Iteration

Paul E. Johnson[1] [2]

[1]Department of Political Science

[2]Center for Research Methods and Data Analysis, University of Kansas

2018

Outline

Outline

R Frame of Mind

- Iteration is commonly needed in R (R Core Team, 2017)
  - repeat the same thing over and over with new samples
  - process several subgroups of data (compare cities)
  - apply various functions to one data set
- Some idioms make code faster.
- Some idioms make code more understandable.

For Loops and Iterators

- I'm cutting out a lot of philosophical BS about iterators here. I hope nobody says "We want to hear a lot more computer science theory about keys, iterators, and index variables"
- All computer languages with which I'm aware have some variant of a **for loop**, a way to say "here are 14 rows, process each one in order"
- R has for loops, as well as a family of "apply" functions that are very widely used.
- Many usages of the "apply" functions require the user to write little functions (that's why it is important to review functions before working on apply).

Outline

R has lots of ways to do things over and over

- for and while loops: similar to (easier to write than) C and Java
- The R (s)(l)(m)(v)apply family functions: similar to less-well-known languages like Lisp
  - `apply` : for matrices. Process all rows or columns
  - `lapply` : process each element in a list
  - `sapply` : lapply with output simplification
  - `vapply` : improved, safer version of sapply
  - `replicate` : shorthand for sapply for simple simulations
  - `mapply` : for functions that need several arguments, separately drawn from separate vectors or lists
- Today, lets contrast for and lapply

What are the key differences

- Most people will emphasize
  - speed
  - code clarity
- Another important difference is "scope".
  - the apply functions operate in an closure, cannot alter objects in the workspace except by the return value
  - for loop can alter objects in the workspace because its calculations are not done in an enclosed environment.

Outline

for looping

- It is easier to teach this with examples than jargon.
- Example 1. Suppose
  - i is integers 1 through 10
  - x and y are 2 vectors.

```
x <- vector()
y <- vector()
for (i in 1:10) {
    x[i] <- log(i)
    y[i] <- exp(x[i])
 }
cbind(i = 1:10, x, y)
```

for looping ...

```
        i          x  y
 [1,]   1 0.0000000  1
 [2,]   2 0.6931472  2
 [3,]   3 1.0986123  3
 [4,]   4 1.3862944  4
 [5,]   5 1.6094379  5
 [6,]   6 1.7917595  6
 [7,]   7 1.9459101  7
 [8,]   8 2.0794415  8
 [9,]   9 2.1972246  9
[10,]  10 2.3025851 10
```

  - Aha! `exp()` undoes `log()`. HS math was correct.
- Example 2. Suppose
  - `x` already exists
  - The recommended method of creating the index is the `seq_along()` function, saves us the trouble of counting how many elements there are.
  - Because I don't want to convey the impression that the index always has to be called "i", I will name this index "johnelway"

for looping ...

```
x <- log(1:10)
y <- vector()
for (johnelway in seq_along(x)){
    y[johnelway] <- exp(x[johnelway])
 }
cbind(johnelway = seq_along(x), x, y)
```

```
      johnelway         x  y
 [1,]         1 0.0000000  1
 [2,]         2 0.6931472  2
 [3,]         3 1.0986123  3
 [4,]         4 1.3862944  4
 [5,]         5 1.6094379  5
 [6,]         6 1.7917595  6
 [7,]         7 1.9459101  7
 [8,]         8 2.0794415  8
 [9,]         9 2.1972246  9
[10,]        10 2.3025851 10
```

for looping ...

- Nervous people say " `vector()` makes this slower. "We should tell R how many elements there are supposed to be first: `vector(mode = "numeric", length = 10)` ". I agree.
- We can take elements out of R lists with "[[" notation. Suppose
  - `myL` is an R list
  - The individual pieces in which are obtained by writing `myL[[i]]`
  - This function "steps through" the 10 elements and replaces them with something else.

```
myL <- list()
## pretend myL is full of some precious objects
for (i in seq_along(myL)){
   myL[[i]] <- someFunctionYouMakeUp(myL[[i]])
 }
```

for looping ...

Results from `someFunctionYouMakeUp` will replace original values in `myL`

- It is not necessary to obliterate your old list elements. We can create a new list to store the output.

```
newL <- list()
for (i in 1:10){
   newL[[i]] <- someFunctionYouMakeUp(myL[[i]])
 }
```

- The important thing to notice is that the for loop is allowed to write on objects in the global workspace.
- Hence it is a handy way to cycle through a collection of data frames.
- Again, the efficiency experts will criticize this, rightly so. In a big problem, it would be much faster to create with `newL <- vector("list", length = 10)`

Why do for loops have a bad reputation?

- People who are unfamiliar with R think that it is "just like" C or Fortran, in which for loops are fast.
  - they also assume that reading elements with `x[i]`, or writing elements with `x[i] <- 7` runs fine.
- A loopy sort of person would want to write this:

```
## Declare a vector heinz57, do something to each element
heinz57 <- vector(mode = "numeric", length = 57)
for(i in 1:57) {
    heinz57[i] <- log(i)
}
```

- It will be much faster in R to simply write this:

Why do for loops have a bad reputation? ...

```
x1 <- log(1:57)
identical(x1, heinz57)
```

```
[1] TRUE
```

- The difference is in **vectorization**.
  - Repeatedly accessing individual pieces with "[" causes a slowdown.
- The story I tell myself is that the second method "pushes computation into the R compute kernel", while the first method requires "a constant interchange of information between the user workspace (to update heinz57[i]) and the R kernel".
- I'm not against for loops on principle, but only because in practice I find most newcomers cause very slow code if they rely on them.
- Example comparing `ifelse()` function and for loop.

Why do for loops have a bad reputation? ...

- The built-in function `ifelse()` offers a convenient method of recoding a variable.

  ifelse(logical_condition, x, y): if logical is true, return `x`; if not, return `y`.

  This is vectorized, so it can be applied to columns in a data frame, as in

```
dat$z <- ifelse(dat$x1 > dat$y, dat$x1,
   dat$x2)
```

- That is faster than a for loop:

Why do for loops have a bad reputation? ...

```
dat$z2 <- NA
for(i in 1:NROW(dat)){
     dat$z2[i] <- if(dat$x1[i] > dat$y[i]){
         dat$x1[i]
     } else {
         dat$x2[i]
     }
 }
```

- `dat$z2` has to be initialized before the for loop
- And the code is a lot longer, more prone to typographical error

- The loopy approach to R coding it is s-l-o-w because of
  - over-use of "[".
  - failure to "preallocate" structures into which values are being filled.

About pre-allocating memory for storage

- In my R website, I have an example "data_structures-lists" which shows that even if we use a `for` loop, we can speed up the result considerably if we allocate a list of a given size before we use it.
- Example, fill 10,000 matrices into a list. This goes much faster if we do not create the storage list by the lazy way ("`list()`") and instead run this:

```
alist <- vector(mode = "list", length = 10000)
for(i in 1:10000){
     alist2[[i]] <- matrix(rnorm(9), ncol = 3)
 }
```

- Because this grabs storage slots for 10,000 items, it does not have to pause and
  - create a new list with one more element
  - copy the old list members to the new list

  every time it goes through the loop.

Outline

"lapply()": Do same thing to all Elements of a List

- `lapply(someList, someFunction)` will
  1. take a list of things
  2. apply the function to each item
  3. returning a new list as result.
- Use case
  - we have 50 data sets on people in 50 states
  - we have a function that can build a summary tables or plot for each of these
  - we lapply those functions to the list

jumboData example

- Suppose there are 150 data frames saved in a list named `jumboData`. Here is code you can run to actually generate 150 data frames:

```
set.seed(234)
getDF <- function(i) {data.frame(ds = i, x1 =
    rnorm(100), x2 = rnorm(100))}
jumboData <- lapply(1:150, getDF)
```

- This creates the data generator function, and "lapplies" it to 1:150. If you want to, investigate that by looking at individual pieces, `jumboData[[144]]` for example.
- We obtain the means of each one with the built-in function `colMeans()`

```
colMresults <- lapply(jumboData, colMeans)
```

- What did we get?

jumboData example ...

```
is.list(colMresults)
```

```
[1] TRUE
```

```
print(colMresults[[1]])
```

```
         ds          x1           x2
1.000000000  0.004130791 -0.085854129
```

```
print(colMresults[[2]])
```

```
        ds          x1          x2
2.00000000 0.02841649 0.06292265
```

The result is a list, with 150 vectors, each summarizing one of the data frames inside `jumboData`.

- We have many (MANY) ways in R to stack those 150 vectors into a matrix. Here's one:

jumboData example ...

```
colMstacked <- do.call(rbind, colMresults)
dim(colMstacked)
```

```
[1] 150   3
```

```
head(colMstacked)
```

```
     ds          x1           x2
[1,]  1  0.004130791 -0.085854129
[2,]  2  0.028416492  0.062922646
[3,]  3 -0.033087563 -0.041779156
[4,]  4  0.110083615 -0.178247853
[5,]  5 -0.080966386 -0.001748287
[6,]  6 -0.019188038 -0.169436776
```

The use of `do.call` puts this lecture into the intermediate, rather than elementary R user range. I can explain, and point to this example where I learned about it in my WorkingExamples collection: efficiency-stackListItems

Functions that require more arguments

- The simplest example will have 2 arguments, a list and a function name

```
aNewList <- lapply(someList, FUN =
    someFunction)
```

- someFunction **MUST** accept an elements from someList *as the first* argument
- Additional arguments `arg2`, `arg3`, to `someFunction` can be provided like this

```
aNewList <- lapply(someList, FUN =
    someFunction, arg2 = 7, arg3 = 5)
```

but it is required that someFunction's first argument must be filled by the element of `someList`

lapply example with more arguments

- My data generator in previous example did not allow any parameters.
- Here is my new candidate:

```
getDF <- function(i, m1 = 0, m2 = 0, s1 = 1, s2 =
    1) {
     data.frame(ds = i, x1 = rnorm(100, m1, s1),
                       x2 = rnorm(100, m2, s2))}
jumboData <- lapply(1:150, getDF, m1 = 90, s1 =
    10, m2 = 33, s2 = 10)
```

- Lets check the column means first

```
colM3T <- t(sapply(jumboData, colMeans))
colM3T[1:5, ]
```

lapply example with more arguments ...

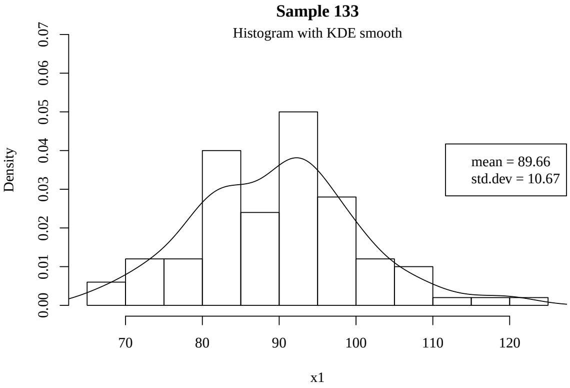
```
      ds       x1       x2
[1,]   1 90.91261 30.94204
[2,]   2 89.98699 34.61032
[3,]   3 87.38849 33.30044
[4,]   4 88.73683 31.78901
[5,]   5 88.99573 33.70151
```

- Pick one data frame for inspection

```
ex133 <- jumboData[[133]]
head(ex133)
```

```
   ds        x1       x2
1 133  92.19604 27.23815
2 133  93.59130 34.71204
3 133  82.58271 53.47827
4 133  78.46469 23.78407
5 133 101.30743 38.52884
6 133  88.72971 19.47875
```

lapply example with more arguments ...



- I had to fight a while to get that legend into shape, and that broke the graph in several ways. Grrr!

lapply example with more arguments ...

```
hist(ex133$x1, xlab = "x1", prob = TRUE, main =
    "Sample 133", ylim = c(0, 0.07))
mtext("Histogram with KDE smooth", 3, -1)
lines(density(ex133$x1))
legend("right", legend = paste(c("mean =",
    "std.dev ="), round(c(mean(ex133$x1),
    sd(ex133$x1, 2)),2)))
```

"sapply()" is only slightly different

- The `colMresults` output from `lapply` is a list with 150 vectors.
- We already found that we can "post process" that list by rbinding the elements into a matrix with `do.call(rbind, colMresults)`. There are, however, one-step solutions.
- We can get back an array if we use `sapply`, or its newer, more-save cousin `vapply`.
- "s" is for "simplify" the result. *Ask R to guess* what each pieces is supposed to give back, then guess how to compactify that.

```
colMresults2 <- sapply(jumboData, colMeans)
dim(colMresults2)
```

```
[1]   3 150
```

"sapply()" is only slightly different ...

```
## That's 150 columns with 3 rows each. The first
   3 columns
colMresults2[ , 1:3]
```

```
        [,1]     [,2]     [,3]
ds   1.00000  2.00000  3.00000
x1  90.91261 89.98699 87.38849
x2  30.94204 34.61032 33.30044
```

The return is a matrix that has one column for each of the input data frames.

- The result seems "sideways".
- I would rather have that information transposed, so I use `t()`

```
colMresults2T <- t(colMresults2)
head(colMresults2T)
```

"sapply()" is only slightly different ...

```
      ds       x1       x2
[1,]   1 90.91261 30.94204
[2,]   2 89.98699 34.61032
[3,]   3 87.38849 33.30044
[4,]   4 88.73683 31.78901
[5,]   5 88.99573 33.70151
[6,]   6 91.24301 34.07683
```

vapply() is safer version of sapply()

- In *Advanced R*, Wickham makes a good argument that `sapply` should not be used in functions or long scripts because it may guess incorrectly about return values
- `vapply` is a similar/newer version. We must specify the structure expected from the return.

```
colMresults3 <- vapply(jumboData, colMeans,
    numeric(3))
## 3rd argument gives structure required in
    output from colMeans
str(colMresults3)
```

```
num [1:3, 1:150] 1 90.9 30.9 2 90 ...
- attr, "dimnames")=List of 2..: chr[1:3]"ds""x1""x2".. :  NULL
```

- Ach! Output is sideways again.

vapply() is safer version of sapply() ...

- The output has 150 columns, too wide to show here. But we can peek at the first 5 columns

```
colMresults3[ , 1:5]
```

```
        [,1]     [,2]     [,3]     [,4]     [,5]
ds  1.00000  2.00000  3.00000  4.00000  5.00000
x1 90.91261 89.98699 87.38849 88.73683 88.99573
x2 30.94204 34.61032 33.30044 31.78901 33.70151
```

If you want more about iterators

- In 2013, I wrote a longer presentation, from which about 10% of this presentation is taken
- There are two large worked out examples of simulations using `lapply`

iteration-1.pdf

Outline

Bootstrapping: Some "Do it Yourself" Work Is Required

- Many R functions require users to write little functions that do little things.
- In many cases (like lapply or apply), look for FUN as an argument.
- Sometimes no builtin-exists. useR must create!

boot Function Requires a Special Function "statistic"

```
library(boot)
?boot
```

```
Bootstrap Resampling

Description:

Generate 'R' bootstrap replicates of a statistic applied to data.
Both parametric and nonparametric resampling are possible. ...

boot(data, statistic, R, sim = ''ordinary'', stype = ''i'',
strata=rep(1, n), L = NULL, m = 0, weights = NULL,
ran.gen=function(d, p) d, mle = NULL, simple = FALSE, ...)

statistic: A function which when applied to data returns a vector
containing the statistic(s) of interest...
```

Bootstrap: Background Explanation

- Bootstrap: draw samples repeatedly and re-estimate θ
- Resulting values approximate a sampling distribution θ
- The "boot" package asks for a data frame and a special function "statistic". statistic must
  - accept a data frame as the first argument
  - accept an "index vector" as the second argument

Don't Panic: This is Confusing to Everybody

This is your DF

index	x1	x2	x3	x4
1	8	1		
2	9	0		
3	8	0	$\ddots$	
4	9	1		
5	7	0		$\ddots$
7	8	1		
8	7	0		
9	6			
10	9			

- All the iterations are the same, they just use different row subsets
- boot will choose a set of rows, say "c(1, 6, 8, 10)". Your statistic function is supposed to do the right thing with the data subset.
  - X[c(1, 6, 8, 10),]
- Then boot re-draws an index, "c(3, 5, 7, 9)".
- Then analysis happens with:
  - X[c(3, 5, 7, 9),]
- Over and over

Example usage

```
boot(data, statistic = yourFunction, R = 1000)
```

- boot will iterate 1000 times, and `yourFunction` will provide the statistic of interest.
- You write yourFunction to make required calculation.
- boot will tell yourFunction which lines to use in the data frame, *over-and-over*.

The Median of a Poisson Distribution

- Suppose you have a sample from a Poisson Process:

```
samp <- rpois(20, lambda=3)
```

- And you calculate the median:

```
median(samp)
```

```
[1] 3
```

- How confident are you in that estimate of the median?

Bootstrap Your Median

- Here is yourFunction, it takes just a column vector as input:

```
calcMed <- function(x, ind){
    median(x[ind])
 }
```

- x[ind] has the effect of "pulling" rows that match "ind" from "x"
- The boot function will send 1000 "case indexes" to your function.

```
library(boot)
bpois <- boot(samp, calcMed, R = 1000)
bpois
```

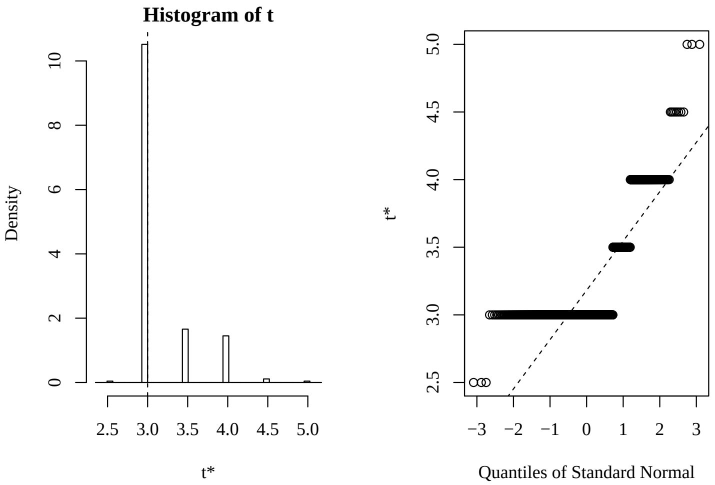
Bootstrap Your Median ...

```
ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = samp, statistic = calcMed, R = 1000)


Bootstrap Statistics :
    original  bias    std. error
t1*        3  0.1815   0.3646103
```

The plot method for boot output



Why Do They Do It That Way?

- Your instinct is to do this the "simple" way
  - (Just) "Manually" draw new random samples of rows from a data frame.
  - But: Creating 1000s of "new" re-sampled data sets would "waste" (exhaust?) memory
  - Would be especially slow if separate data sets have to be copied between systems.
- More efficient to keep 1 data frame, but 1000's of vectors of row numbers.

Outline

Balancing Speed and Comprehension

- I'm not divorced from `for` loops. But I recognize that vectorization is always faster, if we can use it.
- If one is patient with the manuals and documentation, the usage of `lapply`, `vapply`, and `boot` can be elegant, fast.
- If one is impatient, and treats R code as if it were intended for C or fortran, one might have code that is
  - done more quickly
  - harder to debug
  - runs more slowly

References

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Session

```
sessionInfo()
```

```
R version 3.4.4 (2018-03-15)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 18.04 LTS

Matrix products: default
BLAS: /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
      LC_TIME=en_US.UTF-8
 [4] LC_COLLATE=en_US.UTF-8     LC_MONETARY=en_US.UTF-8
      LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C
[10] LC_TELEPHONE=C             LC_MEASUREMENT=en_US.UTF-8
    LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  base

other attached packages:
[1] boot_1.3-20
```

Session ...

```
loaded via a namespace (and not attached):
[1] compiler_3.4.4 tools_3.4.4
```