Introduction to R Workshop Summary

Sara E. Burke

Created 2015-02-08 • Last updated 2020-05-18

This workshop is designed as an introduction to the use of R for basic statistical computing. My main goal is to familiarize you with basic programming tasks and the gist of how R treats common statistical tasks.

When you want to do something that is not covered in this workshop, try searching the internet. There are helpful examples and tutorials all over the web, and this workshop should prepare you to understand many of them.

Code snippets in this document are denoted using a `fixed width font`.

Installing R

Visit www.r-project.org, click "download R," and then choose any mirror site to download it from. Select your operating system and then follow the directions.

If you want a slightly more user-friendly interface, you might consider RStudio (www.rstudio.com). I don't use RStudio, but it has some nice features, like automatically saving your plots from a given session. In these workshops, I will be doing my examples in the regular R GUI, not RStudio.

Unit 1: Basic Programming

R is a programming language, which means it makes sense to start out with the basics of programming rather than with statistics. The idea is to show you that you are in control of what R is doing every step of the way. It is like learning how the steering wheel and pedals work in a car before you start exploring possible driving destinations.

In contrast to many other common programming languages like C or Python, R is specifically designed to make it easy to use for statistics.

Command Line

When you start R, the first thing you see will be the console, which functions like a command line interface. Try typing a command.

```
3 + 3
```

Setting Variables

To tell R to assign a value to a variable, type the variable name followed by an equals sign.

```
x = 3   # = sets the value of the variable x
x       # typing x will show you its value
x + 1
x * 2

x = 4   # to change the value of x, just assign it a new value
x
```

The assignment operator takes the value on the right and gives it the name on the left, never the other way around. This should result in an error message:

```
3 = x
```

Reinforcing this feature of variable assignment, the assignment operator can also be written using a less than sign and a hyphen.

```
x <- 5  # <- also sets the value
x
```

You can use any string of letters as a variable name.

```
turtles = 2 + 7 + 13
```

Note that names in R are case sensitive, so `Turtles` is a different variable from `turtles`.

Basic Arithmetic

```
x = (4 + 2 * 7) / (1 + 1)  # Works as you'd expect
x
x - 3^2
```

Don't make this mistake:

```
3x   # gives an error message
3*x  # works as expected
```

If you don't finish a line of code, the console will prompt you for the rest of it with a plus sign. Many programs feature "lines" of code that actually take up more than one line in the script file.

```
x /
2
```

Functions

```
sqrt(x)
```

Functions have names just like variables. To use them, type the name and then parentheses. The parentheses contain any input you want to give the function.

Don't name your variables the same thing as existing functions. If you aren't sure if a name is already in use, just type it by itself at the console to see what (if anything) it currently represents.

```
# Do I want to use "sqrt" as a variable name?
sqrt
# OK, "sqrt" is a function already. How about "squareroot"?
squareroot
# "squareroot" is an acceptable variable name.
squareroot = sqrt(x)
squareroot
```

String Values

Quotation marks are used to denote strings.

```
x = "asdf"  # Quotation marks denote string values
x
x = 'abcd'  # Use single or double quotation marks
x
```

Vectors

Vectors are useful in statistics because we typically want to deal with whole "columns" of data with many observations in them rather than just one number at a time.

The `c()` function stands for concatenate. It creates a vector—an ordered set of data points.

```
x = c(2,4,6)  # Concatenate a series of values into a vector
x  # x is now the vector of numbers: 2, 4, 6
```

What if we want to retrieve just one value from a vector?

```
x[1]  # Use square brackets to retrieve individual values
x[2]
```

What if we want to retrieve several values from a vector?

```
x = c(1,3,5,7,9,11,13,15,17,19,21)
x[1:5]  # Retrieve the first through fifth values from x
```

Note that the colon in that command generates its own vector, listing the numbers 1, 2, 3, 4, 5.

```
1:5  # The colon creates a vector of numbers in sequence
c(1,2,3,4,5) # ...just like if you typed out all of the numbers
```

So the command `x[1:5]` worked because `1:5` actually listed all the individual values we wanted—the first, the second, the third, the fourth, and the fifth.

Many arithmetic functions are applied to vectors element-by-element. Examples include: `+`, `*`, `/`, `-`, `sqrt()`, and `log()`.

```
x  # remember that x is the odd numbers from 1 to 21
x + 1 # returns a vector like x but with 1 added to each element
x * 2
log(x)
```

The function `c()` can also be used to join two vectors together.

```
x = c(1,2,3)
y = c(10,11,12)
z = c(x,y)
z
```

Other useful functions:

```
x = 10:30
length(x)  # tells you how many elements are in x
sum(x)     # adds together all the elements in x
mean(x)    # returns the mean of x
var(x)     # returns the sample variance of x
sd(x)      # returns the sample standard deviation of x
min(x)     # returns the smallest value in x
max(x)     # returns the largest value in x
median(x)  # returns the median of x
```

You can also have a vector of strings.

```
religion = c("Catholic","Jewish","Buddhist","Baha'i","Muslim")
```

Notice that the apostrophe in "Baha'i" is included in the string because I used double (not single) quotation marks for the string indicator.

A vector, by definition, contains elements that share a type. For example, you cannot have a vector with some numbers and some strings.

```
c(1, 2, "apple")
```

Notice that the whole vector gets automatically coerced to the string type. We will talk more about data types later.

Miscellaneous Information

As you've noticed by now, the comment character is `#`. Use it extensively!

```
mean(x) + sd(x) # compute one standard deviation above the mean
```

If something does not work as you expect, or you want to know more about how it works, look at the documentation. This can be accessed using the question mark (or the `help` function).

```
?rep  # What is rep?
help(rep)  # This line is the same as the above line.
rep(1,10)
c(rep(1,10),rep(0,10))
rep(c(1,0),each=10)
```

In this workshop, we're using a script file separate from the command line to automatically run many lines of code. Such files are typically saved using the **.r** extension.

Unit 2: Statistics

This unit will teach you about some basic statistics functions that are available in R. The default R libraries have a great many useful statistics functions, and there are also countless add-on packages available for free which account for virtually every statistical procedure that has been developed. All of the functions covered in this part of the workshop are available in the default R installation.

Initial Example: T-tests

I am using t-tests as an example to show you some general principles about how most statistics functions work in R.

```
### Set a few variables to use as examples
warmth = c(1,5,1,3,4,5,7,6,1,7)
targetgender = rep(c('m','f'),5)
```

The basic t-test function is `t.test()`. It works like any other function in R—you supply arguments in the parentheses and the function returns a value that you can then look at or use in some more complicated calculation.

```
?t.test  # Start by looking at the documentation.
```

Notice that the documentation shows all the default values of the inputs.

The first argument is a formula specifying what model you want to test. This formula is built like many representations of the "general linear model." In other words, first comes the DV or outcome variable, then the list of IVs or predictors. A t-test has only one predictor. The outcome is separated from the predictor using a tilde character.

```
t.test(warmth~targetgender)
```

Notice that the above line produced output. That output is the value that the function returned, which is actually a **list**. We will talk more about this later, but know that you can store this list in a new variable instead of just seeing the output at the console.

Also notice that the output specifies that a "Welch Two Sample t-test" was conducted. This is a type of t-test that does *not* assume that the two groups have equal variances, and it is the default for this function. You may be more accustomed to the traditional t-test that assumes equal variances:

```
t.test(warmth~targetgender,var.equal=T)
```

Note that `var.equal=F` is listed as the default value of `var.equal` on the help page for this function.

```
table(targetgender) # Also try the handy table function
```

Data Frames

Most of the time, in statistics, we have whole datasets, not just individual variables like warmth and target gender. A data frame in R is a set of vectors of the same length. It is like a worksheet in Excel or a data file in SPSS.

There are some examples of data frames built into R to use as examples. One of them is called "mtcars."

```
mtcars  # Example data frame built into R
```

Each column is a vector. As in any rectangular dataset, all of the columns are the same length.

You can reference an individual cell with [row,col].

```
mtcars[1,4] # Row 1, column 4
mtcars[1,] # All of row 1
mtcars[,4] # All of column 4
```

You can also reference a column by name using the dollar sign.

```
mtcars$hp # Reference column 4 by name
table(mtcars$cyl) # Reference "cyl" column by name
```

For convenience, view all the variable names with `names()`.

```
names(mtcars) # a vector of all the variables in the data frame
```

The `attach()` command is useful if you are used to programs like SPSS that always assume you are working with a specific dataset.

```
# "Attach" the data frame to refer to variables more easily
attach(mtcars)  # This attaches the mtcars data frame.
mpg             # We can now refer to the mpg variable by name
hp              # ... or any other variable in the data frame.
detach(mtcars)  # This command un-attaches mtcars...
mpg             # ... so now typing mpg alone does not work.
attach(mtcars)  # Let's attach it again.
```

Of course, you'll want to work with your own dataset rather than mtcars. You can read in a dataset using functions like `read.table` and `read.csv`.

First, you'll want to tell R where to look for your files. There are several ways to do this, but in this workshop, we are going to change the **working directory** using the "Change dir" option in the "File" menu. Navigate to the folder that our example datasets are stored in. It often makes

sense to keep all the files for a given project in one directory on your computer, and then use that as the working directory when you use R to work on that project.

You can also use `setwd()` to set the working directory, `getwd()` to see the current working directory, and `dir()` to see the contents of the current working directory (e.g., data files you might wish to read in). For example, `setwd("C:/asdf")` would set the working directory to the folder called "asdf" on the C drive in a Windows operating system. Note that backslashes (\) are special characters so you must either replace them with forward slashes (/) or double them up as in `setwd("C:\\asdf")`.

The `read.table()` function and its relatives (along with all R functions) returns an object, so you will want to have a variable name ready to accept that object. Like "mtcars," this will be the name of your dataset.

```
# By default, tabs and spaces are separators
data1 = read.table("tab_separated.txt",header=T)
read.table("tab_separated.txt") # Don't forget header=T

# Specify a different separator with sep=
data2 = read.table("comma_separated.csv", sep=",", header=T)

# For read.csv, it assumes you have a header by default
data3 = read.csv("comma_separated.csv")

# These methods all result in the same data frame:
data1
data2
data3
```

What if we want to `attach()` one of these new data frames? Use the `search()` function to see what is currently attached.

```
search() # mtcars is still attached.
detach(mtcars) # Detach it.
search() # It is not listed anymore.
attach(data1) # Attach our new example data.
search() # Now data1 is attached instead,
t.test(score~gender) # so we can refer to variables directly.
```

If you are working with multiple data frames simultaneously, you may want to avoid using `attach` and instead just refer to each data frame by name every time you wish to retrieve one or more variables.

The Basic Linear Model Function

The `lm()` function is the basic regression function in R, and it can be used for all sorts of things.

```
?lm() # This function has many, many uses.
```

Like the `t.test()` function, `lm()` outputs an **object** that has **attributes**. You can assign this object to a new variable name, like "model.1." Notice that I used a period in the variable name. This doesn't mean anything special – it is just another character that can be used in variable names.

```
model.1 = lm(score~age)

model.1 # Basic information about the regression
summary(model.1) # More detailed summary of the regression

names(model.1) # What components does this object have?
model.1$residuals # one of the components (vector of residuals)

# Often, I just want the summary, so I skip a few steps...
summary(lm(score~age))
```

Some Useful Plots

Scatterplot

```
plot(score~age)
plot(age,score) # Same thing

abline(model.1) # Best fit line generated by the linear model
```

Residual plots

```
plot(model.1$residuals) # Plot the residuals in order
plot(age,model.1$residuals) # Plot the residuals against age

qqnorm(model.1$residuals) # Q-Q plot of residuals
qqline(model.1$residuals) # Adds a line to the Q-Q plot
```

Histogram

```
hist(score) # Not very interesting with so few data points.
hist(mtcars$hp) # More interesting histogram
```

Categorical Predictors

```
summary(lm(score~age)) # Same syntax from above
summary(lm(score~gender)) # R automatically makes indicators
summary(lm(score~race))

# But what if race was coded numerically?
numericrace = c(1,2,3,3,1,2,3,3,1,2,3,1,1,3,2,2,3,3,2,1)
summary(lm(score~numericrace)) # oops!
factor(numericrace) # factor() tells R to treat it as a factor
factorrace = factor(numericrace)
summary(lm(score~factorrace))

summary(lm(score~factor(numericrace)))

# Managing how R represents factors
race # This variable is currently a factor
contrasts(race) # Default pair of indicator variables
detach(data1) # It is best not to alter an attached dataset
data1$race = relevel(data1$race,"White") # Set comparison group
attach(data1) # I will show alternatives to attach() later
contrasts(race) # New pair of indicator variables
summary(lm(score~race)) # Note new comparison group
```

Multiple Predictors

The key symbols to separate multiple predictors in the formula specification are `+`, `:`, and `*`.

```
#  + Separates additive predictors
summary(lm(score ~ age + gender)) # Two main effects
summary(lm(score ~ age + gender + race)) # Three main effects

#  : produces an interaction (multiplicative) term
summary(lm(score ~ age + gender + age:gender))
summary(lm(score ~ age * gender)) # Shortcut
summary(lm(score ~ age * race))
summary(lm(score ~ age * gender * race))
```

Data and Subset Arguments

Most statistical modeling functions, including `lm()`, have useful `data` and `subset` arguments.

```
# Model score~age*gender with female as reference group
summary(lm(score ~ age * gender))

# Determine which subset of cases have gender == "f"
s = (gender == "f")
s  # Note that the data type is logical (boolean)
```

```
# Use the subset argument to fit score~age for women only
summary(lm(score ~ age, subset = s))

# Try working without dataframe attached
detach()

# Specify which dataframe to use with the data argument
summary(lm(score ~ age * gender,data=data1))

# Use data and subset arguments together
summary(lm(score ~ age, data=data1, subset=s))

# It is sometimes convenient to do it this way:
summary(lm(score ~ age, data=data1, subset=(gender == "f") ))
```

Transformations

You can do transformations on the fly without having separate lines of code for the calculations.

```
summary(lm(score ~ log(age), data=data1))
```

Some transformations have to be enclosed in the `I()` function if they use symbols that are special characters in model specifications.

```
summary(lm(score ~ age + I(age^2), data=data1))
```

ANOVA

```
table(data1$race) # Useful to check the distribution
hist(data1$score) # Histogram function

model.2 = lm(score~race, data=data1) # Regular linear model
summary(model.2) # Regression summary
summary.aov(model.2) # ANOVA summary
anova(model.2) # Another way to do the same thing

model.3 = aov(score~race, data=data1) # Built-in ANOVA function
summary(model.3) # So it is yet another way to do the same thing

summary(aov(score~race*gender, data=data1)) # Two-way ANOVA
# Careful not to interpret the lower-order terms

# Unlike in SPSS, you can easily leave out the interaction term
# and still get ANOVA-style output:
summary(aov(score~race+gender, data=data1))
```

```
# This is useful for testing regression interactions with
# multiple terms:
summary(lm(score~race*age, data=data1))
summary.aov(lm(score~race*age, data=data1))
```

If you want a traditional ANOVA table for a full factoral design, none of the above functions will work for you. ANOVA can be thought of as a particular strategy for summarizing the results of one or more regression models. When your most complex model includes a two-way interaction, there is no single definitive way to summarize main effects. The same goes for two-way interactions in the presence of a three-way interaction, and so forth. You must decide how to describe these effects. The options are often described as "types" of sums of squares—type 1, type 2, type 3, etc.

Suppose you have a simple 2x2 design. Type 1 sums of squares would test the terms sequentially, so that the first main effect is tested on its own and the second main effect is tested adjusting for the first. I recommend never using this approach, as most psychologists will find it highly counterintuitive. Unfortunately, it is the default for the built-in `aov()` function, which is why I wrote "careful not to interpret the lower-order terms" in a comment above. Type 2 sums of squares would test the main effects adjusting for each other. In other words, this approach would be similar to fitting one regression model with only main effects and another model for the interaction term. This is my preferred approach, and also the default for the `Anova()` function in the `car` package. Type 3 sums of squares would test all of the terms in the same regression model, meaning that the lower-order terms would be simple effects instead of overall main effects. Some implementations automatically contrast code or center your variables to estimate these simple effects at useful points (the average of the two conditions, possibly weighted by sample size). The default in SPSS is type 3 sums of squares with contrast coding.

If you have questions about these options, feel free to reach out to me. If it all seems confusing, you can always just specify the exact regression models you want using `lm()` without bothering with `aov()` or `Anova()`.

Miscellaneous

See the "one-page intro to R" at the end of this document for some other frequently used statistics functions, including correlation and chi-squared tests.

You can save the R environment using File / Save Workspace. It produces an .RData file, which can be opened later and will retain your variables.

Missing Values

```
attach(data1)
age # Look at the age variable
mean(age)
age[4] = NA # Set a missing value
age
mean(age) # Many functions return NA if an input is NA
```

```
mean(age,na.rm=T) # This option removes missing values instead

is.na(age[4]) # Check if a value is missing
is.na(age)
sum(is.na(age))

# lm() automatically removes missing values:
summary(lm(score~age))

# NaN
sqrt(-1)
x = sqrt(-1)
is.na(x) # is.na returns true for this kind of missing value too
```

Unit 3: More programming

There are a number of other basic programming tasks that you should know about.

Relational Operators

```
3 < 4
4 < 3
3 == 4
4 == 4
3 <= 2
```

Note: don't confuse the double equals sign (==) with the assignment operator (= or <-).

Boolean Operators

```
TRUE & FALSE
TRUE & TRUE
TRUE | FALSE
FALSE | FALSE
! TRUE
! FALSE
3 < 4  &  4 < 3
3 < 4  |  4 < 3
```

Note that these operators work element-by-element with vectors.

```
x = c(1,2,3,4,5)
x > 3

y = c(5,4,3,2,1)
x >= y
x >= y & x < 5
```

Selecting Cases

```
which(y<3) # Returns the indices for which y is less than 3

which(race!='White') # Vector of indices for non-White Ps
nonwhite = which(race!='White') # Store that vector to use below

race[nonwhite] # A vector of the non-White participants' races
score[nonwhite] # A vector of the non-White Ps' scores

# alternatively...
race[race!='White']
score[race!='White']
```

Loops

If you've programmed in other languages before, you have probably encountered "for" loops, "while" loops, and "do-while" loops or, some variation thereof. R has these features, but I'll focus on "for" loops here because they're often the most convenient.

A fairly common reason to use a `for` loop is to perform some action for each case (participant) in the dataset. Before starting the loop, I'm just going to set a variable, `x`, to zero.

```
# To loop through all of the scores:
x = 0  # x starts at 0
```

Next let's initialize the loop.

```
for (item in score) {
```

The first keyword is "for." This means you are going to loop through all the items in a list of items, one at a time. After "for," type some parentheses. The first thing in these parentheses is an arbitrary, new variable name. I used "item" here. Be sure to pick a new name to avoid overwriting a variable you already have. R will temporarily fill in this variable with whichever list item it is currently working on as it loops through the list.

After the temporary variable name, put the keyword "in." Finally, specify the list or vector of values you want to loop through. Think of this syntax as saying "for each `item` in the list `score`, do the following steps."

Finally, type an open brace `{` to tell R that what follows is the list of steps to take.

```
x = x + item  # Add each score to x
```

For this example, we're just going to keep adding each item to x. In the end, we should have added up all the scores.

When you're done listing steps that the program should take for each item, signal that the loop is over by typing an ending brace.

```
}
x  # Now x is 117
sum(score)  # Is that the correct answer? Yes.
```

I picked a very simple example to illustrate the point, but you would almost never want to compute a sum using a `for` loop, because there is already a handy `sum()` function.

Here's another example:

```
length(score) # We have 20 scores
1:length(score) # Integers from 1 to 20
# These integers uniquely identify each observation. It is
# sometimes useful to loop through each observation's index.
x = rep(0,20) # Start out with a vector of 20 zeros
for (i in 1:length(score)) {
x[i] = score[i] + age[i]
}
x
score + age  # Did it work properly?
```

(Again, the last line of code shows that the loop is an overly complicated way to accomplish this particular goal.)

Conditional Statements

```
x = 3  # Set x to 3
if (FALSE) {x = 0}
x  # The input was false, so x has not changed
if (TRUE) {x = 1}
x  # The input was true, so R executed the line "x = 1"

x = NULL
for (i in 1:length(score)) {
if (race[i] == "White" & gender[i] == 'f') {
x = c(x,i)
}}
x
which(race=="White" & gender=="f")  # Did it work properly?
```

Computing Scale Scores

```
x = c(10,9,8,6,5,5,5,4,3,2)
y = c(10,9,9,7,7,5,4,4,2,1)
z = c(10,8,8,6,6,4,3,3,1,1)

cbind(x,y,z)  # Take multiple vectors & make them into a matrix
rowMeans( cbind(x,y,z) )  # Row means of the matrix

z[4] = NA

rowMeans( cbind(x,y,z) )  # Default behavior for missing values
rowMeans( cbind(x,y,z) , na.rm=T )
```

Packages

Many useful functions and variables are available in R by default, including the variable "pi," the square root function, and all the other functions we've learned about so far. But one of the real strengths of R is that it is free and open-source, so people can easily write new packages containing new functions. The Comprehensive R Archive Network hosts many such packages online; all of these packages have been tested to make sure they work properly. You can download these packages and then load them into any R session you're working on. When you load a package into your R session, all of its functions and variables become available to you.

For example, let's install the car package.

```
recode # Not a function by default

# Now install the car package
# How to do so may depend on platform. Try:
# install.packages("car")
# Once you have a package installed, you can load it anytime
library(car)

recode # Now R recognizes this function

race2 = recode(race,  "'Asian'=1;'Black'=2;'White'=3"  )
race2
```

The "foreign" package can be used to read data files created by other statistical software packages. To illustrate, I have included an example SPSS data file called SPSS_data.sav.

```
library(foreign)
spssdata = read.spss("SPSS_data.sav",to.data.frame=T)
# It is important to use to.data.frame=T both because
# it provides a more useful format and because it
# properly deals with missing values.
spssdata # It worked!
```

Other useful packages include:

```
library(psych)   # Many useful functions, including
                 # "principal" for PCA

library(ggplot2) # A widely-loved graphics package

library(nlme)    # These two packages contain functions for
library(lme4)    # regression with nested and random effects.
```

If you talk to other R users or read about R online, you will quickly notice that different fields—and different people within each field—have different assumptions about which packages are normally used. Some people will communicate as though packages like `ggplot2` are simply

defaults, used by everyone. I encourage you to learn about and use all kinds of packages, but for the sake of clarity you should specify which ones you're using when you communicate, rather than assuming that your favorite ones are universal.

Data Types

```
# Vectors contain items of the same type
x # All numeric
race # Factor
as.character(race) # All string
c(1,"a") # Note that the number 1 got changed to the string "1"

as.numeric("1") # Convert between data types
as.character(1)
as.numeric("a") # There is no numeric version of "a"
as.character(c(1,2,3)) # Works for vectors, too

list("a",2) # Lists can contain different data types
list("This is a list!",c(1,2,3,4)) # Lists can contain vectors

testlist = list("This is a list!",c(1,2,3,4))
testlist[[2]] # Double brackets to retrieve list items
testlist[[2]][3] # The 3rd vector item in the 2nd list item

# Many types of statistical output are actually lists
test = summary(lm(score~gender*race*age))
test[5,2] # Cannot retrieve the SE for the age term this way
names(test) # There is an element called "coefficients"
test$coefficients[5,2] # Refer to that element by name
test[[4]][5,2] # Or use double brackets

# Determine the class or type of an object
class(test)
class(c(1,2,3))
class(c("a","b","c"))
class(list("a","b","c"))
class(mtcars)
class(mtcars$mpg)

typeof(mtcars) # Data frames are a type of list
```

User-Defined Functions

Functions in R are objects just like any other objects (e.g., numeric vectors, data frames). You can define new functions yourself using `function()`.

```
# Simple example of a user-defined function
increment = function(x, by=1) { return(x + by) }
increment(1)  # Default "by" amount is 1
increment(1,by=4)  # Specify alternative amount to increment by
increment(1,4)  # If order matches, arguments don't need names
```

Note that you are always allowed to list the arguments out of order as long as you use their correct names. If you wish to omit names, then the order must match.

Here is another example of a user-defined function, this time with a real purpose. This is a short function I wrote to perform min-max normalization using either the empirical minimum and maximum or any other specified minimum and maximum.

```
minmax = function(x,oldmin=NULL,oldmax=NULL,newmin=0,newmax=1) {
if (length(oldmin)==0) {oldmin=min(x,na.rm=T)}
if (length(oldmax)==0) {oldmax=max(x,na.rm=T)}
unitminmax = (x - oldmin) / (oldmax - oldmin)
return(unitminmax * (newmax - newmin) + newmin) }

rating = c(3.5, 4.1, 3.9, 6.0, 5.4)
minmax(rating)  # Use defaults for oldmin/max and newmin/max
minmax(rating,oldmin=1,oldmax=7)  # Specify old min and max
minmax(rating,newmin=0,newmax=100)  # Specify new min and max
minmax(rating,oldmin=1,oldmax=7,newmin=0,newmax=100) # Both
```

Appendix: Some More Useful Functions

Here are more useful functions. To learn more about them, try plugging them into the `help()` function (or using `?`).

```
boxplot()  # Box plot
barplot()  # Bar plot
par()  # Set graphics parameters -- see ?par for help with
       # a wide variety of general plotting features.
seq()  # Returns a sequence of numbers in vector form
matrix()  # Create a matrix from raw values or vectors
%*%  # Since * works elementwise, you need %*% for
     # matrix multiplication.
solve()  # Compute inverse of a matrix
array()  # Create a multidimensional array
dim()  # Tells you the dimensions of a matrix or array
list()  # Create a list of many disparate elements
data.frame()  # Create a data frame from vectors
str()  # Details about the structure of a data frame
write.csv()  # Writes a data frame to a new csv file
levels()  # Lists all levels of a factor
```

One-Page Intro to R

Available at http://saraemilyburke.com/stats/rcheatsheet.pdf

To download R, visit www.r-project.org, click "download R," and then choose any mirror site. Follow the directions for your operating system.

Executing R scripts

- Type commands into the command line to see immediate output.
- Type out lists of commands in a script file. From the R program, execute selected commands from a script using ctrl+R or command+enter.

If you need help

- For help with a specific function or command, type `?` followed by the name, e.g., `?aov`
- Use a search engine (e.g., Google). There are many free online tutorials for specific tasks in R.
- Try looking through this introductory manual: http://cran.r-project.org/doc/manuals/R-intro.html

Important symbols

`* + - / ^ ( )`	Basic arithmetic symbols work intuitively.
`x = 9` or `x <- 9`	variable assignment (these examples set `x` to the numeric value `9`)
`x="aa"` or `x='aa'`	Quotation marks enclose character data rather than numeric data.
`sum(2,3,5)`	Parentheses contain function input; commas separate multiple input values.
`x[2]`	Square brackets allow you to specify items within a vector.
`1:10`	A colon generates a vector of integers (in this example, from `1` to `10`).
`T TRUE F FALSE`	`TRUE` and `FALSE` are the Boolean values. `T` and `F` stand for them.
`< > <= >= != ==`	These symbols all compare two values, returning `True` or `False`.
`x == 9`	returns `True` if x is 9. Don't mix this up with the assignment operator!
`T|F T&F !F`	`|` means "or", `&` means "and", and `!` means "not"
`NA`	missing value
`# comment text`	Text that follows `#` on a line will be ignored by R.

Functions of vectors

`x = c(1,2,3)`	concatenates a set of values to form a vector, then calls that vector `x`
`length(x)`	returns the number of items in vector `x`
`table(x)`	returns a table of the unique values in vector `x`
`sum(x) mean(x)`	`var(x) sd(x) min(x) max(x) median(x)` basic statistics of vectors
`x * 2`	returns a vector containing each item in vector `x` multiplied by `2`
`log(x)`	returns the natural log of each item in `x`
`x == 9`	returns a vector of Boolean values indicating whether each item in `x` is equal to `9`
`sum(x == 9)`	The sum of a Boolean vector treats `True` as 1 and `False` as 0.
`is.na(x)`	returns a vector of Boolean values indicating whether each item in `x` is missing
`x[y==9]`	If `x` and `y` are vectors of equal length, this syntax returns all values of `x` for which the corresponding value of `y` is equal to `9`.
`which(y==9)`	returns the vector of indices for which `y` is equal to `9`

Organizational functions

`ls()`	returns a vector of all variables currently assigned a value
`rm(x)`	removes or deletes the variable `x`
`library()`	attaches a package. Useful packages include `car` and `psych`
`search()`	identifies data frames and packages that are attached

Working with data frames

`read.table()`	creates a data frame from a text file. Remember to specify `header=T`
`read.csv()`	creates a data frame from a comma separated text file
`names(data)`	shows all the column/variable names in the data frame called `data`
`str(data)`	shows basic properties of each variable in the data frame called `data`
`attach(data)`	attaches the data frame called `data` so variables can be easily referenced

Formula syntax

`y~x1`	predict the variable `y` using the variable `x1`
`y~x1+x2`	predict the variable `y` using the main effects of `x1` and `x2`
`y~x1*x2`	predict the variable `y` using the main effects of `x1` and `x2` and their interaction
`factor()`	transforms a variable into a factor so `lm()` or `aov()` will treat it as such

Common statistical tests

Use `?` at the command line to learn how to specify the inputs for these functions.

`lm()`	linear model – regression or ANOVA
`summary()`	Input your `lm()` object and this function will give you a regression summary.
`anova()`	Compare two regression models.
`aov()`	linear model – ANOVA only
`leveneTest()`	Input your `lm()` or `aov()` object and this function will perform Levene's test. This function requires the `car` package. Don't forget the capital T in Test.
`TukeyHSD()`	Input your `aov()` object and this function will perform a Tukey test.
`t.test()`	t test – input can be a single variable, two variables, or a formula
`var.test()`	F test to compare two variances
`chisq.test()`	Pearson chi-squared test – input can be one or two variables
`cor.test()`	correlation significance test

Scale analysis

`cbind()`	Sticks a series of vectors next to each other as columns of a matrix.
`rowMeans()`	Computes row means of a matrix – use this for averaging variables into a scale. Specify `na.rm=T` to compute means ignoring missing values.
`cronbach()`	Cronbach's alpha of matrix columns – requires the `multilevel` package.
`principal()`	principal component analysis of matrix columns – requires the `psych` package. Use the `GPArotation` package for extra rotations.

Plotting

Checking `?par` and `?plot` is very helpful when plotting.

`par()`	set graphics parameters before plotting – see `?par` for details
`plot()`	scatter plot by default – input can be two variables or a formula
`abline()`	draw a line on the current plot – if the input is a `lm` object, it draws the best fit line
`legend()`	add a legend to the existing plot – see `?legend` for details
`hist()`	histogram
`qqnorm()`	normal Q-Q plot
`barplot()`	other basic plot types – see me for custom versions with features I commonly use
`boxplot()`	