R Programming Week 3

Rudy Rupp

13/10/2020

R Programming Week 3: Loop Functions and Debugging

Writing *for* and *while* loops in the command line is cumbersome. The loop functions that contain the word *"apply"* are more convenient. We'll look into several variants.

Learning Objectives
- Use the loop functions *apply*, *lapply*, *tapply*, *mapply* and *split*.
- Define an anonymous function and describe its use in loop functions.
- Describe how to start the R debugger for an arbitrary R function.
- Describe what the traceback() function does and what is the function call stack.

Loop Functions - *lapply*

Loop over a list of objects and apply a function to every element. *sapply* does the same and tries to simplify the results.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x0000000011ee7b98>
## <environment: namespace:base>
```

- *"X"* must be a list or it will be coerced into a list.
- *"FUN"* is a function to apply and "..." are arguments of that function.
- the actual looping is done internally through C code, to make it faster.
- always returns a list. Names are preserved.

```
## Example
x <- list(a = 1:5, b = rnorm(10), c = rnorm(5, 10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.2518791
##
## $c
## [1] 9.748077
```

```
## runif() function generates "n" random numbers between 0 and 1
lapply(1:4, runif)
```

```
## [[1]]
## [1] 0.03315659
##
## [[2]]
## [1] 0.8083147 0.3319269
##
## [[3]]
## [1] 0.8941805 0.8743410 0.2460053
##
## [[4]]
## [1] 0.93040917 0.16453165 0.09571097 0.25044831
```

```
## optional arguments of runif() can be passed into the "..." argument
lapply(1:4, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 6.769555
##
## [[2]]
## [1] 9.682834 7.044356
##
## [[3]]
## [1] 7.185098 8.549528 6.134958
##
## [[4]]
## [1] 0.0312126 8.0916615 8.9492602 2.7501910
```

Using loop functions with anonymous functions

Anonymous functions can be very useful in combination with *apply* loops.

```
## Consider a list of matrices
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
##
## $b
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
## we can create an anonymous function to extract the first column of each
## matrix
lapply(x, function(arg) arg[ ,1])
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

***sapply* will try to simplify the result of lapply**

- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector with the same length, a matrix is returned.
- If sapply can't simplify the output, a list is returned.

```
x <- list(a = 1:5, b = rnorm(10), c = rnorm(5, 10))
sapply(x, mean)  ##returns a vector instead of a list
```

```
##         a         b         c
## 3.0000000 0.3747197 9.7606158
```

```
mean(x) ## doesn't work because mean doesn't apply to lists.
```

```
## Warning in mean.default(x): argument is not numeric or logical: returning NA
```

```
## [1] NA
```

```
x <- list(a = 1:5, b = rnorm(10), c = rnorm(5, 10), d = c("a", "b", "c"))
y <- sapply(x, mean)  ## calculates means where possible and returns NA
```

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA
```

```
y                    ## otherwise
```

```
##          a          b          c          d
##  3.0000000 -0.2734876 10.0587484         NA
```

```r
class(y)
```

```
## [1] "numeric"
```

Loop functions - *apply*

Used to evaluate a function over the margins of an array (matrices or 3d arrays). It used to be faster than using a a for loop, but that's no longer the case. It just requires less typing.

```r
args(apply)
```

```
## function (X, MARGIN, FUN, ...)
## NULL
```

- *"X"* is an array (e.g. a matrix).
- *"MARGIN"* is an integer vector indicating which margins should be retained.
- *"FUN"* is a function.
- *"..."* is for arguments to be passed to the function.

```r
x <- matrix(rnorm(200), 20, 10)  ## an example matrix

## Calculate the mean of each column
## dimension 1 has 20 rows and dimension 2 has 10 columns.
apply(x, 2, mean) ## preserve columns, collapse rows
```

```
##  [1] -0.31711667 -0.29447993 -0.42585033 -0.03082308  0.02077809  0.09387892
##  [7]  0.10289796 -0.30144571  0.24603824  0.21823230
```

```r
## Calculate the sum of each row
apply(x, 1, sum) ## preserve rows, collapse columns.
```

```
##  [1] -1.3019980 -2.0117510 -4.0372581  1.7696536 -1.6381478 -0.1348511
##  [7]  1.5184458 -2.7644167 -1.7894153  0.7569512  4.0495947 -2.8022460
## [13]  0.6852174  1.4408326 -2.7872621  0.5776152  1.9923264 -4.4947406
## [19]  2.0637829 -4.8501374
```

For sums and means of matrix dimensions, however, there are faster, optimized functions:
- *rowSums()* = apply(x, 1, sum)
- *rowMeans()* = apply(x, 1, mean)
- *colSums()* = apply(x, 2, sum)
- *colMeans()* = apply(x, 2, mean)

For functions that return a vector of length > 1, *apply()* can create a matrix.

```r
## Another example
## signif() is used to round to significant figures.
x <- matrix(signif(rnorm(50, 10, 5),3), 5, 10)
x
```

```
##       [,1]  [,2] [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
## [1,] 13.40  9.91 15.0 14.70 12.90 13.00 15.60 15.00  7.07  10.2
## [2,] 13.20  7.67 13.3  7.24 17.40  9.21 12.40  8.17  7.47  11.6
## [3,]  8.20 15.10 10.4  7.34  5.26  7.91 14.00 14.80  3.86   9.5
## [4,]  3.04  9.87 15.1  7.00  4.97  1.42  3.16 14.60 11.10  12.9
## [5,] 14.60 19.80 13.9  7.28  3.92  5.19 11.30 12.50 19.30  13.5
```

```
## the quantile() function calculates percentiles in a population.
apply(x, 1, quantile, probs = c(0, 0.5, 1))
```

```
##       [,1]   [,2]  [,3]   [,4]  [,5]
## 0%    7.07  7.240  3.86  1.420  3.92
## 50%  13.20 10.405  8.85  8.435 13.00
## 100% 15.60 17.400 15.10 15.100 19.80
```

Each row of the matrix is analysed by the quantile function, which gives a vector of length 3. So *apply()* builds a matrix with the rows of "x" as the column names and the quantiles that the *quantile()* function calculates as the rows.

Using *apply()* in an array:

```
## a is an array of dimensions 2 by 2 by 10
a <- array(signif(rnorm(2 * 2 * 10), 2), c(2, 2, 10))

## preserve dimensions 1 and 2, operate the function of the 3rd dimension.
apply(a, c(1, 2), mean)
```

```
##        [,1]   [,2]
## [1,] -0.072 -0.080
## [2,]  0.098  0.171
```

```
## can also be achieved by:
rowMeans(a, dims = 2)
```

```
##        [,1]   [,2]
## [1,] -0.072 -0.080
## [2,]  0.098  0.171
```

Loop functions - *mapply*

is a multivariate apply. It can apply a function in parallel over a set of arguments. It can have multiple lists as arguments.

```
args(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
## NULL
```

- *"FUN"* is a function

- *"..."* contains arguments to apply to the function. (coerced to list)
- *"MoreArgs"* is a list of other arguments to FUN.
- *"SIMPLIFY"* indicates whether the result should be simplified.

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

```
## instead, we can have mapply use the rep() function
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

```
noise <- function(n, mean = 5, sd = 2) {
        signif(rnorm(n, mean, sd), 2)
}

## If I wanted to create sets of 6 to 10 random numbers with noise(), each
## set having average 2, 4, 6, 8 and 10 respectively.
## noise(6:10, c(2, 4, 6, 8, 10), 2) won't work because the arguments must be single elements.
## Using mapply, we can do this:
mapply(noise, n = 6:10, mean = c(2, 4, 6, 8, 10), sd = 2)
```

```
## [[1]]
## [1] -1.1  2.1  4.0  2.7  4.0  2.7
##
## [[2]]
## [1] 6.2 3.3 5.3 1.5 5.2 7.3 6.0
##
## [[3]]
```

```
## [1]  5.2  2.9  5.6  7.3  7.9  4.4  3.0 10.0
##
## [[4]]
## [1]  6.5 11.0  5.7  6.8 11.0  5.1  3.5  7.3  8.7
##
## [[5]]
##  [1] 10.0 10.0  6.5  9.7 13.0 15.0 11.0 13.0 11.0  8.0
```

Use a function that doesn't allow vector inputs and vectorise it.

Loop Functions - *tapply*

used to apply a function over subsets of a vector.

```
args(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
## NULL
```

- *"X"* is a vector.
- *"Index"* is a vector of same length or a list of factors (coerced into list).
- *"FUN"* is a function.
- *"..."* contains other arguments of *FUN*.
- *"simplify"* to simplify the result.

```
x <- c(rnorm(10), runif(10), rnorm(10,1)) ## runif is random of uniform dist.
f <- gl(3, 10)  ## generates a numeric vector, elements are 1 repeated k times,
                ## then 2 repeated k times, then 3, until the number is equal
                ## to n.
f
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
## Levels: 1 2 3
```

```
tapply(x, f, mean)
```

```
##         1         2         3
## 0.6974091 0.4812648 0.8028691
```

- the factor variable determines which elements of the vector will be grouped together.

```
tapply(x, f, range)
```

```
## $`1`
## [1] -1.293618  2.439859
##
```

```
## $`2`
## [1] 0.04905206 0.95447028
##
## $`3`
## [1] -1.231909  2.216888
```

- Here a minimum and a maximum are given for each group, so the result of *tapply* is a list where each element is a vector of length 2.

Loop Functions - *split*

Works very similarly to *tapply*, but it doesn't apply a function to the elements.

```
args(split)
```

```
## function (x, f, drop = FALSE, ...)
## NULL
```

- useful for using before *lapply* or *mapply*
- can be used to split much more complicated objects.
- always results in a list.

```
x <- c(rnorm(10), runif(10), rnorm(10,1))
f <- gl(3, 10)
split(x, f)
```

```
## $`1`
##  [1]  0.03010107  0.04065041 -0.98287600  0.48786172  0.31944439  0.67989656
##  [7]  2.08431053 -0.40198235  1.07840751 -0.75061426
##
## $`2`
##  [1] 0.3980408 0.6863194 0.8799792 0.8338869 0.1626626 0.5308943 0.6361636
##  [8] 0.8092962 0.2918799 0.1712100
##
## $`3`
##  [1]  0.71692261  3.26375365  1.01289939  1.50188407 -0.08173649 -0.08163282
##  [7]  0.88200235  0.97334811  2.49410539  1.92645905
```

```
lapply(split(x, f), mean)  ## exactly same thing as the tapply function
```

```
## $`1`
## [1] 0.25852
##
## $`2`
## [1] 0.5400333
##
## $`3`
## [1] 1.260801
```

```r
library(datasets)
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```r
## For example if I wanted to calculate the mean of Ozone, solar radiation,
## wind and temperature within each month. I can split the data frame into
## monthly pieces.
s <- split(airquality, airquality$Month)
lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

```
## $`5`
##    Ozone  Solar.R     Wind
##       NA       NA 11.62258
##
## $`6`
##     Ozone   Solar.R      Wind
##        NA 190.16667  10.26667
##
## $`7`
##      Ozone    Solar.R       Wind
##         NA 216.483871   8.941935
##
## $`8`
##    Ozone  Solar.R     Wind
##       NA       NA 8.793548
##
## $`9`
##    Ozone  Solar.R     Wind
##       NA 167.4333  10.1800
```

```r
## includes NAs
```

I can then also use *sapply* to give the results a a matrix instead:

```r
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

```
##                5         6          7        8        9
## Ozone         NA        NA         NA       NA       NA
## Solar.R       NA 190.16667 216.483871       NA 167.4333
## Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

```r
## and I can remove NAs with the 'na.rm' argument of colMeans

sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
```

```
##                5         6          7          8         9
## Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
## Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
## Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

Splitting on more than one level

Using two different factors for splitting.

```
x <- rnorm(10)
f1 <- gl(2, 5)
f2 <- gl(5, 2)

## For example if I have a population of measurement and they are tagged
## by both gender and age group.
cbind(x, f1, f2)
```

```
##                x f1 f2
##  [1,] -1.2120558  1  1
##  [2,] -1.5542078  1  1
##  [3,]  1.6840945  1  2
##  [4,] -0.2000256  1  2
##  [5,]  0.0384198  1  3
##  [6,]  0.1348201  2  3
##  [7,] -0.6256178  2  4
##  [8,] -1.3289031  2  4
##  [9,]  1.5861811  2  5
## [10,]  1.2968053  2  5
```

```
interaction(f1, f2)  ## Combines the levels of both factors. There's now 10.
```

```
##  [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
## Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

```
str(split(x, list(f1, f2)))
```

```
## List of 10
##  $ 1.1: num [1:2] -1.21 -1.55
##  $ 2.1: num(0)
##  $ 1.2: num [1:2] 1.68 -0.2
##  $ 2.2: num(0)
##  $ 1.3: num 0.0384
##  $ 2.3: num 0.135
##  $ 1.4: num(0)
##  $ 2.4: num [1:2] -0.626 -1.329
##  $ 1.5: num(0)
##  $ 2.5: num [1:2] 1.59 1.3
```

```
## the split function doesn't require the use of the interaction function.
## Not all levels have observations, they result in an empty vector.

str(split(x, list(f1, f2), drop = TRUE))  ## drops empty levels from the result
```

```
## List of 6
##  $ 1.1: num [1:2] -1.21 -1.55
##  $ 1.2: num [1:2] 1.68 -0.2
##  $ 1.3: num 0.0384
##  $ 2.3: num 0.135
##  $ 2.4: num [1:2] -0.626 -1.329
##  $ 2.5: num [1:2] 1.59 1.3
```

```
## very useful for later using an sapply or lapply.
```

Debugging Tools - Diagnosing the Problem

Indication R will give that something's wrong:

- **message**: an expected notification/diagnostic produced by the *message* function; execution of the function continues. Message appears in the middle.
- **warning**: an indication of something wrong, but not fatal. execution of the function continues. generated by the *warning* function. Message appears after the function.
- **error**: a fatal problem has occurred. execution stops. produced by the *stop* function.
- **condition**: a generic concept for indicating that something unexpected can occur. programmer can create their own conditions.

```
log(-1)  ## Basic warning.
```

```
## Warning in log(-1): NaNs produced
```

```
## [1] NaN
```

```
printmessage <- function(x) {
        if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than of equal to zero")
        invisible(x)  ## prevents auto-printing
}
printmessage(1)
```

```
## [1] "x is greater than zero"
```

```
printmessage(NA)  ## error produced!
```

```
## Error in if (x > 0) print("x is greater than zero") else print("x is less than of equal to zero"): m
```

```
printmessage2 <- function(x) {
        if(is.na(x))
                print("x is a missing value!")
        else if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than of equal to zero")
        invisible(x)
}
x <- log(-1)
```

```
## Warning in log(-1): NaNs produced
```

```
printmessage2(x) ## no error produced, but unexpected.
```

```
## [1] "x is a missing value!"
```

How do you know that something is wrong with your function?

- What was your input? How did you call the function? Arguments, etc?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does it differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem?

Debugging Tools - Basic Tools

The primary tools for debugging functions in R:

- *traceback*: prints out the function call stack after an error occurs: does nothing if there is no error.
- *debug*: flags a function for "debug mode". when the function is next called, it will freeze and you can step through the function line-by-line.
- *browser*: similar to *debug*. when read, execution will suspend and you can go line-by-line. Can stop the function in the middle, instead of the start.
- *trace*: allows you to insert debugging code into a function without changing the function.
- *recover*: allows you to modify the default error behaviour, so you can browse the function call stack.

Debugging Tools - Using the Tools (examples)

Traceback

```
mean(xxx)
```

```
## Error in mean(xxx): object 'xxx' not found
```

```
## traceback()  ## has to be called immediately after error occurs
```

```
lm(xxx ~ yyy)
```

```
## Error in stats::model.frame(formula = xxx ~ yyy, drop.unused.levels = TRUE): object 'xxx' not found
```

```
## traceback()  ## error actually occurred 4 levels deep.
```

Useful for giving details of errors to others, when asking for help.

Debug

debug is hard to see in static form, but here's an example:

```
# debug(lm)  ## prints out the full body of the next function,
           ## then opens the browser.
lm(xxx ~ yyy)
```

```
## Error in stats::model.frame(formula = xxx ~ yyy, drop.unused.levels = TRUE): object 'xxx' not found
```

The browser acts just like a separate workspace. The environment of this workspace is the environment of the function. Type *n* and enter to run the next line. You can then use debug on a function inside that function to browse that function.

Recover

```
# options(error = recover)
  ## Sets a global option for this R session.

read.csv("nosuchfile")
```

```
## Warning in file(file, "rt"): cannot open file 'nosuchfile': No such file or
## directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

Instead of getting the console back, you get a menu. This is similar to the output of the traceback function. Error occurred at 3rd level, so you can type 3 to browse the environment of that function.

Swirl Exercises Week 3

Swirl Lesson 10: lapply and sapply

These functions offer a concise and convenient means of implementing the **Split-Apply-Combine strategy** for data analysis. That means, the function will *split* up the data into smaller pieces, *apply* a function to each piece, then *combine* the results.

Consider a data frame with several columns representing measured variables and rows representing observations (for example countries where the measurements were taken). Suppose we want to know the class of object of each column. We could individually look it up with a subsetting function, but since we need to do this for many columns, we can use *lapply* to do it as a loop for all of them. *lapply(dataframe, class)* command will give a list of all the column names and the class of the object. *sapply* does the same, but, if possible, simplifies the result into a vector or matrix instead of a list.

Swirl Lesson 11: vapply and tapply

vapply works similarly to *sapply*, however you specify the object class you want as an output. If the function cannot convert the object into that class, it throws an error. This can prevent significant problems in code, that might be caused by getting an unexpected return value from *sapply*.

```
x <- data.frame(name = c("a", "b", "c", "d", "e"), min = runif(5, 0, 10), max = runif(5, 10, 200), tree
x
```

```
##   name       min       max  tree num
## 1    a 1.3115975 198.60769  TRUE   1
## 2    b 1.1417039  49.72381 FALSE   2
## 3    c 9.1919985 129.66185 FALSE   3
## 4    d 0.6675367 139.75730 FALSE   4
## 5    e 8.9071517  46.63994 FALSE   5
```

```
sapply(x, class)
```

```
##        name         min         max        tree         num
## "character"   "numeric"   "numeric"   "logical"   "integer"
```

```
vapply(x, class, character(1))
```

```
##        name         min         max        tree         num
## "character"   "numeric"   "numeric"   "logical"   "integer"
```

```
vapply(x, class, numeric(1))  ## error produced because output is not numeric
```

```
## Error in vapply(x, class, numeric(1)): values must be type 'double',
##  but FUN(X[[1]]) result is type 'character'
```

tapply splits the data up into groups, based on the value of a variable vector, then applies a function to each group.

```
x <- data.frame(fus = runif(30), roh = runif(30), dah = runif(30), dborn = gl(5, 6))
x
```

```
##           fus         roh         dah dborn
## 1  0.63346283 0.766646176 0.867128685     1
## 2  0.87257336 0.810512892 0.531337052     1
## 3  0.35319404 0.462969169 0.357239921     1
## 4  0.93961959 0.199853880 0.349744707     1
## 5  0.38607107 0.413154688 0.407286936     1
## 6  0.41952201 0.661327417 0.089264803     1
## 7  0.40233481 0.850871060 0.821538817     2
## 8  0.39239277 0.309023781 0.512897918     2
## 9  0.98001009 0.271726998 0.092222655     2
## 10 0.59124450 0.772949778 0.229891769     2
## 11 0.41082232 0.364996275 0.289829747     2
## 12 0.46766159 0.108976348 0.003256499     2
## 13 0.64735854 0.373825822 0.362571624     3
## 14 0.98936360 0.807769592 0.452850503     3
## 15 0.27812069 0.164341210 0.162833590     3
## 16 0.94809847 0.038464583 0.358747313     3
## 17 0.44864570 0.666225322 0.833050501     3
## 18 0.90898390 0.496261382 0.897057120     3
## 19 0.84519519 0.902760738 0.462128564     4
## 20 0.64579888 0.184461686 0.056073229     4
## 21 0.63135469 0.315118599 0.995264417     4
## 22 0.27766701 0.284037010 0.082959496     4
## 23 0.25499154 0.006580168 0.404383147     4
## 24 0.48921562 0.878033543 0.773876353     4
## 25 0.90477597 0.025854340 0.596415250     5
## 26 0.07009326 0.864312863 0.082043743     5
## 27 0.33796949 0.073303067 0.297872172     5
## 28 0.26295637 0.477888268 0.484710380     5
## 29 0.81332179 0.981796233 0.884145501     5
## 30 0.58999274 0.467699711 0.856112700     5
```

```
tapply(x$fus, x$dborn, mean)
```

```
##         1         2         3         4         5
## 0.6007405 0.5407443 0.7034285 0.5240372 0.4965183
```

```
  ## takes the mean of column "fus", but separated into the groups set
  ## by "dborn"
```

Quiz Week 3

Take a look at the 'iris' dataset that comes with R. The data can be loaded with the code:

```
rm(list = ls())
library(datasets)
data(iris)
data(mtcars)
```

Q: There will be an object called 'iris' in your workspace. In this dataset, what is the mean of 'Sepal.Length' for the species virginica? Please round your answer to the nearest whole number.
A: 7

Q: Continuing with the 'iris' dataset from the previous Question, what R code returns a vector of the means of the variables 'Sepal.Length', 'Sepal.Width', 'Petal.Length', and 'Petal.Width'?
A: apply(iris[, 1:4], 2, mean)

Q: Load the 'mtcars' dataset in R with the following code. There will be an object names 'mtcars' in your workspace. How can one calculate the average miles per gallon (mpg) by number of cylinders in the car (cyl)? Select all that apply.
A:

```
sapply(split(mtcars$mpg, mtcars$cyl), mean)
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

```
tapply(mtcars$mpg, mtcars$cyl, mean)
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

Q: Continuing with the 'mtcars' dataset from the previous Question, what is the absolute difference between the average horsepower of 4-cylinder cars and the average horsepower of 8-cylinder cars? (Please round your final answer to the nearest whole number.
A: 127

Q: If you run *debug(ls)* what happens when you next call the 'ls' function?
A: Execution of 'ls' will suspend at the beginning of the function and you will be in the browser.

Week 3 Assignment

Matrix inversion is usually a costly computation and there may be some benefit to caching the inverse of a matrix rather than compute it repeatedly (there are also alternatives to matrix inversion that we will not discuss here). Your assignment is to write a pair of functions that cache the inverse of a matrix.

Write the following functions:

- makeCacheMatrix: This function creates a special "matrix" object that can cache its inverse.
- cacheSolve: This function computes the inverse of the special "matrix" returned by makeCacheMatrix above. If the inverse has already been calculated (and the matrix has not changed), then the cachesolve should retrieve the inverse from the cache.

Computing the inverse of a square matrix can be done with the solve function in R. For example, if X is a square invertible matrix, then solve(X) returns its inverse.

For this assignment, assume that the matrix supplied is always invertible.

In order to complete this assignment, you must do the following:

- Fork the GitHub repository containing the stub R files at https://github.com/rdpeng/ProgrammingAssignment2 to create a copy under your own account.

- Clone your forked GitHub repository to your computer so that you can edit the files locally on your own machine.
- Edit the R file contained in the git repository and place your solution in that file (please do not rename the file).
- Commit your completed R file into YOUR git repository and push your git branch to the GitHub repository under your account.
- Submit to Coursera the URL to your GitHub repository that contains the completed R code for the assignment.

In addition to submitting the URL for your GitHub repository, you will need to submit the 40 character SHA-1 hash (as string of numbers from 0-9 and letters from a-f) that identifies the repository commit that contains the version of the files you want to submit. You can do this in GitHub by doing the following: 1. Going to your GitHub repository web page for this assignment. 2. Click on the "?? commits" link where ?? is the number of commits you have in the repository. For example, if you made a total of 10 commits to this repository, the link should say "10 commits". 3. You will see a list of commits that you have made to this repository. The most recent commit is at the very top. If this represents the version of the files you want to submit, then just click the "copy to clipboard" button on the right hand side that should appear when you hover over the SHA-1 hash. Paste this SHA-1 hash into the course web site when you submit your assignment. If you don't want to use the most recent commit, then go down and find the commit you want and copy the SHA-1 hash.

```
## This function initializes the objects "x" and "inv", which will store a
## matrix and its inverse, respectively. It then defines functions to "get" and
## "set" the value of "x" and "inv", then outputs the four functions as a
## list. makeCacheMatrix also serves as the parent environment for these
## functions. The "set" type functions can alter the values of "x" and "inv"
## in the parent environment with the double arrow assignment, and the "get"
## functions, by default give the values of "x" and "inv" in the parent
## environment, since these objects are not defined in the "get" functions
## themselves.

makeCacheMatrix <- function(x = matrix()) {
        inv <- NULL
        set <- function(y = matrix()) {
                x <<- y
                inv <<- NULL
        }
        get <- function() x
        setinv <- function(inverse) inv <<- inverse
        getinv <- function() inv
        list(set = set, get = get, setinv = setinv, getinv = getinv)
}


## the cacheSolve function takes as its argument an object produced by the
## makeCacheMatrix function, which has a matrix "x" stored in it. Firstly it
## retreives the value of the inverted matrix, if there is one, then checks if
## it's NULL. If there is a stored value, the function returns the stored value
## along with a message to let the user know the cached value was used.
## Otherwise, it retrieved the matrix "x" from the makeCacheMatrix environment,
## calculates the inverse matrix, stores it into said environment and returns
## it.

cacheSolve <- function(x, ...) {
```

```
        ## Return a matrix that is the inverse of 'x'
        inv <- x$getinv()
        if(!is.null(inv)) {
                print("Getting Cached Data")
                return(inv)
        }
        matr <- x$get()
        inv_matr <- solve(matr, ...)
        x$setinv(inv_matr)
        inv_matr
}
```