A Sufficient Introduction to R

Derek L. Sonderegger

2016-08-23

Contents

Chapter 1

Introduction

R is a open-source program that is commonly used in Statistics. It runs on almost every platform and is completely free and is available at www.r-project.org. Most of the cutting-edge statistical research is first available on R.

R is a script based language, so there is no point and click interface. (Actually there are packages that attempt to provide a point and click interface, but they are still somewhat primitive.) While the initial learning curve will be steeper, understanding how to write scripts will be valuable because it leaves a clear description of what steps you performed in your data analysis. Typically you will want to write a script in a separate file and then run individual lines. The saves you from having to retype a bunch of commands and speeds up the debugging process.

This document is a very brief introduction to using R in my course. I highly recommend downloading and reading/skimming the manual "An Introduction to R" which is located at cran.r-project.org/doc/manuals/R-intro.pdf.

Finding help about a certain function is very easy. At the prompt, just type `help(function.name)` or `?function.name`. If you don't know the name of the function, your best bet is to go the the web page www.rseek.org which will search various R resources for your keyword(s). Another great resource is the coding question and answer site stackoverflow.

The basic editor that comes with R works fairly well, but you should consider running R through the program RStudio which is located at rstudio.com. This is a completely free Integrated Developement Environment that works on Macs, Windows and a couple of flavors of Linux. It simplifies a bunch of more annoying aspects of the standard R GUI and supports things like tab completion.

When you first open up R (or RStudio) the console window gives you some information about the version of R you are running and then it gives the prompt `>`. This prompt is waiting for you to input a command. The prompt + tells you that the current command is spanning multiple lines. In a script file you might have typed something like this:

```
for( i in 1:5 ){
    print(i)
}
```

But when you copy and paste it into the console in R you'll see something like this:

```
> for (i in 1:5){
+    print(i)
+ }
```

If you type your commands into a file, you won't type the `>` or `+` prompts. For the rest of the tutorial, I will show the code as you would type it into a script and I will show the output being shown with two hashtags (`##`) before it to designate that it is output.

1.1 R as a simple calculator

Assuming that you have started R on whatever platform you like, you can use R as a simple calculator. At the prompt, type 2+3 and hit enter. What you should see is the following

```
# Some simple addition
2+3
```

```
## [1] 5
```

In this fashion you can use R as a very capable calculator.

```
6*8
```

```
## [1] 48
```

```
4^3
```

```
## [1] 64
```

```
exp(1)   # exp() is the exponential function
```

```
## [1] 2.718282
```

R has most constants and common mathematical functions you could ever want. `sin()`, `cos()`, and other trigonometry functions are available, as are the exponential and log functions `exp()`, `log()`. The absolute value is given by `abs()`, and `round()` will round a value to the nearest integer.

```
pi     # the constant 3.14159265...
```

```
## [1] 3.141593
```

```
sin(0)
```

```
## [1] 0
```

```
log(5) # unless you specify the base, R will assume base e
```

```
## [1] 1.609438
```

```
log(5, base=10)  # base 10
```

```
## [1] 0.69897
```

Whenever I call a function, there will be some arguments that are mandatory, and some that are optional and the arguments are separated by a comma. In the above statements the function `log()` requires at least one argument, and that is the number(s) to take the log of. However, the base argument is optional. If you do not specify what base to use, R will use a default value. You can see that R will default to using base e by looking at the help page (by typing `help(log)` or `?log` at the command prompt).

Arguments can be specified via the order in which they are passed or by naming the arguments. So for the `log()` function which has arguments `log(x, base=exp(1))`. If I specify which arguments are which using the named values, then order doesn't matter.

```
# Demonstrating order does not matter if you specify
# which argument is which
log(x=5, base=10)
```

```
## [1] 0.69897
```

```
log(base=10, x=5)
```

```
## [1] 0.69897
```

But if we don't specify which argument is which, R will decide that `x` is the first argument, and `base` is the second.

```
# If not specified, R will assume the second value is the base...
log(5, 10)
```

```
## [1] 0.69897
```

```
log(10, 5)
```

```
## [1] 1.430677
```

When I specify the arguments, I have been using the `name=value` notation and a student might be tempted to use the `<-` notation hereSee next section.. Don't do that as the `name=value` notation is making an association mapping and not a permanent assignment.

1.2 Assignment

We need to be able to assign a value to a variable to be able to use it later. R does this by using an arrow `<-` or an equal sign `=`. While R supports either, for readability, I suggest people pick one assignment operator and stick with it. I personally prefer to use the arrow. Variable names cannot start with a number, may not include spaces, and are case sensitive.

```
tau <- 2*pi       # create two variables
my.test.var = 5   # notice they show up in 'Environment' tab in RStudio!
tau
```

```
## [1] 6.283185
```

```
my.test.var
```

```
## [1] 5
```

```
tau * my.test.var
```

```
## [1] 31.41593
```

As your analysis gets more complicated, you'll want to save the results to a variable so that you can access the results later[1]. *If you don't assign the result to a variable, you have no way of accessing the result.* [2]

1.3 Scripts and RMarkdown

One of the worst things about a pocket calculator is there is no good way to go several steps and easily see what you did or fix a mistake (there is nothing more annoying than re-typing something because of a typo. To avoid these issues I always work with script (or RMarkdown) files instead of typing directly into the console. You will quickly learn that it is impossible to write R code correctly the first time and you'll save yourself a huge amount of work by just embracing scripts (and RMarkdown) from the beginning. Furthermore, having a script file fully documents how you did your analysis, which can help when writing the methods section of a paper. Finally, having a script makes it easy to re-run an analysis after a change in the data (additional data values, transformed data, or removal of outliers).

It often makes your script more readable if you break a single command up into multiple lines. R will disregard all whitespace (including line breaks) so you can safely spread your command over as multiple lines. Finally, it is useful to leave comments in the script for things such as explaining a tricky step, who wrote the code and when, or why you chose a particular name for a variable. The `#` sign will denote that the rest of the line is a comment and R will ignore it.

[1] To paraphrase Beyonce, "Cause if you liked it, then you should have put a name on it."

[2] This isn't strictly true, the variable `.Last.value` always has the result of the last expression evaluated, but you can't go any farther back.

1.3.1 R Scripts (.R files)

The first type of file that we'll discuss is a traditional script file. To create a new .R script in RStudio go to `File -> New File -> R Script`. This opens a new window in RStudio where you can type commands and functions as a common text editor. Type whatever you like in the script window and then you can execute the code line by line (using the run button or its keyboard shortcut to run the highlighted region or whatever line the curser is on) or the entire script (using the source button). Other options for what piece of code to run are available under the Code dropdown box.

An R script for a homework assignment might look something like this:

```
# Problem 1
# Calculate the log of a couple of values and make a plot
# of the log function from 0 to 3
log(0)
log(1)
log(2)
x <- seq(.1,3, length=1000)
plot(x, log(x))

# Problem 2
# Calculate the exponential function of a couple of values
# and make a plot of the function from -2 to 2
exp(-2)
exp(0)
exp(2)
x <- seq(-2, 2, length=1000)
plot(x, exp(x))
```

This looks perfectly acceptable as a way of documenting what you did, but this script file doesn't contain the actual results of commands I ran, nor does it show you the plots. Also anytime I want to comment on some output, it needs to be offset with the commenting character `#`. It would be nice to have both the commands and the results merged into one document. This is what the R Markdown file does for us.

1.3.2 R Markdown (.Rmd files)

When I was a graduate student, I had to tediously copy and past tables of output from the R console and figures I had made into my Microsoft Word document. Far too often I would realize I had made a small mistake in part (b) of a problem and would have to go back, correct my mistake, and then redo all the laborious copying. I often wished that I could write both the code for my statistical analysis and the long discussion about the interpretation all in the same document so that I could just re-run the analysis with a click of a button and all the tables and figures would be updated by magic. Fortunately that magic[3] now exists.

To create a new R Markdown document, we use the `File -> New File -> R Markdown...` dropdown option and a menu will appear asking you for the document title, author, and preferred output type. In order to create a PDF, you'll need to have LaTeX installed, but the HTML output nearly always works and I've had good luck with the MS Word output as well.

The R Markdown is an implementation of the Markdown syntax that makes it extremely easy to write webpages and give instructions for how to do typesetting sorts of things. This syntax was extended to allow use to embed R commands directly into the document. Perhaps the easiest way to understand the syntax is to look at an at the RMarkdown website.

[3] Clark's third law states "Any sufficiently advanced technology is indistinguishable from magic."

The R code in my document is nicely separated from my regular text using the three backticks and an instruction that it is R code that needs to be evaluated. The output of this document looks good as a HTML, PDF, or MS Word document. I have actually created this entire document using RMarkdown.

1.4 Exercises

Create an RMarkdown file that solves the following exercises.

1. Calculate $\log(6.2)$ first using base e and second using base 10. To figure out how to do different bases, it might be helpful to look at the help page for the `log` function.
2. Calculate the square root of 2 and save the result as the variable named sqrt2. Have R display the decimal value of sqrt2. *Hint: use Google to find the square root function. Perhaps search on the keywords "R square root function".*

Chapter 2

Vectors

R operates on vectors where we think of a vector as a collection of objects, usually numbers. The first thing we need to be able to do is define an arbitrary collection using the `c()` function[1].

```
# Define the vector of numbers 1, ..., 4
c(1,2,3,4)
```

```
## [1] 1 2 3 4
```

There are many other ways to define vectors. The function `rep(x, times)` just repeats `x` a the number times specified by `times`.

```
rep(2, 5)              # repeat 2 five times... 2 2 2 2 2
```

```
## [1] 2 2 2 2 2
```

```
rep( c('A','B'), 3 )   # repeat A B three times  A B A B A B
```

```
## [1] "A" "B" "A" "B" "A" "B"
```

Finally, we can also define a sequence of numbers using the `seq(from, to, by, length.out)` function which expects the user to supply 3 out of 4 possible arguments. The possible arguments are `from`, `to`, `by`, and `length.out`. From is the starting point of the sequence, to is the ending point, by is the difference between any two successive elements, and `length.out` is the total number of elements in the vector.

```
seq(from=1, to=4, by=1)
```

```
## [1] 1 2 3 4
```

```
seq(1,4)        # 'by' has a default of 1
```

```
## [1] 1 2 3 4
```

```
1:4             # a shortcut for seq(1,4)
```

```
## [1] 1 2 3 4
```

```
seq(1,5, by=.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq(1,5, length.out=11)
```

```
##  [1] 1.0 1.4 1.8 2.2 2.6 3.0 3.4 3.8 4.2 4.6 5.0
```

If we have two vectors and we wish to combine them, we can again use the `c()` function.

[1] The "c" stands for collection.

```
vec1 <- c(1,2,3)
vec2 <- c(4,5,6)
vec3 <- c(vec1, vec2)
vec3
```

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

2.1 Accessing Vector Elements

Suppose I have defined a vector

```
foo <- c('A', 'B', 'C', 'D', 'F')
```

and I am interested in accessing whatever is in the first spot of the vector. Or perhaps the 3rd or 5th element. To do that we use the `[]` notation, where the square bracket represents a subscript.

```
foo[1]  # First element in vector foo
```

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

```
foo[4]  # Fourth element in vector foo
```

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

This subscripting notation can get more complicated. For example I might want the 2nd and 3rd element or the 3rd through 5th elements.

```
foo[c(2,3)]  # elements 2 and 3
```

```
## [1] "B" "C"
```

```
foo[ 3:5 ]   # elements 3 to 5
```

```
## [1] "C" "D" "F"
```

Finally, I might be interested in getting the entire vector except for a certain element. To do this, R allows us to use the square bracket notation with a negative index number.

```
foo[-1]          # everything but the first element
```

```
## [1] "B" "C" "D" "F"
```

```
foo[ -1*c(1,2) ] # everything but the first two elements
```

```
## [1] "C" "D" "F"
```

Now is a good time to address what is the `[1]` doing in our output? Because vectors are often very long and might span multiple lines, R is trying to help us by telling us the index number of the left most value. If we have a very long vector, the second line of values will start with the index of the first value on the second line.

```
# The letters vector is a vector of all 26 lower-case letters
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Here the `[1]` is telling me that `a` is the first element of the vector and the `[18]` is telling me that `r` is the 18th element of the vector.

2.2 Scalar Functions Applied to Vectors

It is very common to want to perform some operation on all the elements of a vector simultaneously. For example, I might want take the absolute value of every element. Functions that are inherently defined on single values will almost always apply the function to each element of the vector if given a vector.

```
x <- -5:5
x
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

```
abs(x)
```

```
##  [1] 5 4 3 2 1 0 1 2 3 4 5
```

```
exp(x)
```

```
##  [1] 6.737947e-03 1.831564e-02 4.978707e-02 1.353353e-01 3.678794e-01
##  [6] 1.000000e+00 2.718282e+00 7.389056e+00 2.008554e+01 5.459815e+01
## [11] 1.484132e+02
```

2.3 Vector Algebra

All algebra done with vectors will be done element-wise by default.For matrix and vector multiplication as usually defined by mathematicians, use `%*%` instead of `*`. So two vectors added together result in their individual elements being summed.

```
x <- 1:4
y <- 5:8
x + y
```

```
## [1]  6  8 10 12
```

```
x * y
```

```
## [1]  5 12 21 32
```

R does another trick when doing vector algebra. If the lengths of the two vectors don't match, R will recycle the elements of the shorter vector to come up with vector the same length as the longer. This is potentially confusing, but is most often used when adding a long vector to a vector of length 1.

```
x <- 1:4
x + 1
```

```
## [1] 2 3 4 5
```

2.4 Commonly Used Vector Functions

Function	Result
`min(x)`	Minimum value in vector x
`max(x)`	Maximum value in vector x
`length(x)`	Number of elements in vector x
`sum(x)`	Sum of all the elements in vector x
`mean(x)`	Mean of the elements in vector x
`median(x)`	Median of the elements in vector x
`var(x)`	Variance of the elements in vector x

Function	Result
`sd(x)`	Standard deviation of the elements in x

Putting this all together, we can easily perform tedious calculations with ease. To demonstrate how scalars, vectors, and functions of them work together, we will calculate the variance of 5 numbers. Recall that variance is defined as

$$Var\left(x\right) = \frac{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}{n-1}$$

```
x <- c(2,4,6,8,10)
xbar <- mean(x)          # calculate the mean
xbar
```

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

```
x - xbar                 # calculate the errors
```

```
## [1] -4 -2  0  2  4
```

```
(x-xbar)^2
```

```
## [1] 16  4  0  4 16
```

```
sum( (x-xbar)^2 )
```

```
## [1] 40
```

```
n <- length(x)           # how many data points do we have
n
```

```
## [1] 5
```

```
sum((x-xbar)^2)/(n-1)    # calculating the variance by hand
```

```
## [1] 10
```

```
var(x)                   # Same thing using the built-in variance function
```

```
## [1] 10
```

2.5 Exercises

1. Create a vector of three elements (2,4,6) and name that vector `vec_a`. Create a second vector, `vec_b`, that contains (8,10,12). Add these two vectors together and name the result `vec_c`.
2. Create a vector, named `vec_d`, that contains only two elements (14,20). Add this vector to `vec_a`. What is the result and what do you think R did (look up the recycling rule using Google)? What is the warning message that R gives you?
3. Next add 5 to the vector vec_a. What is the result and what did R do? Why doesn't in give you a warning message similar to what you saw in the previous problem?
4. Generate the vector of integers $\{1, 2, \ldots 5\}$ in two different ways.
   a) First using the `seq()` function
   b) Using the `a:b` shortcut.
5. Generate the vector of even numbers $\{2, 4, 6, \ldots, 20\}$
   a) Using the seq() function and

b) Using the a:b shortcut and some subsequent algebra. *Hint: Generate the vector 1-10 and then multiple it by 2.*

6. Generate a vector of 1001 elements that are evenly placed between 0 and 1 using the `seq()` command and name this vector `x`.
7. Generate the vector $\{2, 4, 8, 2, 4, 8, 2, 4, 8\}$ using the `rep()` command to replicate the vector c(2,4,8).
8. Generate the vector $\{2, 2, 2, 2, 4, 4, 4, 4, 8, 8, 8, 8\}$ using the `rep()` command. You might need to check the help file for rep() to see all of the options that rep() will accept. In particular, look at the optional argument `each=`.
9. The vector `letters` is a built-in vector to R and contains the lower case English alphabet.
   a) Extract the 9th element of the letters vector.
   b) Extract the sub-vector that contains the 9th, 11th, and 19th elements.
   c) Extract the sub-vector that contains everything except the last two elements.

Chapter 3

Data Types

There are some basic data types that are commonly used.

1. Integers - These are the integer numbers $(\ldots, -2, -1, 0, 1, 2, \ldots)$. To convert a numeric value to an integer you may use the function `as.integer()`.
2. Numeric - These could be any number (whole number or decimal). To convert another type to numeric you may use the function as.numeric().
3. Strings - These are a collection of characters (example: Storing a student's last name). To convert another type to a string, use as.character().
4. Factors - These are strings that can only values from a finite set. For example we might wish to store a variable that records home department of a student. Since the department can only come from a finite set of possibilities, I would use a factor. Factors are categorical variables, but R calls them factors instead of categorical variable. A vector of values of another type can always be converted to a factor using the as.factor() command.
5. Logicals - This is a special case of a factor that can only take on the values `TRUE` and `FALSE`. (Be careful to always capitalize `TRUE` and `FALSE`. Because R is case-sensitive, TRUE is not the same as true. Using the function `as.logical()` you can convert numeric values to `TRUE` and `FALSE` where `0` is `FALSE` and anything else is `TRUE`.

Depending on the command, R will coerce your data if necessary, but it is a good habit to do the coercion yourself. If a variable is a number, R will automatically assume that it is continuous numerical variable. If it is a character string, then R will assume it is a factor when doing any statistical analysis.

To find the type of an object, the str() command gives the type, and if the type is complicated, it describes the structure of the object.

3.1 Integers and Numerics

Integers and numerics are exactly what they sound like. Integers can take on whole number values, while numerics can take on any decimal value. The reason that there are two separate data types is that integers require less memory to store than numerics. For most users, the distinction can be ignored.

```
x <- c(1,2,1,2,1)
# show that x is of type 'numeric'
str(x)   # the str() command show the STRucture of the object
```

```
##  num [1:5] 1 2 1 2 1
```

3.2 Character Strings

In R, we can think of collections of letters and numbers as a single entity called a string. Other programming languages think of strings as vectors of letters, but R does not so you can't just pull off the first character using vector tricks. In practice, there are no limits as to how long string can be.

```
x <- "Goodnight Moon"

# Notice x is of type character (chr)
str(x)
```

```
##  chr "Goodnight Moon"
```

```
# R doesn't care if I use single quotes or double quotes, but don't mix them...
y <- 'Hop on Pop!'

# we can make a vector of character strings
Books <- c(x, y, 'Where the Wild Things Are')
Books
```

```
## [1] "Goodnight Moon"            "Hop on Pop!"
## [3] "Where the Wild Things Are"
```

Character strings can also contain numbers and if the character string is in the correct format for a number, we can convert it to a number.

```
x <- '5.2'
str(x)     # x really is a character string
```

```
##  chr "5.2"
```

```
x
```

```
## [1] "5.2"
```

```
as.numeric(x)
```

```
## [1] 5.2
```

If we try an operation that only makes sense on numeric types (like addition) then R complain unless we first convert it. There are places where R will try to coerce an object to another data type but it happens inconsistently and you should just do the conversion yourself

```
x+1
```

```
## Error in x + 1: non-numeric argument to binary operator
```

```
as.numeric(x) + 1
```

```
## [1] 6.2
```

3.3 Factors

Factors are how R keeps track of categorical variables. R does this in a two step pattern. First it figures out how many categories there are and remembers which category an observation belongs two and second, it keeps a vector character strings that correspond to the names of each of the categories.

```
# A charater vector
y <- c('B','B','A','A','C')
y
```

```
## [1] "B" "B" "A" "A" "C"
```

```
# convert the vector of characters into a vector of factors
z <- factor(y)
str(z)
```

```
##  Factor w/ 3 levels "A","B","C": 2 2 1 1 3
```

Notice that the vector `z` is actually the combination of group assignment vector `2,2,1,1,3` and the group names vector `"A","B","C"`. So we could convert z to a vector of numerics or to a vector of character strings.

```
as.numeric(z)
```

```
## [1] 2 2 1 1 3
```

```
as.character(z)
```

```
## [1] "B" "B" "A" "A" "C"
```

Often we need to know what possible groups there are, and this is done using the `levels()` command.

```
levels(z)
```

```
## [1] "A" "B" "C"
```

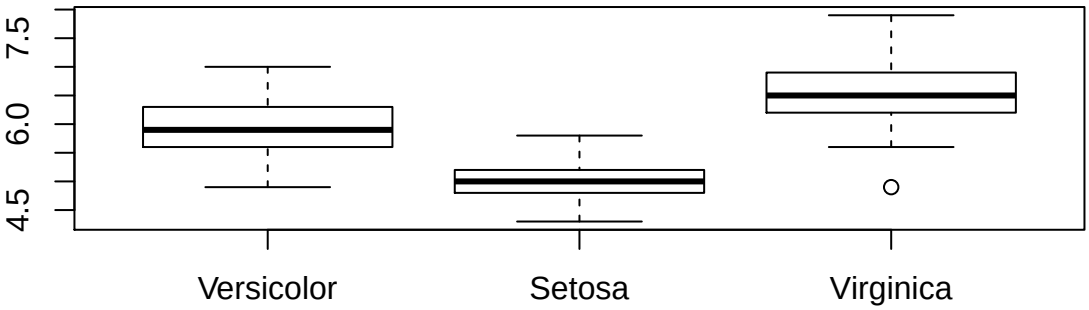
Notice that the order of the group names was done alphabetically, which we did not chose. This ordering of the levels has implications when we do an analysis or make a plot and R will always display information about the factor levels using this order. It would be nice to be able to change the order. Also it would be really nice to give more descriptive names to the groups rather than just the group code in my raw data. I find it is usually easiest to just convert the vector to a character vector, and then convert it back using the `levels=` argument to define the order of the groups, and labels to define the modified names.

```
z <- factor(z,                        # vector of data levels to convert
            levels=c('B','A','C'),    # Order of the levels
            labels=c("B Group", "A Group", "C Group")) # Pretty labels to use
z
```

```
## [1] B Group B Group A Group A Group C Group
## Levels: B Group A Group C Group
```

For the Iris data, the species are ordered alphatetically. We might want to re-order how they appear in a graphs to place Versicolor first. The `Species` names are not capitalized, and perhaps I would like them to begin with a capital letter.

```
iris$Species <- factor( iris$Species,
                        levels = c('versicolor','setosa','virginica'),
                        labels = c('Versicolor','Setosa','Virginica'))
boxplot( Sepal.Length ~ Species, data=iris)
```



Often we wish to take a continuous numerical vector and transform it into a factor. The function cut() takes a vector of numerical data and creates a factor based on your give cut-points.

```
# Define a continuous vector to convert to a factor
x <- 1:10

# divide range of x into three groups of equal length
cut(x, breaks=3)
```

```
##  [1] (0.991,4] (0.991,4] (0.991,4] (0.991,4] (4,7]     (4,7]     (4,7]
##  [8] (7,10]    (7,10]    (7,10]
## Levels: (0.991,4] (4,7] (7,10]
```

```
# divide x into four groups, where I specify all 5 break points
# Notice that the the outside breakpoints must include all the data points.
# That is, the smallest break must be smaller than all the data, and the largest
# must be larger (or equal) to all the data.
cut(x, breaks = c(0, 2.5, 5.0, 7.5, 10))
```

```
##  [1] (0,2.5]  (0,2.5]  (2.5,5]  (2.5,5]  (2.5,5]  (5,7.5]  (5,7.5]
##  [8] (7.5,10] (7.5,10] (7.5,10]
## Levels: (0,2.5] (2.5,5] (5,7.5] (7.5,10]
```

```
# divide x into 3 groups, but give them a nicer
# set of group names
cut(x, breaks=3, labels=c('Low','Medium','High'))
```

```
##  [1] Low    Low    Low    Low    Medium Medium Medium High   High   High
## Levels: Low Medium High
```

3.4 Logicals

Often I wish to know which elements of a vector are equal to some value, or are greater than something. R allows us to make those tests at the vector level.

Very often we need to make a comparison and test if something is equal to something else, or if one thing is bigger than another. To test these, we will use the `<`, `<=`, `==`, `>=`, `>`, and `!=` operators. These can be used similarly to

```
6 < 10    # 6 less than 10?
```

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

```
6 == 10   # 6 equal to 10?
```

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

```
6 != 10   # 6 not equal to 10?
```

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

where we used 6 and 10 just for clarity. The result of each of these is a logical value (a `TRUE` or `FALSE`). In most cases these would be variables you had previously created and were using.

Suppose I have a vector of numbers and I want to get all the values greater than 16. Using the $>$ comparison, I can create a vector of logical values that tells me if the specified value is greater than 16. The `which()` takes a vector of logicals and returns the indices that are true.

```
x <- -10:10     # a vector of 20 values, (11th element is the 0)
x
```

```
##  [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6
## [18]   7   8   9  10
```

```
x > 0           # a vector of 20 logicals
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
which( x > 0 )  # which vector elements are > 0
```

```
##  [1] 12 13 14 15 16 17 18 19 20 21
```

```
x[ which(x>0) ] # Grab the elements > 0
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

On function I find to be occasionally useful is the `is.element(el, set)' function which allows me to figure out which elements of a vector are one of a set of possibilities. For example, I might want to know which elements of the`letters‘ vector are vowels.

```
letters  # this is all 26 english lowercase letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
vowels <- c('a','e','i','o','u')
which( is.element(letters, vowels) )
```

```
## [1]  1  5  9 15 21
```

This shows me the vowels occur at the 1st, 5th, 9th, 15th, and 21st elements of the alphabet.

Often I want to make multiple comparisons. For example given a bunch of students and a vector of their GPAs and another vector of their major, maybe I want to find all undergraduate Forestry majors with a GPA greater than 3.0. Then, given my set of university students, I want ask two questions: Is their major Forestry, and is their GPA greater than 3.0. So I need to combine those two logical results into a single logical that is true if both questions are true.

The command & means “and” and | means “or”. We can combine two logical values using these two similarly:

```
TRUE  & TRUE     # both are true so combo so result is true
```

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

```
TRUE  & FALSE    # one true and one false so result is false
```

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

```
FALSE & FALSE    # both are false so the result is false
```

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

```
TRUE  | TRUE     # at least one is true -> TRUE
```

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

```
TRUE  | FALSE    # at least one is true -> TRUE
```

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

```
FALSE | FALSE    # neither is true -> FALSE
```

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

Chapter 4

Exercises

1. Create a vector of character strings with six elements

```
test <- c('red','red','blue','yellow','blue','green')
```

   and then

   a. Transform the `test` vector just you created into a factor.
   b. Use the `levels()` command to determine the levels (and order) of the factor you just created.
   c. Transform the factor you just created into integers. Comment on the relationship between the integers and the order of the levels you found in part (b).
   d. Use some sort of comparison to create a vector that identifies which factor elements are the red group.

2. Given the vector of ages,

```
ages <- c(17, 18, 16, 20, 22, 23)
```

   create a factor that has levels `Minor` or `Adult` where any observation greater than or equal to 18 qualifies as an adult. Also, make sure that the order of the levels is `Minor` first and `Adult` second.

3. Suppose we vectors that give a students name, their GPA, and their major. We want to come up with a list of forestry students with a GPA of greater than 3.0.

```
Name <- c('Adam','Benjamin','Caleb','Daniel','Ephriam', 'Frank','Gideon')
GPA <- c(3.2, 3.8, 2.6, 2.3, 3.4, 3.7, 4.0)
Major <- c('Math','Forestry','Biology','Forestry','Forestry','Math','Forestry')
```

   a) Create a vector of TRUE/FALSE values that indicate whether the students GPA is greater than 3.0.
   b) Create a vector of TRUE/FALSE values that indicate whether the students' major is forestry.
   c) Create a vector of TRUE/FALSE values that indicates if a student has a GPA greater than 3.0 and is a forestry major.
   d) Convert the vector of TRUE/FALSE values in part (c) to integer values using the `as.numeric()` function. Which numeric value corresponds to TRUE?
   e) Sum (using the `sum()` function) the vector you created to count the number of students with GPA > 3.0 and are a forestry major.

4. Make two variables, and call them `a` and `b` where `a=2` and `b=10`. I want to think of these as defining an interval.

   a. Define the vector `x <- c(-1, 5, 12)`
   b. Using the `&`, come up with a comparison that will test if the value of `x` is in the interval $[a, b]$. (We want the test to return `TRUE` if $a \leq x \leq b$). That is, test if `a` is less than `x` and if `x` is less

than `b`. Confirm that for x defined above you get the correct vector of logical values.

c. Similarly make a comparison that tests if `x` is outside the interval $[a, b]$ using the `|` operator. That is, test if `x < a` or `x > b`. I want the test to return TRUE is x is less than a or if x is greater than b. Confirm that for x defined above you get the correct vector of logical values.

Bibliography