Functions - Intro

* Creating functions in R is very simple.

* Users communicate with R almost entirely through functions anyway.

* You should write a function whenever you find yourself going through the same sequence of steps at the command line, perhaps with small variations.

* You can reuse code that you have found to be useful. You can even package it up and give it to others.

* Once you have "trustworthy" code you can relax and not worry so much about errors.

Functions - Listing Source

In general its easy to see the function definitions of many R functions. Simply type their name.

```
> ls
function (name, pos = -1, envir = as.environment(pos), all.names = FALSE,
    pattern)
{
    if (!missing(name)) {
        nameValue <- try(name, silent = TRUE)
..
..
 }
        grep(pattern, all.names, value = TRUE)
    }
    else all.names
}
<bytecode: 0x10098d0e8>
<environment: namespace:base>
```

Functions - Listing Source

* Sometimes its not so easy to see the contents and you have to hunt for them.

```
> t.test

function (x, ...)
UseMethod("t.test")
<bytecode: 0x1033eca78>
<environment: namespace:stats>
```

* Aha ! "t.test" is a S3-method and you can have a look at implemented methods on objects by doing:

```
> methods(t.test)
[1] t.test.default* t.test.formula*
  Non-visible functions are asterisked
```

Functions - Listing Source

* Sometimes its not so easy to see the contents and you have to hunt for them.

```
> getAnywhere(t.test.default)

A single object matching ‘t.test.default’ was found. It was found in the
following places registered S3 method for t.test from namespace stats
namespace:stats with value

function (x, y = NULL, alternative = c("two.sided", "less", "greater"),
mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95, ...)
{
alternative <- match.arg(alternative)
    if (!missing(mu) && (length(mu) != 1 || is.na(mu)))
        stop("'mu' must be a single number")
..
..
```

Functions - Listing Source

* Sometimes you have to work a little harder:

```
> kruskal.test

function (x, ...)
UseMethod("kruskal.test")
<bytecode: 0x104460c28>
<environment: namespace:stats>

> methods(kruskal.test)
[1] kruskal.test.default* kruskal.test.formula*

> kruskal.test.default
Error: object 'kruskal.test.default' not found
```

Functions - Listing Source

* Sometimes you have to work a little harder:

```
> stats:::kruskal.test.default

function (x, g, ...)
{
    if (is.list(x)) {
        if (length(x) < 2L)
            stop("'x' must be a list with at least 2 elements")
        DNAME <- deparse(substitute(x))
        x <- lapply(x, function(u) u <- u[complete.cases(u)])
        k <- length(x)
        l <- sapply(x, "length")
        if (any(l == 0))
            stop("all groups must contain data")
        g <- factor(rep(1:k, l))
        x <- unlist(x)
..
..
```

Functions - Getting Help

* Use the args and example commands to get more info. Of course use the ? to get even more help

```
> args(ls)
function (name, pos = -1, envir = as.environment(pos), all.names = FALSE, 
pattern)

> args(mean)
function (x, ...)

> example(mean)

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50

> ?mean
```

Functions - Declaring

Functions are created using the **function()** directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
my.cool.function <- function(<arguments>) {

## Do something interesting
## Return a value(s)

}
```

Functions can be passed as arguments to other functions

Functions can be nested, so that you can define a function inside of another function

The return value of a function is the last expression in the function body to be evaluated.

www.stat.berkeley.edu/~statcur/Workshop2/Presentations/functions.pdf

Functions - Declaring

*** Let's look at some formal definitions.**

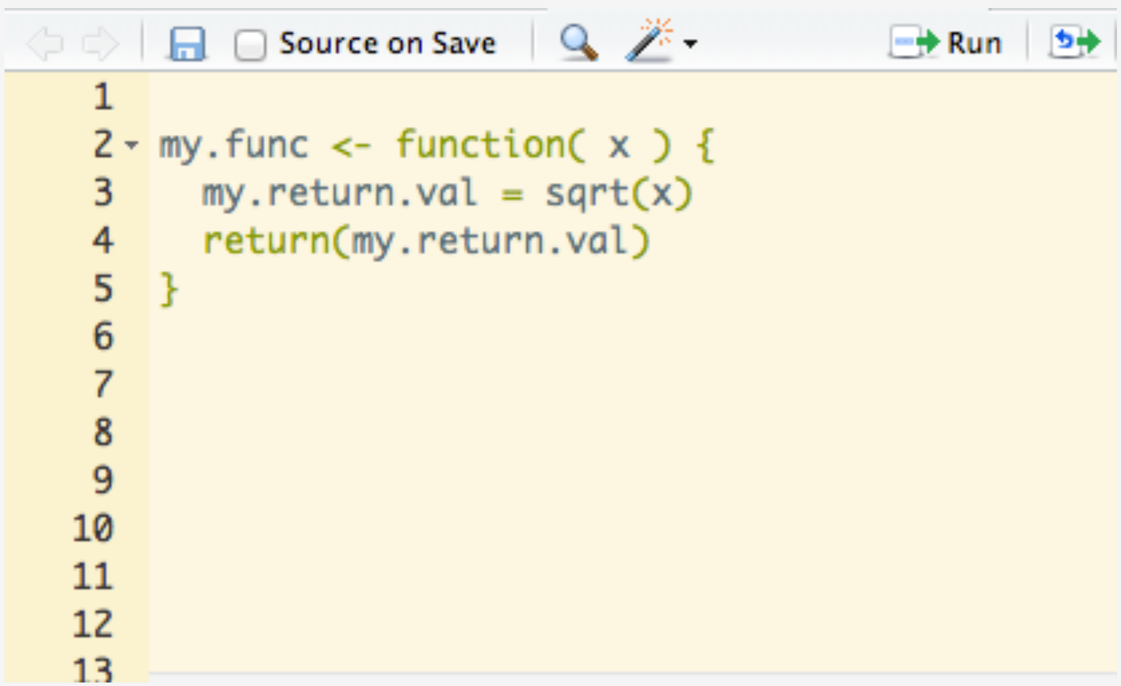
```
my.func <- function(arglist) {
   expr
   return(value) # You should have only ONE return statement
}
```

arglist	**Empty or one or more name or name=expression terms.**
expr	**Some statements / expressions**
value	**An expression**

```
my.func <- function(somenum) {
   my.return.val = sqrt(somenum)
   return(my.return.val)
}

my.func(10)
[1] 3.162278

mycomputation = my.func(10)
```

Functions - Declaring

Note that once you create a function you can retrieve its contents and edit it using the fix function. But better to use the Edit Window in RStudio. Change your function over time and reload it to register new versions by highlighting it and clicking "Run".



Functions - Declaring

You should have only one return statement per function

It should generally be the very last statement in the function

A return is not strictly required although it is more common than not.

You can return a vector, list, matrix, or dataframe.

A list provides the most generality but it might be too much depending on what it is you want to accomplish.

Functions - Declaring

TIPS:

Determine what you are being asked to do. This is easy. You will be told:

1) What the function will accept as input (e.g. vector, matrix, data frame)

2) What arguments the function will accept

3) What to return - what the output will be

Make a shell like the following and build into it:

```
myfunc <- function(somevec) {


}  # End function
```

Put comments in to help you keep up with brackets

Functions - Declaring

Define a function called "pythag" that, given the two side lengths of a triangle, will compute the length of the third side.

```
pythag <- function(a,b) {
    c = sqrt(a^2 + b^2)

    return(c) # You should have ONLY ONE return statement in any function
}

pythag(4,5)
[1] 6.403124

x = 4
y = 5

pythag(x,y)
[1] 6.403124

pythag(a = 4, b = 5)
[1] 6.403124
```

Functions - Returning Stuff

We can return pretty much any kind of R structure we would like. If you remember from the section on lists this is, in part, why lists exist. To let you return a number of things in a single structure. Recall that the lm function does this.

```
data(mtcars)

my.lm = lm(mpg ~ wt, data = mtcars)

typeof(my.lm)
[1] "list"

ls(my.lm)
 [1] "assign"        "call"          "coefficients"  "df.residual"
 [5] "effects"       "fitted.values" "model"         "qr"
 [9] "rank"          "residuals"     "terms"         "xlevels"

my.lm$call
lm(formula = mpg ~ wt, data = mtcars)

my.lm$rank
[1] 2
```

Functions - Returning Stuff

You can create structures also.

```
pythag <- function(a,b) {

    c = sqrt(a^2 + b^2)
    myreturnlist = list(hypoteneuse = c, sidea = a, sideb = b)
    return(myreturnlist)
}

pythag(3,4)   # We get back a list

$hypoteneuse
[1] 5

$sidea
[1] 3

$sideb
[1] 4

pythag(3,4)$hypoteneuse    # We can get specific with what we ask for
[1] 5
```

Functions – Argument Checking

What happens if you give the function bad stuff ?

```
pythag <- function(a,b) {

    c = sqrt(a^2 + b^2)
    myreturnlist = list(hypoteneuse = c, sidea = a, sideb = b)
    return(myreturnlist)
}

> pythag(3,4)
[1] 5

> pythag(3,"a")
Error in b^2 : non-numeric argument to binary operator

> pythag()
Error in a^2 : 'a' is missing

> pythag(3,)
Error in b^2 : 'b' is missing
```

Functions – Argument Checking

Well you could set some default values:

```
pythag <- function(a = 4, b = 5) {
       c = sqrt(a^2 + b^2)
      myreturnlist = list(hypoteneuse = c, sidea = a, sideb = b)
      return(myreturnlist)
}


pythag()
$hypoteneuse
[1] 6.403124

$sidea
[1] 4

$sideb
[1] 5
```

Functions – Argument Checking

Maybe we should do some error checking:

```
pythag <- function(a = 4, b = 5) {
     if (!is.numeric(a) | !is.numeric(b)) {
              stop("I need real values to make this work")
     }
     c = sqrt(a^2 + b^2)
     myreturnlist = list(hypoteneuse = c, sidea = a, sideb = b)
     return(myreturnlist)
}

pythag(3,"5")
Error in pythag(3, "5") : I need real values to make this work

pythag("3",5)
Error in pythag("3", 5) : I need real values to make this work
```

Functions – Argument Checking

Maybe we should do some error checking:

```
pythag <- function(a = 4, b = 5) {
     if (!is.numeric(a) | !is.numeric(b)) {
              stop("I need real values to make this work")
     }
     if (a <=0 | b <= 0) {
              stop("Arguments need to be positive")
     }
     c = sqrt(a^2 + b^2)
     myreturnlist = list(hypoteneuse = c, sidea = a, sideb = b)

     return(myreturnlist)

} # End Function

pythag(-3,3)
Error in pythag(-3, 3) : Arguments need to be positive

pythag(3,3)
[1] 4.242641
```

Functions - Declaring

Always create a function whenever you have some block of code that works well. This will prevent you from having to type it in the code every time you wish to execute it.

It can be edited over time as you need to make changes to it. Functions don't need to be complicated to be useful.

```
# Utility function to determine if a value is odd or even

is.odd <- function(someval) {
     retval = 0  # Set the return value to a default

     if (someval %% 2 != 0) {
        retval = TRUE
     } else {
        retval = FALSE
     }
     return(retval)
}
is.odd(3)
[1] TRUE
```

Functions - Declaring

Ask yourself what are the:

1) input(s) ?   (e.g. single value, vector, matrix, data frame)
2) output(s) ? (e.g. single value, vector, matrix, etc)

```
is.odd <- function(someval) {

     retval = 0  # Set the return value to a default

     if (someval %% 2 != 0) {
        retval = TRUE
     } else {
        retval = FALSE
     }
     return(retval)

} # End function

is.odd(3)
[1] TRUE
```

Functions - Declaring

This works on single values. It could be changed to work with single values or vectors.

```
is.odd <- function(someval) {
  retvec = vector()
  for (ii in 1:length(someval)) {
    if (someval[ii] %% 2 != 0) {
        retvec[ii] = TRUE
    } else {
        retvec[ii] = FALSE
    }
  }
  return(retvec)

}  # End function

is.odd(3)
[1] TRUE

numbers = c(9,9,4,4,6,10,7,18,2,10)
is.odd(numbers)
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```

Functions - Declaring

This works on single values. It could be changed to work with single values or vectors.

```
is.odd(3)
[1] TRUE

numbers = c(9,9,4,4,6,10,7,18,2,10)
is.odd(numbers)
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE

numbers[is.odd(numbers)]   # Very useful
[1] 9 9 7
```

Functions - Vectors

Let's look at some of the structures from last week to see how they might look as functions. We used the following approach to take a series of X values, plug them into a function to get resulting Y values, and then plot them.

```
y = vector()
x = seq(-3,3)
for (ii in 1:length(x)) {
  y[ii] = (x[ii])^2
}

length(x)
[1] 1201

plot(x,y,main="Super Cool Data Plot",type="l")
```

Functions - Vectors

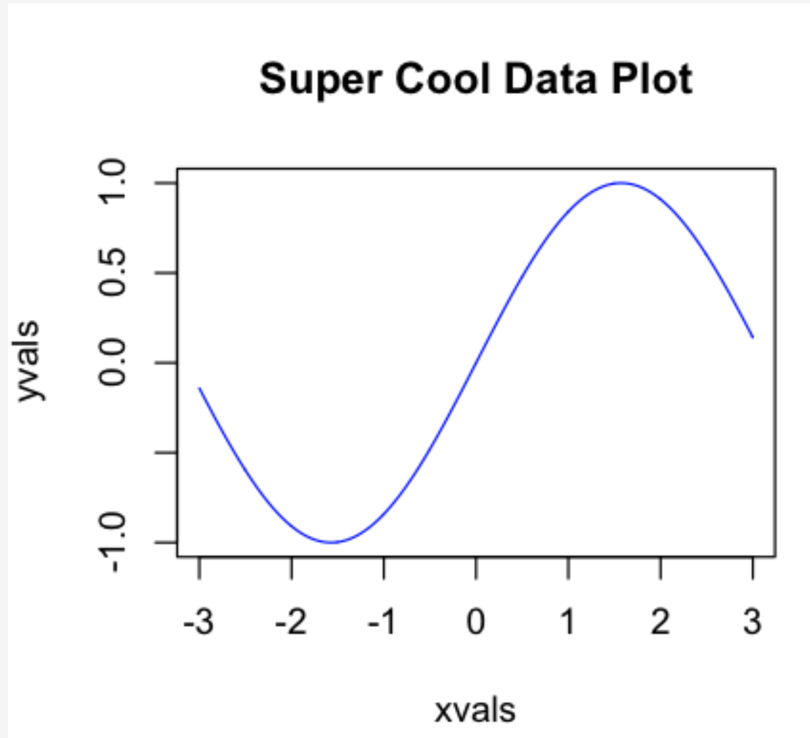
Let's look at some of the structures from last week to see how they might look as functions. We can even make an argument for a function

```
myplotter <- function(xvals, mfunc) {   # begin function

  # Function to print y = x^2
  # Input: xvalues a vector
  # Output: A plot

  yvals = vector()   # setup a blank vector to hold y-values

  for (ii in 1:length(xvals)) {   # begin for loop
    yvals[ii] = mfunc(xvals[ii])
  }                               # end for loop

  plot(xvals, yvals, main="Super Cool Data Plot",type="l",col="blue")

}  # End function
```

Functions - Vectors

Let's look at some of the structures from last week to see how they might look as functions:

```
xvals = seq(-3,3,0.005)

myplotter(xvals,sin)
```



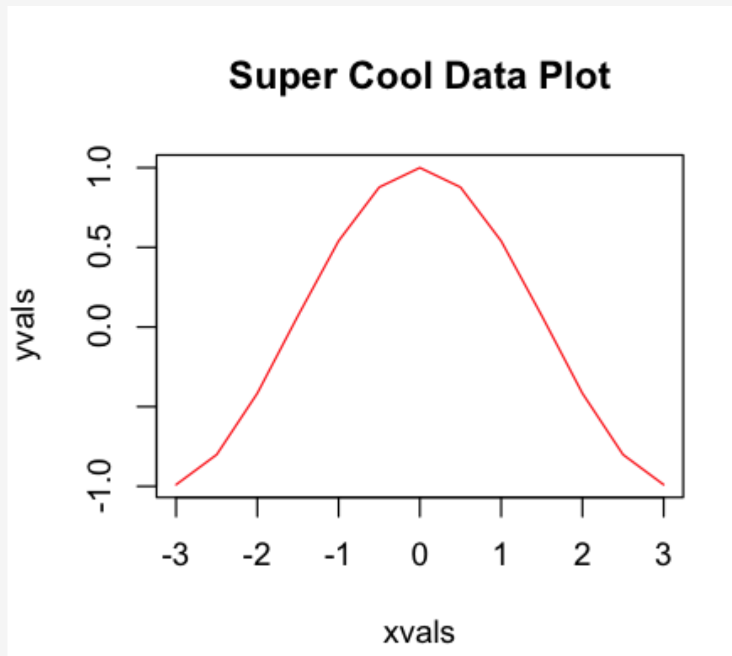
Functions - Vectors

We could add in "Arguments" to influence the color of the plot. We could also return the generated y values if we wanted to.

```
myplotter <- function(xvals, mfunc, plotcolor="blue") {

# Function to print y = x^2
# Input: xvalues
# Output: A plot and the xvals and yvals used to make that plot

   yvals = vector()
   for (ii in 1:length(xvals)) {
     yvals[ii] = mfunc(xvals[ii])
   }

   plot(xvals, yvals, main="Super Cool Data Plot",type="l",col=plotcolor)
   retlist = list(x=xvals, y=yvals)
   return(retlist)
}

xvals = seq(-3,3,0.5)

myplotter(xvals, cos, plotcolor="red")
```

Functions - Vectors

```
xvals = seq(-3,3,0.5)

myplotter(xvals,cos,plotcolor="red")
$x
 [1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0

$y
 [1] -0.9899925 -0.8011436 -0.4161468  0.0707372  0.5403023  0.8775826
1.0000000  0.8775826  0.5403023  0.0707372 -0.4161468
[12] -0.8011436 -0.9899925
```



Functions - Min / Max Example

Write a function that finds the minimum value in a vector. Take this from last week and make it a function. (We don't need to make the set.seed and x = rnorm part of the function).

```
set.seed(188)
x = rnorm(1000)  # 1,000 random elements from a N(20,4)

mymin = somevector[1] # Set the minimum to an arbitrary value

for (ii in 1:length(x)) {
  if (x[ii] < mymin) {
     mymin = x[ii]
  }
}
```

Functions - Min / Max Example

Write a function that finds the minimum value in a vector. Take this from last week and make it a function. (We don't need to make the set.seed and x = rnorm part of the function).

```
mymin <- function(somevector) {

# Function to find the minimum value in a vector
# Input: A numeric vector
# Output: A single value that represents the minimum

  mymin = somevector[1] # Set the minimum to an arbitrary value

# Now loop through the entire vector. If we find a value less than
# mymin then we set mymin to be that value.

  for (ii in 1:length(somevector)) {
    if (somevector[ii] < mymin) {
      mymin = somevector[ii]
    }
  }
  return(mymin)
}
```

Functions - Min / Max Example

Write a function that finds the minimum value in a vector. Take this from last week and make it a function.

```
set.seed(123)

testvec = rnorm(10000)

mymin(testvec)
[1] -3.84532

min(testvec)  # Matches the built in R function
[1] -3.84532
```

Functions - Min / Max Example

Let's make an argument that let's us specify what we want - The min or max

```
myextreme <- function(somevector, action="min") {

  if (action == "min") {
     myval = somevector[1] # Set the minimum to an arbitrary value

     for (ii in 1:length(somevector)) {
       if (somevector[ii] < myval) {
         myval = somevector[ii]
       }
     }    # End for

  } else {   # If action is not "min" then we assume the "max" is wanted

     myval = somevector[1] # Set the minimum to an arbitrary value

     for (ii in 1:length(somevector)) {
       if (somevector[ii] > myval) {
         myval = somevector[ii]
       }
     }            # End for
  }               # End If
  return(myval)
}
```

Functions - Min / Max Example

Let's make an argument that let's us specify what we want - The max or min:

```
myextreme(testvec,"min")
[1] -3.84532

myextreme(testvec,"max")
[1] 3.847768

min(testvec)
[1] -3.84532

max(testvec)
[1] 3.847768
```

Functions - Split Dataframes

Last time we looked at for-loops to process data frames that we had split up by a factor:

```
mysplits = split(mtcars, mtcars$cyl)

for (ii in 1:length(mysplits)) {
    cat("Split ",names(mysplits)[ii]," has ",
          nrow(mysplits[[ii]]),"rows \n")
}

Split  4  has  11 rows
Split  6  has  7 rows
Split  8  has  14 rows
```

Functions - Split Dataframes

```
myfunc <- function(somedf, somefac) {

# Function to split a data frame by a given factor
# Input: A data frame, a factor
# Output: A list containing a count of records in each group

  retlist = list()    # Empty list to return group record count
  mysplits = split(somedf,somefac)  # Split the data frame by somefac

  for (ii in 1:length(mysplits)) {  # loop through the splits
    retlist[[ii]] = nrow(mysplits[[ii]])
  }
  names(retlist) = names(mysplits)
  return(retlist)
}

myfunc(mtcars,mtcars$cyl)
$`4`
[1] 11

$`6`
[1] 7

$`8`
[1] 14
```

Functions - Matrix

Last time we looked at an example wherein we copied a matrix and modified its contents while we were copying it. Specifically, we subtracted each element from the mean of its respective column. This is called "centering".

```
set.seed(123)

mymat = matrix(round(rnorm(6),2),3,2)

newmat = matrix(rep(0,6),3,2) # Setup a new mat of the same size

for (col in 1:ncol(mymat)) {
  for (row in 1:nrow(mymat)) {
    newmat[row,col] = mymat[row,col] - mean(mymat[,col])
  }
}

newmat
           [,1]  [,2]
[1,] -0.8166667 -0.57
[2,] -0.4866667 -0.51
[3,]  1.3033333  1.08
```

Functions - Matrix

```
mtcenter <- function(somemat) {

# Input: A matrix to center
# Output: A matrix that is centered

    retmat = rep(0, length(somemat)) # Recipe to initialize a
    dim(retmat) = dim(somemat)       # matrix the same size as
                                     # another filled with 0

    for (col in 1:ncol(somemat)) {
      for (row in 1:nrow(somemat)) {
        retmat[row, col] = somemat[row, col] - mean(somemat[,col])
      }
    }

    return(retmat)
}
```

Functions - Anonymous Functions

Anonymous functions are those that are created for "one-off" jobs. They usually show up when using the apply family of functions (lapply, apply, and sapply). Think of anonymous functions as being temporary. We don't even bother to name them but they still behave just like any other function.

```
my.mat = as.matrix(mtcars[,c(1,3:6)])
head(my.mat)
                   mpg disp  hp drat    wt
Mazda RX4         21.0  160 110 3.90 2.620
Mazda RX4 Wag     21.0  160 110 3.90 2.875
Datsun 710        22.8  108  93 3.85 2.320
Hornet 4 Drive    21.4  258 110 3.08 3.215
Hornet Sportabout 18.7  360 175 3.15 3.440
Valiant           18.1  225 105 2.76 3.460
```

We've seen something like the following previously. We call the mean function on all the columns in the matrix. Note that the mean function isn't anonymous. It has a name. But what if we wanted to provide our own custom function. For example one that computes the mean, standard deviation, and range for each column ? We can do that easily.

```
apply(my.mat,2, mean)
       mpg       disp         hp       drat         wt
 20.090625 230.721875 146.687500   3.596563   3.217250
```

Functions - Anonymous Functions

```
my.mat = as.matrix(mtcars[,c(1,3:6)])

head(my.mat)
                   mpg disp  hp drat    wt
Mazda RX4         21.0  160 110 3.90 2.620
Mazda RX4 Wag     21.0  160 110 3.90 2.875
Datsun 710        22.8  108  93 3.85 2.320
Hornet 4 Drive    21.4  258 110 3.08 3.215
Hornet Sportabout 18.7  360 175 3.15 3.440
Valiant           18.1  225 105 2.76 3.460


apply(my.mat,2, function(x) {c(mean=mean(x),sd=sd(x),range=range(x))})

             mpg     disp        hp      drat        wt
mean   20.090625 230.7219 146.68750 3.5965625 3.2172500
sd      6.026948 123.9387  68.56287 0.5346787 0.9784574
range1 10.400000  71.1000  52.00000 2.7600000 1.5130000
range2 33.900000 472.0000 335.00000 4.9300000 5.4240000
```

Functions - Anonymous Functions

```
my.mat = as.matrix(mtcars[,c(1,3:6)])

apply(my.mat,2, function(x) {
                     c(mean=mean(x),
                     sd=sd(x),
                     range=range(x))
                  })

             mpg     disp        hp      drat        wt
mean   20.090625 230.7219 146.68750 3.5965625 3.2172500
sd      6.026948 123.9387  68.56287 0.5346787 0.9784574
range1 10.400000  71.1000  52.00000 2.7600000 1.5130000
range2 33.900000 472.0000 335.00000 4.9300000 5.4240000

# Or like this

myfunc <- function(x) {
   retvec =  c(mean=mean(x), sd=sd(x), range=range(x))
   return(retvec)
}

apply(my.mat,2,myfunc)
```

Functions - Function Study

Let's build some functions with a bit more utility than the ones we've been looking at. Here we'll implement Newton's method for computing square roots. We need the following information:

```
n - A number for which we will compute its square root

guess - A guess that we will provide to start the process

abs(n - (guess^2)) - The difference between our guess and
                     the target number

tolerance - the value at which we will accept our guess as being accurate

n ~ (n/guess + guess)/2  - Newton's method that will iteratively
                           improve upon our guess until it falls
                           within our specified tolerance.
```

Functions - Function Study

```
Steps involved to compute square root using Newton's method:

1) Get the target number (e.g. 121)

2) Make a first guess (e.g. 9)

3) Select a tolerance value. How close does our answer need to be to the actual
answer before we will accept it ? (e.g. 0.0001)

4) Compute the difference between our first guess squared from the target
value. Is it close enough ?

5) If it is then we are done. If not then we use Newton's formula to improve
our guess.

n ~ (n/guess + guess)/2

6) Then we repeat steps 4 and 5 for as long as the improved answer isn't close
enough.
```

Functions - Function Study

```
n = 121

iterations = 1

guess = 9

tolerance = 0.0001

diff = n-(guess^2)

while (abs(diff) >= 0.001) {
     cat("Iteration number ",iterations,"\n")
     guess = (n/guess + guess)/2
     diff = n-(guess^2)
     iterations = iterations + 1
}

Iteration number 1
Iteration number 2
Iteration number 3

guess
[1] 11
```

Functions - Function Study

```
mynewton <- function(n,guess,toler=0.0001) {

# Function to compute square root of a number n
# INPUT: "n" a positive number
#        "guess" our initial guess
#        "toler" a tolerance threshold

# OUTPUT: a vector containing our computed answer and the number of
#         iterations necessary to achieve it

   retvec = vector()      # Vector to return our answer
   numofiters = 0         # We keep track of how many iterations we do
   diff = n - (guess^2)   # Compute how close our initial guess came

   while( abs(diff) >= toler) {
      guess = (n/guess + guess)/2
      diff = n - (guess^2)
      numofiters = numofiters + 1
   }
   return(c(lastguess=guess,iterations=numofiters))
}

mynewton(121,9)

 lastguess iterations
  11          3
```

Functions - Function Study

Many times we "factor" out or functions into specialized sub functions to do specific things. This is common if we work on a team. One person does one function, someone else does another, etc.

```
improve <- function(guess, n) {
  return((n/guess + guess)/2)
}

good_enough <- function(n, guess) {
  diff = abs(n - guess^2)
  return(diff < 0.001)
}

square_root <- function(n, guess) {
  while(!good_enough(n, guess)) {
    guess = improve(guess, n)
  }
  return(guess)
}

my_sqrt <- function(n,guess) {
 r = square_root(n, guess)
 return(r)
}

my_sqrt(121,9)
[1] 11
```