Worksheet 1 Getting Started

This worksheet is a quick start guide for using the open source statistical software, **R** which is available to download from https://www.r-project.org

On this course we will use **R** from within another program, called **RStudio** available to download from <https://www.rstudio.com>

**RStudio** provides a user friendly environment for **R** that simplifies many data analysis tasks with **R**.

This worksheet is a quick start guide but I would recommend further reading. The library in CIT has the following two ebooks that you can download:

The R Book by M.J. Crawley and Art of R Programming by Norman Matloff

The R Book has an emphasis on statistics and the Art of R Programming on coding with **R**.

Some of the best resources are online:

R is for Data Science <http://r4ds.had.co.nz/>

Stack Overflow http://stackoverflow.com/tags/r/info

Cookbook for R <http://www.cookbook-r.com>

RStudio Tutorials https://www.rstudio.com/online-learning/

R-Bloggers.com http://www.r-bloggers.com

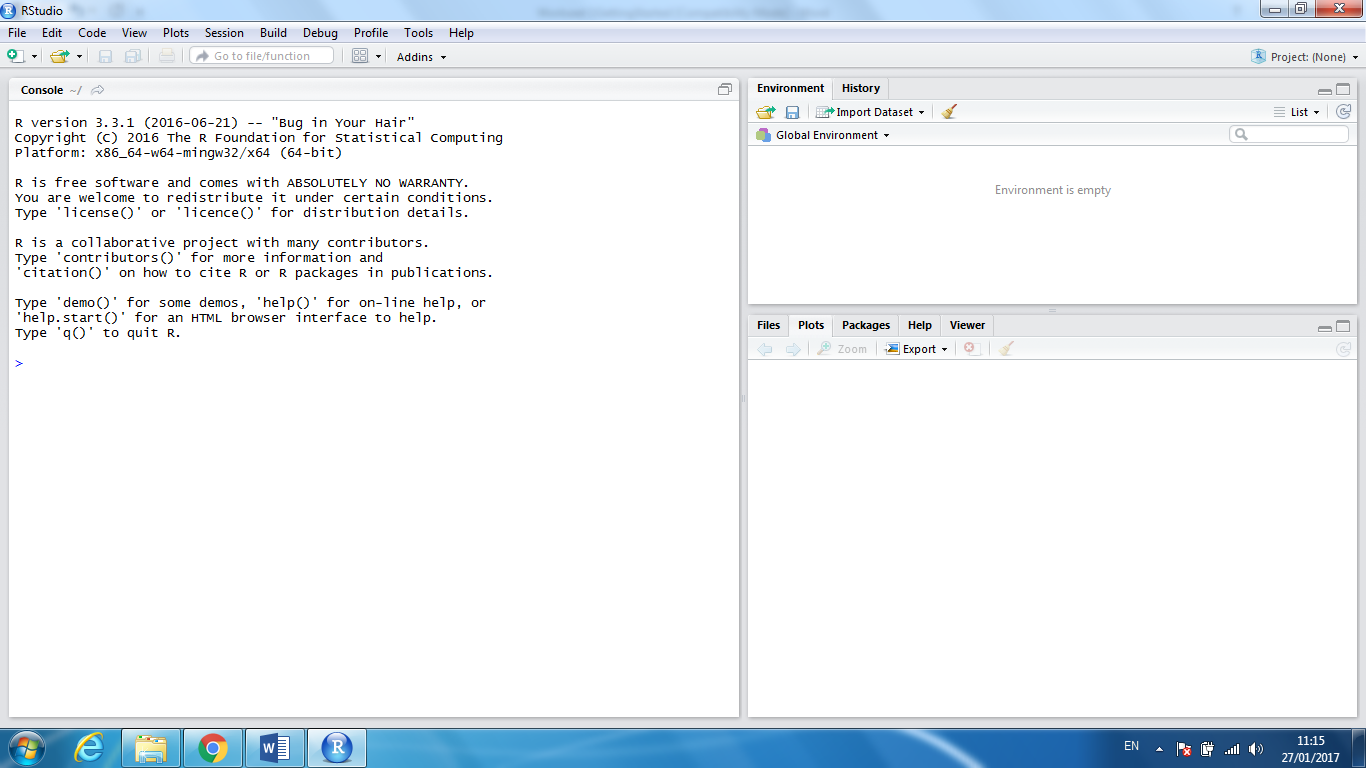
In these worksheets, material from R is shown in Courier New font as shown below:

5+6

[1] 11

**The R console window**

When you start **RStudio** you will see the following screen.



Commands are typed into the console window after the command prompt >

The Environment tab allows you to see what variables **R** has in its working memory.

The History tab allows you to see previously run commands.

**Organising your Work**

With **RStudio** you can organise your work in projects using the menu:

File > New Project

Choose a directory name (Getting Started/Worksheet1)

Choose a location for the directory

**R**’s working directory is the directory on your computer where **R** will look for data files and save files.

To find out which directory RStudio is using you can use the getwd()command.

The working directory can be changed using the setwd() command.

If you are using **RStudio** then you can change your directory using the menu

Session > Set Working Directory > Choose Directory

When a project is opened within **RStudio** the following actions are taken:

* A new **R** session (process) is started
* The .RData file in the project's main directory is loaded (if project options indicate that it should be loaded).
* The .Rhistory file in the project's main directory is loaded into the RStudio History pane (and used for Console Up/Down arrow command history).
* The current working directory is set to the project directory.
* Previously edited scripts are restored into editor tabs
* Other RStudio settings (e.g. active tabs, splitter positions, etc.) are restored to where they were the last time the project was closed.

When you close your project **RStudio** will ask whether you want to save the workspace.

If you are working on a large project it is useful to create sub-folders within the project e.g. a folder for data, a folder for plots, a folder for scripts…

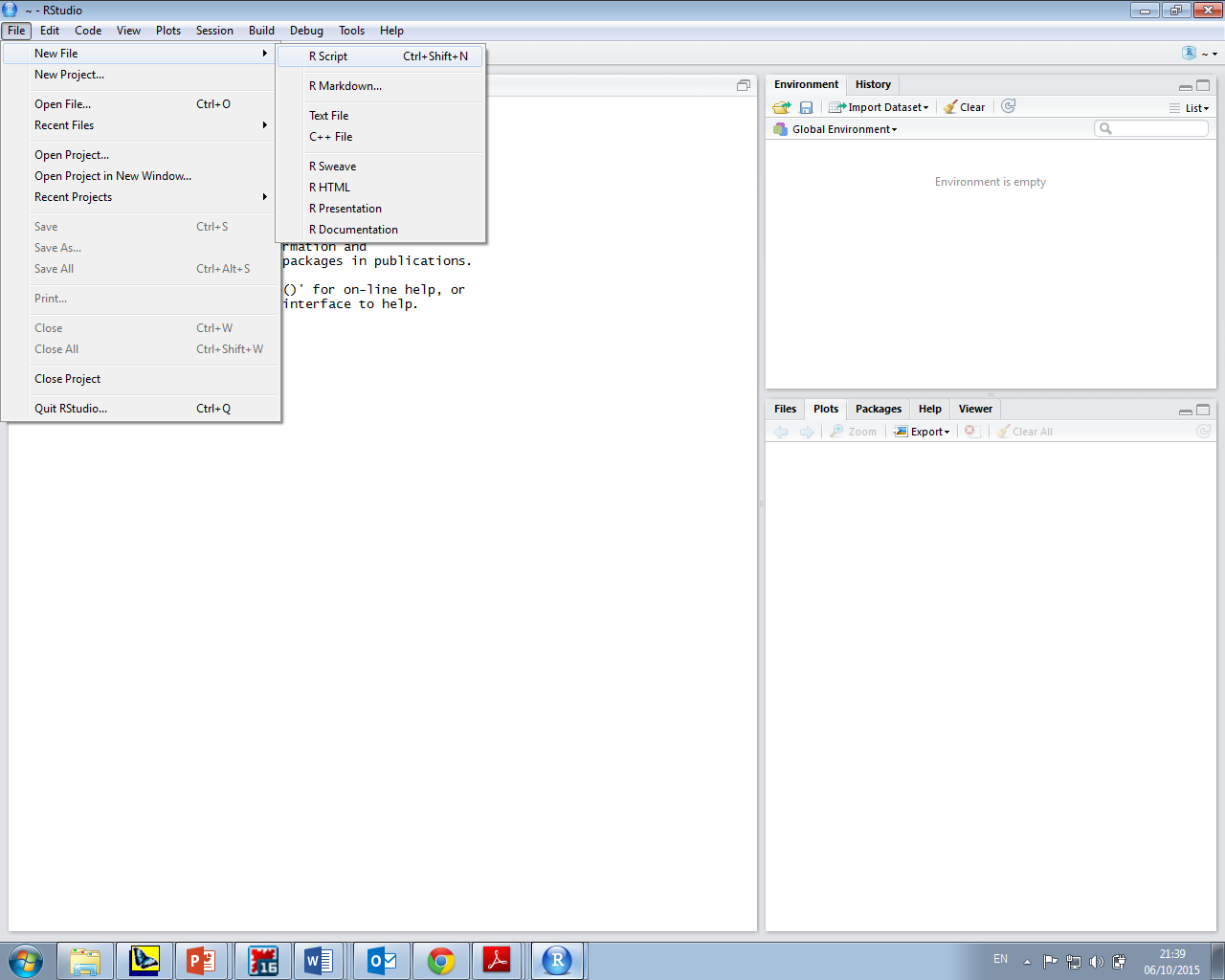
**Scripts**

The best way to keep track of your work is to write an **R** script for each piece of work.

An **R** script is a text file that contains a set of **R** commands and short comments that briefly explain the commands. The commands in an **R** script can be sent to **R**’s command prompt to be executed all at once.

**R** scripts are useful because they record you data analysis, making it easy to repeat the analysis and share with others.

File > New File > R Script



It is very important to annotate your scripts because it keeps a record of the analysis makes it easier for you to return to your work.

Annotations are preceded by the # symbol, which tells **R** to ignore text that follows.

Below is an example of an R script:

###### Project title goes here ######

# Description of what the code does here including

# a description of the data and possibly the date.

# Commands are often preceded by an explanation e.g.

# create a data frame containing the male students only

male<-subset(mytable,Gender=='Male')

**Installing add-on packages**

There are a huge number of add-on packages that can added to **R** for specific analysis. These packages must be installed and loaded before they can be used in **R**.

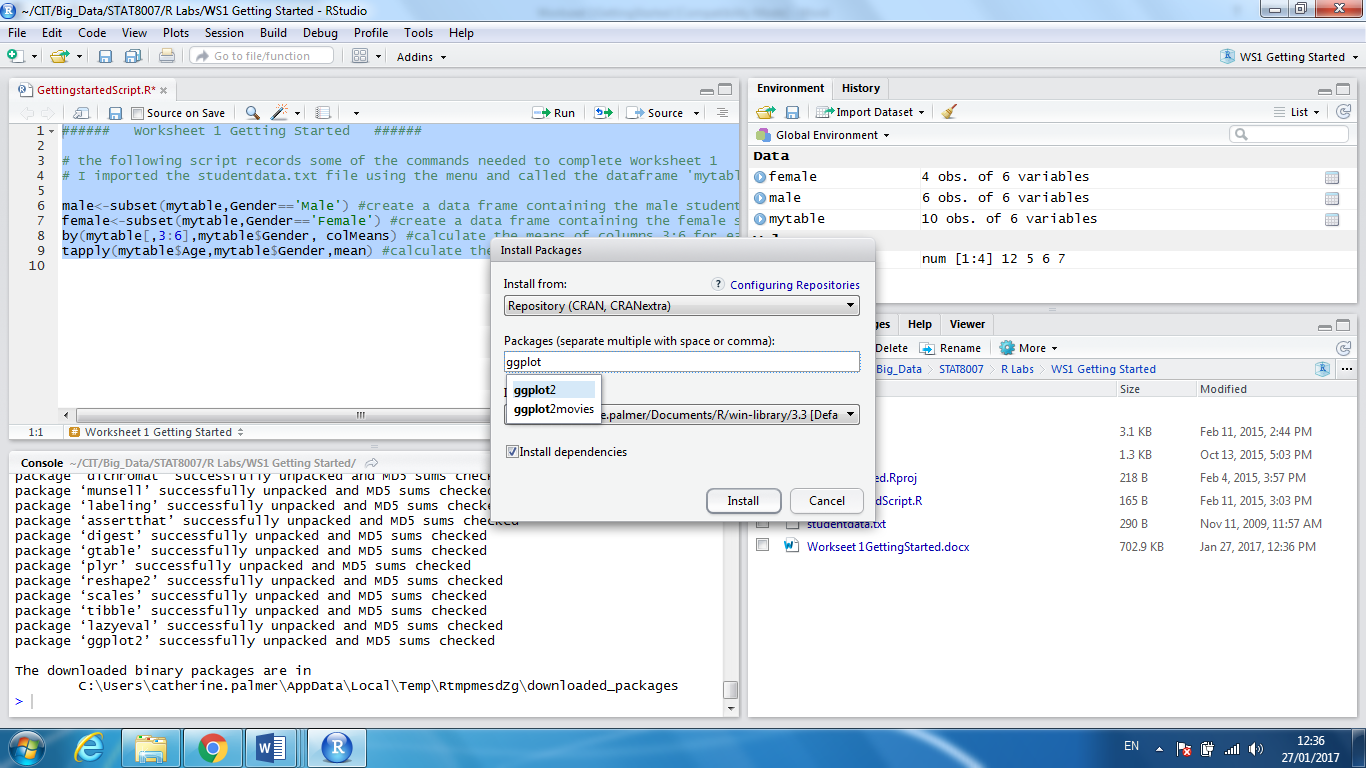
A list of packages is available on the **R** website: https://cran.r-project.org/web/packages/

We will install the package ggplot2

install.packages("ggplot2")

Alternatively we can use the menu:

Tools > Install Packages



Once a package has been installed on your computer it must be loaded each time you want to use it, using the library() command.

library(“ggplot2”)

**The Tidyverse**

The tidyverse consists of a number of key packages that can be used together to create code that is consistent and easy to read.



**Basics**

**The Up arrow key will retrieve your last command (this can save a lot of time).**

**Help**

If you know the name of the function you want help with, you just type a question mark at the command line prompt followed by the name of the function. For example to get help on the function **read.table**, just type:

?read.table

**Assignment**

Objects obtain values in **R** by assignment (`*x* gets a value'). This is achieved by the ‘gets’ arrow **<-** which is a composite symbol made up from `less than' and `minus' with no space between them. Thus to create a scalar constant *y* with value 5 we type:

y <-5

**Calling Functions**

R has a large collection of built-in functions that are called like this:

function\_name(arg1 = val1, arg2 = val2, ...)

Let’s try using seq() which makes regular sequences of numbers and, while we’re at it, learn more helpful features of RStudio.

Type:

se

and hit TAB. A popup shows you possible completions. You can complete the command seq() by finishing typing q or by using ↑/↓ arrows to select. Notice the floating tooltip that pops up, reminding you of the function’s arguments and purpose. If you want more help, press F1 to get all the details in the help tab in the lower right pane.

Press TAB once more when you’ve selected the function you want. RStudio will add matching opening (() and closing ()) parentheses for you. Type the arguments 1, 10 and hit return.

seq(1, 10)

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

Type the code below and notice you get similar assistance with the paired quotation marks:

x <- "hello world"

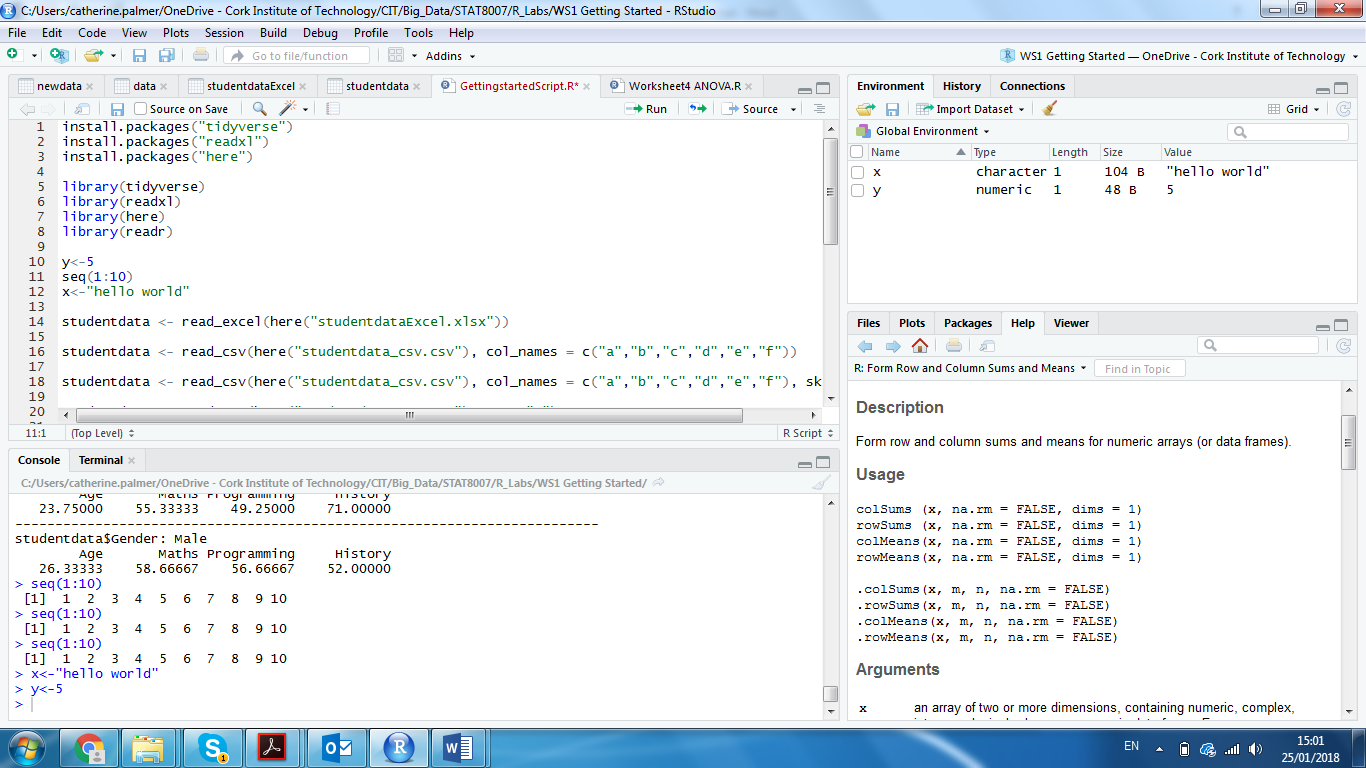
Quotation marks and parentheses must always come in a pair. RStudio does its best to help you, but it’s still possible to mess up and end up with a mismatch. If this happens, **R** will show you the continuation character “+”:

x <- "hello

+

The + tells you that **R** is waiting for more input; it doesn’t think you’re done yet. Usually that means you’ve forgotten either a " or a ). Either add the missing pair, or press ESCAPE to abort the expression and try again.

Now look at your environment in the upper right pane:



Here you can see all of the objects that you’ve created.

**Arithmetic Operations and Mathematical Functions**

**R** has many built in functions and you can use **R** much the same way as you would use a calculator.

At the prompt type, 5+6 then press enter:

5+6

[1] 11

R returns the value 11.

log(5)

[1] 1.609438

Note that in **R** the command log takes the natural log of a number (usually denoted ln) which has base *e*. Log to the base 10 is written as log10

log10(10)

[1] 1

**Types of variable**

Four important classes of variables are:

Real numbers (num)

Whole numbers (int)

Text (chr)

Logicals (logi)

To find out the class of a variable in R try:

x <- 1 #assign the value 1 to the variable x

class(x)

[1] "numeric"

Numbers in **R** are generally treated as numeric objects (i.e. double precision real numbers). If you explicitly want an integer, you need to specify the L suffix

1 #gives you a numeric object

1L #explicitly gives you an integer

There is a special number Inf which represents infinity;

The value NaN represents an undefined value (“not a number”); e.g. 0 / 0

NaN can also be thought of as a missing value (more on that later)

To assign text to a variable, use quotation symbols:

x <- “abc” #assign the characters abc to the variable x

class(x)

[1] "character"

A logical variable has the value TRUE or FALSE

x<-TRUE

class(x)

[1] "logical"

**Vectors**

The fundamental data type in R is the vector. Elements in a vector must all be of the same class i.e. all numeric, or all characters, or all logical etc.

You can type the values of a vector into the command line, using the concatenation function c

y<-c(10, 11, 12, 13, 14, 15, 16) #creates a numeric vector

x<-c(“abc”,”2.4”) #creates a character vector

Note that the quotation marks around the number 2.4 means that R assigns it as a character rather than a number.

**R** can evaluate functions over entire vectors.

max(y)

[1] 16

min(y)

[1] 10

mean(y)

[1] 13

median(y)

[1] 13

range(y)

[1] 10 16

var(y)

[1] 4.666667

sqrt(var(y))

[1] 2.160247

quantile(y)

0% 25% 50% 75% 100%

10.0 11.5 13.0 14.5 16.0

rank(y)

[1] 1 2 3 4 5 6 7

length(y)

[1] 7

Indexing into a vector

Indices select part of a vector, in **R** we use square brackets [].

y<-c(10, 11, 12, 13, 14, 15, 16)

The first element of y is 10. If we wish to select this we type:

y[1]

[1] 10

If we wish to select a range of elements we type:

y[1:3] # this extracts the first three elements of y

[1] 10 11 12

If we want to extract several values (say the 2nd, 3rd and 6th) we use a vector

Index<- c(2,3,6)

y[Index]

[1] 11 12 15

**Matrices**

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

# create a matrix with 2 rows and 3 columns

M <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3, byrow = TRUE)

M

[,1] [,2] [,3]

[1,] 1 2 3

[2,] 4 5 6

We can access the (1, 2) or the (2, 1) element of this matrix using the appropriate indices.

M[1, 2]

[1] 2

M[2, 1]

[1] 4

Indices can also be missing. To access entire rows or columns of a matrix, leave blank:

x[1, ] # Extract the first row

[1] 1 2 3

x[, 2] # Extract the second column

[1] 2 5

**For Vectors (and therefore Matrices), Arithmetic and Logical Operations are applied element wise.**

This is important to remember especially when performing matrix multiplication.

# here is an example of adding two vectors

x<-c(1,2,3)

y<-(4,5,6)

x+y

[1] 5 7 9

# here is an example of multiplying two vectors

x\*y

[1] 4 10 18

If we wish to multiply the vectors x and y using the rules of matrix multiplication we must use the command %\*%

x%\*%y

[,1]

[1,] 32

**Higher – Dimensional Arrays**

While matrices are two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the example below we create an array with two elements which are 2x3 matrices.

M1<-matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3, byrow = TRUE)

M2<-matrix(c("a","b","c","d","e","f"),nrow = 2,ncol=3,byrow=TRUE)

A <- array(c(M1,M2),dim = c(2,3,2))

The dim attribute refers to 2 rows, 3 columns and two # layers (each layer contains a 2x3 matrix).

A

, , 1

[,1] [,2] [,3]

[1,] "1" "2" "3"

[2,] "4" "5" "6"

, , 2

[,1] [,2] [,3]

[1,] "a" "b" "c"

[2,] "d" "e" "f"

**Lists**

A vector can only contain objects of the same class but a list can contain objects of

different classes.

x <- list(1, "a", TRUE, 1 + 4i)

**Factors**

Factors are categorical variables that frequently used in statistics. A classic example of a factor is the gender variable, with two levels Male or Female. Factors are created using the factor() function. The nlevels() function gives the number of levels. **R** stores the vector along with the distinct values of the elements in the vector as labels. The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector.

Gender <- c(“M”, “M”, “M”, “F”)

Gender <- factor(Gender)

Gender

[1] M M M F

Levels: F M

Relational operators

**Symbol** **Definition**

== equals to (note the double equals sign)

!= not equals to

> greater than

>= greater than or equal to

< less than

<= less than or equal to

Logical operators

**Symbol Definition**

! logical NOT

& logical AND

| logical OR

Example:

1 == 2 # Is 1 equal to 2?

[1] FALSE # **R** tells us that the statement 1 == 2 is false.