

A Teaching Script for Learning Functions

Genevieve Nuttall

LEARNING BASIC FUNCTIONS IN R

References:

link 1

link 2

link 3

link 4

Functions:

In R, functions are extremely important. Every command that you run is probably a set of functions that someone else created for you to use. Every package is jammed full of tons of functions that we are able to call with a simple command.

For example, when you type `mean(x)`, R will give you back a number. The only reason it can do this is because the `mean()` command has been programmed to execute a specific function. Take a look at what makes up the function `var`:

```
# Let's look at what makes up the function to find variance:
```

```
var
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##   if (missing(use))
##     use <- if (na.rm)
##       "na.or.complete"
##     else "everything"
##   na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##     "everything", "na.or.complete"))
##   if (is.na(na.method))
##     stop("invalid 'use' argument")
##   if (is.data.frame(x))
##     x <- as.matrix(x)
##   else stopifnot(is.atomic(x))
##   if (is.data.frame(y))
##     y <- as.matrix(y)
##   else stopifnot(is.atomic(y))
##   .Call(C_cov, x, y, na.method, FALSE)
## }
## <bytecode: 0x000000001445fd08>
## <environment: namespace:stats>
```

```
# If you run this, you will see the basic syntax of the function that runs when you run the var() command
# Pretty cool!
```

Basically, using and creating functions comes down to two questions:

1. What do we want the computer to do?
 - Tells us which function to use
2. What information does the computer need to compute this?
 - Tells us which arguments we should use

Here are some examples of built-in functions you may have used before in R:

```
##### Calculating Functions
```

```
sum(6:8)
```

```
## [1] 21
```

```
mean(1:10)
```

```
## [1] 5.5
```

```
log(16)
```

```
## [1] 2.772589
```

```
sqrt(36)
```

```
## [1] 6
```

```
##### Plotting functions
```

```
## Plot info from a sample data set in R: trees
```

```
head(trees)
```

```
##   Girth Height Volume
```

```
## 1   8.3     70   10.3
```

```
## 2   8.6     65   10.3
```

```
## 3   8.8     63   10.2
```

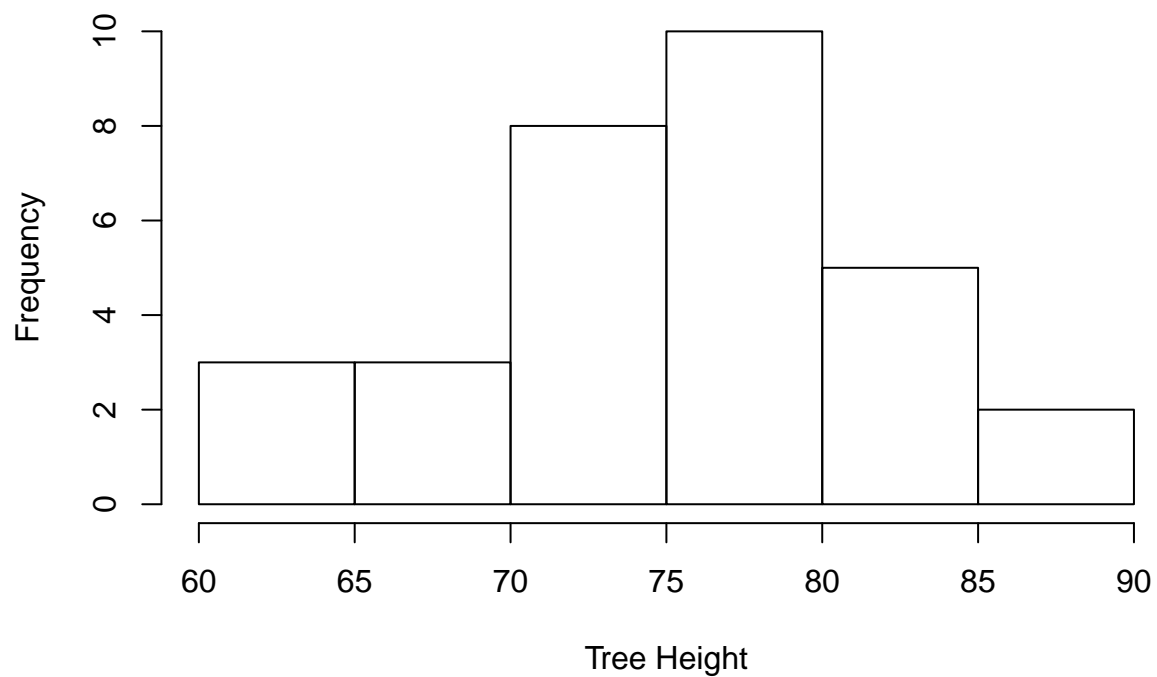
```
## 4  10.5     72   16.4
```

```
## 5  10.7     81   18.8
```

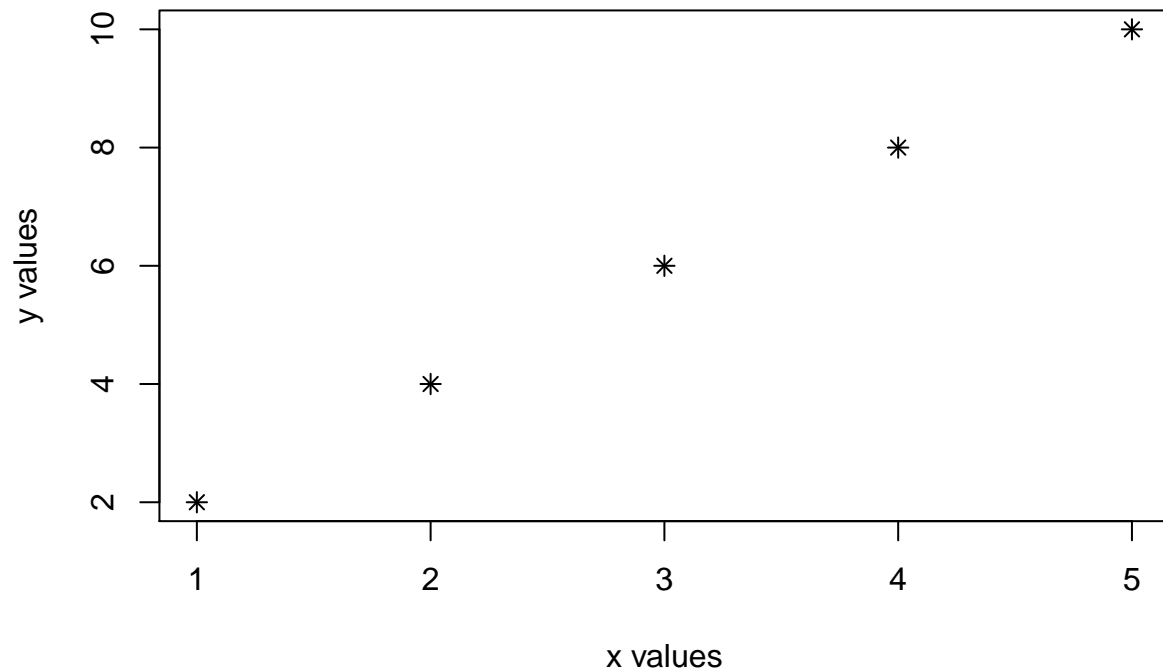
```
## 6  10.8     83   19.7
```

```
hist(trees$Height,xlab="Tree Height")
```

Histogram of trees\$Height



```
## Plot your own graph
plot(x=c(1,2,3,4,5),y=c(2,4,6,8,10),xlab="x values",ylab="y values",pch=8)
```



```
##### Statistical functions
```

```
## t-test
```

```
z<-rnorm(15)
```

```
a<-rnorm(15)
```

```
t.test(z,a)
```

```
##
```

```
## Welch Two Sample t-test
```

```
##
```

```
## data: z and a
```

```
## t = 0.29734, df = 27.86, p-value = 0.7684
```

```
## alternative hypothesis: true difference in means is not equal to 0
```

```
## 95 percent confidence interval:
```

```
## -0.5286056 0.7080751
```

```
## sample estimates:
```

```
## mean of x mean of y
```

```
## 0.03862040 -0.05111435
```

```
# Look at the results - can you find the p value? Is there a significant difference?
```

```
## smooth.spline (fits a cubic smooth line to noisy curves) - one of my favorites!
```

```
peak1.standard<-read.csv("Peak1.csv")
```

```
# An example of VERY noisy data - unfortunately from my own data
```

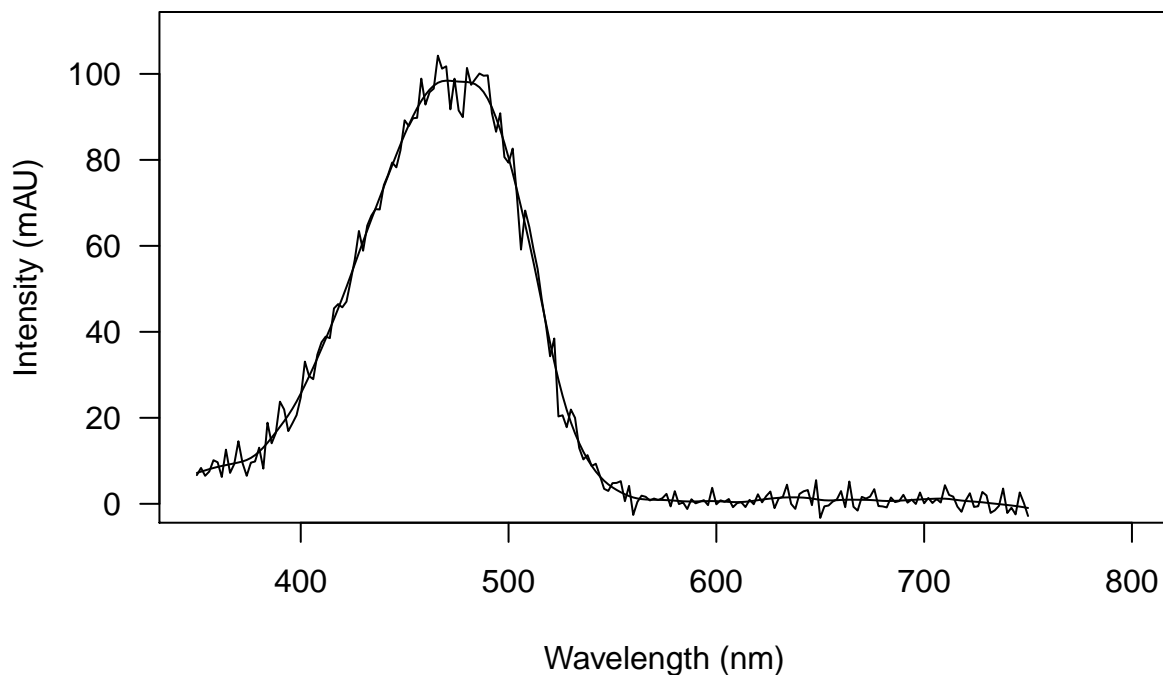
```
plot(peak1.standard$Wavelength,peak1.standard$mAU,xlim=c(350,800), type="l", pch=1,cex=0.6,las=TRUE,
```

```

      ylab="Intensity (mAU)",ylim=c(0,110),xlab="Wavelength (nm)")
# Let's use the smooth.spline function to smooth this line out.
smooth.peak1.standard<-smooth.spline(peak1.standard$Wavelength, peak1.standard$mAU)
# Now plot the smooth line over the noisy line to see how well it fits

## Highlight and run all three lines at same time
plot(peak1.standard$Wavelength,peak1.standard$mAU,xlim=c(350,800), type="l", pch=1,cex=0.6,las=TRUE,
      ylab="Intensity (mAU)",ylim=c(0,110),xlab="Wavelength (nm)")
lines(smooth.peak1.standard)

```



```

# Remember, these look like simple commands but they are actually functions
# that have been built into the software for us. Thanks, R!

```

```

# Look at what makes up the function smooth.spline
smooth.spline

```

```

## function (x, y = NULL, w = NULL, df, spar = NULL, cv = FALSE,
##     all.knots = FALSE, nknots = .nknots.smspl, keep.data = TRUE,
##     df.offset = 0, penalty = 1, control.spar = list(), tol = 1e-06 *
##     IQR(x))
## {
##     contr.sp <- list(low = -1.5, high = 1.5, tol = 1e-04, eps = 2e-08,
##         maxit = 500, trace = getOption("verbose"))
##     contr.sp[names(control.spar)] <- control.spar
##     if (!all(sapply(contr.sp[1:4], is.numeric)) || contr.sp$tol <
##         0 || contr.sp$eps <= 0 || contr.sp$maxit <= 0)

```

```

##      stop("invalid 'control.spar'")
## xy <- xy.coords(x, y)
## y <- xy$y
## x <- xy$x
## if (!all(is.finite(c(x, y))))
##   stop("missing or infinite values in inputs are not allowed")
## n <- length(x)
## if (is.na(n))
##   stop("invalid number of points")
## w <- if (is.null(w))
##   rep_len(1, n)
## else {
##   if (n != length(w))
##     stop("lengths of 'x' and 'w' must match")
##   if (any(w < 0))
##     stop("all weights should be non-negative")
##   if (all(w == 0))
##     stop("some weights should be positive")
##   (w * sum(w > 0))/sum(w)
## }
## if (!is.finite(tol) || tol <= 0)
##   stop("'tol' must be strictly positive and finite")
## xx <- round((x - mean(x))/tol)
## nd <- !duplicated(xx)
## ux <- sort(x[nd])
## uxx <- sort(xx[nd])
## nx <- length(ux)
## if (nx <= 3L)
##   stop("need at least four unique 'x' values")
## if (nx == n) {
##   ox <- TRUE
##   tmp <- cbind(w, w * y, w * y^2)[order(x), ]
## }
## else {
##   ox <- match(xx, uxx)
##   tapply1 <- function(X, INDEX, FUN = NULL, ..., simplify = TRUE) {
##     sapply(X = unname(split(X, INDEX)), FUN = FUN, ...,
##           simplify = simplify, USE.NAMES = FALSE)
##   }
##   tmp <- matrix(unlist(tapply1(seq_len(n), ox, function(i,
##     y, w) c(sum(w[i]), sum(w[i] * y[i]), sum(w[i] * y[i]^2)),
##     y = y, w = w), use.names = FALSE), ncol = 3, byrow = TRUE)
## }
## wbar <- tmp[, 1L]
## ybar <- tmp[, 2L]/ifelse(wbar > 0, wbar, 1)
## yssw <- sum(tmp[, 3L] - wbar * ybar^2)
## if (is.na(cv) && !missing(df))
##   stop("'cv' must not be NA when 'df' is specified")
## CV <- !is.na(cv) && cv
## if (CV && nx < n)
##   warning("cross-validation with non-unique 'x' values seems doubtful")
## r.ux <- ux[nx] - ux[1L]
## xbar <- (ux - ux[1L])/r.ux
## if (all.knots) {

```

```

##         if (!missing(nknots) && !is.null(nknots))
##             warning("'all.knots' is TRUE; 'nknots' specification is disregarded")
##         nknots <- nx
##     }
##     else if (is.null(nknots))
##         nknots <- .nknots.smspl(nx)
##     else {
##         if (is.function(nknots))
##             nknots <- nknots(nx)
##         else if (!is.numeric(nknots))
##             stop("'nknots' must be numeric (in {1,..,n})")
##         if (nknots < 1)
##             stop("'nknots' must be at least 1")
##         else if (nknots > nx)
##             stop("cannot use more inner knots than unique 'x' values")
##     }
##     knot <- c(rep(xbar[1], 3), if (all.knots) xbar else xbar[seq.int(1,
##         nx, length.out = nknots)], rep(xbar[nx], 3))
##     nk <- nknots + 2L
##     ispar <- if (is.null(spar) || missing(spar)) {
##         if (contr.sp$trace)
##             -1L
##         else 0L
##     }
##     else 1L
##     spar <- if (ispar == 1L)
##         as.double(spar)
##     else double(1)
##     if (length(spar) != 1)
##         stop("'spar' must be of length 1")
##     icrit <- if (is.na(cv))
##         0L
##     else if (cv)
##         2L
##     else 1L
##     dofoff <- df.offset
##     if (!missing(df)) {
##         if (df > 1 && df <= nx) {
##             icrit <- 3L
##             dofoff <- df
##         }
##         else warning("not using invalid df; must have 1 < df <= n := #{unique x} = ",
##             nx)
##     }
##     iparms <- c(icrit = icrit, ispar = ispar, iter = as.integer(contr.sp$maxit))
##     keep.stuff <- FALSE
##     ans.names <- c("coef", "ty", "lev", "spar", "parms", "crit",
##         "iparms", "ier", if (keep.stuff) "scratch")
##     fit <- .Fortran(C_rbart, as.double(penalty), as.double(dofoff),
##         x = as.double(xbar), y = as.double(ybar), w = as.double(wbar),
##         ssw = as.double(yssw), as.integer(nx), as.double(knot),
##         as.integer(nk), coef = double(nk), ty = double(nx), lev = double(if (is.na(cv)) 1L else nx),
##         crit = double(1), iparms = iparms, spar = spar, parms = unlist(contr.sp[1:4]),
##         scratch = double(17L * nk + 1L), ld4 = 4L, ldnk = 1L,

```

```

##      ier = integer(1L))[ans.names]
##      stopifnot(identical(wbar, tmp[, 1]))
##      if (is.na(cv))
##        lev <- df <- NA
##      else {
##        lev <- fit$lev
##        df <- sum(lev)
##        if (is.na(df))
##          stop("NA lev[]; probably smoothing parameter 'spar' way too large!")
##      }
##      if (fit$ier > 0L) {
##        sml <- fit$spar < 0.5
##        wtxt <- paste("smoothing parameter value too", if (sml)
##          "small"
##          else "large")
##        if (sml) {
##          stop(wtxt)
##        }
##        else {
##          fit$ty <- rep(mean(y), nx)
##          df <- 1
##          warning(wtxt, "\nsetting df = 1  __use with care!__")
##        }
##      }
##      cv.crit <- if (is.na(cv))
##        NA
##      else if (cv) {
##        ww <- wbar
##        ww[!(ww > 0)] <- 1
##        weighted.mean(((y - fit$ty[ox])/(1 - (lev[ox] * w)/ww[ox]))^2,
##          w)
##      }
##      else weighted.mean((y - fit$ty[ox])^2, w)/(1 - (df.offset +
##        penalty * df)/n)^2
##      pen.crit <- sum(wbar * (ybar - fit$ty)^2)
##      fit.object <- list(knot = knot, nk = nk, min = ux[1L], range = r.ux,
##        coef = fit$coef)
##      class(fit.object) <- "smooth.spline.fit"
##      object <- list(x = ux, y = fit$ty, w = wbar, yin = ybar,
##        data = if (keep.data) list(x = x, y = y, w = w), lev = lev,
##        cv.crit = cv.crit, pen.crit = pen.crit, crit = fit$crit,
##        df = df, spar = fit$spar, lambda = unname(fit$params["low"]),
##        iparms = fit$iparms, fit = fit.object, call = match.call())
##      class(object) <- "smooth.spline"
##      object
##    }
## <bytecode: 0x0000000016452b70>
## <environment: namespace:stats>

```

Luckily, R isn't too selfish, and it lets you make functions too so let's give it a shot!

Writing functions!

This will be useful to you at some point when you need to make a specific function for data analysis.

In R, there is a basic format for writing a function:

function_name <- **function**(argument1,argument2){body of function} *"function" is a keyword here - reserved for defining functions*

There are three components to this command: 1. formal arguments (in parentheses) 2. body {in squiggly brackets} 3. environment - where the function information is stored

When you run a function, R will always return the last statement of the command, which will be your {body} *Make sure you use the {} brackets when writing the body of a function*

Try creating a basic function for practice:

```
fun_function<-function(x,y=5){x*y}

fun_function

## function(x,y=5){x*y}
# Let's look at the components (formal arguments, body, and environment) of this function:
formals(fun_function)

## $x
##
##
## $y
## [1] 5

body(fun_function)

## {
##   x * y
## }

environment(fun_function)

## <environment: R_GlobalEnv>
# Remember, every function is made up of these three basic components.

# Let's run this function now
fun_function(3)

## [1] 15

# This is telling us that our "x" value is 3, and we know that y=5 and the body is {x*y}
# So, the output should be 15 because {x*y} = {3*5}. Yay!
```

Now, let's try something more complicated.

```
absolutely<-function(x)
{
  if(x<0){
    -x
  }
  else{
    x
  }
}

# This code tells us that if an x value is less than 1, then this negative value should
```

```

# be converted to positive
# AKA an absolute value function
# if evaluates a condition (x<0) - if a value fits this condition, then it will execute the body of the
# if the value doesn't fit the condition of x<0, then it moves to the else clause {x} so then R will ju

# Try it!
absolutely(-7)

## [1] 7

absolutely(28)

## [1] 28

```

When will you need to write your own functions? *For loops* are good examples!!!

For loops are functions used to repeat a line of code. They are helpful if you need to cycle through a set of steps - instead of coding each repeated step, you can use a for loop to do it for you.

Incorporate the same idea of arguments and body that we learned before.

```

for (food in c("mac and cheese","watermelon","pizza","blueberries","rice krispie treat")){
  print(paste("My favorite food is", food))}

## [1] "My favorite food is mac and cheese"
## [1] "My favorite food is watermelon"
## [1] "My favorite food is pizza"
## [1] "My favorite food is blueberries"
## [1] "My favorite food is rice krispie treat"

## Hopefully I covered one of your fav foods here, I'm really sorry if not.

## This is a silly example, but it shows you exactly what's going on with the loop.
## In the body, it's taking the text "My favorite food is" and linking it to the arguments specified as
## Each line represents a new loop, and the function will continue until it runs out of arguments.

```

You want to start your for loop with `for()`, have your arguments be the list of the things that the loop should cycle through, and have your body be what you want done with these things.

Let's move on to a more scientific example:

```

# Let's make a vector:

vector1<-c(3,5,6,9,8,11,15,16,22,23)

# Next, set your initial condition to 0. Do you know why we should do this?

condition<-0

## Okay, now we're going to write a loop that counts the total number of even values in our vector:

for(val in vector1){if (val %% 2==0) condition=condition+1}
print(condition)

## [1] 4

# In this loop, "val" corresponds to each values in our vector
# val %% 2 tells R to check the remainder of each value in the vector when divided by 2
# then, this remainder is compared to 0, and if it is 0, then R will evaluate it as TRUE

```

```
# by printing the condition, it tells you how many TRUE values were processed in the loop
```

You can also create nested for loops which are loops inside of loops if you need multiple a function to run through multiple layers of cycles. An example:

```
# Let's make a 10 x 10 matrix and fill it using a nested for loop
```

```
matrix1 <- matrix(nrow=5, ncol=6)
matrix1 ## you can see it's empty right now
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]  NA   NA   NA   NA   NA   NA
## [2,]  NA   NA   NA   NA   NA   NA
## [3,]  NA   NA   NA   NA   NA   NA
## [4,]  NA   NA   NA   NA   NA   NA
## [5,]  NA   NA   NA   NA   NA   NA
```

```
# This loop will assign values in each row/column based on the
# product of the i and j values that we specify here:
```

```
for(i in 1:dim(matrix1)[1]) {
  for(j in 1:dim(matrix1)[2]) {
    matrix1[i,j] = i*j}}
```

```
# Basically, this code tells R to put in a value in each
# cell that is the product of the corresponding product
# of row and column and this code is repeated until all cells are filled
```

```
# [1] is just an index that corresponds to rows, [2] to columns
```

```
# It's "nested" because you have a for() command within another for() command
```

```
# Look at the matrix now:
```

```
matrix1
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
## [2,]    2    4    6    8   10   12
## [3,]    3    6    9   12   15   18
## [4,]    4    8   12   16   20   24
## [5,]    5   10   15   20   25   30
```

For loops are one type of loop. In R, there are three main types of loops:

- for loops
- while loops
- repeat loops

Depending on your needs, you might want to use a different type of loop. To learn more about how to write different loops, visit [datacamp](#)

In some cases, it might be easier to use an `apply()` function which can act as a shortcut for a loop.

Remember, R is a world of functions. The ones we covered today are just a small percentage of what is out there on R.

To go over more commonly used functions, view [this list of functions](#)