Introduction to R

James Savage

12 January 2017

Part 1: The Basics

What is R?

R is a programming language focused on statistical computing and graphics. It was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, based on an earlier programming language called S. The R language is relatively new as programming languages go, with the first stable beta version released in 2000.

R is Open Source software that is freely available under the GNU General Public License, meaning that anyone can use or modify it for any purpose. Pre-compiled binary versions of R are available for various operating systems, and any code you write will work across all platforms.

Why use R?

Advantages

- Free (Both in the sense of beer and in the sense of speech)
- Extensive graphics capabilities (publication-quality plots)
- Command-line interface (once you get used to it!)
- Reproducibility through R scripts
- Huge range of add-on packages that expand what you can do
- Active community of developers
- Thorough (and built-in) documentation

Disadvantages

- Relatively easy to learn, but difficult to master
- Command-line interface (until you get used to it!)
- Little built-in support for dynamic or 3-D graphics (although this is improving
- Available functions are based on demand and user contributions. If no one feels like implementing your favorite analysis method, then you'll need to do it yourself
- Objects must generally be stored in physical memory
- Poorly written code is slow, and hard to read and maintain

About this Course

This R course is presented by Adam Kane and James Savage, postdoctoral fellows at the School of BEES, UCC. The course is aimed at students and staff with little prior experience of R, and covers the basics of the language (this session) and how to perform some common statistical tests and analyses (mext session). Code and further documentation relating to the course can be found on the course's github page: http://github.com/kanead/R-Course-UCC

The aim of this first part of the course is to introduce you to working with R, but more importantly to reduce the trepidation associated with using a command-line interface for the first time. Hopefully, by the end of

the course you will have enough confidence to play around with R and start applying it to your own needs. Please feel free to ask questions at any point if something is unclear or you encounter problems.

At the end of each major section, there are some short exercises that ask you to solve problems in R using the commands from the previous part(s). While you are doing these we will move around the class answering any questions you may have about the exercises or content of the course so far.

Downloading & Installing R and RStudio

To use R on your own computer, you will need to download R from the Comprehensive R Archive Network (CRAN: https://cran.r-project.org/). Choose the appropriate version of R for your operating system (e.g. Windows 7), then download and install the base distribution. The site will automatically give you the most recently released version of R. CRAN is also where you acquire official packages and extenstions to R.

R studio is an Integrated Development Environment (IDE) program for R, and we recommended it whenever using R for anything beyond the very basics. It makes coding easier by including a text editor that highlights R code syntax, and also provides tools for browsing data and the workspace, and giving better control of graphics and directories. RStudio is availale at https://www.rstudio.com/products/rstudio/download/ for a range of operating systems: choose the appropriate installer and follow its instructions to install.

Using R & RStudio

Once both R and Rstudio have been installed, open RStudio. You should be greeted with three panels: one on the left (Console) and two on the right (Environment/History above Files/Plots/Packages/Help/Viewer). The names above the panels are tabs: try clicking on a tab that isn't currently highlighted to change what that panel shows.

The panel on the left is the R console: at the bottom of this is the R prompt, which is where you type commands to R. You can follow the course examples by entering code into the R prompt yourself, and seeing what output R gives (generally printed in the console). Some of the outputs are not given within this document for reasons of space, so run the code yourself if you would like to see it. When entering multiple lines of code, enter them one at a time so you can easily follow what is going on.

In this document, code will be formatted like this:

```
2 + 2
```

```
## [1] 4
```

Enter the above into the R prompt, then press Enter to execute the code.

Expressions & Objects

At the R prompt we type expressions: commands that tell R to do something. One of the most basic things we will want R to do is to give some data a name, and then store it in memory. This is done using the <- symbol, which is known as the assignment operator. These stored, named chunks of data are called objects. For example, we can define an object called x to have the value 5.

```
x <- 5
```

Once assigned, typing just the name of an object will automatically print its value.

```
x
```

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

The [1] before the value indicates that x is a vector (more on this later) and that its first element is 5.

Note that R allows you to overwrite any assignments you have made, and will not give you any warnings or ask for confirmation.

```
x <- 12
x
```

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

Luckily, if you ever want to evaluate an expression again, you can press the up arrow key while in the R prompt to select a recent expression, then press Enter to redo the evaluation.

Objects will be stored in memory until they are removed by you, or until you quit the R session. In R studio, you can see what objects are currently stored using the Environment tab in the top right panel.

Calculations & Comparisons

In addition to creating objects, simple operations such as adding, multiplying, and squaring numbers can be performed easily in R using arithmetic operators. R will evaluate the expression, and auto-print the result.

```
6 + 13
```

```
## [1] 19
```

```
4 - 8
```

```
## [1] -4
```

```
345 * 2356
```

```
## [1] 812820
```

```
567 / 13
```

```
## [1] 43.61538
```

```
2^4
```

```
## [1] 16
```

```
(2+8)^2-(54/23)^3
```

```
## [1] 87.05811
```

You can also use operators on variables you have named, and store the results as objects directly. R does not auto-print the result of the expression if it has been stored as an object.

```
height <- 7
width <- 4
area <- height * width
area
```

```
## [1] 28
```

In addition to arithmetic operators, there are also relational operators that compare objects. These will return either TRUE or FALSE. * < less than * > greater than * <= less than or equal to * >= greater than or equal to * == equality * != not equals

```
height > width
```

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

```
height == 7
```

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

Functions

Most of the work we do with R involves applying Functions to objects or values. Functions are named commands that end with open and closed brackets (). For example, the square-root function sqrt().

```
sqrt(area)
```

```
## [1] 5.291503
```

The objects or values that the function acts on are called "arguments", and go inside the function's brackets. The sqrt() function only has one argument, but some functions have several. Arguments are separated by commas, and have names. For example the function seq() creates a sequence, and has arguments "from" and "to" (among others). The order in which you give the arguments is important, but this is overridden if you name them explicity.

```
seq( 1, 5 )
```

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

```
seq( 5, 1 )
```

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

```
seq( from = 1, to = 5 )
```

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

```
seq( to = 5, from = 1 )
```

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

Many functions have one or two arguments that are necessary, and lots more arguments that act as options to control exactly how the function runs. If these optional arguments are not specified, the function will use default values. For example, seq() includes the argument "by" which controls the increment of the sequence. It has the default value of one, but this can be changed to any other number.

```
seq( from = 2, to = 20)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
seq( from = 2, to = 20, by = 2.3 )
```

```
## [1]  2.0  4.3  6.6  8.9 11.2 13.5 15.8 18.1
```

Getting help

At this point you're probably wondering how you find out what arguments exist for each function. Thankfully, one great thing about R is the amount of built-in help that is available. If you need to know how a function works or what arguments it has, simply type "?"" and then the function name. You can also type help(function), or help("function") if the function name is a special character. In RStudio, these commands will open the relevant help page in the bottom right panel.

```
?seq
help(seq)
help("^")
```

R also provides examples of how to use many common functions. Try example(functionName) if you are still unsure after reading the help.

```
example(seq)
```

Errors & Warning messages

Errors and Warning messages occur when problems occur as R is evaluating code. They are printed in red text and describe what went wrong, often in a way that is difficult to interpret. In the case of Errors, the evaluation of the code stops at the point the error occured. The most common error is probably when you mistakenly ask R to do something with an object that doesn't exist.

```
mass <- volume * density
```

```
## Error in eval(expr, envir, enclos): object 'volume' not found
```

Note that the error only stated that the object "volume" was not found, even though the object "density" does not exist either. The evaluation stopped at the point R hit the first error.

Warning messages, unlike error messages, do NOT stop evaluation of the code. Be careful when using any result obtained after a Warning, especially if you don't understand why the message was generated.

```
f <- sqrt(-1)
```

```
## Warning in sqrt(-1): NaNs produced
```

```
f
```

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

In this case R determined that the square-root of -1 was NaN ("not a number"), but still stored this result as the object f. It is possible to handle complex numbers in R, but we won't be covering them within this course.

Keeping things tidy

It is easy for the working Environment to get cluttered with lots of objects, many of which may no longer be necessary. As well as seeing what objects are currently stored using the Environment panel, you can get a list of current objects using ls(). To remove objects, use the function rm().

```
ls()
```

```
## [1] "area"   "f"      "height" "width"  "x"
```

```
rm(height)
ls()
```

```
## [1] "area"  "f"     "width" "x"
```

Aside from remove unnecessary objects, the best way to keep on top of things is to give objects names that are easy to understand. Some simple rules to follow are: * Make your object names informative (when possible) * Use capital letters, dots, or underscores as spaces in object names (e.g. sampleArea, sample.area, sample_area) * Don't use duplicate (or very similar) names for different objects * Don't use names that are similar to existing functions you might use * Don't use leading digits or symbols in names * Use a consistent naming strategy

Writing code

Rather than evaluating every expression in the R prompt immediately, it is usually better to write your commands in a text file first, to make it easier to remember what you did and reproduce it in future. RStudio helps by providing a built-in text editor. You can open a new text document by clicking on the white square with the green plus in the top-left corner of the screen, or by pressing Ctrl+Shift+N.

Try typing a few R commands in the new script file (one command per line is usually a good idea). Any code in the script can be run directly by selecting it and then pressing Ctrl+Enter. Selecting multiple lines will lead to them all being evaluated.

Within a sctipt, you can add comments using "#". Any text that appears after # on a line will not be evaluated. You should always add as many comments as possible! The more comments there are, the easier it will be to understand the code in future.

```
height <- 5
triArea <- 0.5*width*height # area of a triangle
# You can also use comments on their own line to add sections to your scripts
```

Use the save icon or Ctrl+S to save your script. Script files are saved with the extension .R, but outside of R and RStudio can be opened and altered with notepad or any other text editor just like a normal text file. We recommend you use this script file when working through the exercises below.

Packages

Each function in R belongs to a certain package, and each package is a collection of functions, data, help, and examples with a common theme. The base R system that you install consists of the "base" package (containing the most fundamental functions), along with a few other packages such as "stats" and "utils" that contain commonly needed statistics and utility functions resepctively.

Beyond the packages that come installed with R, you decide which packages (and hence functions) you want available in your R system. There are plenty to choose from: as of January 2017 there are over 9700 packages on CRAN that have been developed by users and programmers around the world. CRAN packages come with Manuals that describe the functions then contain, and often Vignettes that supply worked examples.

To install a new package, use install.packages() to download the package from CRAN, and then library() to load it into your Environment.

```
install.packages("nlme")
library(nlme)
```

You can browse all available packages on CRAN by name here: https://cran.r-project.org/web/packages/available_packages_by_name.html

Exercise 1: The basics

a) Create an object called "sausages" with a value of 126, and an object called "beans" will a value of 2345.
b) Calculate the value of sausages x beans and save the result as a new object (any name you like).
c) Test whether sausages are greater than beans.
d) Remove "sausages" and "beans" from the working Environment.
e) Open the CRAN list of packages, and use the descriptions to find a package on a topic related to your own work. Install and load this package into R using the functions above. Back on CRAN, take a look at the documentation available for the package to find out what functions it contains. In R, open the help for one of the functions in this package using "?" or help().

Part 2: Data Classes

To work effectively with R, it is important to understand how it sees and handles different types of data. These come in two types: "atomic" data classes, which relate to the difference between single elements (e.g. a number vs a string of characters), and other data classes that relate to how the elements are arranged (e.g. a 1D vector vs a 2D matrix). This distinction is important because the type and structure of your data are largely independent, but both alter how you work with it.

Atomic data types

R has five basic or atomic classes of objects: *numeric (numbers)* integer (whole numbers) *character (string)* logical (True/False) *complex (not appearing in this course)

You can find out what type of data any object is using the class() function. For simple objects this will return one of the above types; more complex objects will return their structural type.

```
a <- 134
class(a)
```

```
## [1] "numeric"
```

```
b <- "banana"
class(b)
```

```
## [1] "character"
```

```
c <- TRUE
class(c)
```

```
## [1] "logical"
```

You can also ask R whether an object is of a particular class directly, using a family of functions with the format is.className():

```
is.numeric(a)
```

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

```
is.character(b)
```

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

```
is.logical(c)
```

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

Numeric & Integer data

Numeric data includes all real numbers, to any degree of size or precision, including when given in scientific notation. This includes whole numbers: to be classed as integers, numbers need to be specified as integers explicitly.

```
class(5.34)
```

```
## [1] "numeric"
```

```
class(10e3)
```

```
## [1] "numeric"
```

```
class(5)
```

```
## [1] "numeric"
```

The most important thing to remember about working with numeric data is that because of rounding errors (and the way computers store numbers), you need to be very careful when using relational operators (e.g. ==, <).

```
d <- 3.0-2.9
d           ## seems to have the value of 0.1
```

```
## [1] 0.1
```

```
d == 0.1    ## but is not exactly equal to 0.1 according to R
```

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

```
e <- d*10
e
```

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

```
e == 1      ## the problem carries over to anything created from d, even if it appears to be an integer
```

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

Integers are created by adding "L" after a number, and are useful if you expect to be working exclusively with whole numbers. They never generate rounding errors and can always be safely compared to whole numbers.

```
is.integer(e)
```

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

```
f <- 10L
is.integer(f)
```

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

```
f == 10
```

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

Also included in the numeric class are the special number "Inf" which represents infinity, and "NaN" ("not a number") which represents an undefined value.

Character data

Character or string vectors store text. Character data must always be entered within quotation marks, or R will think you are referencing an object.

```
b <- banana    # generates an error
```

```
## Error in eval(expr, envir, enclos): object 'banana' not found
```

```
b <- "banana"
```

The function paste() is useful for combining text objects.

```
g <- "smoothie"
paste(b,g)
```

```
## [1] "banana smoothie"
```

You can also extract parts of a character object by using the substring function substr():

```
h <- "On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in t
substr(h, start=8, stop=24)
```

```
## [1] "Origin of Species"
```

Similarly, you can find and replace strings inside a character object using the sub() and gsub() functions:

```
sub("Species", "Bananas", h)
```

```
## [1] "On the Origin of Bananas by Means of Natural Selection, or the Preservation of Favoured Races in
```

Logical data

Logical or binary vectors store TRUE/FALSE data. R considers TRUE and FALSE as equivalent to 1 and 0 respectively when performing calculations.

```
i <- 5 > 4
i
```

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

```
i*5
```

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

```
j <- 5 < 4
j
```

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

```
j*5
```

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

You can combine these with relational operators such as "&" (and), "|" (or), and "!" (not/negation) to perform further logical tests on objects you have created.

```
i & j          # Are i AND j true?
i | j          # Are i OR j true?
!i == j        # is NOT i equal to j?
```

Exercise 2: Data types

a) What type of data is the object x, created above? What about the object f (sqrt(-1))?
b) Create an integer object with the value of 34. Divide it by 5, and save the result as a new object. What type of data does the new object have?
c) Create a character object with the value "yourFirstName YourLastName". Use substr() to save your first and last names as two new object.
d) Combine your names together again using paste().
e) What is the value of TRUE/FALSE?

Vectors

Vectors are a sequence of elements with the same basic type, e.g. numeric, character, logical. Vectors are created using c(). For vectors, class() returns the atomic class of each element. Technically all the objects we have created so far have already been vectors: R treats single elements as vectors with length 1.

```
numObs <- c(4, 7, 7, 12, 6) # e.g. number of things observed over 5 sample periods
numObs
```

```
## [1]  4  7  7 12  6
```

```
is.vector(numObs)
```

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

```
class(numObs)
```

```
## [1] "numeric"
```

```
months <- c("Jan","Feb","Mar","Apr","May")
class(months)
```

```
## [1] "character"
```

```
observed <- c(TRUE,FALSE,TRUE,TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,TRUE) # e.g. presense / absence
class(observed)
```

```
## [1] "logical"
```

Vectors can also be combined using c().

```
newObs <- c(3,9,5,3)
c(numObs, newObs)
```

```
## [1]  4  7  7 12  6  3  9  5  3
```

When generating numeric vectors, you can use the ":" operator to create simple sequences.

```
x <- 1:20
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

There are numerous other ways to generate vectors. Two examples are our old friend seq(), which creates a vector sequence given arguments "from"","to"", and "by", and rep(), which creates a repeating vector based on an original vector.

```
sequence <- seq(from = 1,to = 10, by = 2)
sequence
```

```
## [1] 1 3 5 7 9
```

```
rep(sequence, times = 5)
```

```
##  [1] 1 3 5 7 9 1 3 5 7 9 1 3 5 7 9 1 3 5 7 9 1 3 5 7 9
```

Naming vectors

The elements that make up a vector can also be named. This can be done either using the names() function, or by assigning a name to each element as you create the vector.

```
names(numObs) <- months
numObs2 <- c(Jan = 4, Feb = 7, Mar = 7, Apr = 12, May = 6)
numObs3 <- c("Jan" = 4, "Feb" = 7, "Mar" = 7, "Apr" = 12, "May" = 6)
numObs; numObs2; numObs3
```

```
## Jan Feb Mar Apr May
##   4   7   7  12   6
```

```
## Jan Feb Mar Apr May
##   4   7   7  12   6
```

```
## Jan Feb Mar Apr May
##   4   7   7  12   6
```

Note that all three methods give the same result, and that R prints named vectors slightly differently, as a table.

One common attribute you want to know about a vector is its length: how many data elements it contains. You can get this using length(). For a more detailed description of a vector (or any other object) you can use str(), which prints the structure of the object.

```
length(months)
```

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

```
str(numObs)
```

```
##  Named num [1:5] 4 7 7 12 6
##  - attr(*, "names")= chr [1:5] "Jan" "Feb" "Mar" "Apr" ...
```

Vector arithmetic

R applies operations to every element of a vector, so adding, multiplying or otherwise manipulating a vector object affects all the data within it.

```
abundance <- c(46,73,21,40,66)
abundance * 7
```

```
## [1] 322 511 147 280 462
```

```
abundance / 10
```

```
## [1] 4.6 7.3 2.1 4.0 6.6
```

```
abundance + 15
```

```
## [1] 61 88 36 55 81
```

```
abundance - 20
```

```
## [1] 26 53  1 20 46
```

```
abundance^2
```

```
## [1] 2116 5329  441 1600 4356
```

You can also perform artithmetic on two vectors. In this case, R will match up the elements of each vector when performing the calculation.

```
recruitment <- c(50, 100, 30)
mortality <- c(30, 40, 80)
recruitment - mortality
```

```
## [1]  20  60 -50
```

```
recruitment + c(10, 20, 30)
```

```
## [1]  60 120  60
```

```
recruitment * c(1, 2, 3)
```

```
## [1]  50 200  90
```

```
recruitment / c(1, 2, 3)
```

```
## [1] 50 50 10
```

Relational operators work in a similar way, outputting a logical vector of the same length as the input vectors.

```
recruitment > mortality
```

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

Many functions exist to explore vectors. Some of the most commonly used functions include range(), max(), min(), sum(), mean() etc. which each do exactly what you would expect.

```
range(recruitment)
```

```
## [1]  30 100
```

```
max(recruitment)
```

```
## [1] 100
```

```
min(mortality)
```

```
## [1] 30
```

```
change <- recruitment - mortality
sum(change)
```

```
## [1] 30
```

Subsetting

Retrieving values from inside a vector is performed using single square brackets "[]". The simplest method is use the index, for example [3] will give you the third element of the vector. You can subset a single element, or multiple elements using [c()].

```
numObs[2]
```

```
## Feb
##   7
```

```
numObs[c(2,3,4)]
```

```
## Feb Mar Apr
##   7   7  12
```

The same syntax is used to drop elements from a vector. Putting a minus sign before an index drops the specified element(s):

```
numObs[-3]
```

```
## Jan Feb Apr May
##   4   7  12   6
```

```
numObs[-c(1,5)]
```

```
## Feb Mar Apr
##   7   7  12
```

If you try to select an index beyond the length of the vector, R will return NA (missing data).

```
numObs[9]
```

```
## <NA>
##   NA
```

A second method is to subset using names, if the vector is named:

```
numObs["Feb"]
```

```
## Feb
##   7
```

```
numObs[c("Feb","Mar","Apr")]
```

```
## Feb Mar Apr
##   7   7  12
```

A third approach is to create a logical vector that tells R which elements you want to keep (TRUE) and which to discard (FALSE). If this logical vector is shorter than the vector you are selecting from, it will be recycled (e.g. FALSE,TRUE will select every other element).

```
numObs[c(FALSE,TRUE,TRUE,TRUE,FALSE)]
```

```
## Feb Mar Apr
##   7   7  12
```

```
numObs[c(TRUE,FALSE)]
```

```
## Jan Mar May
##   4   7   6
```

A useful trick is creating a logical vector by testing which elements of the vector match a certain condition, and then subsetting using that.

```
numObs > 6
```

```
##   Jan   Feb   Mar   Apr   May
## FALSE  TRUE  TRUE  TRUE FALSE
```

```
numObs[numObs < 7]
```

```
## Jan May
##   4   6
```

Missing values

Missing data in R is represented by NA. You can test whether values are NA using the is.na() function (and for NaN using is.nan()).

```
k <- c(1, 2, NA, 3, NA, 4)
is.na(k)
```

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

```
is.nan(sqrt(-1))
```

```
## Warning in sqrt(-1): NaNs produced
```

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

NAs are treated differently from 0s in a dataset: NAs will usually cause calculations to fail, unlike 0s, but many functions can also be told to strip them beforehand.

```
l <- c(1,2,0,3,0,4)      # same as k but with 0s insteas of NAs
mean(k)                  # calculation fails due to NAs (but no error)
```

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

```
mean(k, na.rm = TRUE)   # the na.rm argument strips NAs before calculation
```

```
## [1] 2.5
```

```
mean(1)
```

```
## [1] 1.666667
```

As a general rule, try to remove NAs when at all possible. The easiest way to do this is by subsetting using the NOT operator:

```
k[!is.na(k)]
```

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

Exercise 3: Vectors

a) Create two numeric vector of length 18 by any means you like. Multiply them together, and sum the resulting vector.
b) Use a logical vector to select every third element of one of your vectors.
c) Create a character vector with at least three unique elements. What do you think sum(), max(), min(), and range() will return on this vector? Try it out.

Matrices

A matrix is a collection of data elements arranged in a rectangular layout. In R, matrices work almost exactly like vectors, apart from being 2D, so much of what you have already learned applies. All matrix elements must be the same basic data type, just like vectors, and you perform calculations on them and subset them in a similar way.

Matrixes are created using matrix(), which takes a vector as input, along with the arguments "nrow" and "ncol" to specify the number of rows and columns.

```
matrix(1:6, nrow = 2)
```

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

```
matrix(c("a","b","c","d","e","f"), nrow = 2, byrow = TRUE) # fills the matrix by row (L->R) rather than
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "b"  "c"
## [2,] "d"  "e"  "f"
```

```
matrix(1:6, nrow = 6, ncol = 2) # the vector is not long enough to fill the matrix, so it gets recycled
```

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

```
m <- matrix(1:12, byrow = TRUE, nrow = 4)
```

Applying class() to a matrix will return "matrix", not (e.g.) "numeric" or "character" as a vector would. To get what type of matrix it is, use mode(). You can also get a description of the structure using str() as before.

```
class(m)
```

```
## [1] "matrix"
```

```
mode(m)
```

```
## [1] "numeric"
```

```
str(m)
```

```
##  int [1:4, 1:3] 1 4 7 10 2 5 8 11 3 6 ...
```

Subsetting a matrix works the same way as a vector, except that you need two indices to specify the row and column position of the data you want.

```
m[1,3]
```

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

```
m[3,2]
```

```
## [1] 8
```

If you only specify the row that you want (e.g. [2,]), R will assume you want all the columns. Likewise, if you only specify a column (e.g. [,4]) R will give you all the rows.

```
m[3,]
```

```
## [1] 7 8 9
```

```
m[,3]
```

```
## [1]  3  6  9 12
```

As with vectors, you can also subset multiple rows/columns at once using c().

```
m[2, c(2, 3)]
```

```
## [1] 5 6
```

```
m[c(1, 2), c(2, 3)]
```

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

Individual elements are not named in a matrix, but row and column names can be applied and then used to subset matrices in the same way as vectors.

```
rownames(m) <- c("r1", "r2", "r3", "r4")
colnames(m) <- c("a", "b", "c")
m["r2","c"]
```

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

```
m[3, c("b", "c")]
```

```
## b c
## 8 9
```

R also contains useful functions for summing and calculating averages of the rows and columns of matrices.

```
colSums(m)
colMeans(m)
rowSums(m)
rowMeans(m)
```

Finally, arithmetic operations are applied to every element of a matrix independently, just as with vectors.

```
m * 5
```

```
##     a  b  c
## r1  5 10 15
## r2 20 25 30
## r3 35 40 45
## r4 50 55 60
```

```
m + 10
```

```
##     a  b  c
## r1 11 12 13
## r2 14 15 16
## r3 17 18 19
## r4 20 21 22
```

```
m + c(100,1)
```

```
##      a   b   c
## r1 101 102 103
## r2   5   6   7
## r3 107 108 109
## r4  11  12  13
```

Two important functions for manipulating data in R are rbind() and cbind(). These are used to add rows and columns (respectively) to an existing matrix.

```
rbind(m, 13:15)
```

```
##     a  b  c
## r1  1  2  3
## r2  4  5  6
## r3  7  8  9
## r4 10 11 12
##    13 14 15
```

```
cbind(m, c(15, 15, 15, 15))
```

```
##     a  b  c
## r1  1  2  3 15
## r2  4  5  6 15
## r3  7  8  9 15
## r4 10 11 12 15
```

Exercise 4: Matrices

a) Generate a vector of the sequence 1,2,3, ... 16 using a function or operator.
b) Create a matrix with 16 elements arragned into four rows and columns using the above vector. Assign the result the name m2.
c) Subset the bottom-right quarter of your new matrix and assign it the name m3.
d) Use a function to find the largest element in m3.

e) What do you think m2[,] will return as output? Guess before trying it out!

Factors

Catgorical variables are a common type of data we want to manipulate, but they can be tricky to work with because they can only take a certain number of distinct values. An example would be the blood types of a group of people:

```
blood <- c("B", "AB", "O", "A", "O", "O", "A", "B", "A")
class(blood)
```

```
## [1] "character"
```

To handle these data properly, R needs to know that these are not characters but factors: a set of categories with a limited number of different values or "levels". You define a factor using factor().

```
blood_factor <- factor(blood)
blood_factor        # note that this also prints the levels
```

```
## [1] B  AB O  A  O  O  A  B  A
## Levels: A AB B O
```

```
class(blood_factor)
```

```
## [1] "factor"
```

```
str(blood_factor)
```

```
##  Factor w/ 4 levels "A","AB","B","O": 3 2 4 1 4 4 1 3 1
```

The function levels() is used to print and/or rename the levels of a factor.

```
levels(blood_factor)
```

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

```
levels(blood_factor) <- c("BT_A", "BT_AB", "BT_B", "BT_O")
blood_factor
```

```
## [1] BT_B  BT_AB BT_O  BT_A  BT_O  BT_O  BT_A  BT_B  BT_A
## Levels: BT_A BT_AB BT_B BT_O
```

Factors can be unordered (like blood types), and this is the default when creating them from character vectors. For unordered factors, comparisons between elements will return NA (and a warning).

```
blood_factor[1] < blood_factor[2]
```

```
## Warning in Ops.factor(blood_factor[1], blood_factor[2]): '<' not meaningful
## for factors
```

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

You can create ordered factors using the optional "ordered" argument within factor(). For example, as vector of shirt sizes:

```
sizes <- c("M", "L", "XS", "L", "M", "XL", "M","XL")
sizes_factor <- factor(sizes, ordered = TRUE, levels = c("XS","S", "M", "L","XL"))
sizes_factor
```

```
## [1] M  L  XS L  M  XL M  XL
## Levels: XS < S < M < L < XL
```

Note that the levels of sizes_factor are printed with "<" in between to indicate the ordering. When using ordered factors, comparisons work in the same way as with numerics.

```
sizes_factor[1] < sizes_factor[2]
```

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

Did you notice that sizes_factor has a level ("S"") that is not present in the data? When creating factors you can include categories that are expected but have not yet been observed.

Coercion

We have seen how you can use class() and the is.dataType() family of functions to identify and test what type of data a vector contains. For example, a simple numeric vector:

```
nums <- c(1,3,5,7,3,7,7,1)
class(nums)
```

```
## [1] "numeric"
```

```
is.numeric(nums)
```

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

But sometimes numbers may actually represent names or categories rather than numeric data, and the vector needs to reflect that before analysis can begin. A family of functions with the format as.dataType() are used to convert or "coerce" objects from one type to another.

```
nums_char <- as.character(nums)
nums_fact <- as.factor(nums)
nums_int <- as.integer(nums)
nums_char; nums_fact; nums_int
```

```
## [1] "1" "3" "5" "7" "3" "7" "7" "1"
```

```
## [1] 1 3 5 7 3 7 7 1
## Levels: 1 3 5 7
```

```
## [1] 1 3 5 7 3 7 7 1
```

```
class(nums_char); class(nums_fact); class(nums_int)
```

```
## [1] "character"
```

```
## [1] "factor"
```

```
## [1] "integer"
```

If you try to coerce objects to a type that doesn't make sense, you will generally get a warning message, and NAs will be generated.

```
as.numeric("apple")
```

```
## Warning: NAs introduced by coercion
```

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

You should also beware of automatic coercion by R. This most commonly occurs when different types of elements are combined into a single vector. Vectors can only have a single atomic class, so (e.g.) when numeric, character, and logical vectors are combined the result is a character vector (whatever the order of combination).

```
c("hedgehog", 0.8, TRUE)
```

```
## [1] "hedgehog" "0.8"      "TRUE"
```

Lists

Lists are data structures that can hold any number of other objects of any type (even other lists), avoiding any problems with coercion. Lists are created using list(), and you can check their class using is.list().

```
animal <- list("hedgehog", 0.8, TRUE)
animal
```

```
## [[1]]
## [1] "hedgehog"
##
## [[2]]
## [1] 0.8
##
## [[3]]
## [1] TRUE
```

```
is.list(animal)
```

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

Note the difference from when we used c() on the same objects above. The objects retain their original classes, but are printed as three separate elements.

Lists can be named using names(), or during creation.

```
names(animal) <- c("name", "mass", "mammal")
animal
```

```
## $name
## [1] "hedgehog"
##
## $mass
## [1] 0.8
##
## $mammal
## [1] TRUE
```

```
animal <- list(name = "hedgehog", mass = 0.8, mammal = TRUE)
```

Lists can include other lists. At this point, str() becomes very useful for inspecting their structure.

```
breedingData <- list(month = "June", gestation = 33, biparental = FALSE)
animal <- list(name = "hedgehog", mass = 0.8, mammal = TRUE, breeding = breedingData)
str(animal)
```

```
## List of 4
##  $ name    : chr "hedgehog"
##  $ mass    : num 0.8
##  $ mammal  : logi TRUE
##  $ breeding:List of 3
##   ..$ month     : chr "June"
##   ..$ gestation : num 33
##   ..$ biparental: logi FALSE
```

Subsetting lists is done using square brackets, as with vectors. However, with lists you have a choice of using either single "[" or double "[[" brackets. Use single brackets to return a (shorter) list, and double brackets to return just the element.

```
x <- animal[1]
x
```

```
## $name
## [1] "hedgehog"
```

```
class(x)
```

```
## [1] "list"
```

```
y <- animal[[1]]
y
```

```
## [1] "hedgehog"
```

```
class(y)
```

```
## [1] "character"
```

Once named, list elements can be called by name in the same way, using either single or double brackets. You can also call named list elements using the syntax "listName$elementName"

```
animal["mass"]
```

```
## $mass
## [1] 0.8
```

```
animal[["mass"]]
```

```
## [1] 0.8
```

```
animal[c("name", "breeding")]
```

```
## $name
## [1] "hedgehog"
##
## $breeding
## $breeding$month
## [1] "June"
##
## $breeding$gestation
## [1] 33
##
## $breeding$biparental
## [1] FALSE
```

```
animal$name
```

```
## [1] "hedgehog"
```

If you have lists within lists, then you need one set of brackets to go down each level of list. For example to select the first element of the sublist "breeding", one possible command would be:

```
animal[["breeding"]][[1]]
```

```
## [1] "June"
```

The "$" operator is also useful when adding to an existing list. For example, imagine we wanted to add a named vector to the list recording how many observations there were of hedgehogs that week.

```
weekObs <- c(mon = 34, tue = 31, wed = 23, thur = 13, fri = 38)
animal$obs <- weekObs
str(animal)
```

```
## List of 5
##  $ name    : chr "hedgehog"
##  $ mass    : num 0.8
##  $ mammal  : logi TRUE
##  $ breeding:List of 3
##   ..$ month     : chr "June"
##   ..$ gestation : num 33
##   ..$ biparental: logi FALSE
##  $ obs     : Named num [1:5] 34 31 23 13 38
##   ..- attr(*, "names")= chr [1:5] "mon" "tue" "wed" "thur" ...
```

You can achieve the same outcome using double brackets, and it is also possible to add data to sublists.

```
animal[["obs"]] <- weekObs
animal$breeding$weaning <- 35
animal
```

```
## $name
## [1] "hedgehog"
##
## $mass
## [1] 0.8
##
## $mammal
## [1] TRUE
##
## $breeding
## $breeding$month
## [1] "June"
##
## $breeding$gestation
## [1] 33
##
## $breeding$biparental
## [1] FALSE
##
## $breeding$weaning
## [1] 35
##
##
## $obs
##  mon  tue  wed thur  fri
##   34   31   23   13   38
```

Exercise 5: Lists

a) Using brackets correctly is the key to working with lists. Two of the next four expressions will give an error, while the other two will work properly. Can you figure out which before inputting the code? For the two that fail, what causes the errors?

```
animal[c(1, 3)]
animal[[c(1, 3)]]
animal[[1]][[3]]
animal[[c(4, 1)]]
```

b) Add a new element to the list "animal". Call the new element "order" and give it the value "Eulipotyphla".
c) Add a new element to the sublist "breeding" within "animal". Call the new element "litter" and give it the value 3.5.

Data frames

A data frame is like a matrix in being a rectangular array of data, but each column in a data frame can be a different atomic data type. For many applications, this provides an ideal compromise between a matrix (which must be all one data type) and a list (which does not form a rectangular array). Most interesting data problems involve a mixture of categorical, logical, and numeric variables (columns) each with a number of observations (rows), making data frames usually the best way to store information in R.

Data frames can be created by combining vectors using the data.frame() function. This function requires all the input vectors to be the same length.

```
speciesName <- c("Aardvark", "Bear", "Cheetah", "Deer", "Elephant", "Fox")
mass <- c(70, 217, 47, 200, 5400, 6.4)
herbivore <- c(FALSE, FALSE, FALSE, TRUE, TRUE, FALSE)
species <- data.frame(speciesName, mass, herbivore)
species
```

```
##   speciesName   mass herbivore
## 1    Aardvark   70.0     FALSE
## 2        Bear  217.0     FALSE
## 3     Cheetah   47.0     FALSE
## 4        Deer  200.0      TRUE
## 5    Elephant 5400.0      TRUE
## 6         Fox    6.4     FALSE
```

As with other kinds of objects, you can name them using names(), rownames(), colnames(), or explicitly on creation. Columns are allowed to have the same name as the overall data frame, and rownames do not need to be numbers.

```
names(species) <- c("species", "mass", "herbivore")
species
```

```
##    species   mass herbivore
## 1 Aardvark   70.0     FALSE
## 2     Bear  217.0     FALSE
## 3  Cheetah   47.0     FALSE
## 4     Deer  200.0      TRUE
## 5 Elephant 5400.0      TRUE
## 6      Fox    6.4     FALSE
```

```
rownames(species) <- c("a","b","c","d","e","f")
species
```

```
##    species   mass herbivore
## a Aardvark   70.0     FALSE
## b     Bear  217.0     FALSE
## c  Cheetah   47.0     FALSE
```

```
## d     Deer  200.0      TRUE
## e Elephant 5400.0      TRUE
## f      Fox    6.4     FALSE
```

The str() function gives us a good picture of the structure of a data frame.

```
str(species)
```

```
## 'data.frame':    6 obs. of  3 variables:
##  $ species  : Factor w/ 6 levels "Aardvark","Bear",..: 1 2 3 4 5 6
##  $ mass     : num  70 217 47 200 5400 6.4
##  $ herbivore: logi  FALSE FALSE FALSE TRUE TRUE FALSE
```

The above output shows us that the character vector "speciesName" was interpreted as a factor (categorical variable) when incorporated into the data frame. This is the default behaviour, but is wrong if we intend for species names to be unique. You suppress this using an optional argument to data.frame().

```
species <- data.frame(species = speciesName, mass = mass, herbivore = herbivore, stringsAsFactors = FALS
str(species)
```

```
## 'data.frame':    6 obs. of  3 variables:
##  $ species  : chr  "Aardvark" "Bear" "Cheetah" "Deer" ...
##  $ mass     : num  70 217 47 200 5400 6.4
##  $ herbivore: logi  FALSE FALSE FALSE TRUE TRUE FALSE
```

Subsetting dataframes works exactly the same way as subsetting matrices.

```
species[3,2]
```

```
## [1] 47
```

```
species[3,"mass"]
```

```
## [1] 47
```

```
species[3,]
```

```
##   species mass herbivore
## 3 Cheetah   47     FALSE
```

```
species[,"mass"]
```

```
## [1]   70.0  217.0   47.0  200.0 5400.0    6.4
```

```
species[c(3, 5), c("mass", "herbivore")]
```

```
##   mass herbivore
## 3   47     FALSE
## 5 5400      TRUE
```

You can also select columns using $ or [[]], in the same way as a list.

```
species$mass
```

```
## [1]   70.0  217.0   47.0  200.0 5400.0    6.4
```

```
species[["mass"]]
```

```
## [1]   70.0  217.0   47.0  200.0 5400.0    6.4
```

```
species[[2]]
```

```
## [1]   70.0  217.0   47.0  200.0 5400.0    6.4
```

To make selecing data frame columns easier, there is a function attach() that allows you to call them directly. This is useful until you have multiple data frames with duplicate column names, or objects with similar names, at which point it becomes a liability. You can use detach() to remove the functionality again.

```
rm(speciesName,mass,herbivore) # remove vectors used to build "species"
herbivore <- "Zebu"
attach(species)
```

```
## The following objects are masked _by_ .GlobalEnv:
##
##     herbivore, species
```

```
mass # Returns column of "species", as intended
```

```
## [1]   70.0  217.0   47.0  200.0 5400.0    6.4
```

```
herbivore # Returns object "name", not column
```

```
## [1] "Zebu"
```

```
detach(species)
```

A less risky method is to use with(), which temporarily attaches dataframes.

```
with(species, mean(mass))
```

```
## [1] 990.0667
```

Adding columns to a data frame can be done either by name (like a list) or using cbind() (like a matrix).

```
order <- c("Tubulidentata", "Carnivora", "Carnivora", "Artiodactyla", "Proboscidea", "Carnivora")
species$order <- order
species[["order"]] <- order

threatened <- c(FALSE, FALSE, TRUE, FALSE, TRUE, FALSE)
species <- cbind(species, threatened)
species
```

```
##    species   mass herbivore         order threatened
## 1 Aardvark   70.0     FALSE Tubulidentata      FALSE
## 2     Bear  217.0     FALSE     Carnivora      FALSE
## 3  Cheetah   47.0     FALSE     Carnivora       TRUE
## 4     Deer  200.0      TRUE  Artiodactyla      FALSE
## 5 Elephant 5400.0      TRUE   Proboscidea       TRUE
## 6      Fox    6.4     FALSE     Carnivora      FALSE
```

New observations can be added to a data frame using rbind(), like a matrix, but only if the number and type of columns match:

```
gorilla <- data.frame("gorilla", 155, TRUE, "Primates")
rbind(species, gorilla) # Gives error
```

```
## Error in rbind(deparse.level, ...): numbers of columns of arguments do not match
```

```
gorilla <- data.frame(species = "gorilla", mass = 155, herbivore = TRUE, order = "Primates", threatened
species <- rbind(species, gorilla)
species
```

```
##    species   mass herbivore         order threatened
## 1 Aardvark   70.0     FALSE Tubulidentata      FALSE
## 2     Bear  217.0     FALSE     Carnivora      FALSE
```

```
## 3  Cheetah   47.0     FALSE     Carnivora       TRUE
## 4     Deer  200.0      TRUE  Artiodactyla      FALSE
## 5 Elephant 5400.0      TRUE   Proboscidea       TRUE
## 6      Fox    6.4     FALSE     Carnivora      FALSE
## 7  gorilla  155.0      TRUE      Primates       TRUE
```

Exercise 6: Data frames

a) Select just the first and third columns of the data frame "species", and save them as a new object.
b) Change the mass of the elephant in "species" to 5450.
c) Create a new row for the data frame "species" with information about an animal of your choice, and add this to "species".

Part 3: Working with Data

The Workspace

The Workspace refers to your current working environment, including all the objects you have defined. The Environment tab (top right panel) lets you see the names and structure of these objects, and if you click on an object in this tab RStudio will open it in a more detailed view. This is very useful for inspecting larger objects such as spreadsheets you have imported into R as data frames. You can also use the function View() for the same effect.

```
View(species)  # note the capital V!
```

When importing and exporting data, it is usually easiest to store all the files in the same place, and set that folder as the default location (workind directory) for R. You can find what R's current default is using the getwd() function, and set a new one using setwd().

```
getwd()
setwd("C:/Program Files/R")    # you may need to modify this for your computer
```

In RStudio, you can also navigate to a folder using the Files tab (bottom right panel), then select More -> Set As Working Directory. This is identical to using setwd() directly.

Reading and Writing Data

The two most common ways for getting data into and out of R are (a) importing and exporting spreadsheets used by another program (such as Excel), and (b) saving and loading R objects for use by you or someone else.

You can save dataframes as a .csv file using the write.csv() function. By default, this will be saved in your current working directory. If you want another type of text file as output, write.table() gives some further options.

```
write.csv(species, file = "species.csv")
rm(species)
species
species <- read.csv(file = "species.csv")
species
```

Other functions such as read.table() and read.delim() are useful for other sorts of text files.

To save amd load R objects, you use the imaginatively titles save() and load() functions. By default, R will look for the file in the current working directory

```
save(gorilla, file = "gorilla.R")
rm(gorilla)
gorilla
load("gorilla.R")
gorilla
```

There is a package called "foreign"" that provides functionality for directly importing SPSS, Stata, Minitab, and other files, but usually it is simpler to just save the output from these programs as csvs and then import then into R using familar functions.

Inspecting data objects

In addition to the familiar str(), it is often useful to have a look at the top or bottom few rows of your data frames. You can do this using the functions head() and tail()

```
head(species,3)
```

```
##    species mass herbivore         order threatened
## 1 Aardvark   70     FALSE Tubulidentata      FALSE
## 2     Bear  217     FALSE     Carnivora      FALSE
## 3  Cheetah   47     FALSE     Carnivora       TRUE
```

```
tail(species,3)
```

```
##    species   mass herbivore       order threatened
## 5 Elephant 5400.0      TRUE Proboscidea       TRUE
## 6      Fox    6.4     FALSE   Carnivora      FALSE
## 7  gorilla  155.0      TRUE    Primates       TRUE
```

Sorting, Splitting, and Removing NAs

You can sort vectors and data frames using the functions sort() and order(). Be careful with sort() - it rearrages vectors (i.e. columns) even inside data frames, and there is no unsort.

```
sort(species$mass)
```

```
## [1]    6.4   47.0   70.0  155.0  200.0  217.0 5400.0
```

```
rankMass <- order(species$mass)
rankMass
```

```
## [1] 6 3 1 7 4 2 5
```

```
species[rankMass, ]
```

```
##    species   mass herbivore         order threatened
## 6      Fox    6.4     FALSE     Carnivora      FALSE
## 3  Cheetah   47.0     FALSE     Carnivora       TRUE
## 1 Aardvark   70.0     FALSE Tubulidentata      FALSE
## 7  gorilla  155.0      TRUE      Primates       TRUE
## 4     Deer  200.0      TRUE  Artiodactyla      FALSE
## 2     Bear  217.0     FALSE     Carnivora      FALSE
## 5 Elephant 5400.0      TRUE   Proboscidea       TRUE
```

```
species[order(species$mass),]  # in one line
```

```
##    species   mass herbivore         order threatened
## 6      Fox    6.4     FALSE     Carnivora      FALSE
## 3  Cheetah   47.0     FALSE     Carnivora       TRUE
## 1 Aardvark   70.0     FALSE Tubulidentata      FALSE
## 7  gorilla  155.0      TRUE      Primates       TRUE
## 4     Deer  200.0      TRUE  Artiodactyla      FALSE
## 2     Bear  217.0     FALSE     Carnivora      FALSE
## 5 Elephant 5400.0      TRUE   Proboscidea       TRUE
```

You can split up data objects using the split() function. This is most commonly used to divide (e.g.) a data frame into two based on one of its columns. The result is a named list.

```
split(species,species$threatened)
```

```
## $`FALSE`
##    species  mass herbivore         order threatened
## 1 Aardvark  70.0     FALSE Tubulidentata      FALSE
## 2     Bear 217.0     FALSE     Carnivora      FALSE
## 4     Deer 200.0      TRUE  Artiodactyla      FALSE
## 6      Fox   6.4     FALSE     Carnivora      FALSE
##
## $`TRUE`
##    species mass herbivore       order threatened
## 3  Cheetah   47     FALSE   Carnivora       TRUE
## 5 Elephant 5400      TRUE Proboscidea       TRUE
## 7  gorilla  155      TRUE    Primates       TRUE
```

We often want to remove NAs from our datasets. Imagine we find that two of our estimates for mass are inaccurate, and want to remove those rows from our dataframe.

```
species[4:5,"mass"] <- NA
species
```

```
##    species  mass herbivore         order threatened
## 1 Aardvark  70.0     FALSE Tubulidentata      FALSE
## 2     Bear 217.0     FALSE     Carnivora      FALSE
## 3  Cheetah  47.0     FALSE     Carnivora       TRUE
## 4     Deer    NA      TRUE  Artiodactyla      FALSE
## 5 Elephant    NA      TRUE   Proboscidea       TRUE
## 6      Fox   6.4     FALSE     Carnivora      FALSE
## 7  gorilla 155.0      TRUE      Primates       TRUE
```

```
species[!is.na(species$mass),]
```

```
##    species  mass herbivore         order threatened
## 1 Aardvark  70.0     FALSE Tubulidentata      FALSE
## 2     Bear 217.0     FALSE     Carnivora      FALSE
## 3  Cheetah  47.0     FALSE     Carnivora       TRUE
## 6      Fox   6.4     FALSE     Carnivora      FALSE
## 7  gorilla 155.0      TRUE      Primates       TRUE
```

If you want to use only those rows with data in all columns, these can be selected quickly using complete.cases() without knowing where the missing data are.

```
species_complete <- species[complete.cases(species),]
species_complete
```

```
##    species  mass herbivore         order threatened
## 1 Aardvark  70.0     FALSE Tubulidentata      FALSE
## 2     Bear 217.0     FALSE     Carnivora      FALSE
## 3  Cheetah  47.0     FALSE     Carnivora       TRUE
## 6      Fox   6.4     FALSE     Carnivora      FALSE
## 7  gorilla 155.0      TRUE      Primates       TRUE
```

Exercise 7: Data manipulation

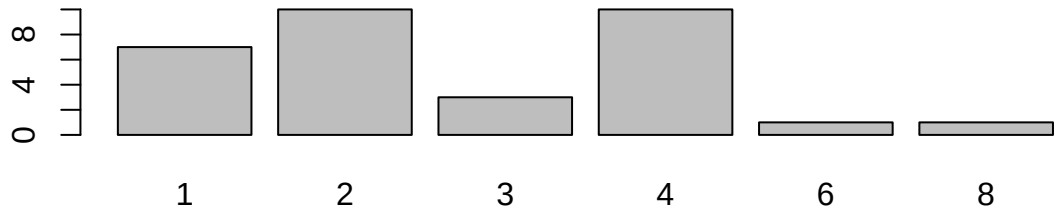
a) Order the data frame "species" by threatened status.
b) Split the data frame "species" into two dataframes of herbivorous and non-herbivorous animals.
c) Create a data frame consisting of only the first two rows of "species", and then save it as a .csv file with a suitable name.

Part 4: Data visualisation

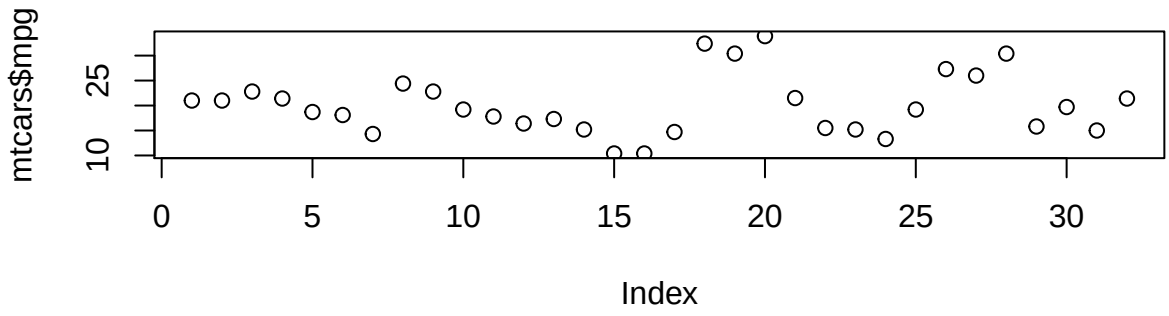
Types of plots

The simplest way to plot data in R is (can you guess?) the plot() function. This is a generic function, so different types of input lead to different types of plot: it will try its best to do something with whatever data you feed it. For these examples, we will be taking advantage of mtcars, one of R's built-in datasets. This one contains data on cars from the 1974 Motor Trend magazine.

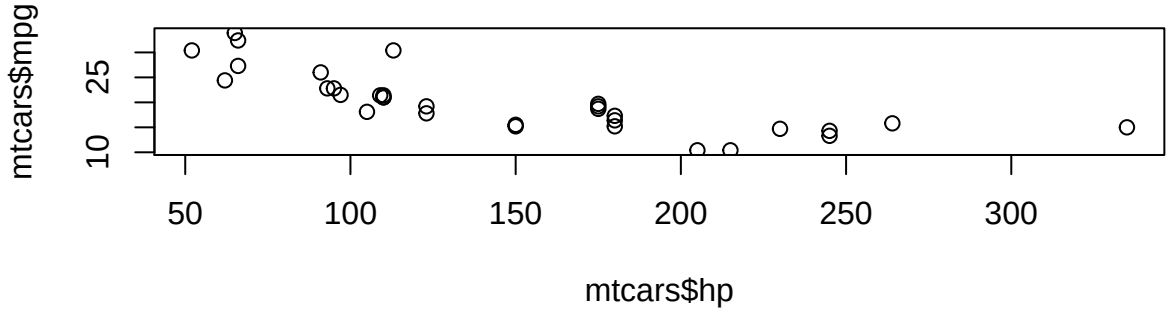
```
data(mtcars)
plot(as.factor(mtcars$carb))
```



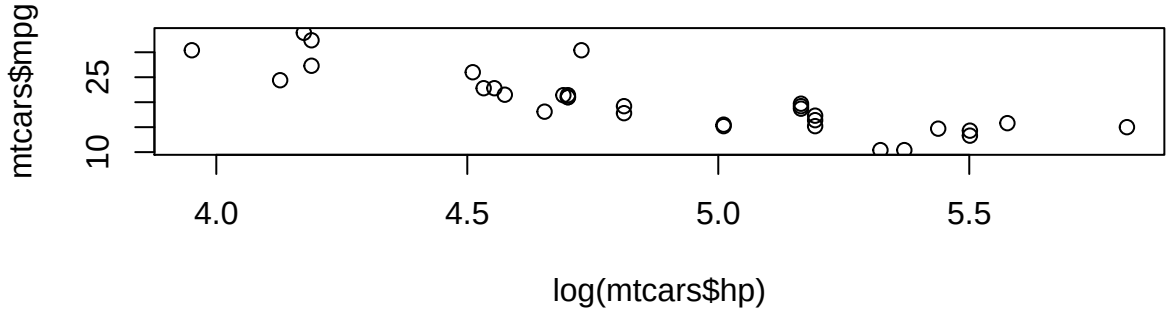

```
plot(mtcars$mpg)
```



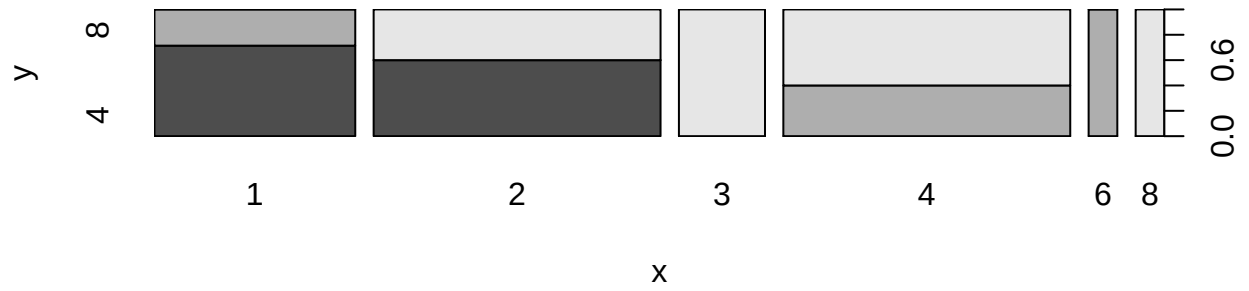

```
plot(mtcars$hp,mtcars$mpg)
```




```
plot(log(mtcars$hp),mtcars$mpg)
```

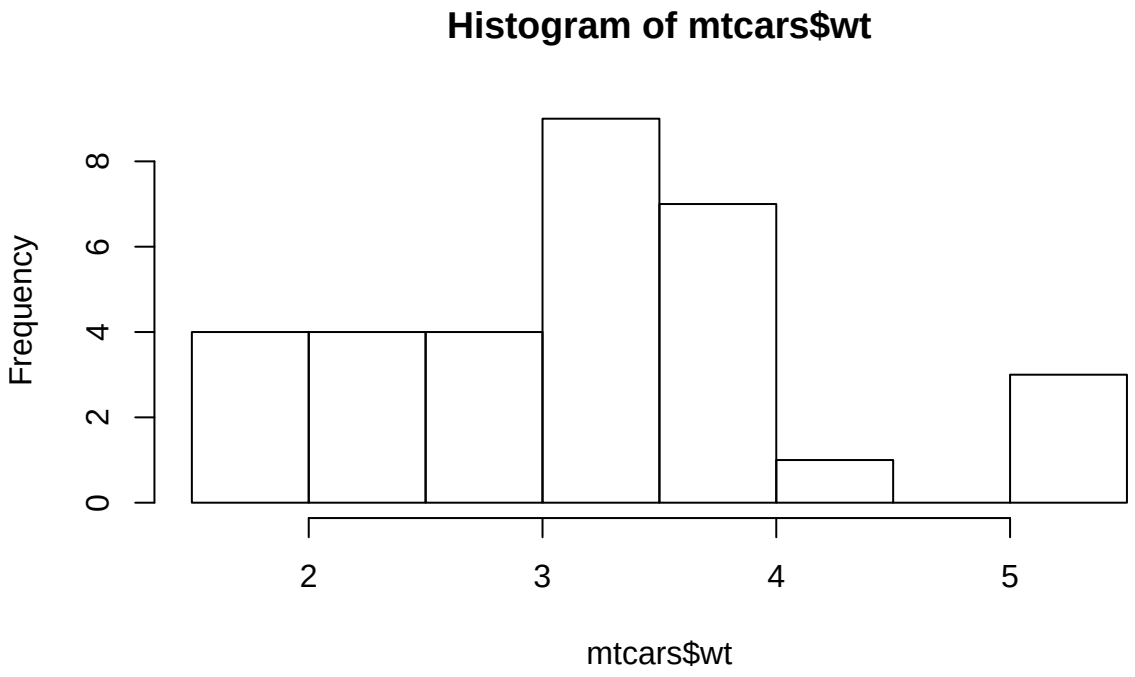



```
plot(as.factor(mtcars$carb),as.factor(mtcars$cyl))
```



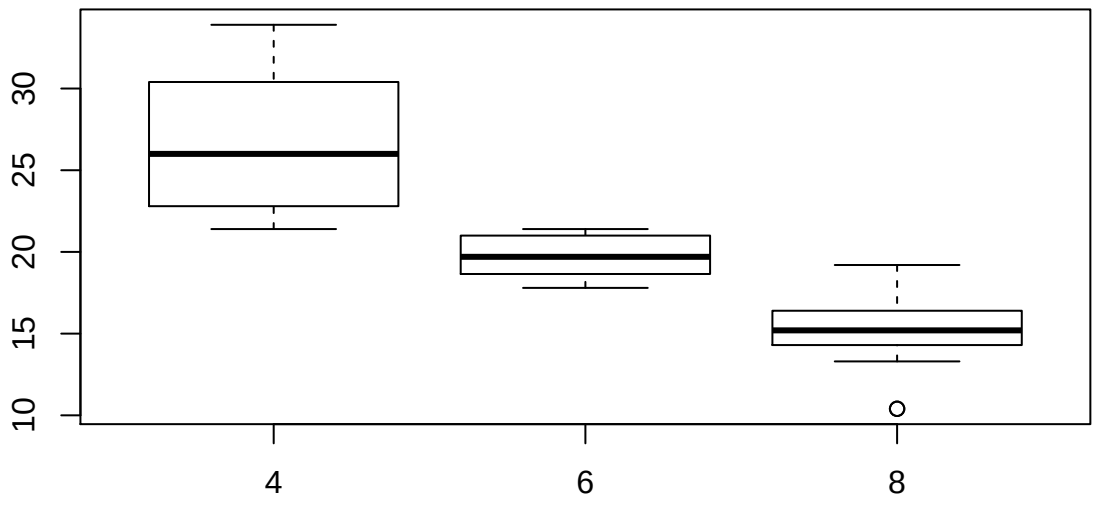
Of course, there are many other types of plot that plot() cannot generate. Histograms are commonly desired for plotting frequency distributions, and are generated using hist().

```
hist(mtcars$wt)
```



To compare numemrical data across categories, often you want to use a boxplot. No prizes for guessing the name of the function.

```
boxplot(mtcars$mpg ~ mtcars$cyl)
```

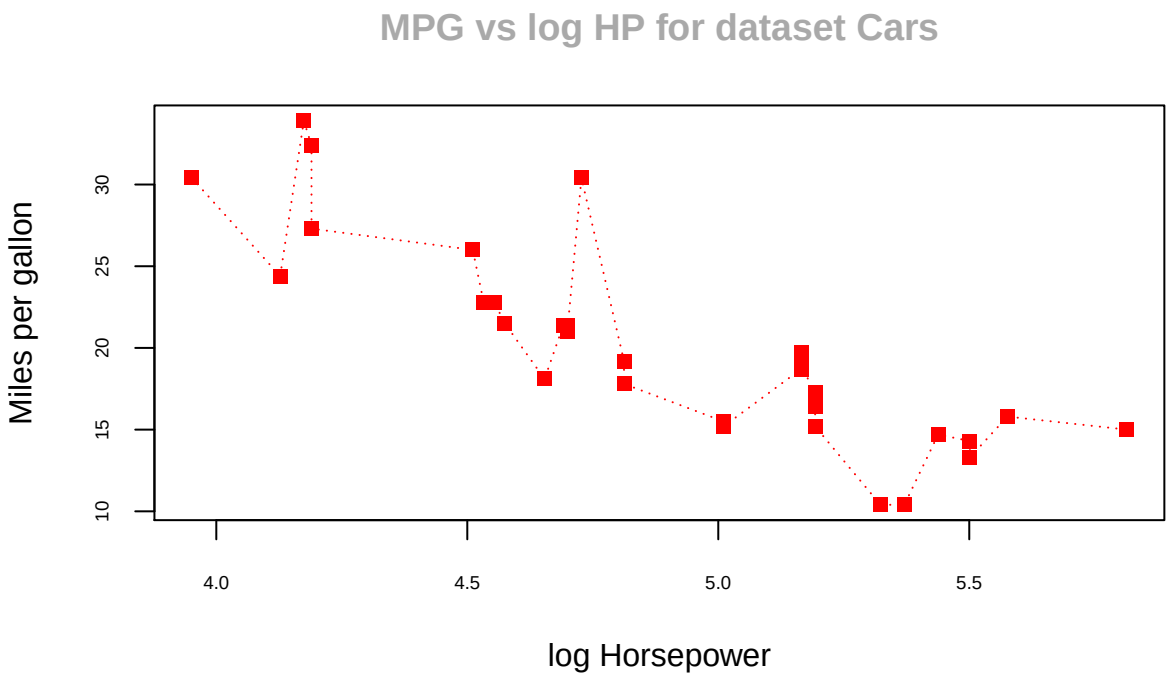


Plot Options

In addition to providing lots of flexibility in how your data are plotted, R also gives you complete flexibility in how your plot appears. The following code changes a large number of the plot options and graphical parameters available, but there are many more.

```
cars <- mtcars[order(mtcars$hp),]

plot(log(cars$hp),cars$mpg,
xlab = "log Horsepower",          # X-axis label
ylab = "Miles per gallon",        # Y-axis label
main = "MPG vs log HP for dataset Cars",    # Plot title
type = "o",         # Plot type (p = points, l = lines, o = both etc.)
col = "red",         # Plot colour
col.main = "darkgray",    # Plot title colour
cex.axis = 0.6,            # Font size of the axis
lty = 3,                  # Line type
pch = 15)                 # Point type
```

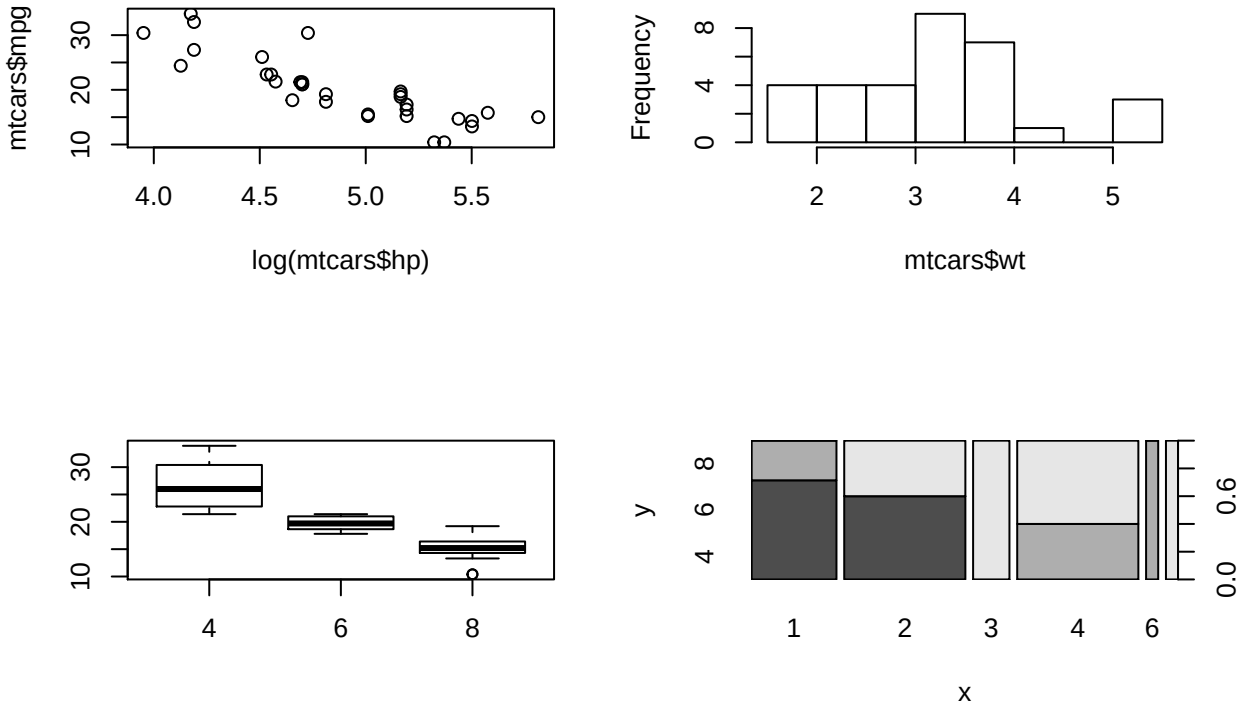


You can get a list of plot() arguments through ?plot, as usual, and a list of the generic graphical parameters through ?par. Entering par() with no arguments outputs a list of the current defaults.

```
par()
```

Changing the default par() values can be useful if you are planning on making a number of plots in a similar style, or want to display a number of plots together. For example, suppose we want to create a 2x2 grid of plots for a 4-panel figure.

```
par(mfrow = c(2,2))
plot(log(mtcars$hp),mtcars$mpg)
hist(mtcars$wt, main = "")
boxplot(mtcars$mpg ~ mtcars$cyl)
plot(as.factor(mtcars$carb),as.factor(mtcars$cyl))
```

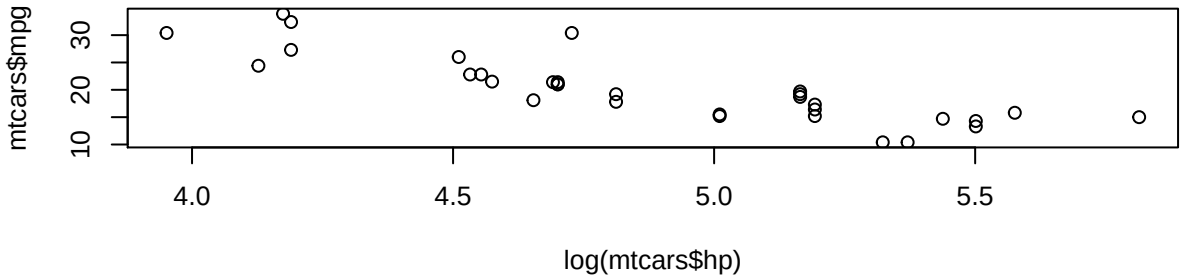


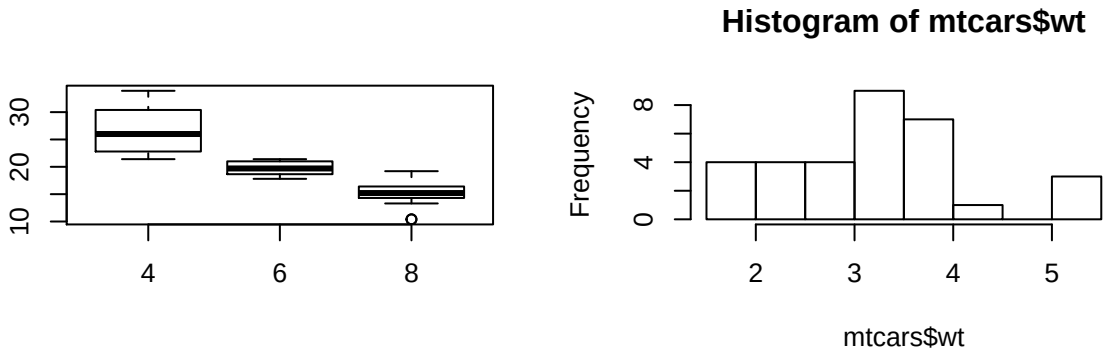

```
par(mfrow = c(1,1)) # set back to default 1x1 plot
```

More complex arragements of plots are also possible, using the layout() function.

```
grid <- matrix(c(1, 1, 2, 3), nrow = 2,
ncol = 2, byrow = TRUE)

layout(grid)
plot(log(mtcars$hp),mtcars$mpg)
boxplot(mtcars$mpg ~ mtcars$cyl)
hist(mtcars$wt)
```






```
layout(1)
```

You can see some more examples of R plotting using example().

```
example(plot)
```

There are many excellent packages in R for producing more complex plots, or plots of different styles. Some of the most popular are ggplot2, ggvis and lattice.

Exercise 8: Plots

a) Create a basic plot of miles per gallon (mpg) against weight (wt) using the mtcars dataset. Add axis labels to your plot.
b) Create a histogram of 1/4 mile time (qsec). Change the title to something descriptive.
c) Create a boxplot of miles per gallon (mpg) grouped by transmission type (am). Change the colour of the boxplots to red (or any another colour). Bonus: make the boxes two different colours.

Wrap-up Day 1

That was a lot to cover! Don't worry if you don't fully understand or remember every topic explored today: the aim was just to introduce you to them so you feel comfortable trying things again in your own time. Hopefully these notes will provide a useful reference point when you come to do your own analyses and data manipulation using R. When you do, the most important thing to remember is that you can always look up how functions works using "?", and if you are still stuck a quick web search will usually bring up someone else with the same problem.

If you want to learn more, detailed R documentation is available at http://www.rdocumentation.org/ and http://cran.r-project.org/doc/manuals/R-lang.html, and there are many excellent resources online to take you further. Springer has a series of books called Use R! that cover a wide range of topics, and a much longer (annotated) list of R books is available at http://www.r-project.org/doc/bib/R-books.html.

Appendix: Some extra tricks

The following are more intermediate-level topics in R, but are included for reference because they can be very useful for the sorts of problems encountered when working with large datasets and multiple files.

The "Apply" family of functions

This family of functions all end in "apply", and are all designed to help you perform a particular function multiple times across a data frame, matrix, list, or vector. This is incredibly useful to avoid running slight variations of the same comman many times.

The main apply() funtion is used to a evaluate a function over a matrix or array. In this example I also use the function rnorm() to generate random numbers.

```
x <- matrix(rnorm(200), 20, 10)  # creates a random matrix
apply(x, 2, mean) # returns the mean of each column
```

```
##  [1] -0.24953466  0.02400524  0.10895512  0.11205886  0.13180581
##  [6]  0.12334070  0.00294864  0.15662697 -0.19536602 -0.27435080
```

```
apply(x, 1, sum)  # returns the sum of each row
```

```
##  [1]  0.8773218 -0.2765526 -2.8961671  2.6313449  1.3475184  3.2369121
##  [7]  8.4341810  4.9196141 -2.6905618 -0.2446130 -0.8876090 -1.3412221
## [13] -0.1085303 -4.6656598 -7.9895295  1.6395027 -2.7654308  2.8786451
## [19] -2.9613070 -0.3280600
```

lapply() evaluates a function over a list. sapply() does the same but tries to simplify the result (returning a vector or matrix instead of a list, if possible).

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] -0.0285791
##
## $c
## [1] 1.178845
```

```
##
## $d
## [1] 4.929041
```

```
sapply(x, mean)
```

```
##          a          b          c          d
##  2.5000000 -0.0285791  1.1788446  4.9290406
```

tapply evaluates a function over subsets of a vector.

```
tapply(species$mass, species$herbivore, mean)
```

```
## FALSE  TRUE
##  85.1    NA
```

Simple loops

Sometimes you want to apply a function a large number of times, but the apply functions don't really fit your needs. In this case you can use a "for"" loop. This is best illustrated using an example. Imagine we have a folder containing data files in .csv format, and want to get a summary of the data in each. We need to: 1. Identify all the files that need analysing 2. Read in each one as a dataframe 3. Do the analysis on each one 4. Save the results As code, this process would look something like this:

```
setwd("C:/path.to.folder") # tell R where to look for files
results <- list() # Create an empty list to receive the results
files.v <- list.files(getwd()) # create a vector of the files in the folder

for (i in 1:length(files.v)) {  # Run a loop i times, from 1 - the number of files present
  current.file <- read.csv(files.v[i]) # read in the ith file as a data frame
  results[[i]] <- summary(current.file) # summarise the ith file, and save it as the ith element of the
}

results
```

As a bonus exercise, try to get the above code working! You can try it on real data if you have some, or just use write.csv() to create some simple files using R.Other control structures such as "if", "else", and "while" are also worth looking into for tackling this sort of problem.