Using R for Mapping and Visualization

Alex Singleton

1 November 2015

Learning Objectives

1. Introduce basic functionality of R including calculations and attribute types
2. Illustrate the storage of data in objects including lists and data frames
3. Learn to create data frames from scratch and also how to import data from a CSV file
4. Develop basic data frame manipulation skills including subsetting, calculations and recoding

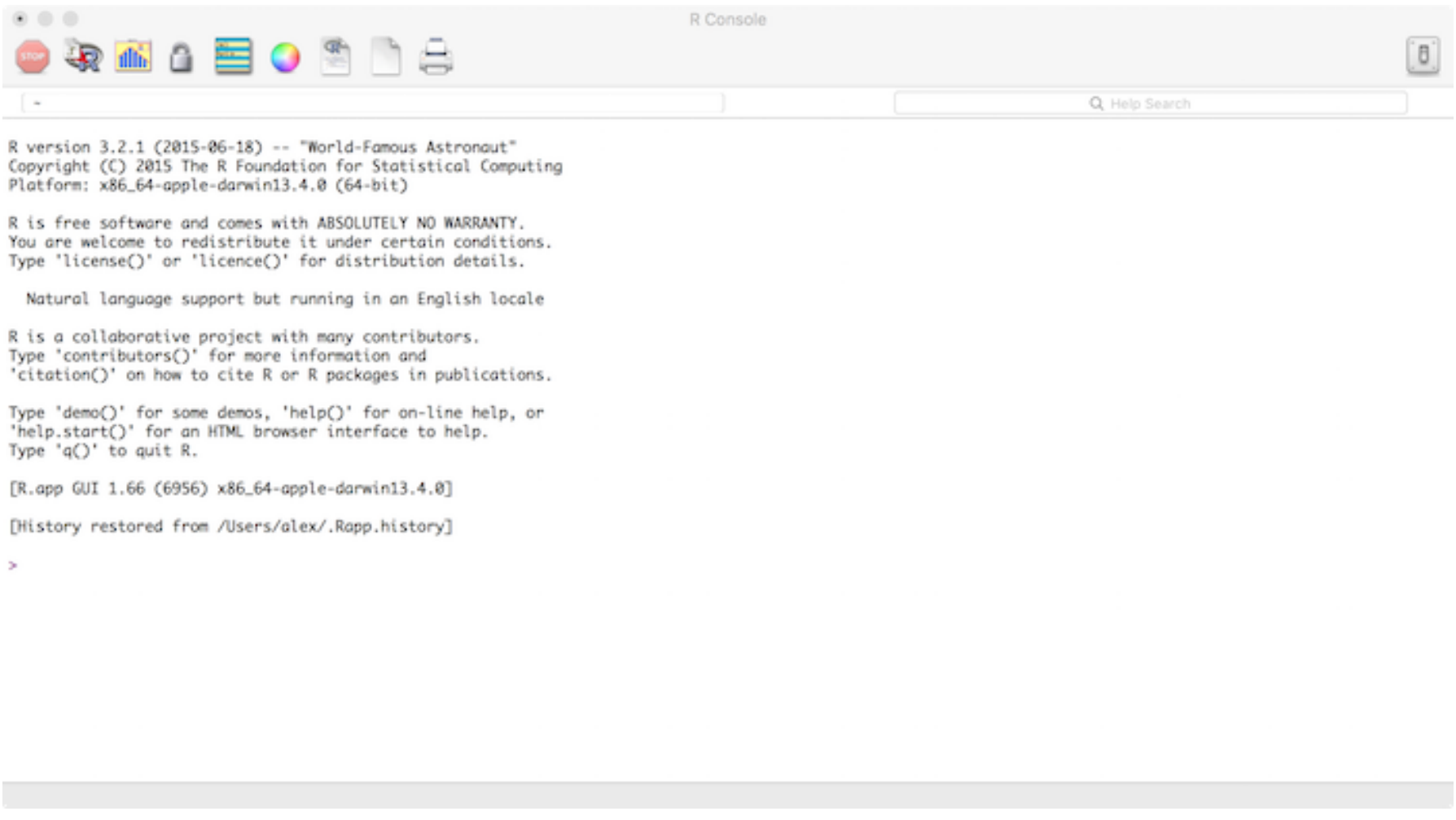
An Introduction to R

R is a popular statistical programming language that has gained widespread use in both academia and industry. R offers a wider array of functionality than a traditional statistics package such as SPSS, and is composed of core (base) functionality, however, is very expandable.R is an Open Source language.

Commands are sent to R using either the terminal / command line or the R Console which is installed with R on either Windows or OS X. On Linux, there is no equivalent of the console, however, third party solutions exist. On your own machine R can be installed here https://www.r-project.org/ (https://www.r-project.org/).

R is installed on all of the university machines, and can be opened from the start menu by clicking on the R icon.

This will open an R console which will look like:



Setting up R for the First Time

When using R for the first time, you need to tell the software where you are going to store your files, including input files (e.g. data) or outputs generated (e.g. graphics). This is called a working directory. For this module, it is recommended that you create a folder for each week. So, in your “M://” drive working directory you might created the following folder structure.

```
>>> M:/
>>>>>>> ENVS456
>>>>>>>>>>>Week_1
```

It is good practice to not include spaces when naming folders, so in this instance we call the working directory for this week “Week_1”. On a Windows machine it does not matter if you use captial or lower case letters as they are interpreted in the same way, however, on Linux or OS X, these are case sensitive. For cross platform compatability it is recomended that you always use the case as specified in your file structure; so “Week_1” rather than “week_1”.

You can see what the current working directory that R is using with the get working directory command (“getwd()”).

```
getwd()
```

```
## [1] "/Users/alex/Dropbox/Teaching/Lectures/ENVS456 - Web Mapping/Practicals/Wee
k1"
```

This has printed the working directory. You will also see “## [1]” before the directory name, however, this is just a line number. Later you will see outputs that span over multiple lines.

You can set a new working directory using the set command.

```
setwd("M:/ENVS456/Week_1")
```

Your first R commands

To get started with R, we will first illustrate how the terminal works by using this to perform some simple calculations. For example, lets begin by adding 10 and 57. Type the following into R and press enter.

```
10 + 57
```

```
## [1] 67
```

You can also use R to perform other numerical operation on these numbers such as multiplication (“*“), division(”/“) or subtraction(”-“).

```
10 * 57
```

```
## [1] 570
```

```
10 / 57
```

```
## [1] 0.1754386
```

```
10 - 57
```

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

There are many other basic mathematical operators built into R such as powers ("^"), square root ("sqrt()") or log ("log()").

```
12^2
```

```
## [1] 144
```

```
sqrt(12)
```

```
## [1] 3.464102
```

```
log(12)
```

```
## [1] 2.484907
```

You can also combine operators together; for example

```
log(3+3*(6/4))
```

```
## [1] 2.014903
```

Variables and Objects

Variables are very important to all programming languages. They enable you to store "values", and then use these at a later stage. In R we can use the symbols `<-` to assign a value to a variable name. The variable name is entered on the left of `<-` and the value on the right. A numeric value can be assigned as follows.

```
product <- 900
```

This has stored the value 900 in the variable product. You can always find out what the value of a variable is by typing it into the terminal.

```
product
```

```
## [1] 900
```

As was mentioned previously, you can then use variables in you R statements. For example, we might want to find out the new value if a 30% discount was applied to the product.

```
product * 0.7
```

```
## [1] 630
```

We might also want to store the value of the calculation in a new variable called “sale_product” which we can again do using the “<-” symbol.

```
sale_product <- product * 0.7
```

If you now print the “sale_product” variable you get the new stored value.

```
sale_product
```

```
## [1] 630
```

Another type of object in R is a “list” which can store a number of different values. To create a list we use a concatenation function `c`. For example, we might want to store a range of numeric values in new variable called “milk”, recording the values of the price of a pint of milk in a number of shops.

```
milk <- c(50,58,70,45,99,56,64,45,46,55,80,62)
milk
```

```
##  [1] 50 58 70 45 99 56 64 45 46 55 80 62
```

Using the list we can then apply a number of functions to generate statistics about the numbers stored in the list. For example you might be interested in calculating the minimum, maximum, average or standard deviation.

```
min(milk)
```

```
## [1] 45
```

```
max(milk)
```

```
## [1] 99
```

```
mean(milk)
```

```
## [1] 60.83333
```

```
median(milk)
```

```
## [1] 57
```

```
sd(milk)
```

```
## [1] 16.02177
```

Another very useful feature of lists is that we you can apply calculations to them. For example, if the price of milk were to go down by 20%, we can see the effect this has on the price as follows.

```
milk * 0.8
```

```
##  [1] 40.0 46.4 56.0 36.0 79.2 44.8 51.2 36.0 36.8 44.0 64.0 49.6
```

Logic

Logic is another important concept in R and is used to test whether a statement is true or false. For example, is one number larger than another. There are a range of logical operators in R which include

```
#greater than
10 > 5
```

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

```
#less than
10 < 5
```

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

```
#equal to
10 == 5
```

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

```
#not equal to
10 != 5
```

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

You can also join together logical statements where either *both* statements have to be true to return true. This uses the & character and is sometimes called an “and” statement

```
#Will return true
(10 > 5) & (6 > 1)
```

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

```
#Will return false because both statements are not true
(10 > 5) & (6 > 7)
```

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

You can also test to see if *either* statements are true, and if so, this returns true. This uses the | character and is called and “or” statement.

```
#Will return true because one of the statements is true
(10 > 5) | (6 > 7)
```

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

Introducing Tabular Data

Many analysis tasks will require data that are in tabular format. Conceptually, this is very similar to a table that you might find in a database or a spreadsheet; which is made up of columns and rows of values. These objects are called a “data frame” in R. Later we will illustrate how a data frame can be created by importing an external text file, however, below illustrates creating one from scratch using two lists of numbers.

First we need to create two new variables, with each containing a list of numbers. For this example we use 2011 census data counts for the total people (“all_people”) and those who identified themselves in good health (“good_health”) for wards in Liverpool.

```
all_people <- c(14853,14510,15004,20340,13908,13974,15272,14045,14503,
                14561,14782,16786,16132,15377,16115,13312,13816,15047,
                16461,17009,17104,18422,12991,20300,16489,16481,14772,
                14382,12921,16746)

good_health <- c(7274,6124,6129,11925,7219,7461,6403,5930,7094,6992,
                 5517,7879,8990,6495,6662,5981,7322,6529,7192,7953,
                 7636,9001,6450,8973,7302,7521,7268,7013,6025,7717)
```

When creating a data frame from multiple lists you need to make sure that they are of the same length; which we can do using another function called `length()`.

```
length(all_people)
```

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

```
length(good_health)
```

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

Both should be `30`. If not, you may have missed a number if you typed these in.

If you remember earlier, we mentioned line numbers. If you now print `all_people` you will see that the numbers output span more than one line. You will see on the second line `## 23` which means that the first number on the second line is 23rd in the sequence of numbers stored in `all_people`.

```
length(all_people)
```

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

We can now join the two objects together to create a data frame by entering:

```
Liverpool_Good_Health <- data.frame(Tot_Pop = all_people, Good_Health = good_healt
h)
```

You can then view the contents of the data frame object by typing in the object name - `Liverpool_Good_Health`.

```
Liverpool_Good_Health
```

The function `data.frame()` has combined the two variables that you specified into columns within the data frame. The columns are assigned names using the '=' sign. Thus, "Tot_Pop = all_people" puts the contents of "all_people" into a new column called "Tot_Pop".

Another column that would be useful are labels for the wards within Liverpool. As such, we will create a new version of the data frame with these added. First we create a new variable `wards`.

```
wards <- c("Allerton and Hunts Cross","Anfield","Belle Vale","Central","Childwall"
,"Church","Clubmoor","County","Cressington","Croxteth","Everton","Fazakerley","Gre
enbank","Kensington and Fairfield","Kirkdale","Knotty Ash","Mossley Hill","Norris
Green","Old Swan","Picton","Princes Park","Riverside","St Michael's","Speke-Garsto
n","Tuebrook and Stoneycroft","Warbreck","Wavertree","West Derby","Woolton","Yew T
ree")
```

Unlike the previous variables which were numeric, `wards` contains a list of character strings. Each element of the string is separated by """. We will then re-create the data frame. Note that the old data frame object is replaced.

```
Liverpool_Good_Health <- data.frame(Ward = wards,Tot_Pop = all_people, Good_Health
= good_health)
Liverpool_Good_Health
```

```
##                        Ward Tot_Pop Good_Health
## 1  Allerton and Hunts Cross   14853        7274
## 2                   Anfield   14510        6124
## 3                Belle Vale   15004        6129
## 4                   Central   20340       11925
## 5                 Childwall   13908        7219
## 6                    Church   13974        7461
## 7                 Clubmoor   15272        6403
## 8                    County   14045        5930
## 9                Cressington   14503        7094
## 10                  Croxteth   14561        6992
## 11                   Everton   14782        5517
## 12                Fazakerley   16786        7879
## 13                 Greenbank   16132        8990
## 14 Kensington and Fairfield   15377        6495
## 15                  Kirkdale   16115        6662
## 16                Knotty Ash   13312        5981
## 17              Mossley Hill   13816        7322
## 18              Norris Green   15047        6529
## 19                  Old Swan   16461        7192
## 20                    Picton   17009        7953
## 21              Princes Park   17104        7636
## 22                 Riverside   18422        9001
## 23              St Michael's   12991        6450
## 24             Speke-Garston   20300        8973
## 25 Tuebrook and Stoneycroft   16489        7302
## 26                  Warbreck   16481        7521
## 27                 Wavertree   14772        7268
## 28               West Derby   14382        7013
## 29                  Woolton   12921        6025
## 30                 Yew Tree   16746        7717
```

Although we have created a data frame from scratch, sometimes we need to know more about the dimensions of the object. We can use the `ncol()` and `nrow()` functions to see how many rows and columns are in the data frame. For example.

```
ncol(Liverpool_Good_Health)
```

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

```
nrow(Liverpool_Good_Health)
```

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

We may also want to know what type of variable are stored in each of the columns. We can find this out using the combination of the `lapply` function combined with the `class` function. The `class` function is used to describe what and object is. So we could use it to find out that the object "Liverpool_Good_Health" is a data frame as follows.

```
class(Liverpool_Good_Health)
```

```
## [1] "data.frame"
```

However, using `lapply()` we can explore each “column” of a data frame as follows to print out the format that each variable is stored in, returning “Ward” as a factor, “Tot_Pop” and “Good_Health” both as numeric. This is useful when reading in data that you haven’t created yourself as you could have variables which look like numbers, e.g. 1; however, are actually read in as characters - e.g. “1”.

```
lapply(Liverpool_Good_Health,class)
```

```
## $Ward
## [1] "factor"
##
## $Tot_Pop
## [1] "numeric"
##
## $Good_Health
## [1] "numeric"
```

The `lapply()` function applies a given function (in this case `class`) to a list. You are probably thinking, but the “Liverpool_Good_Health” object is a data frame not a list. Well, this is sort of correct. A data frame is essentially a special type of list, but you can think of it as a table. You could also use the `lapply()` function on one of the list objects we created earlier to illustrate this.

Factors are used in R for storage efficiency. Rather than storing multiple versions of text strings, a set of integer values are used as a “lookup” to a list of stored character values. You can use the function `factor()` to view the factors used in a list. We can look at the “class” object we created earlier as follows

```
factor(wards)
```

```
##  [1] Allerton and Hunts Cross Anfield
##  [3] Belle Vale               Central
##  [5] Childwall                Church
##  [7] Clubmoor                 County
##  [9] Cressington              Croxteth
## [11] Everton                  Fazakerley
## [13] Greenbank                Kensington and Fairfield
## [15] Kirkdale                 Knotty Ash
## [17] Mossley Hill             Norris Green
## [19] Old Swan                 Picton
## [21] Princes Park             Riverside
## [23] St Michael's             Speke-Garston
## [25] Tuebrook and Stoneycroft Warbreck
## [27] Wavertree                West Derby
## [29] Woolton                  Yew Tree
## 30 Levels: Allerton and Hunts Cross Anfield Belle Vale ... Yew Tree
```

The result is not especially exciting, but reports that there are 30 levels in the object alongside the different factors used with their numeric reference. E.g. 6 is “Church” and 26 is “Warbreck”.

We can also get R to give summary of values in the data frame:

```
summary(Liverpool_Good_Health)
```

```
##                          Ward       Tot_Pop       Good_Health
##  Allerton and Hunts Cross: 1   Min.   :12921   Min.   : 5517
##  Anfield                 : 1   1st Qu.:14412   1st Qu.: 6461
##  Belle Vale              : 1   Median :15026   Median : 7206
##  Central                 : 1   Mean   :15547   Mean   : 7266
##  Childwall               : 1   3rd Qu.:16487   3rd Qu.: 7607
##  Church                  : 1   Max.   :20340   Max.   :11925
##  (Other)                 :24
```

For each of the numeric columns, a number of values are listed:

Item	Description
Min.	The smallest value in the column
1st. Qu.	The first quartile (the value ¼ of the way along a sorted list of values)
Median	The median (the value ½ of the way along a sorted list of values)
Mean	The average of the column
3rd. Qu.	The third quartile (the value ¾ of the way along a sorted list of values)
Max.	The largest value in the column

For the “Ward” column which was a factor, this displays the frequency of occurrence. In this instance all the values are 1 because they are unique, i.e. you don’t have more than one ward in Liverpool with the same name.

Working with data frames

A data frame can be hundreds of rows long and we may not want to print all of the values on the terminal. We can use the `head()` function to print the top 6 rows.

```
head(Liverpool_Good_Health)
```

```
##                         Ward Tot_Pop Good_Health
## 1 Allerton and Hunts Cross   14853        7274
## 2                   Anfield   14510        6124
## 3                Belle Vale   15004        6129
## 4                   Central   20340       11925
## 5                 Childwall   13908        7219
## 6                    Church   13974        7461
```

We can also specify a specific row and/or column id using numeric values stored in square brackets. For example (note that there are some comments in this block of R code, which start with a # - these are not run by R, but are notes for your reference and can be useful when documenting your code).

```
# Row 1, Column 3
Liverpool_Good_Health[1,3]
# Row 2, all columns
Liverpool_Good_Health[2,]
```

Another way in which you can refer to columns in a data frame is by using their names. For example, we will get row 10, and the value in the “Good_Health” column.

```
Liverpool_Good_Health[10,"Good_Health"]
```

You can also get a range of rows, using a colon:

```
Liverpool_Good_Health[1:5,"Good_Health"]
```

This has listed the burglary rates in rows 1:5 for the “Good_Health” column. You can also use the `c` function illustrated earlier to select variables - these can even include ranges using the colon.

```
#Rows 1 through 4 and 7 through 9, Column 1 and 3

Liverpool_Good_Health[c(1:4,7:9),c(1,3)]
```

```
##                       Ward Good_Health
## 1 Allerton and Hunts Cross        7274
## 2                 Anfield        6124
## 3              Belle Vale        6129
## 4                 Central       11925
## 7               Clubmoor        6403
## 8                 County        5930
## 9             Cressington        7094
```

A further way in which you can extract a value from a data frame is by using the $ symbol.

```
Liverpool_Good_Health$Good_Health
```

This prints all the values stored in the column called `Good_Health`.

Importing external data

In the previous section we have worked with data we created within R, however, R has a very wide range of functions available to import data from external files. A very simple and commonly used format to store data in is a CSV file - these are text based, with each row separated by a carriage return and the columns by a comma. These files can be imported using the `read.csv()` function. We will use this to read a CSV file into a new data frame object called census.

```
census <- read.csv("census_data.csv")
head(census)
```

```
##        Code                     Ward People_16_74 Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        10930               1103
## 2 E05000887                  Anfield        10712                312
## 3 E05000888              Belle Vale        10987                432
## 4 E05000889                 Central        19174               1346
## 5 E05000890               Childwall        10410               1123
## 6 E05000891                  Church        10569               1843
##   Tot_Pop Good_Health
## 1   14853        7274
## 2   14510        6124
## 3   15004        6129
## 4   20340       11925
## 5   13908        7219
## 6   13974        7461
```

You will see that this CSV contains some of the data we created manually, but also two other census variables and a “Code” column which contains a unique ID for each Ward.

Adding New Columns to Data Frames

Earlier we illustrated how you can use R as a calculator to return new values. We can also calculate and store new values within a data frame object. In this example we will use the census data we just imported to calculate two new sets of percentages for the proportion of people in good health, and the proportion of people in higher managerial forms of occupation.

In addition to using the $ symbol as a way of referring to a column within a data frame, we can also use this to create a new column - which in this case will be called “PCT_Good_Health”. The values for this column will be created by dividing the columns “Good_Health” by “Tot_Pop” and multiplying by 100.

```
census$PCT_Good_Health <- census$Good_Health / census$Tot_Pop * 100
head(census)
```

```
##        Code                     Ward People_16_74 Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        10930               1103
## 2 E05000887                  Anfield        10712                312
## 3 E05000888              Belle Vale        10987                432
## 4 E05000889                 Central        19174               1346
## 5 E05000890               Childwall        10410               1123
## 6 E05000891                  Church        10569               1843
##   Tot_Pop Good_Health PCT_Good_Health
## 1   14853        7274        48.97327
## 2   14510        6124        42.20538
## 3   15004        6129        40.84911
## 4   20340       11925        58.62832
## 5   13908        7219        51.90538
## 6   13974        7461        53.39201
```

We will now calculate another new variable, however, this time using the column indexes rather than the $ symbol. This will be called “PCT_Higher_Managerial” and involve dividing “Higher_Managerial” by “People_16_74” (think about why the total population isn’t used…) and again, multiplying by 100. Numerically, this id dividing column 4 in the data frame by column 3.

```
census[,"PCT_Higher_Managerial"] <- census[,4] / census[,3] * 100
head(census)
```

```
##        Code                    Ward People_16_74 Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        10930              1103
## 2 E05000887                Anfield        10712               312
## 3 E05000888             Belle Vale        10987               432
## 4 E05000889                Central        19174              1346
## 5 E05000890              Childwall        10410              1123
## 6 E05000891                 Church        10569              1843
##   Tot_Pop Good_Health PCT_Good_Health PCT_Higher_Managerial
## 1   14853        7274        48.97327             10.091491
## 2   14510        6124        42.20538              2.912621
## 3   15004        6129        40.84911              3.931920
## 4   20340       11925        58.62832              7.019923
## 5   13908        7219        51.90538             10.787704
## 6   13974        7461        53.39201             17.437790
```

Subsetting Data Frames

For the rest of this practical we are only interested in the two numeric variables “PCT_Good_Health” and “PCT_Higher_Managerial” and we will need to take a subset of the dataframe. There are a number of ways in which we can do this using the square brackets with either numeric or column label references, or using the `subset` function. The following examples are wrapped in the `head()` so only the top six rows are printed. The output should all be the same, however, the method of subsetting are different.

```
head(census[,c("Code","Ward","PCT_Good_Health","PCT_Higher_Managerial")])
```

```
##        Code                    Ward PCT_Good_Health PCT_Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        48.97327             10.091491
## 2 E05000887                Anfield        42.20538              2.912621
## 3 E05000888             Belle Vale        40.84911              3.931920
## 4 E05000889                Central        58.62832              7.019923
## 5 E05000890              Childwall        51.90538             10.787704
## 6 E05000891                 Church        53.39201             17.437790
```

```
head(census[,c(1:2,7:8)])
```

```
##        Code                    Ward PCT_Good_Health PCT_Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        48.97327             10.091491
## 2 E05000887                Anfield        42.20538              2.912621
## 3 E05000888             Belle Vale        40.84911              3.931920
## 4 E05000889                Central        58.62832              7.019923
## 5 E05000890              Childwall        51.90538             10.787704
## 6 E05000891                 Church        53.39201             17.437790
```

```
head(subset(census, select = c("Code","Ward","PCT_Good_Health","PCT_Higher_Manager
ial")))
```

```
##        Code                    Ward PCT_Good_Health PCT_Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        48.97327             10.091491
## 2 E05000887                Anfield        42.20538              2.912621
## 3 E05000888             Belle Vale        40.84911              3.931920
## 4 E05000889                Central        58.62832              7.019923
## 5 E05000890              Childwall        51.90538             10.787704
## 6 E05000891                 Church        53.39201             17.437790
```

We can also make a new object using one of these methods - note that we now remove the `head()` function as we want the new object to contain all rows.

```
census_small <- census[,c(1:2,7:8)]
```

We can also use logic to subset data frames. For example, supposing we wanted to find those wards in Liverpool where there were greater than 10% of those employed in higher managerial occupations. We can achieve this using a logic statement. We can break this process down so it is more obvious what is happening. First we will look at the values of the “PCT_Higher_Managerial” column and see how many are above 10%.

```
census_small$PCT_Higher_Managerial > 10
```

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

This has printed a list of TRUE and FALSE which correspond to the numbers (and their order) in the “PCT_Higher_Managerial” column. Another useful function in R is `table()` which can turn a list into contingency table. As such, if we wrap the previous statement with `table` this will generate a summary of TRUE and FALSE values.

```
table(census_small$PCT_Higher_Managerial > 10)
```

```
##
## FALSE  TRUE
##    22     8
```

You can also use a list of TRUE and FALSE to subset a row in a data frame. As such, we can use this with the square brackets, so rather than selecting a row by a numeric value, we use the TRUE and FALSE list.

```
census_small[census_small$PCT_Higher_Managerial > 10,]
```

```
##         Code                     Ward PCT_Good_Health
## 1  E05000886 Allerton and Hunts Cross        48.97327
## 5  E05000890                Childwall        51.90538
## 6  E05000891                   Church        53.39201
## 9  E05000894              Cressington        48.91402
## 17 E05000902             Mossley Hill        52.99653
## 22 E05000907                Riverside        48.86006
## 23 E05000908             St Michael's        49.64976
## 29 E05000914                  Woolton        46.62952
##    PCT_Higher_Managerial
## 1               10.09149
## 5               10.78770
## 6               17.43779
## 9               11.47240
## 17              14.36355
## 22              10.70365
## 23              14.71319
## 29              15.39853
```

Or, you might just be interested in the Ward names.

```
census_small[census_small$PCT_Higher_Managerial > 10,"Ward"]
```

```
## [1] Allerton and Hunts Cross Childwall
## [3] Church                   Cressington
## [5] Mossley Hill             Riverside
## [7] St Michael's             Woolton
## 30 Levels: Allerton and Hunts Cross Anfield Belle Vale ... Yew Tree
```

Recoding Values in Data Frames

Recoding is a way of creating a new value using the attributes of an old value. For example, we might be interested in creating a new variable in our “census_small” data frame which shows those wards with the proportion of people in good health at a rate less than the mean. The mean can be calculated as follows

```
mean(census_small$PCT_Good_Health)
```

```
## [1] 46.67722
```

We can then use this to test each of numbers contained in the “PCT_Good_Health” column.

```
census_small$PCT_Good_Health < mean(census_small$PCT_Good_Health)
```

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

Which returns some TRUE and some FALSE values. We can then combine this with an `ifelse()` function to create a new variable called “target”. The `ifelse()` returns (rather than TRUE and FALSE) a value specified by the latter two parameters of the function. In this case, these are the strings “Yes” and “No”.

```
census_small$target <- ifelse(census_small$PCT_Good_Health < mean(census_small$PCT
_Good_Health),"Yes","No")
```

You will now see that these values have been added as a new variable in the data frame object “census_small”.

```
head(census_small)
```

```
##        Code                     Ward PCT_Good_Health PCT_Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        48.97327              10.091491
## 2 E05000887                 Anfield        42.20538               2.912621
## 3 E05000888              Belle Vale        40.84911               3.931920
## 4 E05000889                 Central        58.62832               7.019923
## 5 E05000890               Childwall        51.90538              10.787704
## 6 E05000891                  Church        53.39201              17.437790
##   target
## 1     No
## 2    Yes
## 3    Yes
## 4     No
## 5     No
## 6     No
```

Joining Data Frames

A common operation in R is joining two data frames using a common ID. For example, we may have an additional table of census data for Liverpool, and wish to join these attributes onto “census_small”. First we will import another CSV - “census_data2.csv”, calculate the percentage of socially rented households and then cut the table down to this newly created variable and an ID.

```
census2 <- read.csv("census_data2.csv")
census2$PCT_Social_Rented_Households <- census2$Social_Rented_Households / census2
$Households * 100
census2 <- census2[,c("GEO_CODE","PCT_Social_Rented_Households")]
```

This leaves us with a table as follows

```
head(census2)
```

```
##    GEO_CODE PCT_Social_Rented_Households
## 1 E05000886                     13.005189
## 2 E05000887                     22.772576
## 3 E05000888                     42.555119
## 4 E05000889                     18.363917
## 5 E05000890                      6.937488
## 6 E05000891                      3.025153
```

The variable GEO_CODE is the ID variable and can be used to join onto the matching variable in the object "census_small", however, this has used a different name. We can see the column names used in a data frame using the `colnames()` function.

```
colnames(census_small)
```

```
## [1] "Code"                  "Ward"                  "PCT_Good_Health"
## [4] "PCT_Higher_Managerial" "target"
```

The ID column is called "Code", and if you look at the data frame you will see codes in a similar format.

```
head(census_small)
```

```
##        Code                    Ward PCT_Good_Health PCT_Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        48.97327              10.091491
## 2 E05000887                 Anfield        42.20538               2.912621
## 3 E05000888              Belle Vale        40.84911               3.931920
## 4 E05000889                 Central        58.62832               7.019923
## 5 E05000890               Childwall        51.90538              10.787704
## 6 E05000891                  Church        53.39201              17.437790
##   target
## 1     No
## 2    Yes
## 3    Yes
## 4     No
## 5     No
## 6     No
```

We can now join the two objects using the `merge()` function. We will refer to the two data frames as `x` and `y`. The x data frame is "census_small""; and the y is"census2". In `x`, the column containing the ID is called"Code", and in `y`, it is"GEO_CODE". The parameters of the merge function first accept the two table names, and then the lookup columns as `by.x` or `by.y`. You should also include `all.x=TRUE` as a final parameter. This tells the function to keep all the records in `x`, but only those in `y` that match.

In this example we replace the census_small with the newly merged data frame.

```
census_small <- merge(census_small,census2,by.x="Code",by.y="GEO_CODE",all.x=TRUE)
```

If you view the new object, this should now have the matched values.

```
head(census_small)
```

```
##        Code                       Ward PCT_Good_Health PCT_Higher_Managerial
## 1 E05000886 Allerton and Hunts Cross        48.97327             10.091491
## 2 E05000887                   Anfield        42.20538              2.912621
## 3 E05000888                Belle Vale        40.84911              3.931920
## 4 E05000889                   Central        58.62832              7.019923
## 5 E05000890                 Childwall        51.90538             10.787704
## 6 E05000891                    Church        53.39201             17.437790
##   target PCT_Social_Rented_Households
## 1     No                    13.005189
## 2    Yes                    22.772576
## 3    Yes                    42.555119
## 4     No                    18.363917
## 5     No                     6.937488
## 6     No                     3.025153
```

Saving your analysis

You can save the "R environment" which contains all the objects that you have created using the `save.image()` function

```
save.image("Practical_1.RData")
```

This creates the file "Practical_1.RData" in your working directory. You can load this at a later stage using the `load()` function.

```
load("Practical_1.RData")
```

Another way in which we can save tabular data is to export a CSV file. This works in a similar way to importing. In this example we will export the "census_small" data frame object. Keep this safe as we will use this in the next practical. The first input in the function is the object name, and then the name of the CSV we want to create.

```
write.csv(census_small,"census_small.csv")
```

Some final tips

R store the objects that you create in memory; however, you may forget what objects you have created. To list the objects currently in memory use the `ls()` function.

```
ls()
```

```
##  [1] "all_people"           "census"
##  [3] "census_small"         "census2"
##  [5] "good_health"          "Liverpool_Good_Health"
##  [7] "milk"                 "product"
##  [9] "sale_product"         "wards"
```

You might also want to remove an object from memory. This can be achieved with the `rm()` function - in this example we remove the wards object.

```
rm(wards)
```

Finally, if you want some details about a particular function, you can use the `?` symbol followed by the function name. This will create a help page. For example, to find out more details about he `ls()` function type.

```
?ls()
```