

Univariate Statistics and Methodology using R

Department of Psychology, The University of Edinburgh

Academic year 2020-2021

Contents

Course overview	5
Course description	5
Team	5
Schedule	5
Textbook	7
Intro to R, Rstudio, and data in R	9
0.1 Introducing R and Rstudio	9
0.2 Take a breather	18
0.3 Starting a new .Rmd document	20
0.4 Data in R	23
0.5 Compiling a .Rmd document	26
0.6 Checklist for today	26
Glossary	26
1 Types of data	29
Pre-requisites	29
Learning Objectives	29
1.1 Accessing subsections of data	30
1.2 Editing subsections of data	36
1.3 Types of data	40
1.4 Exercises	43
1.5 Glossary	46

Course overview

Course description

Univariate Statistics and Methodology in R (USMR) is a semester long crash-course aimed at providing Masters students in psychology with a competence in standard univariate methodology and analysis using R. Design and analysis are taught under a unifying framework which shows a) how research problems and design should inform which statistical method to use and b) that many statistical methods are special cases of a more general model. This course will introduce you to statistical modelling and empower you with tools to analyse richer data and answer a broader set of research questions of interest to you.

This course introduces you to statistics and R software. We begin with introductions to the use of statistical methods in research and to R for statistics. We then move to learning about inferential statistics, covering concepts such as hypothesis testing, Type I vs. Type II errors, p-values and power. Common tests such as t-tests and chi-squared tests are introduced before building to a thorough explanation of simple linear regression and its application and assumptions. Extending this to multiple regression with interaction terms and categorical coding schemes, the course culminates in introducing the generalised linear model and the different families of models.

Team

- Professor Martin Corley: martin.corley@ed.ac.uk
- Dr Josiah King: ug.ppls.stats@ed.ac.uk
- Dr Umberto Noe: ug.ppls.stats@ed.ac.uk
- And not forgetting your friendly tutoring team! Ask them anything!

Schedule

Week	Lecture	Lab
1	Introduction to statistics and research methods	Introduction to RCollecting dataTypes of data
2	Logic, sets, probability, sampling, descriptive statistics, and variables	Visualising and describing distributions Visualising and describing relationships
3	Functions, normal distribution, central limit theorem, data visualisation	Probability theoryRandom variablesSampling variability and sampling distributionsBias-variance trade-off
4	Models, hypotheses, probability distributions, significance testing	Bootstrap & Confidence IntervalsHypothesis testing with the p-value approachHypothesis testing with the critical values approachHypothesis testing & Confidence IntervalsMaking decisions - Effect sizes, Power, Errors
5	Statistical tests: z , t , χ^2 , F	Test for two mean (independent samples)Test for one mean and test for two means (paired samples)Chi-square test
Break		
6	GLM 1: Correlation and bivariate regression	Correlation (different types)Move to regressionSimple linear regressionInterpreting coefficientsAssumptions and violations

Week	Lecture	Lab
7	GLM 2: Multiple regression and model checking	Multiple linear regression Interpreting coefficients Assumptions Model selection Model issues
8	GLM 3: Interactions, effects coding, standardisation	Interactions (quant \times quant) Interactions (mixed) ANOVA as special case Coding (dummy, effect) Interactions (cat \times cat)
9	GLM 4: Generalising the GLM - link functions and estimation	Logistic regression Assumptions Multinomial Logistic Other GLM
10	GLM and ANOVA	Recap - common tests as linear model

Textbook

The course textbook is *Learning Statistics with R* (version 0.4 or higher) by D Navarro.

Download the book (version 0.6; PDF) or purchase a printed copy for around £20)

All datasets are available at: <https://learningstatisticswithr.com>

Intro to R, Rstudio, and data in R

Learning Objectives

- LO1: Install R & Rstudio, and get comfortable with the layout
- LO2: Learn about how to read in and store data in R
- LO3: Produce your first Rmarkdown document

0.1 Introducing R and Rstudio

Installing

You have two options for how you use R and Rstudio:

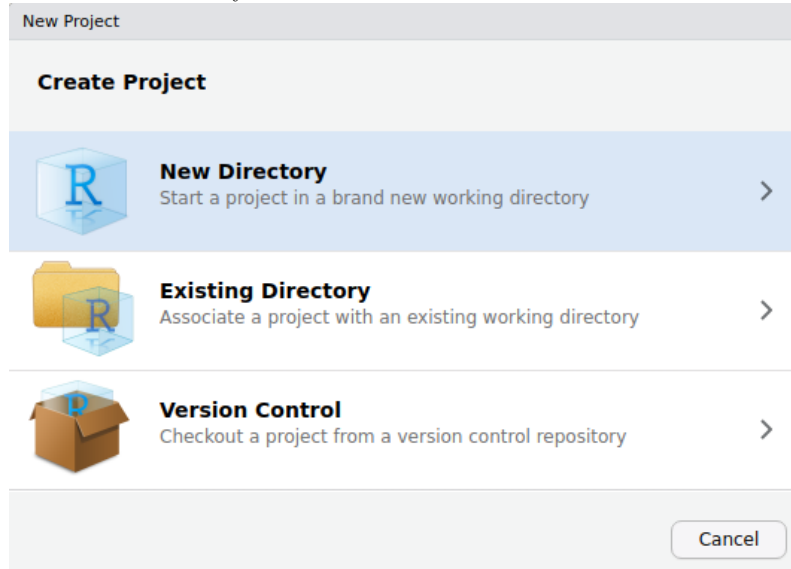
A: Download R and Rstudio onto your computer (recommended)

B: Use R and Rstudio online via a RstudioCloud in a web browser (for people using chromebooks).

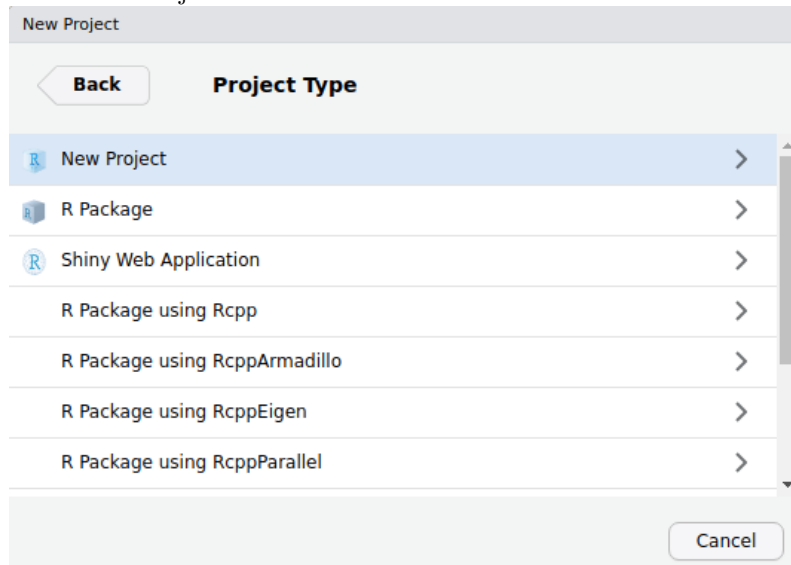
Option A: Installing R and Rstudio (recommended)

1. Download and install the most recent version of R:
 - If you are on a Mac: <https://cran.r-project.org/bin/macosx/>
 - If you are on Windows: <https://cran.r-project.org/bin/windows/base/>
2. Download and install Rstudio:
 - Choose the appropriate downloaded for your computer (e.g., MacOS/Windows): <https://www.rstudio.com/products/rstudio/download/#download>
3. Open Rstudio:
4. Create a new project:
 - File > New Project..

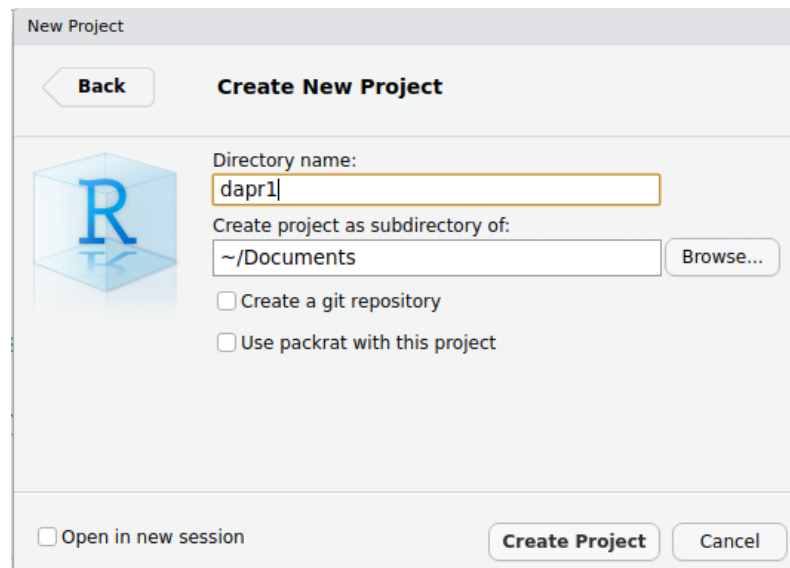
- Click New Directory:



- Click New Project:

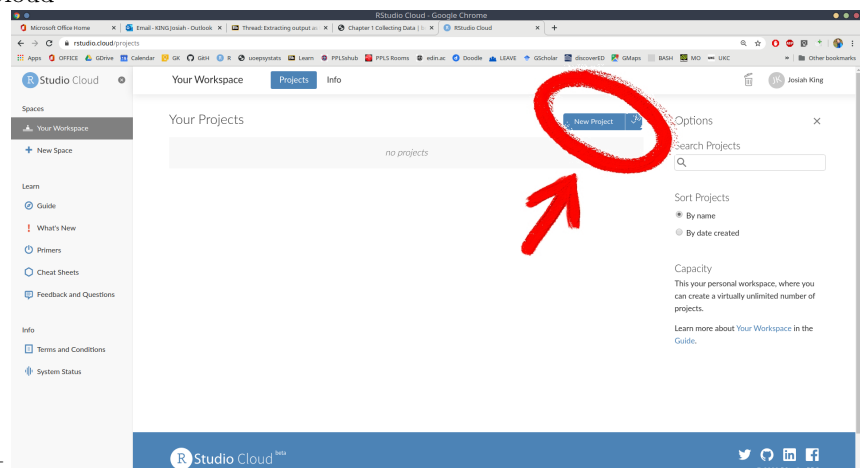


- Name the project, and decide where you want to save it on your computer by clicking on browse. Then click Create Project:

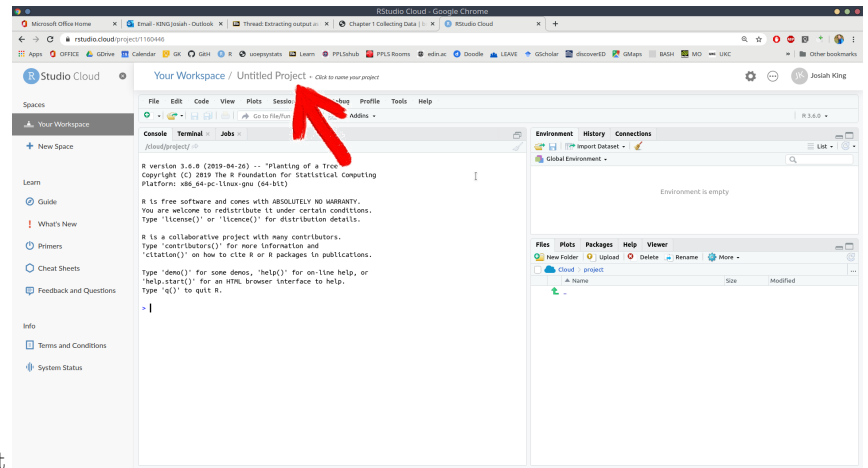


Option B: Rstudio Cloud (for chromebook users)

1. Register for Rstudio Cloud (<https://rstudio.cloud/>).
2. Log in to Rstudio Cloud



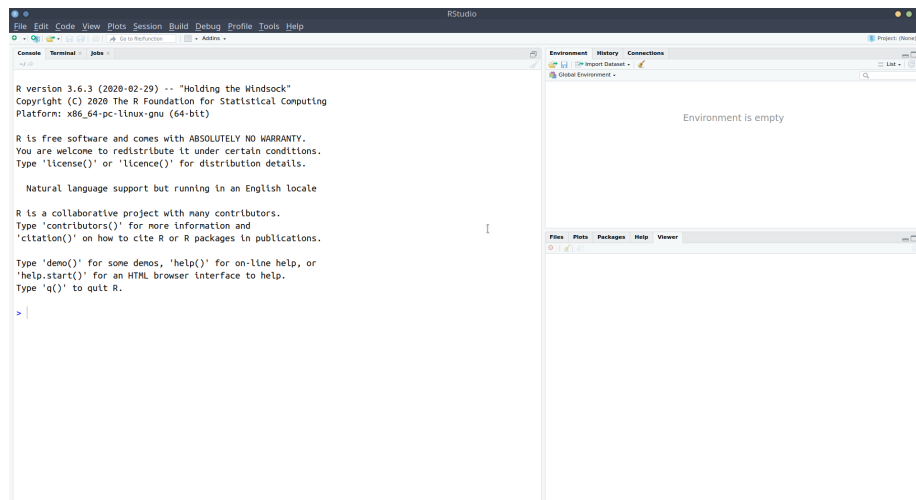
3. Create a new project



4. Rename the project

First look at Rstudio

Okay, now you should have a project open, and you should see something which looks more or less like the below, where there are several little windows.



We're going to explore what each of these little windows offer by just diving in and starting to do things.

R as a calculator

Starting in the left-hand window, you'll notice the little blue arrow `>`. This is where we R code gets *executed*. Type `2+2`, and hit enter

It's a calculator!

Let's work through some of the basic operations (adding, subtracting etc).
Try these commands yourself:

- `2+5`
- `10-4`
- `2*5`
- `10-(2*5)`
- `(10-2)*5`
- `10/2`
- `3^2` (Hint, interpret the `^` symbol as “to the power of”)

Helpful tip

Whenever you see the blue arrow (`>`), it means R is ready and waiting for a command.

If you type `10+` and press enter, you'll see that instead of `>` you are left with `+`. This means that R is waiting for more. Either give it more, or cancel the command by pressing the escape key on your keyboard.

Now let's take a sidestep.

As well as performing calculations, we can *ask* R things, such as “Is 3 less than 5?”:

```
3<5
```

```
[1] TRUE
```

Try the following:

- `3>5` - “is 3 greater than 5?”
- `3<=5` - “is 3 less than OR equal to 5?”
- `3>=3` - “is 3 greater than OR equal to 3?”
- `3==5` - “is 3 equal to 5?”
- `(2*5)==10` “is 2 times 5 equal to 10?”
- `(2*5)!=11` “is 2 times 5 NOT equal to 11?”

R as a calculator with a memory

We can also store things in R's memory, and to that we just need to give them a name.

Type `x <- 5` and press enter.

What has happened? We've just stored something named `x` which has the value 5. We can now refer to the name and it will give us the value! Try typing `x` and hitting enter. It should give you the number 5.

What about `x*3`?

Storing things in R

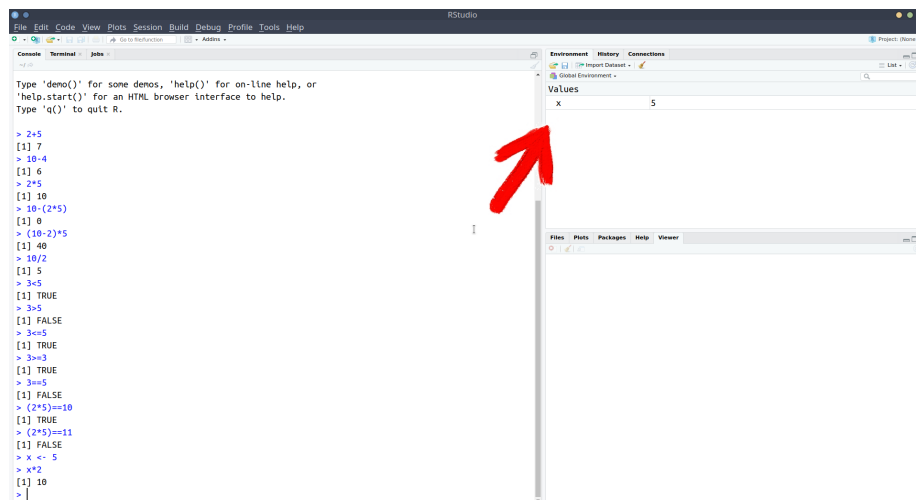
The `<-` symbol is used to *assign* a value to a named object.

`[name] <- [value]`

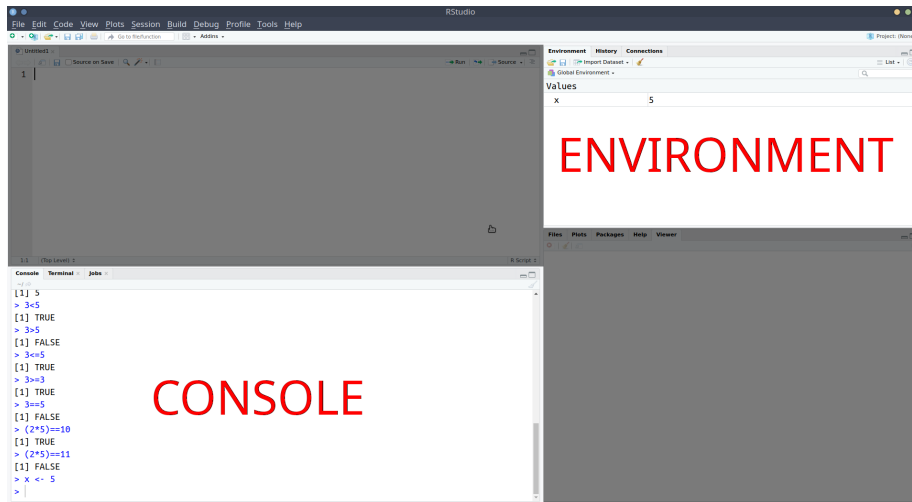
Note, there are a few rules about names in R:

- No spaces - spaces *inside* a name are not allowed (the spaces around the `<-` don't matter):
 - `lucky_number <- 5` `lucky number <- 5`
- Names must start with a letter:
 - `lucky_number <- 5` `1lucky_number <- 5`
- Case sensitive:
 - `lucky_number` is different from `Lucky_Number`
- Reserved words - there is a set of words you can't use as names, including: `if`, `else`, `for`, `in`, `TRUE`, `FALSE`, `NULL`, `NA`, `NaN`, `function` (Don't worry about remembering these, R will tell you if you make the mistake of trying to name a variable after one of these).

You might have noticed that something else happened when you executed the code `x<-5`. The thing we named `x` with a value of `5` suddenly appeared in the top-right window. This is known as the **environment**, and it shows everything that we store things in R:



We've now used a couple of the windows - we've been executing R code in the **console**, and learned about how we can store things in R's memory (the **environment**) by assigning a name to them:



Notice that in the screenshot above, we have moved the **console** down to the bottom-left, and introduced a new window above it. This is the one that we're going to talk about next.

Rscripts and Rmarkdown

What if we want to edit our code?

Whatever we write in the console just disappears upwards. What if we want to change things we did earlier on?

Well, we can write and edit our code in a separate place *before* sending it to the **console** to be executed!!

R scripts

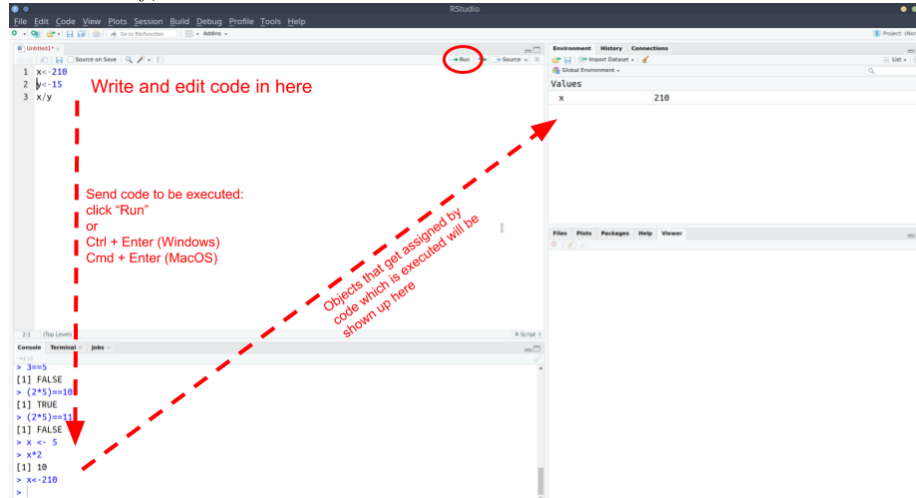
Task 1. Open an R script + **File > New File > R script 1**. Copy and paste the following into the R script

```
x<-210
y<-15
x/y
```

1. With your text-cursor (blinking vertical line) on the top line:
 - Ctrl + enter (Windows)
 - Cmd + enter (MacOS)

Notice what has happened - it has sent the command `x<-210` to the console, where it has been executed, and `x` is now in your environment.

Additionally, it has moved the text-cursor to the next line.



Task Press Ctrl + enter (Windows) or Cmd + enter (MacOS) again. Do it twice (this will run the next two lines).

Then, change `x` to some other number in your R script, and run the lines again (starting at the top).

Task Add the following line to your Rscript and execute it (send it to the console pressing Ctrl/Cmd + Enter):

```
plot(1,5)
```

A very basic plot should have appeared in the bottom-right of Rstudio. The bottom-right window actually does some other useful things.

Task 1. Save the Rscript you have been working with: + File > Save + give it an appropriate name, and click save. 1. Check that you can now see that file in the project, by clicking on the "Files" tab of the bottom-right window.

Rmarkdown

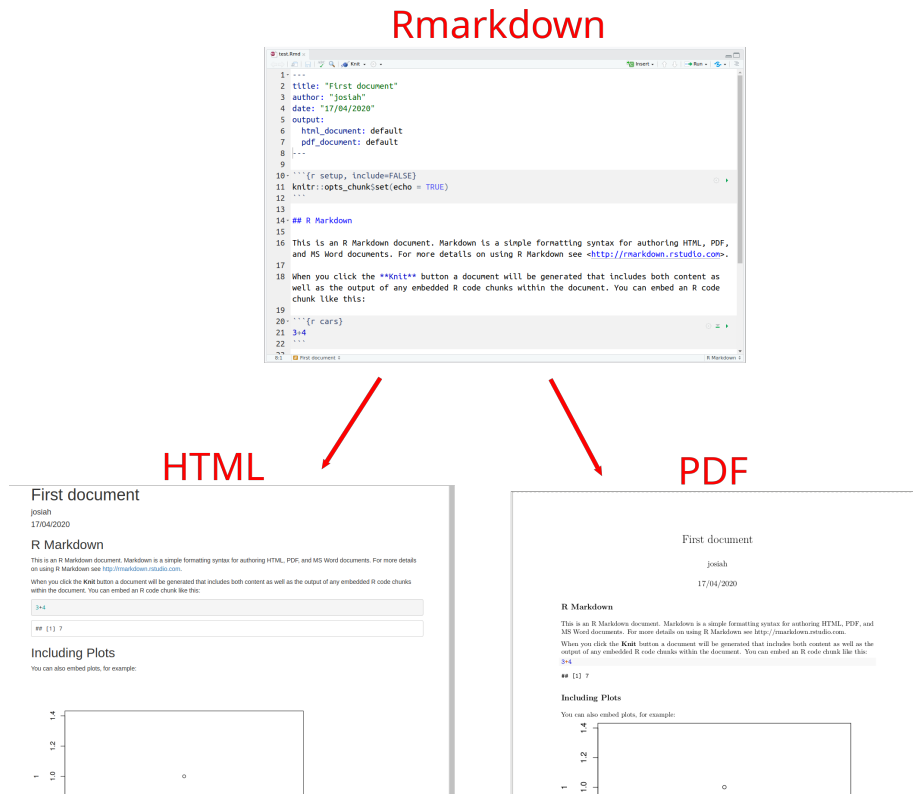
In addition to R scripts, there is another type of document we can create, known as an "Rmarkdown".

Rmarkdown documents combine the analytical power of R and the utility of a text-processor. We can have one document which contains all of our analysis as well as our written text, and can be *compiled* into a nicely formatted report. This saves us doing analysis in R and copying results across to Microsoft Word. It ensures our report accurately reflects our analysis. Everything that you're reading now has all been written in Rmarkdown!

We're going to use Rmarkdown documents throughout this course. We'll get into it how to write them lower down, but it basically involves writing normal text interspersed with "code-chunks" (i.e., chunks of code!).

In the example below, you can see the grey boxes indicating the R code, with text in between.

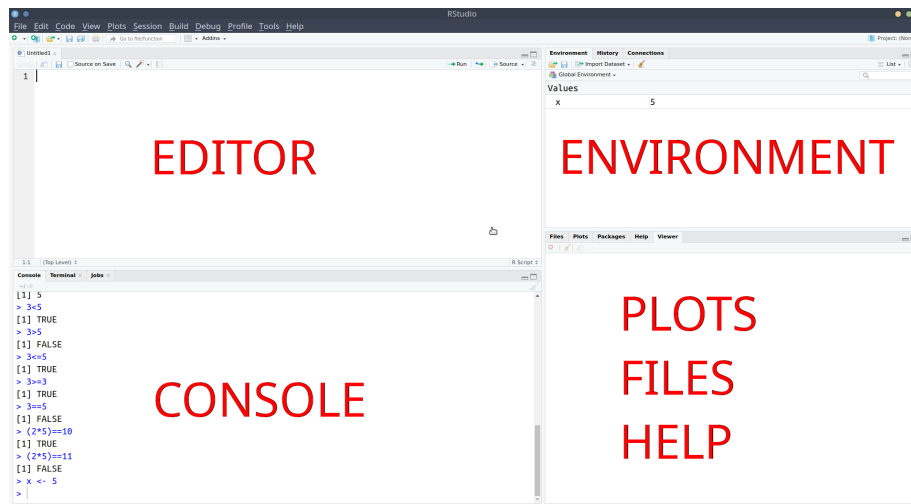
We can then compile the document into either a **.pdf** or a **.html**.



Recap

Okay, so we've now seen all of the different windows in Rstudio in action:

- The **console** is where R code gets executed
- The **environment** is R's memory, you can *assign* something a name and store it here, and then refer to it by name in your code.
- The **editor** is where you can write and edit R code and Rmarkdown documents. You can then send this to the console for it to be executed.
- The bottom-right window shows you the **plots** that you create, the **files** in your project, and some other things (we'll get to these later).



0.2 Take a breather

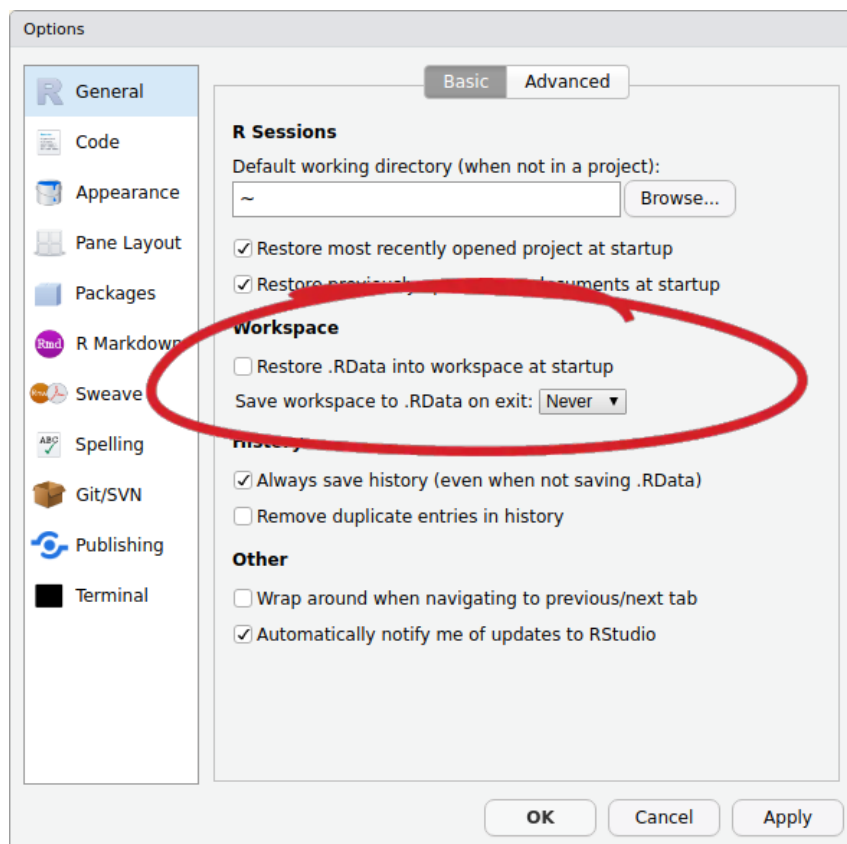
Below are a couple of our recommended settings for you to change as you begin your journey in R. After you've changed them, take a 5 minute break before moving on to learning about how we store data in R.

Useful Settings 1: Clean environments

As you use R more, you will store lots of things with different names. Throughout this course alone, you'll probably name hundreds of different things. This could quickly get messy within our project.

We can make it so that we have a clean environment each time you open Rstudio. This will be really handy.

1. In the top menu, click
Tools > Global Options...
2. Then, *untick* the box for "Restore .RData into workspace at startup", and change "Save workspace to .RData on exit" to "Never":



Useful Settings 2: Wrapping code

In the editor, you might end up with a line of code which is really long:

```
x <- 1+2+3+6+3+45+8467+356+8565+34+34+657+6756+456+456+54+3+78+3+3476+8+4+67+456+567+3+34575+45+2
```

You can make Rstudio ‘wrap’ the line, so that you can see it all, without having to scroll.

```
x <- 1+2+3+6+3+45+8467+356+8565+34+34+657+6756+456+456+54+3+78+3+3476+8+4+67+
456+567+3+34575+45+2+6+9+5+6
```

1. In the top menu, click **Tools > Global Options...**
2. In the left menu of the box, click “Code”
3. *Tick* the box for “Soft-wrap R source files”

0.3 Starting a new .Rmd document

Pre-requisites: Installing R packages

Alongside the basic installation of R and Rstudio, there are many add-on packages which the R community create and maintain.

The thousands of packages are part of what makes R such a powerful and useful tool - there is a package for almost everything you could want to do in R.

In order to be able to write and compile Rmarkdown documents (and do a whole load of other things which we are going to need throughout the course) we are now going to install a set of packages known collectively as the “tidyverse” (this includes the “rmarkdown” package).

Task In the **console**, type `install.packages("tidyverse")` and hit enter.

Lots of red text will come up, and it will take a bit of time.

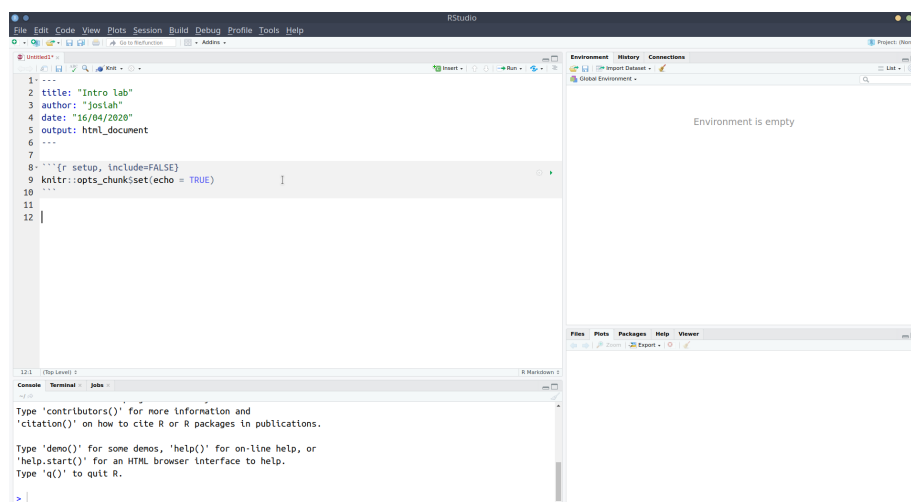
When it has finished, and R is ready for you to use again, you will see the little blue arrow `>`.

Task Open a new Rmarkdown document.

File > New File > R Markdown..

When the box pops-up, give a title of your choice (“Intro lab”, maybe?) and your name as the author.

The file which opens will have some template stuff in it. Delete everything below the first code chunk to start with a fresh document:



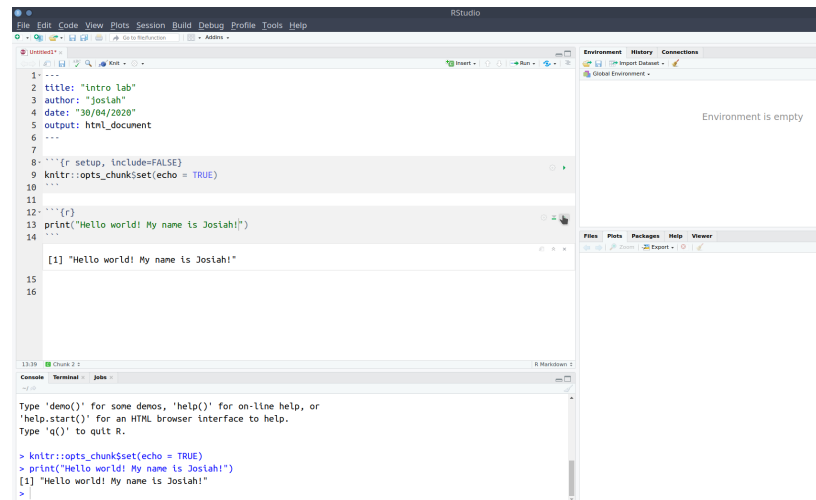
Task Insert a new code chunk by either using the Insert button in the top right of the document, and clicking R, or typing `Ctrl + Alt + i`

Inside the chunk, type:

```
print("Hello world! My name is ?").
```

To execute the code inside the chunk, you can either:

- do as you did in the R script - put the text-cursor on the first line, and hit Ctrl/Cmd + Enter to run the lines sequentially
- click the little green arrow at the top right of your code-chunk to run all of the code inside the chunk.



```
1- ...
2 title: "intro lab"
3 author: "Josiah"
4 date: "30/04/2020"
5 output: html_document
6 ...
7
8- ```{r setup, include=FALSE}
9 knitr::opts_chunkset(echo = TRUE)
10 ...
11
12- ```{r}
13 print("Hello world! My name is Josiah!")
14 ...
15
16
```

[1] "Hello world! My name is Josiah!"

13:39 Chunk 2.1 8 Workbooks 1

Console Terminal Jobs

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

```
> knitr::opts_chunkset(echo = TRUE)
> print("Hello world! My name is Josiah!")
[1] "Hello world! My name is Josiah!"
>
```

You can see that the output gets printed below.

Using an R package

We're going to use some functions which are in the **tidyverse** package, which already installed above.

However, it's not enough just to install it - to actually *use* the package, we need to load it using `library(tidyverse)`.



Images sourced from <https://www.wikihow.com/Change-a-Light-Bulb>

(source: <https://twitter.com/visnut/status/1248087845589274624>)

When writing analysis code, we want it to be **reproducible** - we want to be able to give somebody else our code and the data, and ensure that they can get the same results. To do this, we need to show what packages we use.

It is good practice to load any packages you use at the top of your code.

Task In your first code chunk, type:

```
#I'm going to use these packages in this document:
library(tidyverse)
```

and run the chunk.

Note, you might get various messages popping up below when you run this chunk, that is fine).

Comments in code

Note that using **#** in R code makes that line a comment, which basically means that R will ignore the line. Comments are useful for you to remind yourself of what your code is doing.

Writing

Task Below the code chunk, add a new line with the following:

```
# R code examples
```

Note that when the **#** is used in a Rmarkdown file **outside** of a code-chunk, it will make that line a heading when we finally get to *compiling* the document. Below, what you see on the left will be compiled to look like those on the right:

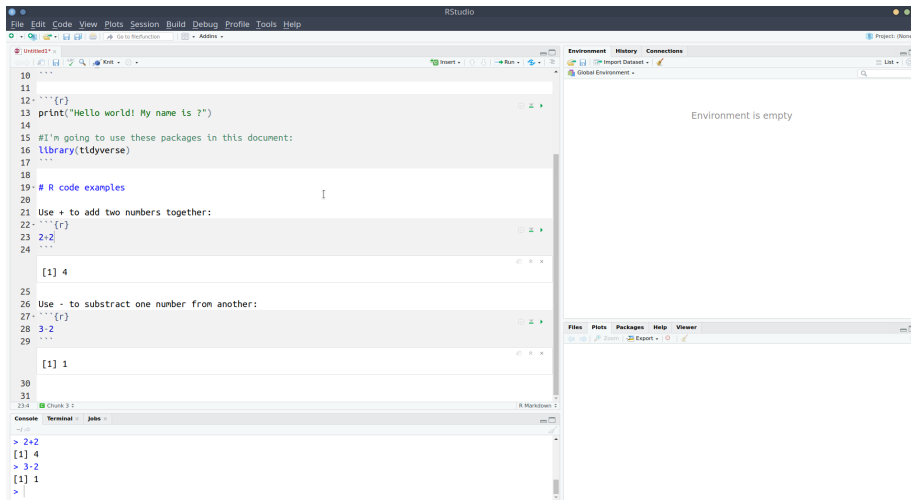
```
# Header 1
## Header 2
### Header 3
#### Header 4
##### Header 5
##### Header 6
```

Header 1
Header 2
Header 3
Header 4
Header 5
Header 6

Task In your Rmarkdown document, choose a few of the symbols below, and write an explanation of what it does, giving an example in a code chunk. You can see an example of the first few below.

- +
- -
- *
- /
- ()
- ^

- `<-`
- `<`
- `>`
- `<=`
- `>=`
- `==`
- `!=`



0.4 Data in R

Storing data in R: Sequences (“Vectors”) of values

We’ve already seen how to assign a value to a name/symbol using `<-`. However, we’ve only seen how to assign a single number, e.g. `x<-5`.

To assign a sequence of values to R, we combine the values using `c()`.

```
myfirstvector <- c(1,5,3,7)
myfirstvector
```

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

We can perform arithmetic operations to each value in the vector:

```
myfirstvector + 5
```

```
[1] 6 10 8 12
```

Values don’t have to be numbers, but note what happens when we try to add 5 to a sequence which includes some non-numbers:

```
mysecondvector <- c(1,4,"cat","dog","parrot","peppapig")
mysecondvector + 5
```

Error in mysecondvector + 5 : non-numeric argument to binary operator

Reading data into R

While we can manually input data like we did above, more often, we will need to read in data which has been created elsewhere (like in excel, or by some software which is used to present participants with experiments).

Task Add a new heading by typing the following:

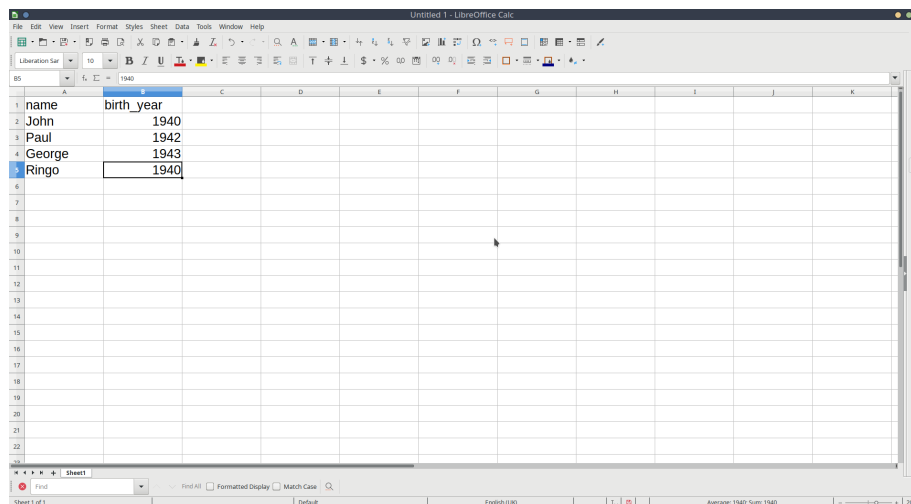
```
# Reading and storing data
```

Remember: We make headings using the # *outside* of a code chunk.

Task Open Microsoft Excel, or LibreOffice Calc, or whatever spreadsheet software you have available to you, and create some data with more than one variable.

It can be whatever you want, but we've used a very small example here for you to follow, so feel free to use it if you like.

We've got two sets of values here: the names and the birth-years of each member of the beatles. The easiest way to think of this would be to have a row for each Beatle, and a column for each of name and birth-year.



Task Save the data as a .csv file.

Although R can read data when it's saved in Microsoft/LibreOffice formats, the simplest, and most universal way to save data is as simple text, with the values

separated by some character - **.csv** stands for **comma separated values**.

In Microsoft Excel, if you go to: **File > Save as**

In the Save as Type box, choose to save the file as **CSV (Comma delimited)**.

Important: save your data in the project folder you created at the start of this lab.

Back in Rstudio...

Next, we're going to read the data into R. We can do this by using the `read_csv()` function, and directing it to the file you just saved.

Task Create a new code-chunk in your Rmarkdown, and in the chunk, type: `read_csv("name-of-your-data.csv")`, where you replace *name-of-your-data* with whatever you just saved your data as in your spreadsheet software.

Helpful tip

If you have your text-cursor inside the quotation marks, and press the tab key on your keyboard, it will show you the files inside your project. You can then use the arrow keys to choose between them and press enter to add the code:

TODO - screencap gif

When you run the line of code you just wrote, it will print out the data, but will not store it. To do that, we need to assign it as something:

```
beatles <- read_csv("data_from_excel.csv")
```

Note that this will now turn up in the *Environment* pane of Rstudio.

Now that we've got our data in R, we can print it out by simply using its name:

```
beatles
```

```
# A tibble: 4 x 2
  name    birth_year
  <chr>      <dbl>
1 John         1940
2 Paul         1942
3 George       1943
4 Ringo        1940
```

And we can do things such as ask R how many rows and columns there are:

```
dim(beatles)
```

```
[1] 4 2
```

```
str(beatles)
```

```
Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame':   4 obs. of  2 variables:
 $ name      : chr  "John" "Paul" "George" "Ringo"
```

```
$ birth_year: num 1940 1942 1943 1940
- attr(*, "spec")=
.. cols(
..   name = col_character(),
..   birth_year = col_double()
.. )
```

Task Use `dim()` to confirm how many rows and columns are in your data.

Use `str()` to take a look at the structure of the data.

Don't worry about the output of `str()` right now, we'll pick up with this in the next chapter.

0.5 Compiling a .Rmd document

Task By now, you should have an Rmarkdown document (**.Rmd**) with your answers to the tasks we've been through today.

Compile the document by clicking on the **Knit** button at the top (it will ask you to save your document first). The little arrow to the right of the Knit button allows you to compile to either **.pdf** or **.html**.

0.6 Checklist for today

1. **EITHER:**
 - Option A: Install R and Rstudio
 - Option B: Register for RstudioCloud (free) and log in
2. Start a new project for the course
3. Change a few Rstudio settings (recommended)
4. Install some R packages (the “tidyverse”)
5. Create a new Rmarkdown document
6. Complete today's tasks and exercises on storing data in R
7. Compile your Rmarkdown document
8. Celebrate!

Glossary

- console
- environment
- editor
- r script

- rmarkdown

Symbol	Description	Example
+	Adds two numbers together	2+2 - two plus two
-	Subtract one number from another	3-1 - three minus one
*	Multiply two numbers together	3*3 - three times three
/	Divide one number by another	9/3 - nine divided by three
()	group operations together	(2+2)/4 is different from 2+2/4
^	to the power of..	4^2 - four to the power of two, or four squared
<-	stores an object in R with the left hand side (LHS) as the name, and the RHS as the value	x<-10
=	stores an object in R with the left hand side (LHS) as the name, and the RHS as the value	x = 10
<	is less than?	2<3
>	is greater than?	2>3
<=	is less than or equal to?	2<=3
>=	is greater than or equal to?	2>=2
==	is equal to?	(5+5) == 10
!=	is not equal to?	(2+3) != 4
c()	combines values into a vector (a sequence of values)	c(1,2,3,4)

SHOW_SOLs = TRUE

Chapter 1

Types of data

Pre-requisites

Please ensure you have successfully installed R and RStudio, or are working on RStudioCloud, and that you have completed the tasks on the Introduction to R page - Included as the previous Chapter of this book, which illustrated the basics of a) how to create a Rmarkdown document, b) how to read data into R, and c) how to use R to do basic arithmetic.

In this chapter, we are going to take a look at how to access specific sections of data, before we move on to talking about different types of data.

We encourage you to work along with the reading - open a new RMarkdown document, and run the code which we discuss below.

At the end of the chapter, there is a set of exercises for you to complete.

Learning Objectives

- LO1: Learn how to access entries within data stored in R
- LO2: Learn how to modify entries/variables in data
- LO3: Understand the distinction between types of variable and how to code data as different types (classes) in R

1.1 Accessing subsections of data

Suppose we read some data in to R:

```
library(tidyverse)

starwars<-dplyr::starwars %>% select(-mass, -skin_color, -films, -vehicles,-starships,
starwars[87,]<-c("Homer Simpson",180,NA,NA,"Springfield","unknown")
starwars[40,]<-c("Marge Simpson",170,"Blue",NA,"Springfield","unknown")
starwars[2,"species"]<-"Human"
starwars[7,"height"]<-9420
starwars$height<-as.numeric(starwars$height)
starwars<-starwars %>% filter(!is.na(homeworld),!is.na(species))
```

Reading data from the internet

Note that when you have a url for some data, such as link, you can read it in directly by giving functions like `read_csv()` the url inside quotation marks.

We can print out the top of the data by using the name we just gave it:

```
starwars

# A tibble: 75 x 6
  name          height hair_color eye_color homeworld species
  <chr>         <dbl> <chr>    <chr>    <chr>    <chr>
1 Luke Skywalker   172 blond    blue    Tatooine Human
2 C-3PO            167 <NA>     yellow  Tatooine Human
3 R2-D2             96 <NA>     red     Naboo   Droid
4 Darth Vader      202 none     yellow  Tatooine Human
5 Leia Organa      150 brown    brown   Alderaan Human
6 Owen Lars        178 brown, grey blue    Tatooine Human
7 Beru Whitesun lars 9420 brown    blue    Tatooine Human
8 R5-D4             97 <NA>     red     Tatooine Droid
9 Biggs Darklighter 183 black    brown   Tatooine Human
10 Obi-Wan Kenobi   182 auburn, white blue-gray Stewjon Human
# ... with 65 more rows
```

(Don't worry about the NAs for now, they are just how R tells you an entry is missing)

The data contains information on various characteristics of characters from Star Wars.

Tip: Try clicking on the data in your **environment** (the top right window of Rstudio). It will open the data in a tab in the editor window - this is another way of looking at the data, more like you would in spreadsheet software like Microsoft Excel.

We can take a look at how big the data is (the dimensions), using `dim()`

```
dim(starwars)
```

```
[1] 75  6
```

There's a reasonable amount of data in there - 75 rows and 6 variables (columns). What if we want to extract certain bits of it?

This is where we learn about two important bits of R code used to access parts of data - the dollar sign \$, and the square brackets [].

The dollar sign \$

The dollar sign allows us to select a specific variable.

For instance, we can pull out the variable named "eye_color" in the data, by using \$eye_color after the name that we gave our dataset:

```
starwars$eye_color
```

[1]	"blue"	"yellow"	"red"	"yellow"	"brown"	"blue"
[8]	"red"	"brown"	"blue-gray"	"blue"	"blue"	"blue"
[15]	"black"	"orange"	"hazel"	"blue"	"yellow"	"brown"
[22]	"brown"	"blue"	"orange"	"blue"	"brown"	"black"
[29]	"blue"	"orange"	"orange"	"orange"	"yellow"	"orange"
[36]	"brown"	"yellow"	"pink"	"hazel"	"yellow"	"black"
[43]	"brown"	"yellow"	"black"	"brown"	"blue"	"orange"
[50]	"black"	"blue"	"brown"	"brown"	"blue"	"yellow"
[57]	"blue"	"brown"	"brown"	"brown"	"brown"	"yellow"
[64]	"black"	"black"	"blue"	"unknown"	"unknown"	"gold"
[71]	"green, yellow"	"blue"	"brown"	"black"	NA	

The square brackets []

Square brackets are used to do what is known as **indexing** (finding specific entries in your data).

We can retrieve bits of data by identifying the i^{th} entry(s) inside the square brackets, for instance:

```
somevalues <- c(10,20,30,40,50,60,70,80,90,100)
```

```
# pull out the 3rd entry
```

```
somevalues[3]
```

```
[1] 30
```

In the above example, we have a **vector** (a single sequence of values), and so we can retrieve entries with the syntax:

```
vector[entry]
```

In a **dataframe** we have an extra dimension - we have *rows* **and** *columns*. Using square brackets with a dataframe needs us to specify both:

dataframe[rows, columns]

For instance:

```
# first row, fourth column:
starwars[1,4]
```

```
# A tibble: 1 x 1
  eye_color
  <chr>
1 blue
```

```
# tenth row, first column:
starwars[10,1]
```

```
# A tibble: 1 x 1
  name
  <chr>
1 Obi-Wan Kenobi
```

If we leave either rows or columns blank, then we will get out *all* of them:

```
# tenth row, all columns:
starwars[10, ]
```

```
# A tibble: 1 x 6
  name          height hair_color eye_color homeworld species
  <chr>          <dbl> <chr>      <chr>      <chr>      <chr>
1 Obi-Wan Kenobi 182 auburn, white blue-gray Stewjon Human
```

```
# all rows, 2nd column:
starwars[ , 2]
```

```
# A tibble: 75 x 1
  height
  <dbl>
1 172
2 167
3 96
4 202
5 150
6 178
7 9420
8 97
9 183
10 182
# ... with 65 more rows
```


There is another way to identify column - we can use the name in quotation marks:

```
# first row, "species" column
starwars[1, "species"]
```

```
# A tibble: 1 x 1
  species
  <chr>
1 Human
```

Finally, we can also ask for multiple rows, or multiple columns, or both!

```
# the 1st AND the 6th row,
# and the 1st AND 3rd columns:
starwars[c(1,6), c(1,3)]
```

```
# A tibble: 2 x 2
  name          hair_color
  <chr>         <chr>
1 Luke Skywalker blond
2 Owen Lars     brown, grey
```

And we can specify a sequence using the colon, **from:to**:

```
# FROM the 1st TO the 6th row, all columns:
starwars[1:6, ]
```

```
# A tibble: 6 x 6
  name          height hair_color eye_color homeworld species
  <chr>          <dbl> <chr>    <chr>    <chr>    <chr>
1 Luke Skywalker   172 blond    blue     Tatooine Human
2 C-3PO            167 <NA>     yellow   Tatooine Human
3 R2-D2            96 <NA>     red      Naboo    Droid
4 Darth Vader      202 none     yellow   Tatooine Human
5 Leia Organa      150 brown    brown    Alderaan Human
6 Owen Lars        178 brown, grey blue     Tatooine Human
```

Extra We can use the two accessors in combination:

```
# extract the variable called "name" and show the 20th entry
starwars$name[20]
```

```
[1] "Boba Fett"
```

Note: When we do this, we don't have the comma inside the square brackets. When we use the **\$** to pull out a variable, such as `starwars$name`, we no longer have a dataframe - `starwars$name` doesn't have rows and columns, it just has a series of values - *it's a vector!*.

So when you are using `[]` with a **vector** (1 dimension) rather than a **dataframe** (2 dimensions), you don't specify `[rows, columns]`, but simply `[entry]`.

Accessors

The dollar sign `$`

Used to extract a variable from a dataframe:

- `dataframe$variable`

The square brackets `[]`

Used to extract parts of an R object by identifying rows and/or columns, or more generally, “entries”. Left blank will return all.

- `dataframe[rows, columns]`
- `vector[entries]`

Accessing by a condition

We can also do something really useful, which is to access all the entries in the data for which *a specific condition* is true.

Let’s take a simple example to start:

```
somevalues <- c(10,10,0,20,15,40,10,40,50,35)
```

If we want to access all the values which are `>20`, we can use:

```
somevalues[somevalues>20]
```

```
[1] 40 40 50 35
```

Let’s unpack what this is doing.. First, let’s look at what `somevalues>20` does:

```
somevalues>20
```

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

It gives us `FALSE` for the entries of `somevalues` which are less than (or equal to) 20, and `TRUE` for the entries which are greater. This statement `>20` is the **condition**.

Now consider the following, where we have taken that set of `TRUE/FALSE`’s and put them inside the square brackets:

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

```
[1] 40 40 50 35
```

It gives us out the entries of `somevalues` which we have specified are `TRUE`. This is the same as what our initial line was doing:

```
somevalues[somevalues>20]
```

```
[1] 40 40 50 35
```

We can extend this same logic to a dataframe.

Let's suppose we want to access all the entries in our Star Wars data who have the value "Droid" in the *species* variable.

To work out how to do this, we first need a line of code which defines our **condition** - one which returns TRUE for each entry of the *species* variable which is "Droid", and FALSE for those that are not "Droid".

We can use the dollar sign to pull out the *species* variable:

```
starwars$species
```

[1]	"Human"	"Human"	"Droid"	"Human"	"Human"	"Human"	"Human"
[8]	"Droid"	"Human"	"Human"	"Human"	"Human"	"Wookiee"	"Human"
[15]	"Rodian"	"Hutt"	"Human"	"Human"	"Human"	"Human"	"Human"
[22]	"Human"	"Human"	"Mon Calamari"	"Human"	"Ewok"	"Sullustan"	"Human"
[29]	"Human"	"Gungan"	"Gungan"	"Gungan"	"Toydarian"	"Dug"	"Human"
[36]	"Human"	"Zabrak"	"Twi'lek"	"Twi'lek"	"Vulptereen"	"Xexto"	"Human"
[43]	"Human"	"Cerean"	"Nautolan"	"Zabrak"	"Tholothian"	"Iktotchi"	"Human"
[50]	"Kel Dor"	"Chagrian"	"Human"	"Human"	"Human"	"Geonosian"	"Human"
[57]	"Mirialan"	"Human"	"Human"	"Human"	"Human"	"Clawdite"	"Human"
[64]	"Kaminoan"	"Kaminoan"	"Human"	"Aleena"	"Skakoan"	"Muun"	"Human"
[71]	"Kaleesh"	"Wookiee"	"Human"	"Pau'an"	"unknown"		

And we can ask R whether each value is equal to "Droid".

Remember: in R, we ask whether something is equal to something else by using a double-equals, ==.

```
starwars$species == "Droid"
```

```
[1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[20] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[39] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[58] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Finally, we can use this list of TRUE/FALSEs inside our square brackets to access the entries of the data for which this condition is TRUE:

```
# I would read the code below as:
```

```
# "In the starwars dataframe, give me all the rows for which the
# condition starwars$species=="Droid" is TRUE, and give me all the columns."
```

```
starwars[starwars$species == "Droid", ]
```

```
# A tibble: 2 x 6
```

	name	height	hair_color	eye_color	homeworld	species
	<chr>	<dbl>	<chr>	<chr>	<chr>	<chr>
1	R2-D2	96	<NA>	red	Naboo	Droid
2	R5-D4	97	<NA>	red	Tatooine	Droid

1.2 Editing subsections of data

Now that we’ve seen how to *access* sections of data, we can learn how to edit them!

Data Cleaning

One of the most common reasons you will need to edit entries in your data is in **data cleaning**. This is the process of identifying incorrect/incomplete/irrelevant data, and replacing/modifying/deleting them.

Modifying specific entries

Above, we looked at the subsection of the data where the *species* variable had the entry “Droid”. Some of you may have noticed earlier that we had some data on C3PO. Is he not also a droid?



(Looks pretty Droid-y to me! *disclaimer: I know nothing about Star Wars*)

Just as we saw above how to *access* specific entries, e.g.:

```
# 2nd row, all columns
starwars[2, ]
```

```
# A tibble: 1 x 6
  name height hair_color eye_color homeworld species
  <chr>   <dbl> <chr>      <chr>    <chr>    <chr>
1 C-3PO   167 <NA>      yellow   Tatooine Human
```

```
# 2nd row, 6th column (the "species" column)
starwars[2,6]
```

```
# A tibble: 1 x 1
  species
  <chr>
1 Human
```

We can change these by **assigning them a new value** (remember the <- symbol):

```
# C3PO is a droid, not a human
starwars[2,6]<-"Droid"
```

```
# Look at the 2nd row now -
# the entry in the "species" column has changed:
starwars[2, ]
```

```
# A tibble: 1 x 6
  name height hair_color eye_color homeworld species
  <chr>  <dbl> <chr>      <chr>      <chr>      <chr>
1 C-3PO   167 <NA>      yellow     Tatooine   Droid
```

think of it as “overwriting”

We have *overwritten* the entry in the 2nd row, 6th column of the data (starwars[2,6]) with the value “Droid”.

Modifying entries based on a condition

We saw above how we can access subsections of data based on a **condition**, with code such as:

```
# "In the starwars dataframe, give me all the rows for which the
# condition starwars$homeworld=="Naboo" is TRUE, and give me all the columns."

# remember, we're asking for all the columns by leaving it blank *after* the
# comma inside the square brackets: data[rows, columns]
starwars[starwars$homeworld=="Naboo", ]
```

```
# A tibble: 8 x 6
  name          height hair_color eye_color homeworld species
  <chr>         <dbl> <chr>      <chr>      <chr>      <chr>
1 R2-D2          96 <NA>      red         Naboo      Droid
2 Palpatine     170 grey      yellow     Naboo      Human
3 Jar Jar Binks 196 none      orange     Naboo      Gungan
4 Roos Tarpals  224 none      orange     Naboo      Gungan
5 Rugor Nass    206 none      orange     Naboo      Gungan
```

6	Gregar Typho	185	black	brown	Naboo	Human
7	Cordé	157	brown	brown	Naboo	Human
8	Dormé	165	brown	brown	Naboo	Human

What if we wanted to modify it so that every character from “Naboo” was actually of species “Nabooian”?

We can do that in a number of ways, all of which do the same thing - namely, they access parts of the data and assign them the new value “Nabooian”.

Study the lines of code below and their interpretations:

```
# In the starwars data, give the rows for which condition starwars$homeworld=="Naboo"
starwars[starwars$homeworld=="Naboo", "species"] <- "Nabooian"

# In the starwars data, give the rows for which condition starwars$homeworld=="Naboo"
starwars[starwars$homeworld=="Naboo", 6] <- "Nabooian"

# In the species variable in the starwars data, give the entries for which condition s
starwars$species[starwars$homeworld=="Naboo"] <- "Nabooian"
```

Changing a whole column

Another thing we might want to do is change a whole column in some way. The logic is the same, for instance:

```
starwars$height <- starwars$height/100
```

What we have done above is assign the variable “height” inside the dataframe “starwars”, and give it the values which we get from `starwars$height/100`.

Equally, we *could also have a new* column, called “height2” with these values:

```
starwars$height2 <- starwars$height/100
```

This would have left the “height” variable as-is, and created a new one called “height2” which was the values in “height” divided by 100.

Changing the shape of the data

Lastly, we might want to change the data by removing a row or a column. Again, the logic remains the same, in that we use `<-` to assign the edited data to a name (either a new name, thus creating a new object, or an existing name, thereby *overwriting* that object).

For instance, notice that the 40th and 87th rows of our data probably aren’t a valid observation - I’m reasonably sure that Marge and Homer Simpson never appeared in Star Wars:

```
starwars[c(40,87), ]
```

```
# A tibble: 2 x 6
  name      height hair_color eye_color homeworld species
<chr>      <dbl> <chr>      <chr>      <chr>      <chr>
1 Dud Bolt   0.94 none      yellow     Vulpter   Vulptereen
2 <NA>       NA   <NA>      <NA>      <NA>      <NA>
```

We can remove a certain row(s) by using a minus sign - inside the square brackets

```
# everything minus the 87th row
starwars[-87, ]
```

```
# A tibble: 75 x 6
  name      height hair_color eye_color homeworld species
<chr>      <dbl> <chr>      <chr>      <chr>      <chr>
1 Luke Skywalker   1.72 blond      blue      Tatooine   Human
2 C-3P0            1.67 <NA>      yellow     Tatooine   Droid
3 R2-D2            0.96 <NA>      red        Naboo      Nabooian
4 Darth Vader      2.02 none      yellow     Tatooine   Human
5 Leia Organa      1.5  brown     brown      Alderaan   Human
6 Owen Lars        1.78 brown, grey blue      Tatooine   Human
7 Beru Whitesun lars 94.2 brown     blue      Tatooine   Human
8 R5-D4            0.97 <NA>      red        Tatooine   Droid
9 Biggs Darklighter 1.83 black     brown      Tatooine   Human
10 Obi-Wan Kenobi   1.82 auburn, white blue-gray Stewjon    Human
# ... with 65 more rows
```

```
# everything minus the (40th and 87th rows)
starwars[-c(40, 87), ]
```

```
# A tibble: 74 x 6
  name      height hair_color eye_color homeworld species
<chr>      <dbl> <chr>      <chr>      <chr>      <chr>
1 Luke Skywalker   1.72 blond      blue      Tatooine   Human
2 C-3P0            1.67 <NA>      yellow     Tatooine   Droid
3 R2-D2            0.96 <NA>      red        Naboo      Nabooian
4 Darth Vader      2.02 none      yellow     Tatooine   Human
5 Leia Organa      1.5  brown     brown      Alderaan   Human
6 Owen Lars        1.78 brown, grey blue      Tatooine   Human
7 Beru Whitesun lars 94.2 brown     blue      Tatooine   Human
8 R5-D4            0.97 <NA>      red        Tatooine   Droid
9 Biggs Darklighter 1.83 black     brown      Tatooine   Human
10 Obi-Wan Kenobi   1.82 auburn, white blue-gray Stewjon    Human
# ... with 64 more rows
```

And we can simply *re-use* the name “starwars” to overwrite the data and make

this change take effect (rather than just print out the result, which the code above did):

```
starwars <- starwars[-c(40,87), ]
```

(now, in the **environment** pane of Rstudio, the object named “starwars” will say 85 observations, rather than 87, which it had before - we’ve removed the 2 rows)

The same logic applies for columns:

```
# Create a new object called "anonymous_starwars" and assign it
# to the values which are the "starwars" dataset minus the
# 1st column (the "name" column):
anonymous_starwars <- starwars[, -1]
```

```
# print out anonymous_starwars
anonymous_starwars
```

```
# A tibble: 74 x 5
  height hair_color eye_color homeworld species
  <dbl> <chr>      <chr>      <chr>      <chr>
1  1.72 blond      blue       Tatooine   Human
2  1.67 <NA>       yellow     Tatooine   Droid
3  0.96 <NA>       red        Naboo      Nabooian
4  2.02 none       yellow     Tatooine   Human
5  1.5  brown        brown      Alderaan   Human
6  1.78 brown, grey blue       Tatooine   Human
7  94.2 brown        blue       Tatooine   Human
8  0.97 <NA>       red        Tatooine   Droid
9  1.83 black        brown      Tatooine   Human
10 1.82 auburn, white blue-gray Stewjon    Human
# ... with 64 more rows
```

1.3 Types of data

There are so many different ways we can measure things, and so the data which results from our measurements vary also.

For instance, we could measure the colour of people’s hair as any of “black”, “brown”, “blond”, and so on, or we could use reflective spectrophotometry to obtain a number for the black/white-ness, a number for the red/green-ness, and a number for the blue/yellow-ness. One approach gives us a set of possible categories, the other gives us numbers.

Type	Description	Example
Categorical	Variables with a discrete number of response options Binary data is a special case with only 2 possible values	Species: <i>Human, Droid, Wookie, Hutt</i> , ...Is_Human: <i>Yes, No</i> .
Continuous	Variables which can take any real number value within the specified range of measurement	Height: <i>172, 165.2, 183</i> , ...
Count	Variables which can only take non-negative integer values (0,1,2,3 etc.)	Number_of_movies: <i>5, 6, 7, 4, 5</i> , ...

In R, different types of data get treated differently by functions, and often we need to tell R explicitly what type of data each variable is. We can use some specific functions to both tell and ask R what type some data are:

Type	Set as...	Check is...
Categorical	<code>as.factor()</code> <code>factor()</code>	<code>is.factor(variable)</code>
Continuous	<code>as.numeric()</code>	<code>is.numeric(variable)</code>
Character	<code>as.character()</code>	<code>is.character(variable)</code>

We can check whether variables of a certain class by using functions such as:

```
is.character(starwars$name)
```

```
[1] TRUE
```

```
is.numeric(starwars$height)
```

```
[1] TRUE
```

```
is.factor(starwars$species)
```

```
[1] FALSE
```

Alternatively, we can also use the function `class()`:

```
class(starwars$height)
```

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

And we can modify the class of a variable by following the same syntax as we modified variables earlier, and using functions such as `as.factor()`, `as.numeric()`:

```
# overwrite the "species" variable with a 'factorised' "species" variable:
starwars$species <- as.factor(starwars$species)
```

Check that it is now a factor:

```
class(starwars$species)
```

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

Factors have certain **levels** that values can take:

```
levels(starwars$species)
```

```
[1] "Aleena"      "Besalisk"    "Cerean"      "Chagrian"    "Clawdite"    "Droid"
[8] "Ewok"        "Geonosian"   "Human"       "Hutt"        "Iktotchi"    "Kalees"
[15] "Kel Dor"     "Mirialan"    "Mon Calamari" "Muun"        "Nabooian"    "Nautoli"
[22] "Pau'an"      "Quermian"    "Rodian"      "Skakoan"     "Sullustan"   "Tholovian"
[29] "Toong"       "Toydarian"   "Trandoshan"  "Twi'lek"     "unknown"     "Wookiee"
[36] "Zabrak"
```

Levels

In categorical data, each case has a value which is one of a fixed set of possibilities. The set of possibilities are the **levels** of a categorical variable.

If we were to try and set an entry as a value which is not one of those levels, it won't work:

```
# set the 1st row, 6th column (the species variable) to be "Peppapig"
starwars[1, 6] <- "Peppapig"
```

invalid factor level, NA generated

One useful function which treats different classes of variable differently, is `summary()`.

Notice how the output differs between variables such as *height*, which we know is numeric, *species*, which we just set to be categorical, and *homeworlds* which is currently character (just text):

```
summary(starwars)
```

name	height	hair_color	eye_color	homeworld
Length:74	Min. : 0.79	Length:74	Length:74	Length:74
Class :character	1st Qu.: 1.70	Class :character	Class :character	Class :character
Mode :character	Median : 1.80	Mode :character	Mode :character	Mode :character
	Mean : 3.02			
	3rd Qu.: 1.91			
	Max. : 94.20			

1.4 Exercises

Question 1 Open a new Rmarkdown document.

File > New File > R Markdown..

Question 2 In your first code-chunk, load the *tidyverse* packages with the following command:

```
library(tidyverse)
```

Make sure you run the chunk.

For the exercises, we have a dataset on the most popular internet passwords, their strength, and how long it took for an algorithm to crack it.

```
pwords <- read_csv("https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2020/01/01/data.csv")
rename(cracked = "value", type="category")

tibble(
  variable = names(pwords),
  description = c("popularity in their database of released passwords",
                  "password", "category of password", "Time to crack by online guessing", "Strength")
) %>% knitr::kable()
```

variable	description
rank	popularity in their database of released passwords
password	password
type	category of password
cracked	Time to crack by online guessing
strength	Strength = quality of password where 10 is highest, 1 is lowest

The data is available online at link.

If you click on that link, you will notice that the values are separated by commas. You may also notice that the url ends with **.csv**. This means we can use the **read_csv()** function to read it into R.

Question Read in the data from the link above.

Be sure to assign it a name, otherwise it will just print it out, and not store it in R's environment!

Solution

```
pwords <- read_csv("....")
```

Question Look at the 90th entry of the data, using the square brackets.

Solution

```
# from the pwords data, show the 90th row, and all columns
pwords[90, ]
```

```
# A tibble: 1 x 5
  rank password type   cracked strength
  <dbl> <chr>    <chr>    <dbl>    <dbl>
1    90 bigdog  animal    3.72      7
```

Question Show the 1st to 20th rows, and the *password* variable.

Solution These will all do the same thing:

```
pwords[c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20), "password"]
pwords[1:20, "password"]
pwords[1:20, 2]
```

```
# A tibble: 20 x 1
  password
  <chr>
1 password
2 123456
3 12345678
4 1234
5 qwerty
6 12345
7 dragon
8 baseball
9 football
10 letmein
11 monkey
12 696969
13 abc123
14 mustang
15 michael
16 shadow
17 master
18 jennifer
19 111111
```

20 2000

Question Is the *type* variable being treated as categorical by R? Check using the `is.factor()` function.

If it is not, then make it a factor.

Solution

```
is.factor(pwords$type)
```

```
[1] FALSE
```

```
pwords$type <- as.factor(pwords$type)
```

Question What **levels** does the *type* variable have?

Solution

```
levels(pwords$type)
```

```
[1] "animal"           "cool-macho"       "fluffy"           "food"             "na"
[6] "nerdy-pop"        "password-related" "rebellious-rude"  "simple-alphanumeric" "spo"
```

Question Show all the data for passwords categorised as “fluffy”. Assign them to a new object called *fluffy_passwords*

Solution

```
# "In the pwords dataframe, give me all the rows for which the
# condition pwords$type=="fluffy" is TRUE, and give me all the columns."
# assign them as a new object called "fluffy_passwords"
fluffy_passwords <- pwords[pwords$type=="fluffy", ]
```

We can now look at our new object by printing it:

```
fluffy_passwords
```

```
# A tibble: 44 x 5
   rank password type   cracked strength
  <dbl> <chr>    <fct>   <dbl>    <dbl>
1    42 love    fluffy    7.92      6
2    43 sunshine fluffy    6.91      9
3    54 silver  fluffy    3.72      8
4    63 orange  fluffy    3.72      8
5    70 ginger  fluffy    3.72      7
6    73 summer  fluffy    3.72      6
7    83 princess fluffy    6.91      8
```

```

8    88 diamond  fluffy  3.19      8
9    89 yellow   fluffy  3.72      6
10   99 iloveyou fluffy  6.91      9
# ... with 34 more rows

```

1.5 Glossary

- data cleaning
- continuous
- count
- categorical
- factor
- level
- ordinal

Symbol	Description	Example
<code>[]</code>	used to extract the 1st, 2nd, ... i^{th} elements in a set of numbers	<code>myvector[3]</code>
<code>\$</code>	used to extract a named column from a dataframe	<code>mydata\$age_variable</code>

Not | `!` | `!(1==1)` | FALSE Or | `|` | `(1==1)` | `(1==2)` | TRUE And | `&` | `(1==1)` & `(1==2)` | FALSE