**Lab**

**Introduction to R**

Lab objectives:

* To become familiar with the basic constructs in R.

In this lab you will create various objects using R, which is an open source tool for data visualisation and analysis. It contains a vast (and increasing!) library of specialised packages for data manipulation, analysis and visualisation.

R can be downloaded from

<http://www.r-project.org/>

In the labs, we will be using RStudio, which is a user interface for R and can be downloaded from

<https://posit.co/>

If you are installing R and RStudio in your own device, you must install R before you install RStudio.

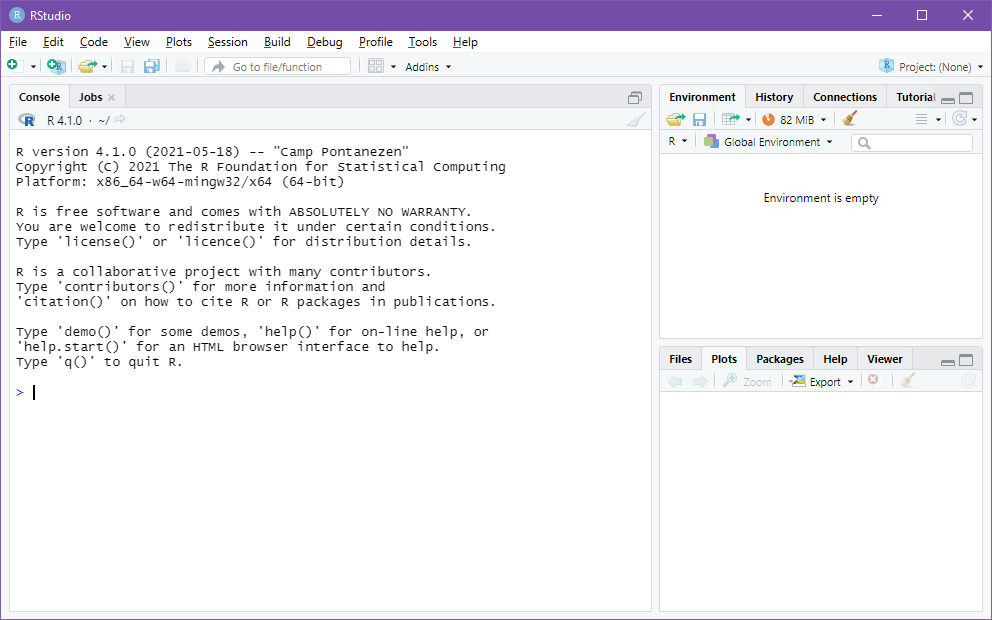
You should use a school machine during lab sessions as it is important that you familiarise yourself with their settings. You will be using a school machine during the CMM020 Practical Examination.

**Notes**

1. R is **case sensitive**!
2. If you cut and **paste something containing quotes**, you may need to **retype the quotes** as these are not always in the correct format. In any case, cut and paste is discouraged for this lab, as typing the command will help you learn the functions and format and you will develop the ability to spot errors.
3. The command “cursor” is >. If instead there is a + it means the previous command is incomplete, usually because one or more quotes and/or a bracket has not been closed.

**Getting started**

In the machine where RStudio is installed, go to the *start*  icon and search for RStudio. When you select it, the following window will appear. Resize the window if it is too small.



There are 3 main parts:

* On the left, you will find the console, which is where you can type your commands. In the future, we will see that the left pane can be split into two, if we use an editor for our code. We will be doing this from next week.
* On the top right, you will find your environment variables as well as the history and a tutorial facility.
* On the bottom right, there are several tabs that you can use. You will be mainly using this for viewing output and also for installing packages (libraries).

**Variables**

R uses variables to store its data. For example, to store value 7 in variable x we can type the following command on the console panel (bottom left)

x <- 7

“<-“ is the assign operator. To view a variable’s content, simply type its name, e.g.

x

Will return

[1] 7

We can create more complex structures to hold our data. The function *c()* can be used for this, as it concatenates given data into a vector. For example, to define an object called *energy\_sources* with values *oil, gas, wood, sun, wind, wave, hydroelectric* and *nuclear* type

energy\_source <- c("oil","gas","wood","sun","wind","wave","hydroelectric", "nuclear")

The object *energy\_source* now contains the desired values. To display its contents type

energy\_source

It will return

[1] "oil" "gas" "wood" "sun"

[5] "wind" "wave" "hydroelectric" "nuclear"

To display the 3rd item on the list, type

energy\_source[3]

The system will display

[1] "wood"

To display items 4-6 type

energy\_source[4:6]

The system will display

[1] "sun" "wind" "wave"

Alternatively, you can look at the values in the Environment Variables panel (top right).

You can make new vectors from existing. For example

traditional\_source <- c(energy\_source[1:3],energy\_source[5], energy\_source[7])

Will give traditional\_source value

"oil" "gas" "wood" "wind" "hydroelectric"

You may obtain the length of a vector using the *length()* command. For example,

length(traditional\_source)

will return 5.

**Helpful commands**

To get help regarding a command, you can precede the command with ?. For example, to know more about what the concatenation function c() does type

?c

Help information will appear on the bottom right panel.

If you make a mistake when typing, or want to reuse a previous command, you can use the arrow-up (and arrow-down) key to go through previous commands. Once a command is up, you can modify it.

**Arithmetic operations**

Simple arithmetic operations (+,-,\*,/,^) can be performed easily in R. For example type

x <- 4

y <- x + 20

z <- y/x

what value does z have?

When given two vectors (a list of values) of the same length, R simply performs the given arithmetic element-by-element. If the vectors are not of equal length, R 'recycles' (repeats) the shorter vector until it is the same length as the longer vector. If the length of the longer vector is not a multiple of the shorter vector, it will warn you about it, but it will still perform the operation.

For example,

|  |
| --- |
| numbers <- c(1,2,3,4,5,6,7,8,9,10)  To multiply each element of number by 10 you can use  numbers \* 10  It returns  [1] 10 20 30 40 50 60 70 80 90 100 |
|  |
| |  | | --- | |  | |

The result is a vector of length 10 where each element in numbers has been multiplied by 10.

Another example

numbers \* c(10,20,30)

Returns

[1] 10 40 90 40 100 180 70 160 270 100

As the length of the longer vector is not a multiple of the shorter vector, it warns you about it, but it will still perform the operation. It is equivalent to

Numbers \* c(10,20,30,10,20,30,10,20,30,10)

**Logical operators**

Logical operators include `<`,`>`, `<=`, `>=` `==` (equals) and `!=` (not equals). For example, to see which numbers are greater than 7

|  |
| --- |
| numbers > 7 |
|  |
| |  | | --- | |  | |

It returns

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE

Logical expressions can be combined using `&` (and), `|` (or) or ! (not). For example,

(numbers > 7) | (numbers< 2)

Returns

[1] TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE

**Libraries**

In this course we will use a number of libraries which provide functions to help us code our visualisations and analysis. Libraries are loaded using the library function. In this lab we will use the tidyverse library. You can load it with the following code.

library(tidyverse)

**Tibbles**

We will use “tables” of data which are called tibbles or data frames. Thus, the data is arranged in columns with each column having a different name. For example, we may want to have a table with numbers 1 to 6 in one column, and their power (i.e. the number multiplied by itself) in another column.

my\_nums powers

1 1

2 4

3 9

4 16

5 25

6 36

We can obtain this by first creating each column using a vector (we store this in variables called my\_nums and powers) and then putting them together in a tibble which we store in a variable called powerData.

my\_nums <-(1:6)

Here, 1:6 is a shorthand for the sequence of numbers from 1 to 6.

powers <- my\_nums^2

Here, ^ is the symbol used for power.

Below we put both sets of numbers together in a tibble.

powerData <- tibble(my\_nums,powers)

We can inspect the table by typing the name of the variable it is held in (powerData)

powerData

**Plots**

In this module we will be using a library (ggplot2) to plot data. This allows the data plot to be constructed in pieces, which each piece being used to define a specific aspect of the plot.

To build a plot, we need to call function ggplot, and tell it which dataset we want to use. In this example we will use the powerData dataset that we have just created.

ggplot(data=powerData)

The above code will return an empty box, because we have not defined the x and y axis, or what type of plot we want to use. Let’s try again, this time specifying that we want to use points to display the data, and specifying that we want column my\_nums on the x axis and column powers on the y axis.

ggplot(data=powerData) + geom\_point(aes(x=my\_nums, y=powers))

The above code works well. However, as plots can become complicated, we will divide the code into its two components, and store it in a variable (normally we will call it p) until we are ready to plot. Thus the above code could be run as follows:

p <- ggplot(data=powerData)

p <- p + geom\_point(aes(x=my\_nums, y=powers))

p

The last bit of code (a single p in a line) is the call to excute the code for the plot (i.e., all the code stored in p).

**Exercise 1:** In the lecture, you saw the Anscombe quartet. For each of the data sets, enter it into 2 vectors and plot it. Make sure you use different variable names for each of your datasets as we will use these later. For example, you may use set1\_x and set1\_y to represent the X and Y values for the first dataset.

**Labels and titles**

You can add a titles and labels using function labs(). For example, we can add to the code above (which we conveniently stored in variable p, a title as follows:

p <- p + labs(title= "Numbers against their powers")

p

Similarly, we can add the x and y labels. For example,

p <- p + labs(x = "Number", y= "Power of number")

p

Note that we could have added title and labels in the same line of code by using

p <- p + labs(title= "Numbers against their powers",

x = "Number", y= "Power of number")

p

You can even include a subtitle such as “Simple scatterplot”.

p <- p + labs(subtitle = "Simple scatterplot")

**Exercise 2:** You previously plotted the datasets in the Anscombe’s quartet. Repeat this exercise but this time give the title “Anscombe’s quartet” and the subtitle “Dataset x” where x is 1,2,3 or 4 depending on the dataset. The x label should read “x values” and the y label should read “y values” (this is not very informative but it gives you practice on labelling axis!).

**Colour in plots**

Our plots are a little boring. There’s no colour. Type

colours()

To see the colours available. Choose one of those colours, e.g. *violetred1*. You can now change the colour of your plot with *colour in the “geom”* function (in this case within the geom\_point. For example, check the second line of the code below, where we change the colour of the dot to red.

p <- ggplot(data=powerData)

p <- p + geom\_point(aes(x=my\_nums, y=powers), colour="red")

p <- p + labs(title= "Numbers against their powers")

p <- p + labs( x = "Number", y = "Power of number")

p <- p + labs(subtitle = "Simple scatterplot")

p

**Exercise 3:** plot each of the Anscombe’s datasets, each using a different colour.