Learning *R*

Carl James Schwarz

StatMathComp Consulting by Schwarz
cschwarz.stat.sfu.ca @ gmail.com

Functions - custom code

Table of Contents I

Calling *R* Functions

R basics - Functions

A FUNCTION is a transformation from input to output.
It has the following parts:

- Function Name - how the function is invoked.
- Arguments - list of arguments to which will be supplied data from the call.
  Arguments can be any type or data structure.
- Body - the code that defines what the function does.
  No limit on what a function can do.
- Output - value of the function. Any data type or structure.

Examples of function calls.

```
1 mean.calories <- mean(cereal$calories)
2 result.lm <- lm( calories ~ fat, data=cereal)
3 plot1 <- ggplot(data=cereal, aes(y=calories, x=fat))+
4             geom_point()
```

A FUNCTION is a transformation from input to output.
Internals of a function e.g. coefficient of variation = sd/mean

```
1  my.cv <- function( x ){
2  # Compute the coefficient of variation
3       my.mean <- mean(x)
4       my.sd  <- sd(x)
5       cv   <- my.sd / my.mean
6       names(cv) <- 'cv'
7       return(cv)
8  } # end of my.cv
9  my.data <- c(1:10);  my.data
10 my.cv(my.data)
11 my.cv(x=my.data)
12 my.cv(y=my.data)
```

A FUNCTION is a transformation from input to output.
Internals of a function e.g. coefficient of variation = sd/mean

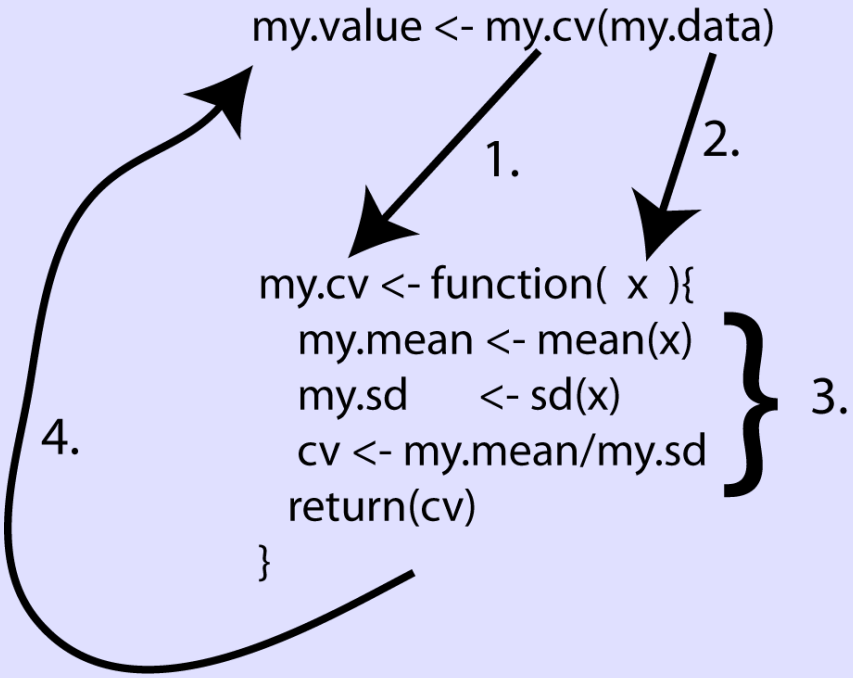
```
> my.data <- c(1:10)
> my.data
 [1]  1  2  3  4  5  6  7  8  9 10

> my.cv(my.data)
       cv
0.5504819

> my.cv(x=my.data)
       cv
0.5504819

> my.cv(y=my.data)
Error in my.cv(y = my.data) : unused argument (y = my.data)
```

What actually happens when a function is called?



1. See if function is defined in workspace?
2. Make a COPY of data objects, and pass COPY to arguments of function.
3. Do the computations. All NEW objects are local.
4. Take results of *return()*, and send back to calling statement.
   Destroy any LOCAL objects.
   Destroy COPY of initial data objects passed to arguments.

How do I find out how to use a function?

- *help(function name)*; e.g. help(mean)
- ??mean - finds all keywords in function descriptions
- Google is your friend!

How do I find out how to use a function? *help(mean)*

```
Description
Generic function for the (trimmed) arithmetic mean.

Usage
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
.....
```

```
1 mean(cereal$calories)
2 mean(cereal$calories, trim=.10)
3 mean(cereal$weight)
4 mean(cereal$weight, na.rm=TRUE)
5 mean(cereal$weight, na.rm=TRUE, trim=0.10)
6 mean(cereal$weight, na.rm=TRUE,
7        trim=5/length(cereal$weight))
```

Types of arguments

```
help(mean)
mean(x, trim = 0, na.rm = FALSE, ...)
```

- Arguments with NO default, e.g. *x*.
- Arguments with default, e.g. *trim* or *na.rm*.
- Additional arguments, e.g. ... that are passed to function. ADVANCED USAGE

```
1 mean(cereal$calories)
2 mean(cereal$calories, trim=.10)
3 mean(cereal$weight)
4 mean(cereal$weight, na.rm=TRUE)
5 mean(cereal$weight, na.rm=TRUE, trim=0.10)
```

How does *R* match calling arguments to function arguments

```
help(mean)
mean(x, trim = 0, na.rm = FALSE, ...)
```

```
1  mean(cereal$calories)
2  mean(cereal$weight, na.rm=TRUE)
3  mean(cereal$weight, na.rm=TRUE, trim=0.10)
4  mean(na.rm=TRUE, x=cereal$weight, .10)  # AVOID
5  mean(na.rm=TRUE, x=cereal$weight, trim=.10, blahblah=3)  #
```

1. Exact argument names are matched. Default values are replaced.
2. Partial match of argument names (CAUTION - AVOID)
3. Positional matching left to right of remaining arguments
4. Any other arguments go into the ....

Useful to pass functions as arguments to functions

```
1 ddply(cereal, "shelf", summarize, mean=mean(calories))
2 aaply(matrix, 1, sum)
3 optim(function, parms)
```

Key problems:

- passing arguments to the passed function via the ... argument.
- very common in more advanced usages such as bootstrapping or optimization

- Very common to automate actions - more general than a script.
- Function should be SELF CONTAINED and so ALL DATA must be passed to function.
- CAUTION: Avoid argument matching with partial matching or in strange order.
- Many possible data structures that can be returned. Most common are:
  - Single value
  - Vector
  - Dataframe
  - List

R Writing *R* Functions

R basics - Functions

A FUNCTION is a transformation from input to output.
It has the following parts:

- Function Name - how the function is invoked.
- Arguments - list of arguments to which will be supplied data from the call.
  Arguments can be any type or data structure.
- Body - the code that defines what the function does.
  No limit on what a function can do.
- Output - value of the function. Any data type or structure.

Examples of function calls.

```
1 mean.calories <- mean(cereal$calories)
2 result.lm <- lm( calories ~ fat, data=cereal)
3 plot1 <- ggplot(data=cereal, aes(y=calories, x=fat))+
4             geom_point()
```

A FUNCTION is a transformation from input to output.
Defining a function e.g. coefficient of variation = sd/mean

```
1  my.cv <- function( x ){
2  # Compute the coefficient of variation
3       my.mean <- mean(x)
4       my.sd  <- sd(x)
5       cv   <- my.sd / my.mean
6       names(cv) <- 'cv'
7       return(cv)
8  } # end of my.cv
9  my.data <- c(1:10);  my.data
10 my.cv(my.data)
11 my.cv(x=my.data)
12 my.cv(y=my.data)
```

- INPUT - list of arguments to which will be supplied data from the call
- OUTPUT - value of *return()* - good form to put as last statement

A FUNCTION is a transformation from input to output.
Internals of a function e.g. coefficient of variation = sd/mean

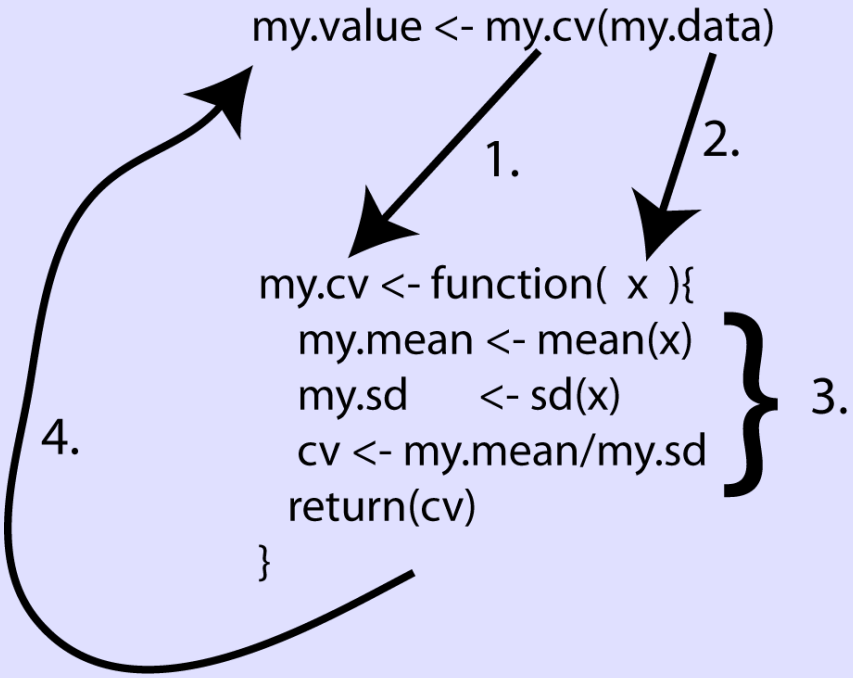
```
> my.data <- c(1:10)
> my.data
 [1]  1  2  3  4  5  6  7  8  9 10

> my.cv(my.data)
       cv
0.5504819

> my.cv(x=my.data)
       cv
0.5504819

> my.cv(y=my.data)
Error in my.cv(y = my.data) : unused argument (y = my.data)
```

What actually happens when a function is called?



1. See if function is defined in workspace?
2. Make a COPY of data objects, and pass COPY to arguments of function.
3. Do the computations. All NEW objects are local.
4. Take results of *return()*, and send back to calling statement.
   Destroy any LOCAL objects.
   Destroy COPY of initial data objects passed to arguments.

Good programming practices:

- For all but trivial functions, use lots of comments describing the purpose and the arguments required for the function.
- Pass all necessary data to functions.
  Don't rely on GLOBAL variables in calling environment.
- Explicitly return final object.
- No side effects, i.e. do not modify any global variable using «-
- Try and handle missing values gracefully.
- ALWAYS test your function with known data.

Creating the function:

- Use text editor to define the function as shown above.
- Highlight and Run in *Rstudio* to compile the function.
- Debug the function.
- Use the function.

If your modify the function, you MUST recompile it before using it again.

Try

```
1 my.cv(cereal$calories)
2 my.cv(cereal$weight)
```

Modify the *my.cv()* to incorporate one more argument:

- Removing missing values before computations (*remove.na=TRUE*)

```
1 my.cv(cereal$calories)
2 my.cv(cereal$weight)
3 my.cv(cereal$weight, remove.na=TRUE)
```

More on arguments to function.

```
1  # Adding complexity. More arguments to the function
2  my.cv <- function( x ,remove.na=FALSE){
3  # Compute the coefficient of variation
4  # after removing  na's
5       my.mean <- mean(x, na.rm=remove.na)
6       my.sd   <- sd(x,   na.rm=remove.na)
7       cv   <- my.mean / my.sd
8       names(cv) <- 'cv'
9       return(cv)
10 } # end of my.cv
11
12 my.cv(cereal$calories)
13 my.cv(cereal$weight)
14 my.cv(cereal$weight, remove.na=TRUE)
```

"Reusing" argument names. Often done and is confusing until you get used to it.

```
1  # Adding complexity. More arguments to the function
2  my.cv <- function( x ,na.rm=FALSE){
3  # Compute the coefficient of variation
4  # after removing na's
5      my.mean <- mean(x, na.rm=na.rm)
6      my.sd   <- sd(x,   na.rm=na.rm)
7      cv   <- my.mean / my.sd
8      names(cv) <- 'cv'
9      return(cv)
10 } # end of my.cv
11
12 my.cv(cereal$calories)
13 my.cv(cereal$weight)
14 my.cv(cereal$weight, na.rm=TRUE)
```

Keep track of where the *na.rm* comes from and is going to with code such as *na.rm=na.rm*.

Add a *chop* argument than trims the top 'chop' and bottom 'chop' fraction of observations before computing the mean and sd.
Hint: *low <- quantile(x, prob=chop, na.rm=TRUE)* returns the *chop* quantile.
Hint: *high <- quantile(x, prob=1-chop, na.rm=TRUE)* returnes the $1 - chop$ quantile.
Hint: *x[x >= low & x <= high]* selects middle observations .

```
> my.cv(cereal$calories)
4.859589
> my.cv(cereal$weight)
NA
> my.cv(cereal$weight, na.rm=TRUE)
6.760404
> my.cv(cereal$weight, na.rm=TRUE, chop=0.10)
22.98736
```

```
1  my.cv <- function( x , chop=0, na.rm=FALSE){
2  # Compute the coefficient of variation
3  # after removing a fraction 'chop' from top and bottom and
4  # dealing with na's
5      newx <- x[ x ≥ quantile(x, probs=chop, na.rm=TRUE) &
6                 x ≤ quantile(x, probs=1-chop,na.rm=TRUE)]
7      my.mean <- mean(newx, na.rm=na.rm)
8      my.sd   <- sd(newx, na.rm=na.rm)
9      cv    <- my.mean / my.sd
10     names(cv) <- 'cv'
11     return(cv)
12 } # end of my.cv
```

Useful functions for debugging functions:

- *browser()* - stops when hit. *Rstudio* can also set a break point.

```
1  my.cv <- function( x , trimfrac=0, na.action=FALSE){
2  # Compute the coefficient of variation
3  # after removing trimfrac from top and bottom and
4  # dealing with na's
5      newx <- x[ x ≥ quantile(x, probs=trimfrac) &
6                 x ≤ quantile(x, probs=1-trimfrac)]
7      browser()
8      ....
9  }
10 my.data <- c(0, 50:60, 100);  my.data
```

Don't forget to recompile function after inserting/removing *browser()*

Actions after the *browser()* is hit

- any *R* expression. Check out local variables etc.
- *n*, *c*, *Q* for *next*, *continue*, *Quit* respectively.
- CAUTION: blank lines will quit the *browser()* - GROAN

Write a function *my.summary(x)* which returns

- Total number of observations (including NAs)
- Total number of NAs
- Mean (ignoring NAs)
- Std dev (ignoring NAs)
- CV (ignoring NAs)

It should return a data.frame.

```
> my.summary(cereal$calories)
  nobs nmiss     mean       sd       cv
1   77     0 105.0649 21.62013 4.859589
> my.summary(cereal$weight)
  nobs nmiss   mean         sd       cv
1   77     2 1.0304 0.1524169 6.760404
```

```
 1  my.summary <- function(x){
 2  # Compute the number, number missing, mean, sd, cv
 3  # with all the NA removed
 4     nobs <- length(x)
 5     nmiss <- sum(is.na(x))
 6     mean <- mean(x, na.rm=TRUE)
 7     sd   <- sd(x, na.rm=TRUE)
 8     cv   <- sd/mean
 9     res  <- c(nobs,nmiss,mean,sd,cv)
10     res  <- data.frame(nobs,nmiss,mean,sd,cv,
11                        stringsAsFactors=FALSE)
12     return(res)
13  }# end of my.summary
```

Recall cereal data.
Write a function *my.lines(cereal.df)* which

- Estimates the regression line between calories and fat using *lm()*
- Returns the estimated slope, its se, and the 95% confidence interval as a data frame.

```
> my.line(cereal)
       slope slope.se      lcl      ucl
fat 9.806005 2.206897 5.409642 14.20237
```

```
1  my.line <- function(df){
2    # do a regression of Calories vs Fat and return
3    # the slope, its se, and the 95% ci on the slope
4    fit <- lm(Calories ~ Fat, data=df)
5    slope <- coef(fit)[2]
6    slope.se <- sqrt(diag(vcov(fit)))[2]
7    slope.ci <- confint(fit)[2,]
8    lcl <- slope.ci[1]
9    ucl <- slope.ci[2]
10   res <- data.frame(slope, slope.se, lcl, ucl, stringsAsFact
11   return(res)
12 } # end my.line
```

Modify your function to specify the names of the X and Y variables in the call.
Hint: Use

```
1   fit <- lm(df[,Yname] ~ df[,Xname], data=df)
```

in your function body to do the fit.

```
> my.line2(cereal, Xname="fat",     Yname="calories")
                X        Y    slope slope.se      lcl      ucl
df[, Xname] fat calories 9.806005 2.206897 5.409642 14.2023
> my.line2(cereal, Xname="protein", Yname="calories")
                  X        Y     slope slope.se       lcl
df[, Xname] protein calories 0.4091816 2.279836 -4.132485 4.
```

```
my.line2 <- function(df, Xname, Yname){
  # do a regression of Yname vs Xname and return
  # the slope, its se, and the 95% ci on the slope
  fit <- lm(df[,Yname] ~ df[,Xname], data=df)
  slope <- coef(fit)[2]
  slope.se <- sqrt(diag(vcov(fit)))[2]
  slope.ci <- confint(fit)[2,]
  lcl <- slope.ci[1]
  ucl <- slope.ci[2]
  res <- data.frame(X=Xname, Y=Yname, slope,
            slope.se, lcl, ucl)
  return(res)
} # end my.line2
```

Modify your function to specify the names of the X and Y variables in the call.
Return a list with the plot, the fitted object, and the summary.

Hint: You will use *aes_string()* in *ggplot()*.

```
> my.line3(cereal, Xname="fat",      Yname="calories")
```

returns the plot, the fit object, and the final result

R basics - Functions - Exercise IV

```
1  my.line3 <- function(df, Xname, Yname){
2    # Get the plot
3    plot <- ggplot(data=df, aes_string(x=Xname, y=Yname))+
4       ggtitle(paste("Plot of ", Yname," vs ", Xname, sep=""))
5       geom_point( position=position_jitter(h=.1, w=.1))+
6       geom_smooth(method="lm", se=FALSE)
7
8    # do a regression of Y vs X and return
9    # the slope, its se, and the 95% ci on the slope
10   fit <- lm(df[,Yname] ~ df[,Xname], data=df)
11   slope <- coef(fit)[2]
12   slope.se <- sqrt(diag(vcov(fit)))[2]
13   slope.ci <- confint(fit)[2,]
14   lcl <- slope.ci[1]
15   ucl <- slope.ci[2]
16   res <- data.frame(X=Xname, Y=Yname, slope, slope.se, lcl,
17
18   return(list(plot=plot, fit=fit, res=res))
19 } # end my.line3
```

Useful to pass functions as arguments to functions

- report <- ddply(df, "byvar", function) # SAC paradigm
- bootres <- boot(data, function, reps) # bootstrapping
- max <- optim(function, parms)

Key problems:

- passing arguments to the passed function, the ... argument

Use your *my.line()* function and *ddply()* to fit a separate line between calories and fat for the 3 shelves.

```
> ddply(cereal, "shelf", my.line)
  shelf      slope slope.se       lcl       ucl
1     1  0.3703704 3.581444 -7.153964  7.894705
2     2 10.8333333 2.626300  5.336424 16.330243
3     3 11.7948718 3.698559  4.278496 19.311248
```

Use your *my.line2()* function and *ddply()* to fit a separate line between *Xname* and *Yname* for the 3 shelves.

```
 > ddply(cereal, "shelf", my.line2, Xname="fat", Yname="calo
  shelf   X         Y      slope slope.se       lcl       ucl
1     1 fat calories  0.3703704 3.581444 -7.153964  7.894705
2     2 fat calories 10.8333333 2.626300  5.336424 16.330243
3     3 fat calories 11.7948718 3.698559  4.278496 19.311248
```

Notice how the last two arguments pass values to the function — more on this later.

Use your *my.line3()* function and *ddply()* to fit a separate line between *Xname* and *Yname* for the 3 shelves and return the plots, the fits, and the results. [More on this later]

```
 > ddply(cereal, "shelf", my.line2, Xname="fat", Yname="calo
  shelf   X        Y      slope slope.se       lcl       ucl
1     1 fat calories  0.3703704 3.581444 -7.153964  7.894705
2     2 fat calories 10.8333333 2.626300  5.336424 16.330243
3     3 fat calories 11.7948718 3.698559  4.278496 19.311248
```

Notice how the last two arguments pass values to the function — more on this later.

Use your *my.line3()* function and *ddply()* to fit a separate line between *Xname* and *Yname* for the 3 shelves and return the plots, the fits, and the results. [More on this later]

```
results <- dlply(cereal, "shelf", my.line3, Xname="fat", Yna
names(results)
results[[1]]
```

Notice how the last two arguments pass values to the function — more on this later.

Bootstrapping

- In some cases, the SE is not easily found because the problem is non-standard or certain assumptions (e.g. normality) is violated.
- Bootstrapping provides a way to compute SE for statistics similar to "means". Does not work well for statistics that are related to order statistics such as max, min, median, etc.

Key Idea:

- Define a function that computes a statistic based on some data
- Create a boot-strap sample which is a sample with replacement from original data
- Compute the statistic on the boot-strap sample.
- Repeat previous two steps many (typically more than 1000) times.
- Look at SD of statistics and 2.5^{th} and 97.5^{th} percentiles for SE and CI

Bootstrapping

```
1 library(boot)
2 help(boot)
```

```
boot(data, statisticfunction, R, sim = "ordinary", stype = c
     strata = rep(1,n), L = NULL, m = 0, weights = NULL,
     ran.gen = function(d, p) d, mle = NULL,
     simple = FALSE, ...,                          <= NOTICE
     parallel = c("no", "multicore", "snow"),
     ncpus = getOption("boot.ncpus", 1L), cl = NULL)
```

Allows you pass data (and other arguments) to your function.

Bootstrapping Example: ratio of mean calories/serving / mean fat/serving

```
1 ratio.meanY.meanX <- function(df, ind, Y,X, na.rm=FALSE){
2 #  Compute the ratio of the mean of Y to
3 #  mean of X potentially removing missing values
4    res <- mean(df[ind,X],na.rm=na.rm)/
5           mean(df[ind,Y],na.rm=na.rm)
6    names(res) <- "ratio"
7    return(res)
8 }
```

First argument to function is *data* and second argument is a vector of indices, indicating which rows of data frame to use.

```
> ratio.meanY.meanX(df=cereal,1:nrow(cereal), X="fat",Y="cal
     ratio
0.009641533
```

Bootstrapping Example: ratio of mean calories/serving / mean fat/serving

```
1 library(boot)
2 bootres <- boot(cereal, ratio.meanY.meanX, R=10,
3              X="Fat",Y="Calories", na.rm=TRUE )
4 bootres
```

```
> bootres
ORDINARY NONPARAMETRIC BOOTSTRAP

Call: boot(data = cereal, statistic = ratio.meanY.meanX, R =
    Y = "calories", na.rm = TRUE)

Bootstrap Statistics :
       original        bias    std. error
t1* 0.009641533 9.710211e-05 0.001037265
```

Bootstrapping Example: ratio of mean calories / mean fat
Examine structure of *bootres* further:

```
1 str(bootres)
2 bootres$t0
3 bootres$t
4 quantile(bootres$t, prob=c(0.25, .975))
```

```
> bootres$t0
      ratio
0.009641533
> bootres$t
              [,1]
  [1,] 0.008798017
  [2,] 0.008663366
    .....
> quantile(bootres$t, prob=c(0.25, .975))
       25%       97.5%
0.009076569 0.011911880
```

Bootstrapping Example: ratio of mean calories/serving / mean fat/serving

Add browser to my function to see what happens, esp. the calling arguments

Find the SE and 95% CI for R^2 on the regression of two variables from the cereal dataset

Find the SE and 95% CI for R^2 on the regression of two variables from the cereal dataset

```
1 r2YX <- function(df, ind, Y,X){
2 #  Compute the regression of Y on X and then find R2
3     fit <- lm( df[ind,Y] ~ df[ind, X])
4     res <- summary(fit)$r.squared
5     names(res) <- "R2"
6     return(res)
7 } # end r2YX
```

First argument to function is *data* and second argument is a vector of indices, indicating which rows of data frame to use.

```
> fit <- lm( calories ~ fat, data=cereal)
> summary(fit)$r.squared
[1] 0.2083875

> r2YX(cereal, 1:nrow(cereal), Y="Calories", X="Fat")
       R2
0.2083875
```

Find the SE and 95% CI for R^2 on the regression of two variables from the cereal dataset

```
1 bootres <- boot(cereal, r2YX, R=100,
2              X="fat",Y="calories" )
3 bootres
4 str(bootres)
5 bootres$t0
6 bootres$t[1:10]
7 quantile(bootres$t, prob=c(0.25, .975))
```

```
> bootres
ORDINARY NONPARAMETRIC BOOTSTRAP

Bootstrap Statistics :
      original        bias    std. error
t1* 0.2083875 -0.004412033  0.06778598
```

Find the SE and 95% CI for R^2 on the regression of two variables from the cereal dataset

```
> bootres
ORDINARY NONPARAMETRIC BOOTSTRAP
Call: boot(data = cereal, statistic = r2YX, R = 100, X = "Fa
     original        bias    std. error
t1* 0.2083875 -0.004412033  0.06778598
> quantile(bootres$t, prob=c(0.25, .975))
      25%     97.5%
0.1553856 0.3548753
```

R basics - Writing Functions - Summary

- Very common to automate actions - more general than a script.
- All functions should be SELF CONTAINED
  - Do not reference variables NOT in argument list
  - Do not create side effects
- Many possible data structures that can be returned. Most common are:
  - Single value
  - Vector
  - Dataframe
  - List
- Don't forget to use the ... argument to pass arguments to passed functions.