

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.

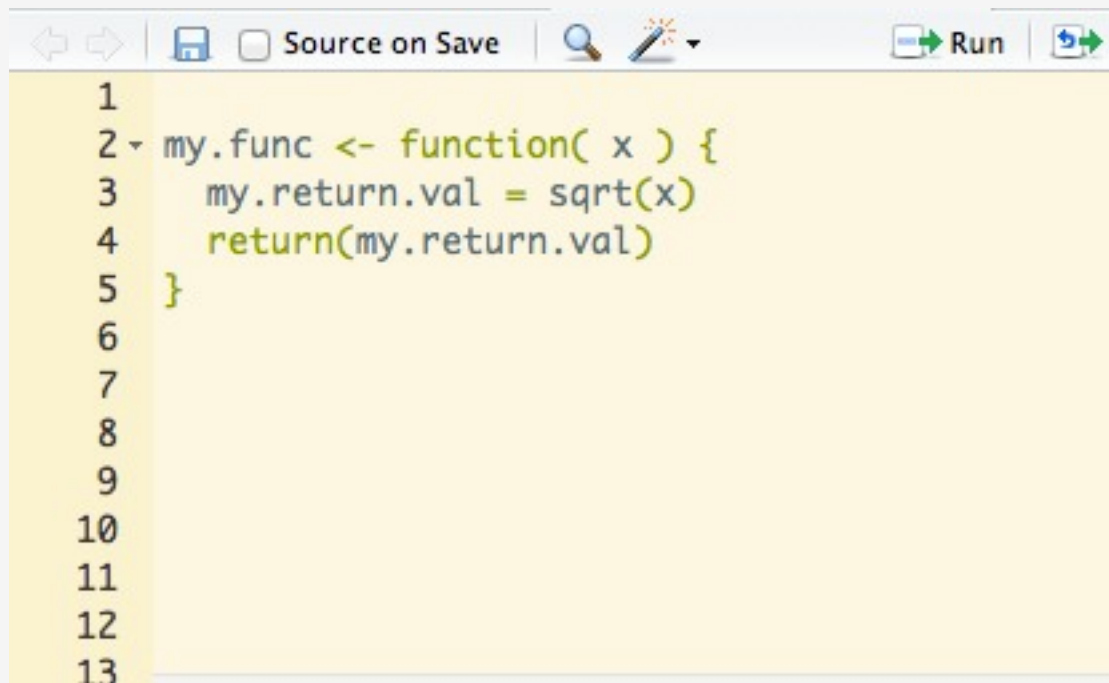
```
my.func <- function( x ) {  
  my.return.val = sqrt(x)  
  return(my.return.val)  
}
```

```
fix(my.func)
```

```
-- STARTS AN EDIT SESSION ON THE FUNCTION --
```

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".



The screenshot shows the RStudio Edit Window with a function declaration. The window has a toolbar at the top with icons for navigation, saving, and running. The code is as follows:

```
1
2 my.func <- function( x ) {
3   my.return.val = sqrt(x)
4   return(my.return.val)
5 }
6
7
8
9
10
11
12
13
```

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)  
}
```

```
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)  
}  
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)  
}
```

```
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:

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

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

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

  for (ii in 1:length(xvals)) {  # begin for loop
    yvals[ii] = xvals[ii]^2
  }                               # end for loop

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

}  # End function
```

Functions - Vectors

This version returns the xvals and yvals in a list. We mimic what many functions in R do (e.g. lm).

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

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

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

  for (ii in 1:length(xvals)) {  # begin for loop
    yvals[ii] = xvals[ii]^2
  }                               # end for loop

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

  retlist = list(x=xvals, y=yvals)
  return(retlist)

}  # End function
```

Functions - Vectors

We could add in "Arguments" to influence the color of the plot.

```
myplotter <- function(xvals, 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] = xvals[ii]^2  
  }  
  
  plot(xvals, yvals, main="Super Cool Data Plot",type="l",col=plotcolor)  
  retlist = list(x=xvals, y=yvals)  
  return(retlist)  
}  
  
xvals = seq(-3,3)  
  
myplotter(xvals,plotcolor="red")
```

Functions - Vectors

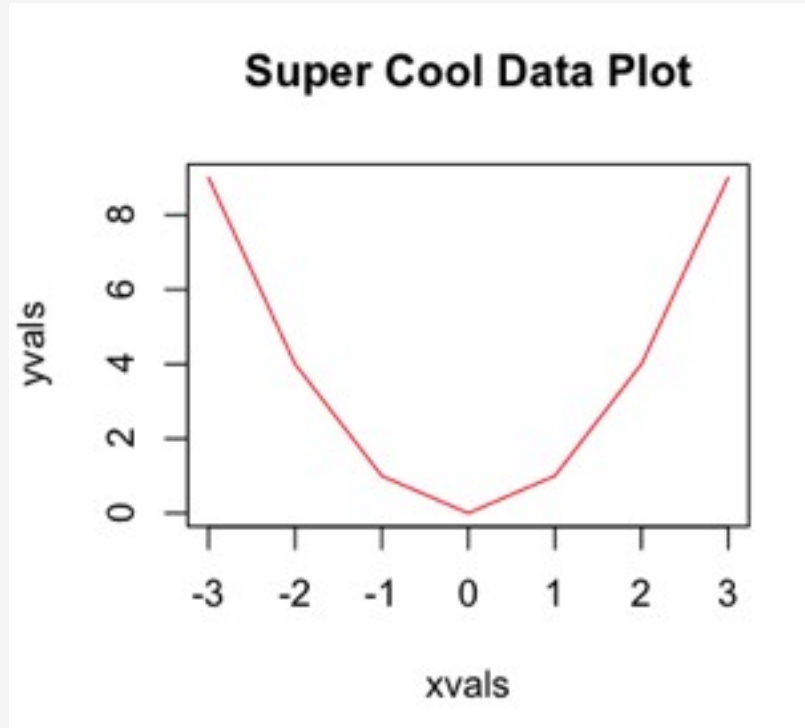
```
> myplotter(seq(-3,3),plotcolor="red")
```

```
$x
```

```
[1] -3 -2 -1  0  1  2  3
```

```
$y
```

```
[1] 9 4 1 0 1 4 9
```



Functions - Multiple Branch if Statement

Let's look at some of the structures from last week to see how they might look as functions. Let's put the grading loop into a function:

```
score = c(74,68,98,90,100,67,59)

for (ii in 1:length(score)) {
  if (score[ii] >= 100) {
    grade = "A+"
  } else if (score[ii] >= 90 & score[ii] < 100 ) {
    grade = "A"
  } else if (score[ii] >= 80 & score[ii] < 90) {
    grade = "B"
  } else if (score[ii] >= 70 & score[ii] < 80) {
    grade = "C"
  } else if (score[ii] >= 60 & score[ii] < 70) {
    grade = "D"
  }
  else {
    grade = "F"
  }
  print(grade)
}
```

Functions - Multiple Branch if Statement

Let's look at some of the structures from last week to see how they might look as functions.
Let's put the grading loop into a function:

```
score = c(74,68,98,90,100,67,59)

mygrader <- function(somescores) {

  for (ii in 1:length(somescores)) {      # Begin for loop
    if (score[ii] >= 100) {                # Begin if
      grade = "A+"
    } else if (score[ii] >= 90 & score[ii] < 100 ) {
      grade = "A"
    } else if (score[ii] >= 80 & score[ii] < 90) {
      grade = "B"
    } else if (score[ii] >= 70 & score[ii] < 80) {
      grade = "C"
    } else if (score[ii] >= 60 & score[ii] < 70) {
      grade = "D"
    }
    else {
      grade = "F"
    }
    print(grade)      # return the student's grade
  } # End for loop

} # End function
```

Functions - Multiple Branch if Statement

```
score = c(74,68,98,90,100,67,59)
```

```
mygrader(score)
```

```
[1] "C"
```

```
[1] "D"
```

```
[1] "A"
```

```
[1] "A"
```

```
[1] "A+"
```

```
[1] "D"
```

```
[1] "F"
```

Okay but let's return a vector that we could use as possible input to another function:

Functions - Multiple Branch if Statement

```
mygrader <- function(somescores) {  
  gradevec = vector() # setup a blank vector to contain grades  
  for (ii in 1:length(somescores)) {  
    if (somescores[ii] >= 100) {  
      gradevec[ii] = "A+"  
    } else if (somescores[ii] >= 90 & somescores[ii] < 100 ) {  
      gradevec[ii] = "A"  
    } else if (somescores[ii] >= 80 & somescores[ii] < 90) {  
      gradevec[ii] = "B"  
    } else if (somescores[ii] >= 70 & somescores[ii] < 80) {  
      gradevec[ii] = "C"  
    } else if (somescores[ii] >= 60 & somescores[ii] < 70) {  
      gradevec[ii] = "D"  
    }  
    else {  
      gradevec[ii] = "F"  
    }  
    # End if statement  
  } # End For Loop  
  return(gradevec) # return the student's grade  
} # End function definition
```

Functions - Multiple Branch if Statement

```
mygrader(score)
```

```
> mygrader(score)
```

```
[1] "C" "D" "A" "A" "A+" "D" "F"
```

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
```

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]  
      }  
    }  
  } 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]  
      }  
    }  
  }  
  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)) {  
  tempdf = mysplits[[ii]]  
  recstosample = sample(1:nrow(tempdf),2,F)  
  print(tempdf[recstosample,])  
}
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
Mazda RX4	21	6	160	110	3.9	2.620	16.46	0	1	4	4

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4

Functions - Split Dataframes

What would this look like in a function ?

```
myfunc <- function(somedf, somefac) {  
  
  # Function to split a data frame by a given factor  
  # Input: A data frame, a factor  
  # Output: Sampled records from each split  
  
  mysplits = split(somedf,somefac) # Split the data frame by somefac  
  
  for (ii in 1:length(mysplits)) { # loop through the splits  
    tempdf = mysplits[[ii]]      # Create a variable to hold each split  
  
    recstosample = sample(1:nrow(tempdf),2,F) # Sample from the split  
    print(tempdf[recstosample,])  
  }  
}
```

Notice here that we don't really return anything. We just print out the records. We could stash the sampled records into a list.

Functions - Split Dataframes

Let's return a list that contains the sampled records from each list

```
myfunc <- function(somedf, somefac) {  
  
  # Function to split a data frame by a given factor  
  # Input: A data frame, a factor  
  # Output: A list containing the sampled records from each split  
  
  retlist = list()    # Empty list to return the sampled records  
  mysplits = split(somedf,somefac) # Split the data frame by somefac  
  
  for (ii in 1:length(mysplits)) { # loop through the splits  
    tempdf = mysplits[[ii]]        # Create a variable to hold each split  
  
    recstosample = sample(1:nrow(tempdf),2,F) # Sample from the split  
    retlist[[ii]] = tempdf[recstosample,]  
  }  
  return(retlist)  
}
```

Functions - Split Dataframes

What would this look like in a function ?

```
myfunc(mtcars,mtcars$cyl)
```

```
[[1]]
```

		mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Porsche	914-2	26.0	4	120.3	91	4.43	2.14	16.70	0	1	5	2
Datsun	710	22.8	4	108.0	93	3.85	2.32	18.61	1	1	4	1

```
[[2]]
```

		mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Valiant	18.1	6	225.0	105	2.76	3.46	20.22	1	0	3	1	
Merc	280	19.2	6	167.6	123	3.92	3.44	18.30	1	0	4	4

```
[[3]]
```

		mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Merc	450SL	17.3	8	275.8	180	3.07	3.73	17.6	0	0	3	3
Merc	450SLC	15.2	8	275.8	180	3.07	3.78	18.0	0	0	3	3

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

Write a function that given a data frame and some column numbers (or names) the function will return just those columns:

```
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 - Matrix

Write a function that given a data frame and some column numbers (or names) the function will return just those columns:

```
mtcenter(ab)
```

```
[1,]  -1  -1  -1
[2,]   0   0   0
[3,]   1   1   1
```

```
apply(ab,2,function(x) x-mean(x))
```

```
  [,1] [,2] [,3]
[1,]  -1  -1  -1
[2,]   0   0   0
[3,]   1   1   1
```

Functions - Practice

Given a single number "x" and a single number "n" write a function to compute:

$$x^1 + x^2 + x^3 + \dots x^n$$

For example if $x = 0.5$ and $n = 5$.

$$(0.5)^1 + (0.5)^2 + (0.5)^3 + (0.5)^4 + (0.5)^5 = 0.96875$$

```
myseries <- function(x,n) {
```

```
# Input: two single numbers
```

```
# Output: a single number
```

```
# What goes here ? A loop ? An if statement ? Both ? Neither ?
```

```
}
```

```
myseries(0.5,5)
```

```
[1] 0.96875
```