

BIOS 545 Week 4, Lecture 1

Department of Biostatistics and Bioinformatics

Steve Pittard wsp@emory.edu

February 1, 2016

Functions

- 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

Version 1 - Simple Square root

```
myfunc <- function(somenum) {  
  retval <- sqrt(somenum)  
  return(retval)  
}
```

Version 2 - Square root and complex numbers

```
myfunc <- function(somenum) {  
  if (somenum < 0 ) {  
    retval <- sqrt(as.complex(somenum))  
  } else {  
    retval <- sqrt(somenum)  
  }  
  return(retval)  
}
```

Functions

Version 3 - Provide an argument to select complex capability or not

```
myfunc2 <- function(somenum,complex="n") {  
  if (somenum < 0 & complex == "n") {  
    stop("HEY !! - Cannot take square root of negative number")  
  } else if (somenum < 0 & complex == "y") {  
    retval <- sqrt(as.complex(somenum))  
  } else {  
    retval <- sqrt(somenum)  
  }  
  return(retval)  
}
```

Functions

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

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

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

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

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

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

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.

Functions

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** A computed expression to be returned

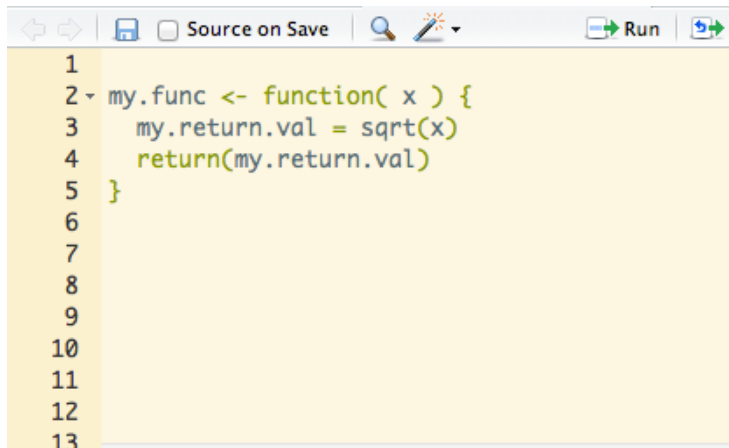
```
my.func <- function(somenum) {  
  my.return.val <- sqrt(somenum)  
  return(my.return.val)  
}
```

```
my.func(10)  
[1] 3.162278
```

```
mycomputation <- my.func(10)
```

Functions

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. The top toolbar includes navigation arrows, a 'Source on Save' checkbox, a search icon, a 'Run' button with a green arrow, and a 'Refresh' button. The main text area has a yellow background and contains the following R code:

```
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

Some guidelines

- 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 in my opinion it should be
- 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
- If you want to return data of different types then returning a list is your only choice (e.g. look at the results from the **lm** function)

Functions

Determine what you are being asked to do. This is easy for purposes of this class. You will be told:

- What the function will accept as input (e.g. vector, matrix, data frame)
- What arguments the function will accept
- What to return - what the output will be
- Make a shell like the following and build into it:

```
myfunc <- function(somevec) {
```

```
# Code goes here
```

```
}
```

Functions

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)  
}
```

```
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

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

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

What happens if you give the function some 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

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

Maybe we should so 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

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(hypotenuse = 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

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

Ask yourself what are the :

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

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)  
  
} # End function
```


Functions

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

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

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(0,3)
for (ii in 1:length(x)) {
  y[ii] <- (x[ii])^2
}
```

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

Functions

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 an arbitrary y function

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

  # Function at which to evaluate each x (must be valid)
  # Input: xvalues - a vector
  #       : mfunc - a function to apply to each value of xvalues
  # 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")

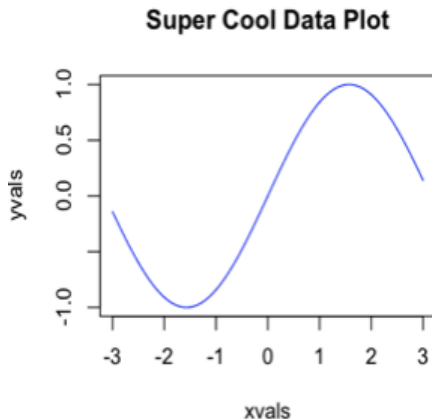
}  # End function
```

Functions

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

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

```
myplotter(xvals,sin)
```



Functions

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 at which to evaluate each x (must be valid)  
  # Input: xvalues - a vector of values  
  #       : mfunc - a function to apply to xvalues  
  #       : plotcolor - the color used when plotting  
  # 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)
```

Functions

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

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

```
$x
```

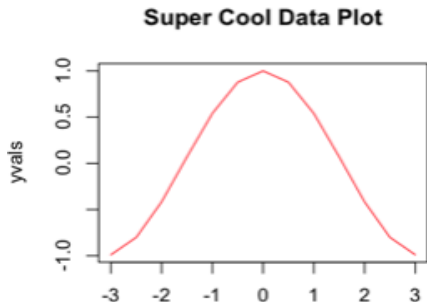
```
[1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0
```

```
$y
```

```
[1] -0.9899925 -0.8011436 -0.4161468  0.0707372  0.5403023  0.8775826
```

```
[7] 1.0000000  0.8775826  0.5403023  0.0707372 -0.4161468
```

```
[12] -0.8011436 -0.9899925
```



Functions

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

```
set.seed(188)
somevector <- 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]
  }
}

mymin
[1] -3.422185
```


Functions

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

```
mymin <- function(somevector) {  
  
  # Function to find the minimum value in a vector  
  # Input: somevector - 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

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

Let's make an argument that let's us specify 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 maximum to an arbitrary value  
  
    for (ii in 1:length(somevector)) {  
      if (somevector[ii] > myval) {  
        myval <- somevector[ii]  
      }  
    } # End for  
  } # End If  
  return(myval)  
}
```

Functions

Let's make an argument that let's us specify the min or max

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

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

```
min(testvec)  
[1] -3.84532
```

```
max(testvec)  
[1] 3.847768
```

Functions

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

```
myfunc <- function(somedf, somefac) {  
  
  # Function to split a data frame by a given factor  
  # Input: somedf - A data frame, somefac - a factor by which to split somedf  
  # 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

It is worth it to note that the previous function could be rewritten using the `lapply` function

```
myfunc <- function(somedf, somefac) {  
  
  # Function to split a data frame by a given factor  
  # Input: somedf - A data frame, somefac - a factor by which to split somedf  
  # Output: A list containing a count of records in each group  
  
  mysplits <- split(somedf,somefac)  
  return(lapply(mysplits,function(x) nrow(x)))  
}  
  
> myfunc(mtcars,mtcars$am)  
$`0`  
[1] 19  
  
$`1`  
[1] 13
```

Functions - Anonymous

- 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	dis	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

Functions - Anonymous

In this example we call the mean function on all the columns in the matrix. Note that the mean function isn't anonymous. It has a name.

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

mpg	disp	hp	drat	wt
20.090625	230.721875	146.687500	3.596563	3.217250

But what if we wanted to provide our own custom function ? Well we could write one in advance and then use it with apply. Here we write a function to sum all elements in a vector.

```
mysum <- function(x) {  
  return(sum(x))  
}
```

```
# We now use it to sum each column in the matrix
```

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

mpg	disp	hp	drat	wt
642.900	7383.100	4694.000	115.090	102.952

Functions - Anonymous

But since this function is so simple we could define it as we make the call to apply.

```
apply(my.mat, 2, function(x) sum(x))
```

mpg	disp	hp	drat	wt
642.900	7383.100	4694.000	115.090	102.952

This function lives only for the length of the call to apply. It is so “temporary” that we don’t even bother to give it a name so it is an *anonymous* function.

Functions - Anonymous

```
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

```
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

Functions - Anonymous

In my opinion it is better to first define the function before using it. I feel that this makes it easier since you can fully test and debug the function independently of the apply command.

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

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

	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

Let's write a function to compute the median of a vector. Computing the median of a vector involves finding the middle value in a vector.

To do this you have to first determine if a vector is of even length or odd length. Based on that answer you will need to apply a different formula.

As an example consider the vector below:

```
exampodd <- c(3,6,9,1,10)
```

```
svec <- sort(exampodd)  
[1] 1 3 6 9 10
```

Functions

So what element is in the middle ? The third element whose value is 6.

Divide the length of the vector in half and use one of the numeric functions (round or ceiling) to get the middle element number and then use that to index into the vector.

So in our case we have "3" as the middle element so `svec[3]` is equal to 6. Does this match what the built in R function returns ?

```
exampodd <- c(3,6,9,1,10)
```

```
(svec <- sort(exampodd))
```

```
[1] 1 3 6 9 10
```

```
idx <- ceiling((length(svec))/2)
```

```
svec[idx]
```

```
[1] 6
```

```
median(svec)
```

```
[1] 6
```

Functions

But what about the case wherein the length of the vector is even ? How do we find its median ?

```
( exampeven <- c(11,9,4,7) )
```

```
svec <- sort(exampeven)
[1] 4 7 9 11
```

```
idx <- (length(svec))/2      # element 2 is one of the middle
mean(c(svec[idx],svec[idx+1])) # Add one to element number (r
median(exampeven)
[1] 8
```


Functions

Here we sort the vector, then divide its length in half to find the middle two values after which we take their average. As before we'll need to use one or more of the numeric functions to access the correct elements.

Here, the middle two elements are 7 and 9 so we take their mean to get a value of 8.

```
exampeven <- c(11,9,4,7)
```

```
exampeven
```

```
[1] 11 9 4 7
```

```
mys <- sort(exampeven)
```

```
[1] 4 7 9 11
```

```
median(exampeven)
```

```
[1] 8
```

Functions

Here is some “psuedo code” for finding the median of a vector

If the length of the vector is odd

- 1 Sort the vector
- 2 Divide the length by 2 and pass it to the ceiling function
- 3 Use this to index into the sorted vector to get the median
- 4 Store the result in a return variable

else

- 1 Sort the vector
- 2 Divide its length by 2
- 3 Take the mean of the numeric result and the numeric result + 1
- 4 Store the result in a return variable

Functions

Use this format. you can do some basic argument checking if you wish. for example check that the vector being passed is numeric.

```
mymedian <- function(somevec) {  
  
  # Input: A vector of numbers  
  # Output: A single value representing the median value  
  
  return(compmed)  # Or whatever you want to your computed median  
}
```

Functions

Let's write a function that simulates the roll of a single die. This is easy using the sample function. We don't even have to write a function

```
sample(1:6,1)    # That's it. This gives us a result of 1,2,...6
```

But let's say we want to roll the dice a certain number of times. We could put this into a function for later use if we wanted.

```
roller <- function(timestoroll=5) {  
  results <- sample(1:6,timestoroll,replace=TRUE)  
  return(results)  
}
```

Okay so let's play a dice game wherein we keep rolling a single dice until we get a certain number. So we won't know in advance how many times we need to roll the dice to reach the target number. For example how many times do we need to roll the dice before we get a 6 ? We don't know. To program this then perhaps a while loop might be effective.

Functions

Here is some pseudo code for doing this

- Pick a target number (set a default for the argument)
- Set up a counter variable to determine how many rolls it took before seeing the target
- Roll the dice and save the result into a variable
- While that variable is not equal to the target number then keep rolling the dice and increment the counter
- Once you roll the target number return the counter variable.

```
roller <- function(target=6) {  
  # Input: target (the number we are trying to get)  
  # Output: a single numeric value representing the number of rolls it  
            took to get the value  
  
  counter <- 1   # initialize the counter  
  
  (your code goes here)  
  
  return(counter)  
}
```