BIOS 545 - Functions Part 2

Some hints when writing functions

- Keep functions short. If your function is long then split it up into different functions
- Functions can call other functions
- Keeping functions separate makes it easy to identify possible problems
- Try your function with a large variety of data to find possible problems
- Use if statements up front to catch bad input values
- Put comments in your code so you and others who read your code will know what you are trying to do

In R, the global frame is called the global environment or the workspace, and is kept in memory

- Scoping Rules determine where the interpreter looks for values of free variables
- An environment is a sequence of frames
- A value bound to a variable in a frame earlier in the sequence will take precedence over a value bound to the same variable in a frame later in the sequence. The first value is said to shadow or mask the second.

Environments

The environment command shows you the current environment. Normally you don't worry about this although it is important to understand since it is linked to scope.

```
environment()
```

```
## <environment: R_GlobalEnv>
```

```
ls()
```

```
## character(0)
```

You can remove all variables in your current environment. This is for when you want to "clean house" and start over with an empty environment which is also known as a "namespace".

```
rm(list=ls())
```

As proof that functions run within their own environment consider this example:

```
environment()
```

```
## <environment: R_GlobalEnv>
```

```
# Write a function to print the current environment

myenvfunc <- function() {
  print(environment())
}
myenvfunc()
```

```
## <environment: 0x7f9b80028380>
```

So myenvfunc runs in its OWN environment that is separate from the Global environment.

In this next example an object x will be defined with value zero. Inside myfunc, the x is defined with value 3.

Executing the function myfunc will not affect the value of the global variable "x".

```
# Set x to zero in the global environment (your console)
x <- 0

myfunc <- function(x) {
  x <- 3
  return(x) }

cat("Result from myfunc is: ",myfunc(),"\n")
```

```
## Result from myfunc is:  3
```

```
cat("The value of x in the Global env is: ",x,"\n")
```

```
## The value of x in the Global env is:  0
```

This means a normal assignment within a function will not overwrite objects outside the function. An object created within a function will be lost when the function has finished.

```
# First, remove all references to variables
rm(list=ls()) # Clears all variables from your environment.

# Define a function that references x and a albeit
# from within a function

exampf <- function(x) {
  return(x + a)
}
ls()
```

```
## [1] "exampf"
```

Let's call the function exampf with a value of 2. This will fail. Why ?

```
exampf(2)
```

When f is called it passes the value of 2. So "x" assumes a local value of 2. Then the function wants to add x = 2 to the value of a though non has been specified so the function results in an error. This seems reasonable since R can't find a variable called "a" anywhere.

Let's try this again except this time we will set values for **a** and **x** in the global environment before we call **exampf**

```
exampf <- function(x) {
  return(x + a)
}

a <- 10
x <- 5

ls()
```

```
## [1] "a"      "exampf" "x"
```

```
exampf(2)
```

```
## [1] 12
```

When exampf is called, the *local* binding of x <- 2 shadows the *global* binding x <- 5. The variable a is a free variable in the frame of the function call, and so the global binding of a <- 10 will apply.

```
# Set the variables in the global environment
a <- 10
x <- 5

# Define a function f in the global environment
f <- function(x) {
  a <- 5
  return(g(x))
}

# Define a function g in the global environment
g <- function(y) {
  return(y + a)
}
```

In R, the global binding a <- 10 is used when f calls g, because a is a free variable in g, and g is defined at the command prompt in the global frame.

```
f(2)
```

```
## [1] 12
```

```
# Define a and x in the global fame / environment
a <- 10
x <- 5

# Define f in the Global frame / environment
f <- function(x) {
  a <- 5
  # Define g within the environment of f
  g <- function(y) {
    return(y + a)
  }
  g(x)
}

f(2)
```

```
## [1] 7
```

What does this all mean ?

The arguments and variables that you use within a function are private to that function even if you use the same name as a previously defined variable

It is a common mistake to refer to a variable within a function without initializing it to something first. If it has the same name as a variable in the **global environment** then it will pick up that variable's value

So make sure you keep track of what you are doing within your function

Use descriptive and unambiguous variable names

```
exampf <- function(x) {
  a <- 3
```

```
  return(x + a)
}

# Maybe do something like:

exampf <- function(exampx) {
  exampa <- 3
  return(exampx + exampa)
}

#

exampf <- function(exampx, exampa) {
  return(exampx + exampa)
}
```

Functions can always call other functions that exist within the same environment. It happens all the time and is one of the more common activities in R

```
exampf <- function(myvec) {
  # mean and sd are known
  retval <- c(mean(myvec), sd(myvec))
  return(retval)
}

exampf(1:10)
```

```
## [1] 5.50000 3.02765
```

Define this function within RStudio either at the console or from the edit window.

It is now in your "Global Environment"and can be used at the prompt or by other functions that you might write

```
is_odd <- function(somenumber) {
  retval <- 0
  if (somenumber %% 2 != 0) {
       retval <- TRUE
  } else {
       retval <- FALSE
  }
  return(retval)
}

is_odd(3)
```

```
## [1] TRUE
```

Now we could perhaps use this function to help us when computing the median of a function. Remember that we wrote our own function to do this even though there is an existing built-in function. My version looks something like this:

```
find_median <- function(somevector) {
# Author: wsp@emory.edu
# Purpose: To find the median of an input vector
# INPUT:  somevec - a numeric vector
# OUTPUT: median - a single number representing the
```

```
#         median of somevector
#
  # Do some basic error checking
  if (!is.numeric(somevector)) {
    stop("I need a numeric vector to do this work")
  }

  # Sort the input vector - we have to do this for both cases
  # so we just do it here
  sorted_vector <- sort(somevector)

  # Next we figure out if the vector is of even or odd length
  sorted_vector_length <- length(sorted_vector)

  if (sorted_vector_length %% 2 !=0) {
     # Odd length means we get the middle value

     idx <- ceiling(sorted_vector_length/2)
     median <- sorted_vector[idx]

  } else {
     # Even length which indicates we take the mean of
     # the two middle values

     idx <- (sorted_vector_length/2)
     median <- mean(c(sorted_vector[idx],sorted_vector[idx+1]))
  }

  # Return the computed median variable
  return(median)
}
```

We could run this now:

```
find_median(1:20)
```

```
## [1] 10.5
```

One thing to notice is that part of our coding logic is to determine if the length of the input vector is even or odd. While we could do this in our function, we could also use our **is_odd** function. There are a couple of ways to proceed.

```
find_median <- function(somevector) {
# Author: wsp@emory.edu
# Purpose: To find the median of an input vector
# INPUT:  somevec - a numeric vector
# OUTPUT: median - a single number representing the
#         median of somevector
#
  # Do some basic error checking
  if (!is.numeric(somevector)) {
    stop("I need a numeric vector to do this work")
  }

  # Sort the input vector - we have to do this for both cases
  # so we just do it here
```

```
  sorted_vector <- sort(somevector)

  # Next we figure out if the vector is of even or odd length
  sorted_vector_length <- length(sorted_vector)

  if (is_odd(sorted_vector_length)) {
     # Odd length means we get the middle value

     idx <- ceiling(sorted_vector_length/2)
     median <- sorted_vector[idx]

  } else {
     # Even length which indicates we take the mean of
     # the two middle values

     idx <- (sorted_vector_length/2)
     median <- mean(c(sorted_vector[idx],sorted_vector[idx+1]))
  }

  # Return the computed median variable
  return(median)
}
```

Let's execute the function:

```
find_median(1:20)
```

```
## [1] 10.5
```

Another approach would be to define the **is_odd** function within the **find_median** function.

```
find_median <- function(somevector) {
# Author: wsp@emory.edu
# Purpose: To find the median of an input vector
# INPUT:  somevec - a numeric vector
# OUTPUT: median - a single number representing the
#         median of somevector
#
  # Do some basic error checking
  if (!is.numeric(somevector)) {
    stop("I need a numeric vector to do this work")
  }

  # Define the is_odd function
  is_odd <- function(somenumber) {
     retval <- 0
     if (somenumber %% 2 != 0) {
         retval <- TRUE
     } else {
         retval <- FALSE
     }
     return(retval)
  }

  # Sort the input vector - we have to do this for both cases
```

```
  # so we just do it here
  sorted_vector <- sort(somevector)

  # Next we figure out if the vector is of even or odd length
  sorted_vector_length <- length(sorted_vector)

  if (is_odd(sorted_vector_length)) {
     # Odd length means we get the middle value

     idx <- ceiling(sorted_vector_length/2)
     median <- sorted_vector[idx]

  } else {
     # Even length which indicates we take the mean of
     # the two middle values

     idx <- (sorted_vector_length/2)
     median <- mean(c(sorted_vector[idx],sorted_vector[idx+1]))
  }

  # Return the computed median variable
  return(median)
}
```

Let's run this function:

```
find_median(1:20)
```

```
## [1] 10.5
```

Function Arguments

Let's revisit our function to implement the Pythagorean theorem.

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

We can call this function a couple of ways. In this call to the function, the number 4 is match to the argument "a" and 5 is matched to the argument "b". This is called "positional" matching.

```
pythag(4,5)
```

```
## $hypoteneuse
## [1] 6.403124
##
## $sidea
## [1] 4
##
## $sideb
## [1] 5
```

In the following call to the function, the number 5 is matched to "a" and 4 is mathced to "b". This does not make a major difference in the result since the math is basically the same.

```
pythag(5,4)
```

```
## $hypoteneuse
## [1] 6.403124
##
## $sidea
## [1] 5
##
## $sideb
## [1] 4
```

Now, let's look at this situation in general. Consider the **mean** function which takes three arguments (look at the help pages):

x: a value or vector to take the mean of trim: a value that let's you trim the vector by na.rm: ignore missing values

```
set.seed(1)
myx <- rnorm(20)

mean(myx)
```

```
## [1] 0.1905239
```

Okay, that was straightforward in that **myx** matches the "x" argument which is expected to be numeric. It is also the first argument to be expected. So what about the following:

```
# myx MATCHES "x" and 0.05 MATCHES "trim"
mean(myx,0.05)
```

```
## [1] 0.2461054
```

And consider the following:

```
# myx MATCHES "x", 0.05 MATCHES "trim", TRUE matches na.rm
mean(myx,0.05,TRUE)
```

```
## [1] 0.2461054
```

So thus far, all of the values we supplied are of the required type for the **mean** function to do its job. As long as we know what the function expectes then we don't necessarily have to name the arguments as we supply values for them. For simple and straightforward functions, this is usually easy to do.

We could explicitly name the arguments as we provide values for them. This way, R will never be confused about what value corresponds to what argument.

```
set.seed(1)
mean(x = myx, trim = 0.05, na.rm = TRUE)
```

```
## [1] 0.2461054
```

```
# As long as you name the arguments you type them in any order
mean(trim = 0.05, na.rm = TRUE, x = myx)
```

```
## [1] 0.2461054
```

```
# BUT THE FOLLOWING WON'T WORK !
mean(TRUE,0.05,myx)
```

```
## Warning in if (na.rm) x <- x[!is.na(x)]: the condition has length > 1 and only
## the first element will be used
```

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

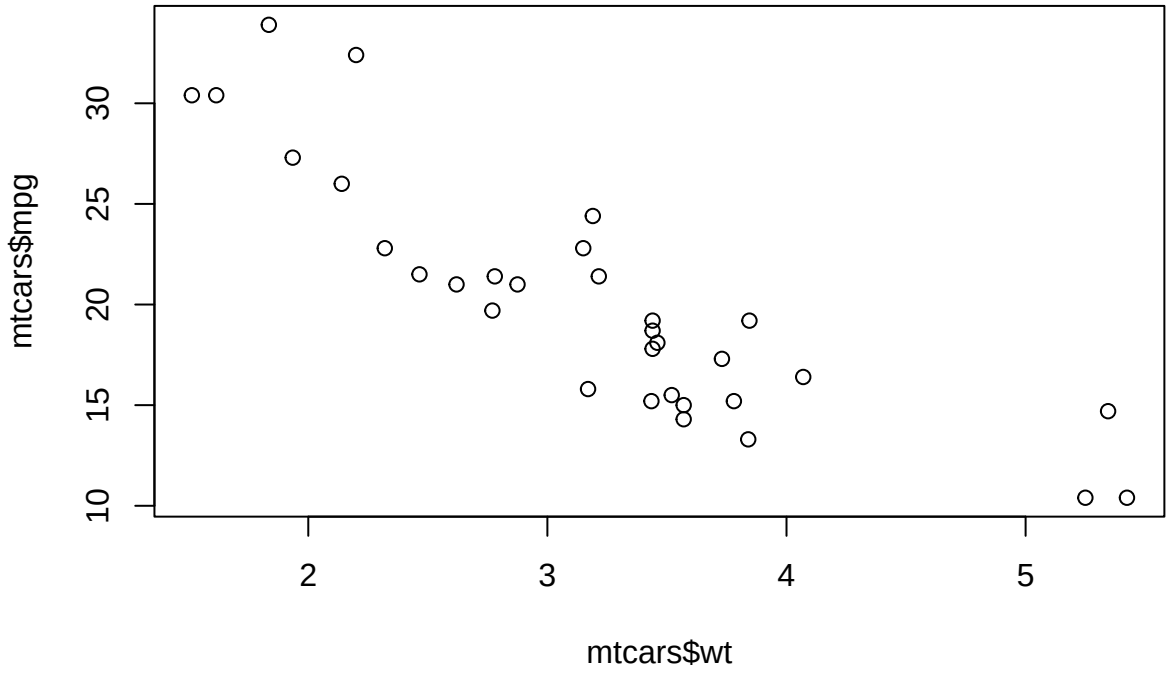
Use named arguments especially in cases where there is a long list of arguments

If you use named arguments then you don't have to remember the position of the arguments.

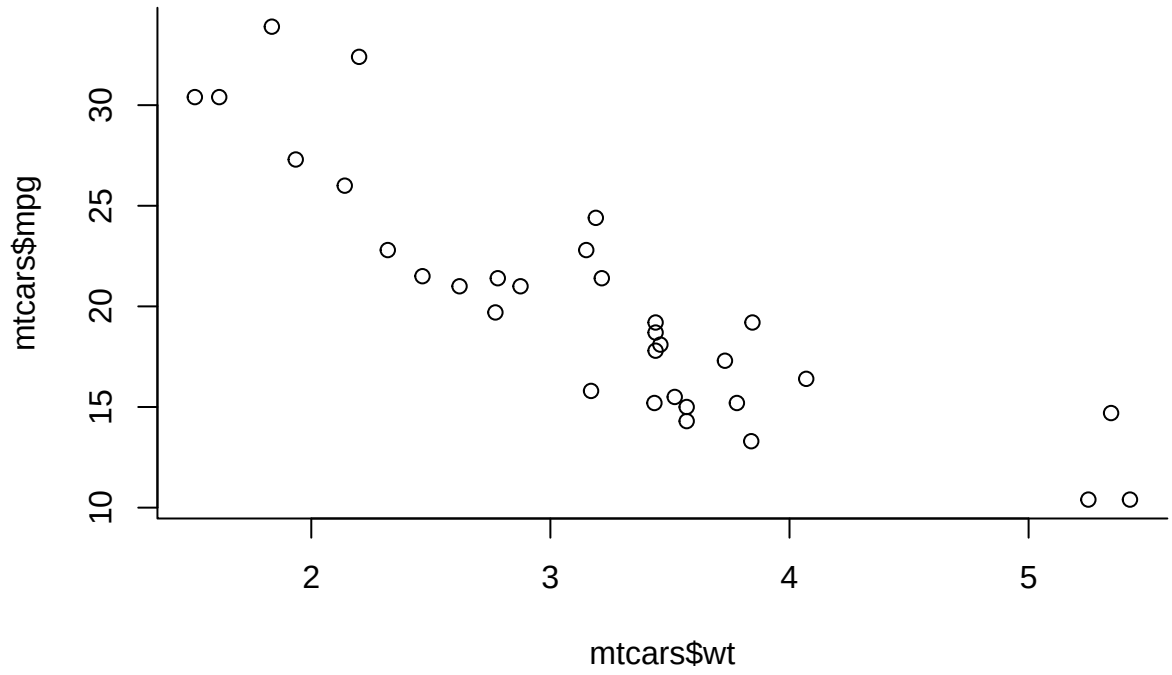
Check out the plot command, which has way too many options for anyone to reasonably remember (well most people anyway)

There is no convenient way to remember the arguments by position unless you have a photographic memory. Consider the **plot** function.

```
plot(x=mtcars$wt, y=mtcars$mpg)
```

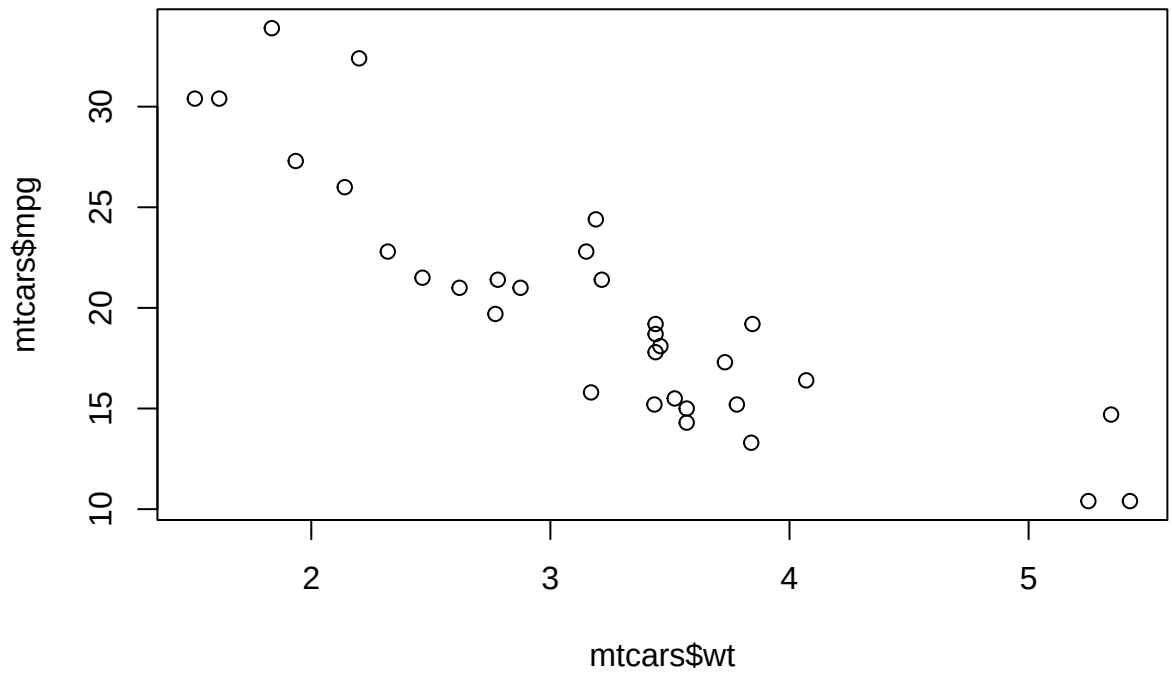



```
plot(x=mtcars$wt, y=mtcars$mpg, bty="l")
```

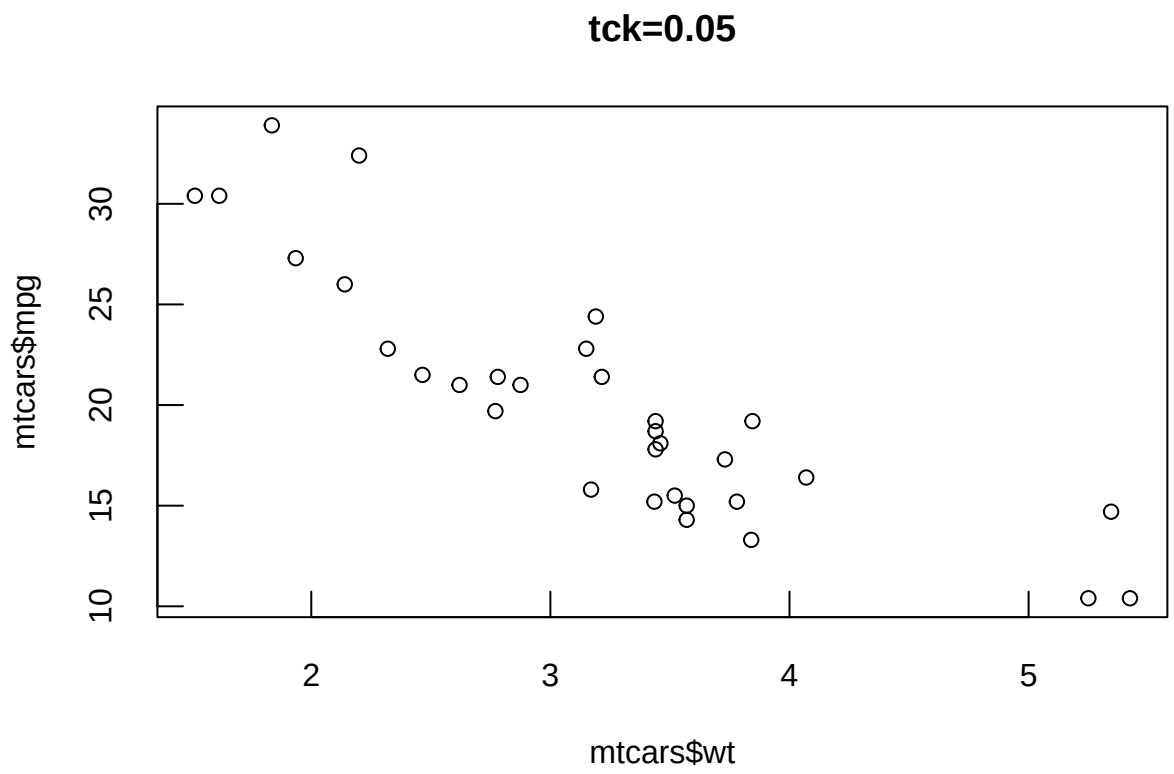



```
plot(x=mtcars$wt, y=mtcars$mpg, main="Default")
```

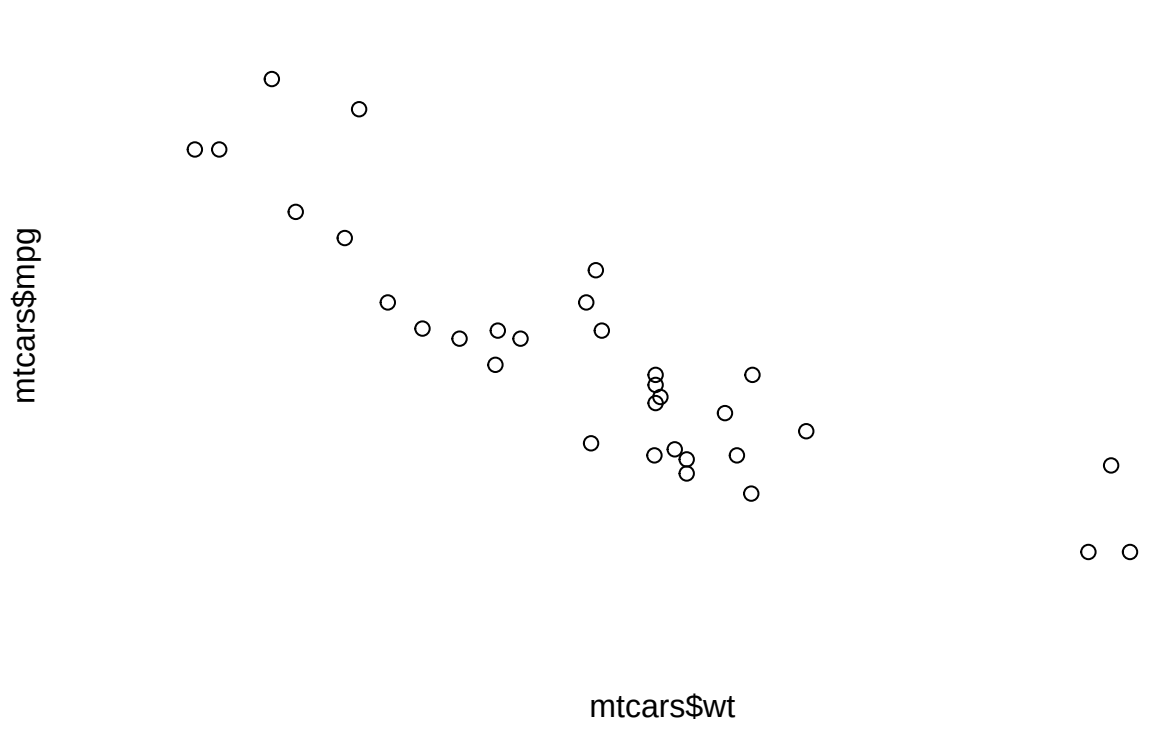
Default



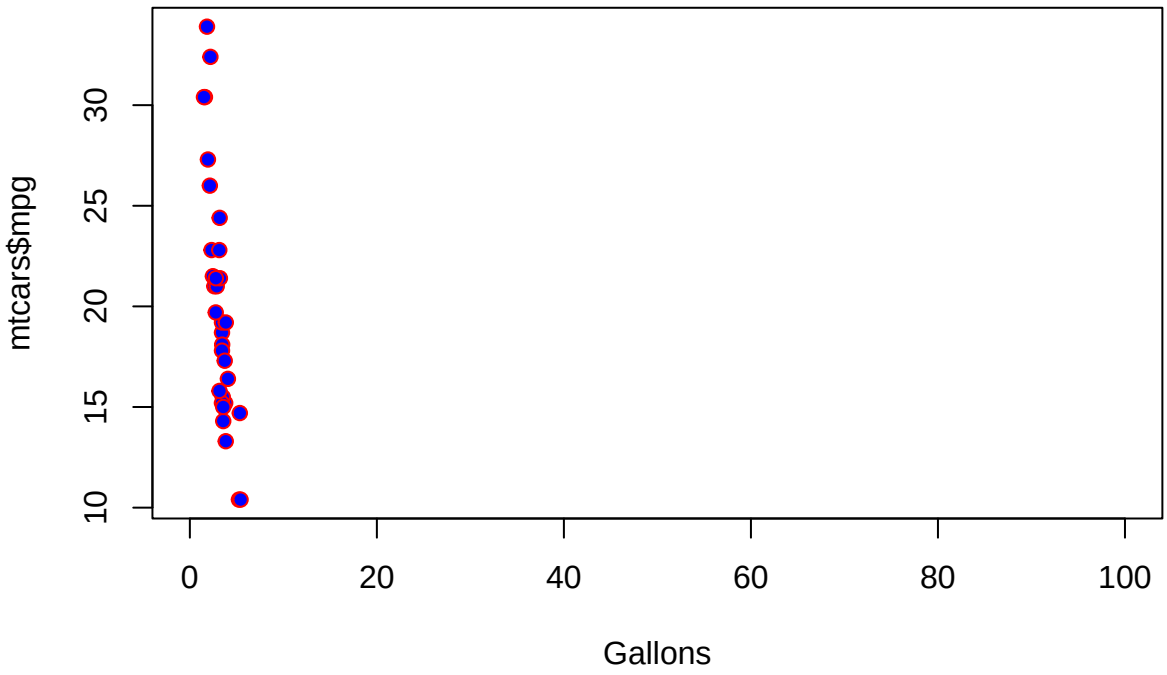

```
plot(x=mtcars$wt, y=mtcars$mpg, tck=0.05, main="tck=0.05")
```




```
plot(x=mtcars$wt, y=mtcars$mpg, axes=F,
     main="Different tick marks for each axis")
```




```
plot(x=mtcars$wt, y=mtcars$mpg,xlim=c(0,100),
     xlab="Gallons",pch=21,bg="blue",col="red")
```




```
?mean
```

The "..." argument

We can pass an unspecified number of parameters to a function by using the ... notation in the argument list

This is usually done when you want to give the user the option of passing any number of arguments that are intended for another function

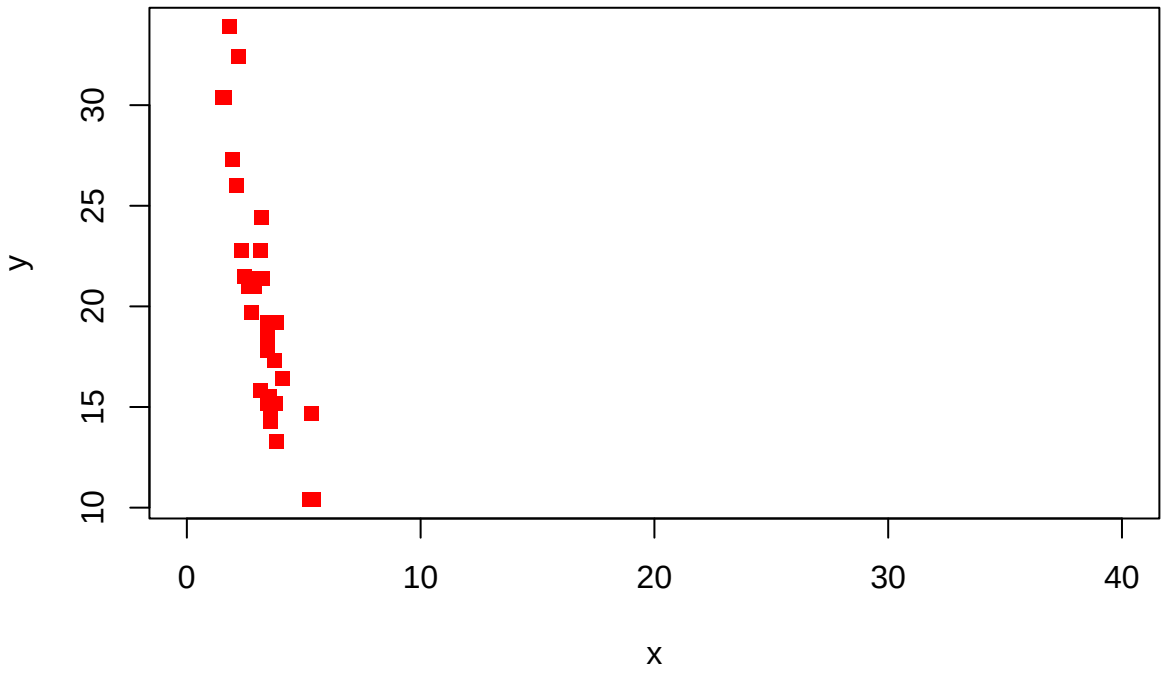
Suppose you write a small function to do some plotting. You want to hard code in the color red for all plots. That is you want to force the color red on anyone who uses your function (not nice but this is just an example)

```
my.plot <- function(x,y, ...) {
  plot(x,y, col="red",...)
}
```

Suppose you write a small function to do some plotting. You want to hard code in the color red for all plots. You want to force the color red on anyone who uses your function (not nice but this is just an example)

```
my.plot <- function(x,y, ...) {
  plot(x, y, col="red", ...)
}

my.plot(x=mtcars$wt, y=mtcars$mpg, pch=15, lty=2, xlim=c(0,40))
```

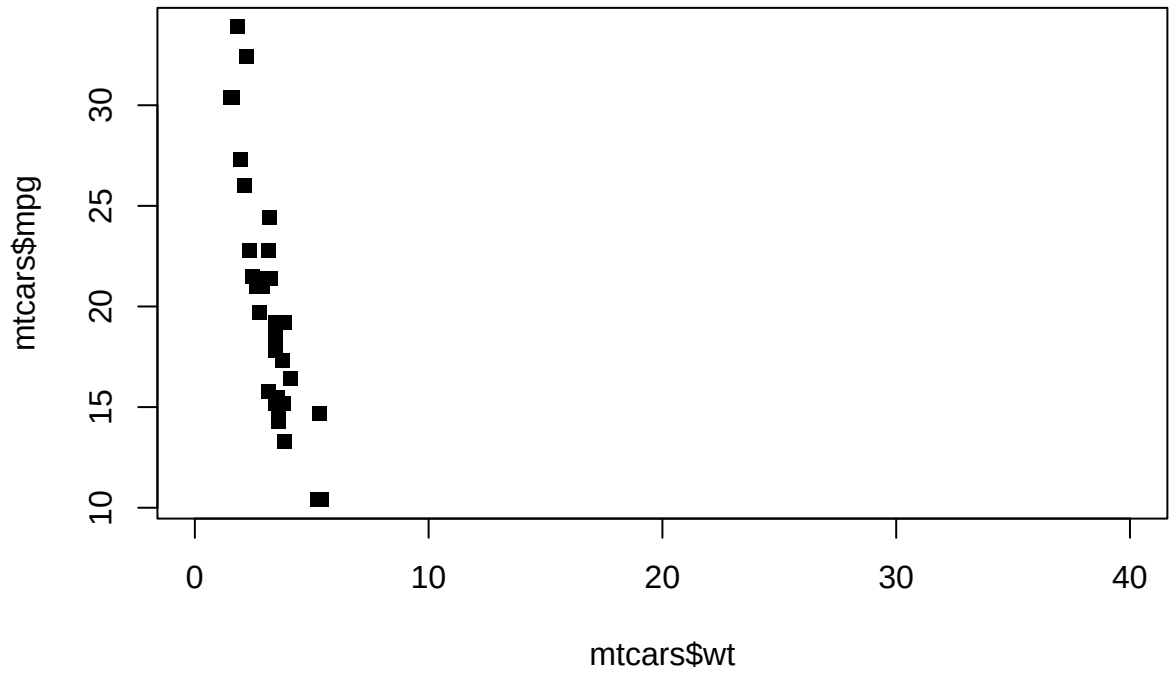


The function my.plot now accepts any argument that can be passed to the plot function (like col, xlab, etc.) without needing to specify those arguments in the header of my.plot.

Note that, technically, you could by-pass the x-y arguments altogether in this case but then why would you even bother to write your own function ?

```
my.plot <- function(...) {
  plot(...)
}

my.plot(x=mtcars$wt, y=mtcars$mpg, pch=15, lty=2, xlim=c(0,40))
```



The function my.plot now accepts any argument that can be passed to the plot function (like col, xlab, etc.) without needing to specify those arguments in the header of my.plot.

Debugging

Note that many of the examples presented in this section come from"An Introduction to the Interactive Debugging Tools in R"by Roger Peng. The associated website can be found at: http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Initial attempts at debugging usually involve some form of inserting print statements to view variables as they are recognized in the function

```
my.func <- function(x,y) {
  z <- x + y
  return(z)
}

my.func(1,"two")
```

```
my.func <- function(x,y) {
  cat("x is",x,"y is",y,"\n")
  z <- x + y
  return(z)
}

my.func(1,"two")
```

Oh so we forgot that the function can't handle arguments if they are characters. This was pretty obvious of course.

Most of the trouble comes from people providing bad input to your function. So if you do some initial error checking on the arguments then you will save yourself a lot of trouble.

```
my.func <- function(x,y) {
  if (!is.numeric(x) || !is.numeric(y) ) {
      stop("Check your input")
  }
  cat("x is",x,"y is",y,"\n")
  z <- x + y
  return(z)
}

my.func(1,"2")
```

Most of the trouble comes from people providing bad input to your function. So if you do some initial error checking on the arguments then you will save yourself a lot of trouble.

```
my.func <- function(x,y) {
   if (!is.numeric(x) || !is.numeric(y) ) {
       warning("Check your input")
   }
   cat("x is",x,"y is",y,"\n")
   z <- x + y
   return(z)
}

my.func(1,"2")
```

The warning function tells us there is a problem but the program moves on anyway even though it is doomed to fail.

```
mymedian <- function(xvec, debug=TRUE) {
   veclength <- length(xvec) # Get vector length
   sortedvec <- sort(xvec) # Get sorted vector

   if (debug) {
     cat("veclength is",veclength,"sortedvec is",sortedvec,"\n")
   }
   if (veclength %% 2 != 0) {
     index <- ceiling(veclength/2)
     retval <- sortedvec[index]
     if (debug) {
         cat("index is",index,"retval is",retval,"\n")
      }
    } else {
      # Other code
  }
}

mymedian(1:5)
```

```
## veclength is 5 sortedvec is 1 2 3 4 5
## index is 3 retval is 3
```

R provides several approaches for ”formal”or ”proper”debugging:

1) Use traceback() when your program fails to see relevant messages
2) When you want to interact with your R program on a step-by-step basis, you can use the debug() function. debug() accepts a single argument, the name of a function.
3) The trace() function allows you to temporarily add arbitrary code to a function; This allows inserting code in a function without permanently changing it.

There can be overlap between these methods and its not always clear which is the best since you might not know how deep you will need to go into your code. I prefer number 2.

```
foo <- function(x) {
  print(1)
  bar(2)
}

bar <- function(x) {
   x + some.nonexistent.variable
}

foo(1)
```

So next call the traceback() function to see ”the stack”of function calls that led to the error

```
traceback()
```

```
## No traceback available
```

So we see that the problem happened in function “Bar” called at line 3 from function foo(). We already know that the reason that “bar” failed was because of the absence of “some.nonexistent.variable”.

```
foo <- function(x) {
  print(1)
  bar(2)      # last call was to foo here at line #3
}

bar <- function(x) {
    x + some.nonexistent.variable
}
```

Let's look at a more involved example

```
f <- function(x) {
  r <- x - g(x)
  return(r)
}

g <- function(y) {
  r <- y * h(y)
  return(r)
}

h <- function(x) {
  if (x < 0) {
    stop("got a non positive number")
  }
  r <- log(x)
  if(r < 10) {
     return(r^2)
  } else {
     return(r^3)
  }
}

f(2)
```

```
## [1] 1.039094
```

```
f(-1)
```

```
traceback()
```

```
## No traceback available
```

A powerful tool is the debug function that allows us to step through a function as it is being executed.

```
SS <- function(mu, x) {
  d <- x - mu
  d2 <- d^2
  ss <- sum(d2)
  # Compute Sum of Squares
  return(ss)
}

set.seed(100)
x <- rnorm(100)
SS(1,x)
```

```
## [1] 202.5615
```

We can debug this function if we would like. It's easy to do and when you are developing functions, you should use this approach to help you.

We could also look again at our 3 functions from before. Let's debug f and, while we are at it, g and h.

Tracing

The trace function is very useful for making minor modifications to functions "on the fly".

It is especially useful if you need to track down an error which occurs in a base function. Since base functions cannot be edited by the user, trace may be the only option available for making modifications.

Note that trace has an extensive help page which should be read in its entirety. We will try to summarize the highlights here.

When using browser, you can only browse the environment in the current function call.

You cannot poke around in the environments for previous function calls.

There may be a situation where you want to suspend execution of a function in one location, but then browse a previous function call to hunt down the bug.

In other words, you may want to"jump up"to a higher level in the function call stack.

```
as.list(body(h))
```

```
## [[1]]
## `{`
##
## [[2]]
## if (x < 0) {
##     stop("got a non positive number")
## }
##
## [[3]]
## r <- log(x)
##
## [[4]]
## if (r < 10) {
##     return(r^2)
## } else {
##     return(r^3)
## }
```

```
trace("h", quote( if(is.nan(r)) { recover() } ),
      at = 3, print = F)
```

```
## [1] "h"
```

```
body(h)
```

```
## {
##     if (x < 0) {
##         stop("got a non positive number")
##     }
##     {
##         .doTrace(if (is.nan(r)) {
##             recover()
```

```
##         })
##         r <- log(x)
##     }
##     if (r < 10) {
##         return(r^2)
##     }
##     else {
##         return(r^3)
##     }
## }
```

The first argument to trace is the name of a function The second argument is the code you want to insert

This can either be the name of a function or it can be an un-evaluated expression

The "at" argument tells trace were to insert the new code. Here we've instructed trace to insert the code before the third statement.

Some Functions for Examination

The following function will simulate the roll of a single die until a value of 6 (the default) is observed after which it will return the number of times it took to get the 6. It will also return all values observed prior to rolling the 6. We can also supply a different value other than 6. This version works.

```
roller <- function(value=6) {

  if (!is.numeric(value) || (value <= 0 || value > 6)) {
    stop("illegal value")
  }

  collector <- vector()
  # simulate a single roll of a die
  sim_roll <- sample(1:6,1)

  # We now roll the die one time
  times_rolled <- 1

  collector[times_rolled] <- sim_roll

  # While the die is not showing the specified value
  # then keep rolling
  while (sim_roll != value) {
    sim_roll <- sample(1:6,1)
    times_rolled <- times_rolled + 1
    collector[times_rolled] <- sim_roll
  }
  # Once we get here, it means we saw the specified value
  return(list(observed_values=collector,
              times_rolled=times_rolled))
}
```

```
roller <- function(value=6) {

  if (!is.numeric(value) || (value <= 0 || value > 6)) {
```

```
    stop("illegal value")
  }

  collector <- vector()
  # simulate a single roll of a die
  sim_roll <- sample(1:6,1)

  # We now roll the die one time
  times_rolled <- 1

  collector[times_rolled] <- sim_roll

  # While the die is not showing the specified value
  # then keep rolling
  while (sim_roll != value) {
    sim_roll <- sample(1:6,1)
    times_rolled <- times_rolled + 1
    collector[times_rolled] <- sim_roll
  }
  # Once we get here, it means we saw the specified value
  return(list(observed_values=collector,
              times_rolled=times_rolled))
}
```