R Cheat Sheet: Avoiding For-Loops

What is wrong with using for-loops?

Nothing! R's (for-while-repeat) loops are intuitive, and easy to code and maintain. Some tasks are best managed within loops.

So why discourage the use of for-loops?

1) Side effects and detritus from inline code. Replacing a loop with a function call means that what happened in the function stayed in the function. 2) In some cases increased speed (especially so with nested loops and from poor loop-coding practice).

How to make the paradigm shift?

1) Use R's vectorisation features. 2) See if object indexing and subset assignment can replace the for-loop. 3) If not, find an "apply" function that slices your object the way you need. 4) Find (or write) a function to do what you would have done in the body of the for-loop. Anonymous functions can be very useful for this task. 5) if all else fails: move as much code as possible outside of the loop body

Play data (for the examples following)

```
require('zoo'); require('plyr'); n <- 100;
u <- 1:n; v <- rnorm(n, 10, 10) + 1:n
w <- round(runif(n, 0.6, 9.4)) #min=1 max=9
df <- data.frame(month=u, x=u, y=v, z=w)
l <- list(x=u, y=v, z=w, yz=v*w, xyz=u*v*w)
trivial.add <- function(a, b) { a + b }
```

Use R's vectorisation features

```
tot <- sum(log(u))   # replaces the C-like:
# tot <- 0;                          # YUK
# for(i in seq_along(u))             # YUK
#     tot <- tot + log(u[i])         # YUK
```

Clever indexing and subset assignment

```
df[df$z == 5, 'y'] <- -1        # replaces:
# for(row in seq_len(nrow(df)))       # YUK
#    if(df[row, 'z'] == 5)            # YUK
#        df[row, 'y'] <- -1           # YUK
df[is.na(df)] <- 0 # remove NAs from the df
```

The base apply family of functions

```
# apply(X, MARGIN, FUN, ...)
# lapply(X, FUN, ...)
# sapply(X, FUN, ...) # has more options
# vapply(X, FUN, FUN.VALUE, ...) # ditto
# tapply(X, INDEX, FUN = NULL, ...) # "
# mapply(FUN, ..., MoreArgs = NULL) # "
# eapply(env, FUN, ...) # has more options
# replicate(n, expr, simplify = "array")
# by(data, INDICES, FUN, ...) # more opts
# aggregate(x, by, FUN, ...) # for a df
# rapply() # see help for options!?
```

lapply (on vector or list, return list)

```
lapply(l, mean) # returns a list of means
unlist( lapply(u, trivial.add, 5) )
# Last case: vapply() or sapply() better
```

sapply (a simplified lapply on v or l)

```
# Object: v, l; Returns: usually a vector
sapply(l, mean) # returns a vector
sapply(u, function(a) a*a) # vec of squares
sapply(u, trivial.add, -1) # function above
```

tapply (group v/l by factor & apply fn)

```
count.table <- tapply(v, w, length)
min.1 <- with(df, tapply(y, z, min))
```

by (on l or v, returns "by" objects)

```
min.2 <- by(df$y, df$z, min) # like above
min.3 <- by(df[, c('x', 'y')], df$z, min)
# last one: finds min from two columns
```

aggregate

```
ag <- aggregate(df, by=list(df$z), mean)
aggregate(df, by=list(w, 1+(u%%12)), mean)
# Trap: variables must be in a list
```

apply (by row/column on two+ dim object)

```
# Object: m, t, df, a (has 2+ dimensions)
# Returns: v, l, m (depends on input & fn)
column.mean <- apply(df, 2, mean)
row.product <- apply(df, 1, prod)
# Traps: apply coerces a df to a matrix to
#        do its magic. Col names are lost.
```

rollapply - from the zoo package

```
# A 5-term, centred, rolling average
v.ma5 <- rollapply(v, 5, mean, fill=NA)
# Sum 3 months data for a quarterly total
v.qtrly <- rollapply(v, 3, sum, fill=NA,
            align='right') # align window
# Note: zoo has rollmean(), rollmax() and
# rollmedian() functions
```

Inside a data.frame

```
# Use transform() or within() to apply a
# function to a column in a data.frame. Eg:
df <- within(df, v.qtrly <- rollapply(v,
        3, sum, fill=NA, align='right'))
# use with() to simplify column access
```

The plyr package

Plyr is a fantastic family of apply like functions with a common naming system for the input-to and output-from split-apply-combine procedures. I use ddply() the most.

```
# ddply(.data, .var, .fun=NULL, ...)
ddply( df, .(z), summarise, min = min(y),
    max = max(y) )
ddply( df, .(z), transform, span = x - y )
```

Other packages worth looking at

```
# foreach - a set of apply-like fns
# snow - parallelised apply-like functions
# snowfall - a usability wrapper for snow
```

Abbreviations

v=vector, l=list, m=matrix, df=data.frame, a=array, t=table, f=factor, d=dates

