certainly does the job.

```
> fibonacci <- function() {
+   ab = c(0, 1)
+   n <- function() {
+     ab <<- c(ab[2], sum(ab))
+     ab[1]
+   }
+   obj <- list(nextElem = n)
+   class(obj) <- c('fibonacci', 'abstractiter', 'iter')
+   obj
+ }
```

Now we can instantiate a Fibonacci iterator and start generating terms…

```
> ifib = fibonacci()
> nextElem(ifib)
[1] 1
> nextElem(ifib)
[1] 1
> nextElem(ifib)
[1] 2
> nextElem(ifib)
[1] 3
> nextElem(ifib)
[1] 5
> nextElem(ifib)
[1] 8
> #
> unlist(as.list(ifib, n = 10))
 [1]  13  21  34  55  89 144 233 377 610 987
```

Again, you might be wondering what advantages this provides over simply having a function which generates the sequence. Consider a situation where you need to have two independent sequences. Certainly the state of each sequence could be stored and passed as an argument. An object oriented solution would be natural. However, iterators will do the job very nicely too.

```
> i1 = fibonacci()
> i2 = fibonacci()
> unlist(as.list(i1, n = 7))
[1]  1  1  2  3  5  8 13
> unlist(as.list(i2, n = 10))
 [1]  1  1  2  3  5  8 13 21 34 55
```

We have generated some terms from each of the iterators. Not surprisingly, when we return to generate further terms, we find that each of the iterators picks up at just the right position in the sequence.

```
> unlist(as.list(i1, n = 7))
[1]  21  34  55  89 144 233 377
> unlist(as.list(i2, n = 10))
 [1]   89  144  233  377  610  987 1597 2584 4181 6765
```

itertools Package

The itertools package is written and maintained by Steve Weston (Yale University) and Hadley Wickham (RStudio). It provides a range of extensions to the basic iterator functionality.

```
> library(itertools)
```

Stop When You Reach the End

The Fibonacci iterator implemented above will keep on generating terms indefinitely. What if we wanted it to terminate after a finite number of terms? We could introduce a parameter for the required number of terms and generate a StopIteration error when we try to exceed this limit.

```
> fibonacci <- function(count = NA) {
+   ab = c(0, 1)
+   n <- function() {
+     if (!is.na(count)) {
+       if (count > 0) count <<- count -1
+       else stop('StopIteration')
+     }
+     #
+     ab <<- c(ab[2], sum(ab))
+     ab[1]
+   }
+   obj <- list(nextElem = n)
+   class(obj) <- c('fibonacci', 'abstractiter', 'iter')
+   obj
+ }
```

In practice this works fine…

```
> ifib = fibonacci(4)
> nextElem(ifib)
[1] 1
> nextElem(ifib)
[1] 1
> nextElem(ifib)
[1] 2
> nextElem(ifib)
[1] 3
> nextElem(ifib)
 Show Traceback

 Rerun with Debug
 Error in obj$nextElem() : StopIteration
```

… but it generates a rather rude error message if you try to iterate beyond the prescribed number of terms. How can we handle this more elegantly? Enter the ihasNext() wrapper which enables a hasNext() function for an iterator.

```
> ifib = ihasNext(fibonacci(4))
> while (hasNext(ifib)) {
+   print(nextElem(ifib))
+ }
[1] 1
[1] 1
[1] 2
[1] 3
```

So, instead of blindly trying to proceed to the next item, we first check whether there are still items available in the iterator. This pattern can be applied to any situation where the iterator has only a finite number of terms. Iterating through the lines in a file would be a prime example!

```
> ifile <- ihasNext(ireadLines('/etc/passwd'))
> while (hasNext(ifile)) {
+   print(nextElem(ifile))
+ }
[1] "root:x:0:0:root:/root:/bin/bash"
[1] "daemon:x:1:1:daemon:/usr/sbin:/bin/sh"
[1] "bin:x:2:2:bin:/bin:/bin/sh"
#
# content omitted for brevity...
#
[1] "ntp:x:121:129::/home/ntp:/bin/false"
[1] "nx:x:122:1010::/var/NX/nx:/etc/NX/nxserver"
```

That's much better!

Limiting the Limitless

Adapting our Fibonacci iterator to stop after a finite number of terms was a bit of a kludge. Not to worry: the ilimit() wrapper allows us to stipulate a limit on the number of terms.

```
> ifib <- ilimit(fibonacci(), 12)
> unlist(as.list(ifib))
 [1]   1   1   2   3   5   8  13  21  34  55  89 144
```

Recycle Everything

We have seen how to truncate an effectively infinite series of iterates. How about replicating a finite series? Using recycle() you can run through the iterable a number of times. As a first example, we create an iterator which will run through a integer sequence (1, 2 and 3) twice.

```
> irec <- recycle(1:3, 2)
> nextElem(irec)
[1] 1
> nextElem(irec)
[1] 2
> nextElem(irec)
[1] 3
> nextElem(irec)
[1] 1
> nextElem(irec)
[1] 2
> nextElem(irec)
[1] 3
```

Next we generate the first six terms from the Fibonacci sequence and repeat them twice.

```
> irec <- recycle(ilimit(fibonacci(), 6), 2)
> unlist(as.list(irec))
 [1] 1 1 2 3 5 8 1 1 2 3 5 8
```

Enumerating

The enumerate() function returns an iterator which returns each element of the iterable as a list with two elements: an index and the corresponding value. So, for example, we can number each of the elements in our list of names.

```
> iname <- enumerate(name)
> #
> cat(sapply(iname, function(n) sprintf("%d -> %s\n", n$index, n$value)), sep = "")
1 -> Bob
2 -> Mary
3 -> Jack
4 -> Jane
```

Zip Them Up!

A generalisation of enumeration is created by zipping up arbitrary iterables.

```
> ifib <- izip(a = letters[1:6], b = fibonacci())
> #
> cat(sapply(ifib, function(n) sprintf("%s -> %d", n$a, n$b)), sep = "\n")
a -> 1
b -> 1
c -> 2
d -> 3
```

```
e -> 5
f -> 8
```

Naturally, we can use this to replicate the functionality of enumerate().

```
> iname <- izip(index = 1:4, value = name)
> cat(sapply(iname, function(n) sprintf("%d -> %s\n", n$index, n$value)), sep = "")
1 -> Bob
2 -> Mary
3 -> Jack
4 -> Jane
```

Getting Chunky!

What about returning groups of values from an iterator? The ichunk() wrapper takes two arguments: the first is an iterator, the second is the number of terms to be wrapped in a chunk. Here is an example of generating groups of three successive terms from the Fibonacci sequence.

```
> ifib <- ichunk(fibonacci(), 3)
> unlist(nextElem(ifib))
[1] 1 1 2
> unlist(nextElem(ifib))
[1] 3 5 8
> unlist(nextElem(ifib))
[1] 13 21 34
```

Until I Say “Stop!”

Finally, an iterator which continues to generate terms until a time limit is reached. How many terms will our Fibonacci iterator generate in 0.1 seconds?

```
> length(as.list(timeout(fibonacci(), 0.1)))
[1] 701
```

Conclusion

I have already identified a few older projects where the code can be significantly improved by the use of iterators. And I will certainly be using them in new projects. I hope that you too will be able to apply them in your work. Iterate and prosper!

References

- Weston, S. (2012). Writing Custom Iterators.

This entry was posted in R and tagged Fibonacci, filtering, foreach, Iterator, prime numbers, R on 2013/11/14

[http://www.exegetic.biz/blog/2013/11/iterators-in-r/] .

10 thoughts on “Iterators in R”

MySchizoBuddy
2013/11/14 at 23:14

What would be the doMC equivalent on windows and for R 3.x

Andrew Post author
2013/11/15 at 16:12

Being a Linux guy, I don’t have much experience with multicore programming on Windows, but I think that doParallel package will do the job.

Russell Pierce
2013/11/20 at 01:58

In Windows the equivalent is just in the parallel package. You could…

```
library(parallel)
cl <- makeCluster(2)
Z <- matrix(rnorm(10000),ncol=100)
parApply(cl,Z,MARGIN=2,mean)
stopCluster(cl)
```

… but if you really love foreach then…

```
library(parallel)
library(doParallel)
cl <- makeCluster(2)
```

```
Z <- matrix(rnorm(10000),ncol=100)
foreach(i = iter(Z, by = "row"), .combine = c) %dopar% mean(i)
stopCluster(cl)
```

Kamil
2013/11/15 at 21:16

Thank you for an interesting post. I'm used to python's iterators and whenever using R, I missed them a lot. It is nice to see that there is a way towards them

---

Matt Weller
2013/12/28 at 01:59

Is it possible to use an iterator to define groups of rows in a data.frame? I'm trying to use foreach in R and I want to process a set of records (which form 313 periods of multivariate time series data) within the loop on different cores. The code is working currently but it copies the big dataset over into each new R process so that's 8 times the RAM usage!!

I've probably explained it a little better with a reproducible example on Stackoverflow.

Andrew Post author
2013/12/28 at 03:13

Hi! Without looking at your code, it's hard to say. I suspect that iterators may not be the most efficient way to accomplish this though. Please fill in the missing link in your post and I will check out your SO post. Thanks, Andrew.

Russell Pierce

2013/12/28 at 03:21

I'm on mobile and can't pull up your stack exchange link. Are you using Windows or Linux? I think the multiple copies of your target dataset is something that tends to happen with Windows multiple core processing. If you are on Linux perhaps see http://stackoverflow.com/questions/1395191/how-to-split-a-data-frame-by-rows-and-then-process-the-blocks.

Matt Weller
2013/12/28 at 13:06

Whilst it may not be the optimal solution I found the Stackoverflow answer about using ichunk did the trick. I've linked to my original SO question here: http://stackoverflow.com/questions/20810181/memory-issue-with-foreach-loop-in-r-on-windows-8-64-bit-doparallel-package/20814851#20814851

Many thanks for the pointer Russell and also to the developers for this great package!!

Steve Weston
2013/12/29 at 01:28

Nice article. For the record, I wrote itertools after leaving Revolution Analytics about four years ago. I'm now working at Yale University.

Andrew Post author
2013/12/29 at 18:06

Hi Steve, thanks for the comment and apologies for the erroneous association. I have corrected the article. Best regards, Andrew.