R Cheat Sheet: Avoiding For-Loops

What is wrong with using for-loops? sapply (a simplified lapply on v or l)
Nothing! R's (for-while-repeat) loops are # Object: v, l; Returns: usually a vector
intuitive, and easy to code and maintain. sapply(l, mean) # returns a vector
Some tasks are best managed within loops. sapply(u, function(a) a*a) # vec of squares
sapply(u, trivial.add, -1) # function above
So why discourage the use of for-loops?
1) Side effects and detritus from inline tapply (group v/l by factor & apply fn)
code. Replacing a loop with a function call count.table <- tapply(v, w, length)
means that what happened in the function min.1 <- with(df, tapply(y, z, min))
stayed in the function. 2) In some cases
increased speed (especially so with nested by (on l or v, returns "by" objects)
loops and from poor loop-coding practice). min.2 <- by(dfy, dfz, min) # like above
min.3 <- by(df[, c('x', 'y')], df$z, min)
How to make the paradigm shift? # last one: finds min from two columns
1) Use R's vectorisation features. 2) See
if object indexing and subset assignment aggregate
can replace the for-loop. 3) If not, find ag <- aggregate(df, by=list(df$z), mean)
an "apply" function that slices your object aggregate(df, by=list(w, 1+(u%%12)), mean)
the way you need. 4) Find (or write) a # Trap: variables must be in a list
function to do what you would have done in
the body of the for-loop. Anonymous apply (by row/column on two+ dim object)
functions can be very useful for this task. # Object: m, t, df, a (has 2+ dimensions)
5) if all else fails: move as much code as # Returns: v, l, m (depends on input & fn)
possible outside of the loop body column.mean <- apply(df, 2, mean)
row.product <- apply(df, 1, prod)
Play data (for the examples following) # Traps: apply coerces a df to a matrix to
require('zoo'); require('plyr'); n <- 100; # do its magic. Col names are lost.
u <- 1:n; v <- rnorm(n, 10, 10) + 1:n
w <- round(runif(n, 0.6, 9.4)) #min=1 max=9 rollapply – from the zoo package
df <- data.frame(month=u, x=u, y=v, z=w) # A 5-term, centred, rolling average
l <- list(x=u, y=v, z=w, yz=v*w, xyz=u*v*w) v.ma5 <- rollapply(v, 5, mean, fill=NA)
trivial.add <- function(a, b) { a + b } # Sum 3 months data for a quarterly total
v.qtrly <- rollapply(v, 3, sum, fill=NA,
Use R's vectorisation features align='right') # align window
tot <- sum(log(u)) # replaces the C-like: # Note: zoo has rollmean(), rollmax() and
# tot <- 0; # YUK # rollmedian() functions
# for(i in seq_along(u)) # YUK
# tot <- tot + log(u[i]) # YUK Inside a data.frame
# Use transform() or within() to apply a
Clever indexing and subset assignment # function to a column in a data.frame. Eg:
df[df$z == 5, 'y'] <- -1 # replaces: df <- within(df, v.qtrly <- rollapply(v,
for(row in seq_len(nrow(df))) # YUK 3, sum, fill=NA, align='right'))
if(df[row, 'z'] == 5) # YUK # use with() to simplify column access
df[row, 'y'] <- -1 # YUK
df[is.na(df)] <- 0 # remove NAs from the df The plyr package
Plyr is a fantastic family of apply like
The base apply family of functions functions with a common naming system for
apply(X, MARGIN, FUN, ...) the input-to and output-from split-apply-
lapply(X, FUN, ...) combine procedures. I use ddply() the most.
sapply(X, FUN, ...) # has more options # ddply(.data, .var, .fun=NULL, ...)
vapply(X, FUN, FUN.VALUE, ...) # ditto ddply(df, .(z), summarise, min = min(y),
tapply(X, INDEX, FUN = NULL, ...) # " max = max(y))
mapply(FUN, ..., MoreArgs = NULL) # " ddply(df, .(z), transform, span = x - y)
eapply(env, FUN, ...) # has more options
replicate(n, expr, simplify = "array") Other packages worth looking at
by(data, INDICES, FUN, ...) # more opts # foreach – a set of apply-like fns
aggregate(x, by, FUN, ...) # for a df # snow - parallelised apply-like functions
rapply() # see help for options!? # snowfall – a usability wrapper for snow

lapply (on vector or list, return list) Abbreviations
lapply(l, mean) # returns a list of means v=vector, l=list, m=matrix, df=data.frame,
unlist(lapply(u, trivial.add, 5)) a=array, t=table, f=factor, d=dates
Last case: vapply() or sapply() better