Introduction To Statistics For Data Science **Workshop 1**
Dr. Ric Crossman

This worksheet is designed to get you used to using `R` and `RStudio`. There will be additional, similar worksheets in later workshops.

We're going to be asking you to do things that you (very likely) don't yet know how to do with `R`. Learning a new programming language can be tricky, particularly for any of you for whom this is your **first** programming language. You have to get used to not only thinking carefully about not just what you want your computer to do, but the language you need to use in order to ask the computer to do it.

For some of you, this is familiar territory. For others, it might be quite new. Don't forget that there are multiple staff members in the room with you, ready to help you start making sense of `R`. Just as no-one would expect you to speak a new language fluently after your first lesson, we're not expecting you to master `R` in this first workshop. Don't be afraid to try things, and don't be discouraged when (at least) some of them don't work out.

Part I: The Basics Of `R`

1. Step 1 is to load `RStudio`. How to download and install this program was covered in Video 1.1. If you are using a machine borrowed from Durham University, remember you need to load up `AppsAnywhere`, and select RStudio. The most recent version of `RStudio` currently available on `AppsAnywhere` is `RStudio 2023 with RStan and Rtools`. Click on "Launch Laptop" to start RStudio.

   We will start off using the `console` window - this should be in the bottom-left of your screen when running `RStudio`. This console window is the only thing you see at first when loading up `R` on its own.

   We can use the console window to directly input commands into R, writing our commands beside the blue "greater than" sign, and hitting Enter each time we want to give `R` a command. I will refer to the "greater than" sign as the **command prompt** from this point on.

2. Provided you are online, there are several ways you can get assistance in `R` if you are stuck. Typing `help.start()` into the command prompt and pressing Enter will bring up a series of manuals for `R` that you can look at. I recommend reading through "An Introduction To `R`".

   You might not find that **too** helpful, of course, if you have a specific question. With so much information on offer, finding what you actually need can be quite daunting. If you want help with, say, a specific function, you can request the help file for that function. To do this, type a question mark followed by the name of the function into the command prompt, and press Enter. For instance, if you're not sure how to use the `sqrt` function, type `?sqrt` into the command prompt, and press Enter. Try it now. (You can achieve the same effect by typing in `help(sqrt)`.)

Even the help files can sometimes not be so helpful; it takes time to learn how to read them. Often though they have an example at the bottom which you can play around with without reading the rest of the file.

3. R is capable of performing basic arithmetic, using the operations `+,-,/,*`. Try using each of these to calculate a value. Do they work the way you expect? You can use `^` to calculate powers, as well.

4. One of the most fundamental tasks in `R` is to assign a **value** to a **variable**, so you can use that value in functions, graphs, etc.

   Let's assign the value 3 to a variable named `Steve`. To do this, you can enter `Steve<-3` or `3->Steve`. Alternatively, you can enter `Steve=3`. I recommend though that you **don't** assign values that way. In `R`, the commands "=" and "==" have very different meanings, and it's easy to accidentally use the wrong one, and cause yourself problems.

   Try performing arithmetic using Steve. What values would you expect to get from typing in:

   (a) `Steve+3`?

   (b) `1/Steve`?

   (c) `Steve^3-Steve*2`?

   Did you get the values you expected?

5. Use the `ls` *function* to obtain a *list* of the variables defined in your **workspace**. What `R` shows you in response will depend on how much you've been using `R` already, but you should at least see `Steve` listed, since you have defined this as a variable. If you want to be rid of Steve, type `rm(Steve)` and press Enter. Try this now, and use the `ls` function to confirm Steve is gone.

   Remember you can use `help(ls)` to find out how this command works.

6. Using the command prompt has its disadvantages. If you accidentally hit Enter partway through writing a command, you can spoil what you were trying to get `R` to do. If you are entering complex code line by line, and make a mistake, you might have to start from the very beginning. There are ways to mitigate this - you can use the up and

down keys to move between previous inputs, and make changes to an input - but it can be a real pain to correct mistakes even so.

One way to solve this problem is to use the `R Script` window, at the top left of the `RStudio` display. This window lets you write as many lines of code as you like at once, with Enter being used not to input commands, but to provide a break that will be **interpreted** as pressing Enter later on.

To input code from the `R Script` window to the `console` window, highlight the code and press `Run`, at the top-right of the `R Script` window. Everything you highlighted will be entered into the console, with line breaks interpreted as instances of pressing Enter.

Load up the `D100000000` code from the Week 1 folder in the Ultra module page. Copy it into the `R Script` window. Highlight it and press `Run`. This will feed the whole of the code into `R` in one go, so you can start using the function immediately. Now, it's true that you could also just paste the code directly into the console with the same effect, but if there's just one error in the code, you would have to fix it, then paste the whole thing into the console again. It is much easier to find a mistake in the `R Script` window, fix it there, and then use `Run` to plug the new code in to `R`.

7. Sometimes we might want to write a comment in our code, to remind us what a given line or section of code is doing (important hint: **never** assume you'll remember why you did something the way you did after a few weeks, or even days, have passed). Doing this in `RStudio` is extremely easy. Just write whatever you want to write, and stick a hash sign at the start.

   Try it now. Write a sentence in the `R Script` window and put a hash sign at the front. Notice what happens? Try running the sentence through the console, first without the hash sign, and then with it.

8. Eventually, you'll want to stop using `RStudio` to do something else. You can save what you have in the `R Script` window by just going to `File` and choosing `Save As`, just as with many other programs.

   Note though that, depending on your settings, saving an `R` session in this way will **only** save what you've written in the `R Script` window. If you want to save the variables you've defined, and you want to save the inputs you've put into the console, you need to click on `Save All` instead. This will save three files:

   (a) A .R file containing the `R Script` window (or more than one, if you've been working with multiple windows);

   (b) A .RData file, containing the variables you've defined;

(c) A .txt file, containing the inputs you put through the console.

So long as you save all three files in the same folder, loading up the .R file will automatically load the .Rdata and .txt files as well.

Part 2: Functions

In this module, we're not going to be focusing on how to create your own functions in `R` - there are enough pre-created functions to keep us busy. That said, I do want to show you how you can make your own functions, as there are many situations in which writing something yourself is more practical than trying to find a function someone else has written that precisely matches your needs.

The following code defines a simple `R` function, which checks whether the input it is given is precisely divisible by 5 or not.

```
test5<-function(n){
if(n%%5==0){return(‘‘Yes’’)}
else{return(‘‘N’’)}
}
```

1. Copy this function into your `R Script` window and use `Run` to enter it into the console. Try a few inputs to check the function is working as intended. **Note**: you will need to make sure the quote marks you enter into `R are` quote marks - if you copy and paste the text from this sheet, you may find the code won't work without replacing the quote marks.

2. What role does `%%` play here? Can you rewrite the function so that it checks whether or not the input is precisely divisible by 7?

3. What role does `==` play here? What does replacing it with `!=` do to the way the function operates? What does replacing it with `>` do to the way the function operates?

4. (**Tricky!**) What role are `if` and `else` playing in the function? Can you rewrite the function in such a way that it checks first if the input is divisible by 5, and, if it isn't, then check if the answer is divisible by 7? **Hint:** There's nothing stopping you from using `if` more than once in a function.

Important Note: I have used the function `return` above to generate the output of the function `test5`. It is also possible to use the `print` function to do something similar. In general, this is something to avoid, however, since `return` allows an output to be used elsewhere, whereas print does not. Try assigning the output of `test5` to a variable, and using that variable in a calculation. Try the same thing, with the `return` function in `test5` replaced with the `print` function.

Part 3: Descriptive Statistics

In this section, we will use data of various forms to calculate statistical values, and draw appropriate figures.

1. Type `uspop` into `R` to display the `uspop` data. Using this data:
    (a) Use the `mean` and `median` function to find the mean and median value.
    (b) Use the `summary` function to generate what we call a six-number summary (the five-number summary plus the mean). Call this summary `popsum`. What does inputting `popsum[3]` give you? What about `popsum[3:4]`? What about `popsum[c(1,4)]`?
    (c) Find the range of the data. Find the interquartile range of the data.
    (d) The difference between a mean and median (and in particular whether it is positive or negative) can often give an indication of whether data is skewed. Check whether the `uspop` data is skewed using the `hist` function. Is this data skewed? If so, is the skew positive or negative?
    (e) Find the variance using the `var` function. Find the standard deviation using the `sd` function. Verify that the value of the variance is indeed the squared value of the standard deviation.
2. Use the `read.csv` function to load the `hair.csv` and `stop.csv` data files (you can find these on the module Ultra page). Remember that you will have to save the files on your machine first, and use your own machine's file location information inside the `read.csv` function. Remember as well that `R` will not process file location information written with single backslashes - you need to either add an additional backslash each time, or replace each backslash with a forward slash.
3. Using the `hair` data:
    (a) Create a bar chart using the `barplot` function.
    (b) Create the bar chart again, with all bars blue.
    (c) Create a pie chart using the `pie` function.
    (d) Create the pie chart again, with a different combination of five colours.
4. Using the `stop` data:

(a) Create a stem and leaf diagram.

(b) Find the mode of the data.

(c) A helpful function for summarising data is the `quantile` function. Can you use this function to find the quintile values of the data? (Hint: the `quantile` function needs you to include one or more decimal values to tell it how far along the ordered data you want it to measure.)

5. It is often claimed that the second year of a university degree is the hardest; one student believes that there is evidence for this in the average marks for students at Durham, and obtains the mean grades for students on the Mathematics BSc and Mathematics & Statistics BSc from the previous academic year (note that these values have been made up for this sheet).

	Year 1	**Year 2**	**Year 3**
Maths	63.4%	65.1%	64.4%
Math & Stats	64.7%	58.1%	63.2%

Looking at these figures, the student concludes there is clear evidence that it is harder to score highly in the second year than in other years.

We are able to obtain five number summaries (which I covered in this week's videos) and sample sizes for each group:

Maths	**Year 1**	(18.2,58.5,64.4,73.4,91.1)	$n = 139$
Maths	**Year 2**	(22.3,56.4,64.3,75.7,97.4)	$n = 132$
Maths	**Year 3**	(29.1,57.9,66.6,73.4,86.0)	$n = 131$
MathStats	**Year 1**	(46.8,54.8,66.1,71.4,84.1)	$n = 21$
MathStats	**Year 2**	(35.1,47.8,57.8,65.7,82.8)	$n = 20$
MathStats	**Year 3**	(32.6,56.3,61.1,75.6,89.4)	$n = 14$

(a) We can use the `boxplot` function to produce a figure which allows us to compare the five number summaries from each group. You will have to input the summaries into `R` first, though, using the `c` function (or through some other method).

(b) What does the figure suggest to you about the suggestion that second year is the hardest year to score highly? Can you write a two-sentence summary of what the figure allows you to conclude?

(c) (**Tricky!**) For each of the six year groups, find whether any outliers were present. For each case where such values are found, state, giving a reason, whether the presence of outliers requires investigation.