

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
 1 Intro to R and Rstudio, and data in R	 9
1.1 Introducing R and Rstudio	9
1.2 Take a breather	18
1.3 Starting a new .Rmd document	20
1.4 Data in R	23
1.5 Compiling a .Rmd document	26
1.6 Checklist for today	26
Glossary	26
 2 Types of data	 29
Learning Objectives	29
2.1 Accessing subsections of data	29
2.2 Editing subsections of data	34
2.3 Types of data	34
2.4 Glossary	37

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>

Chapter 1

Intro to R and 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

1.1 Introducing R and Rstudio

Installing

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

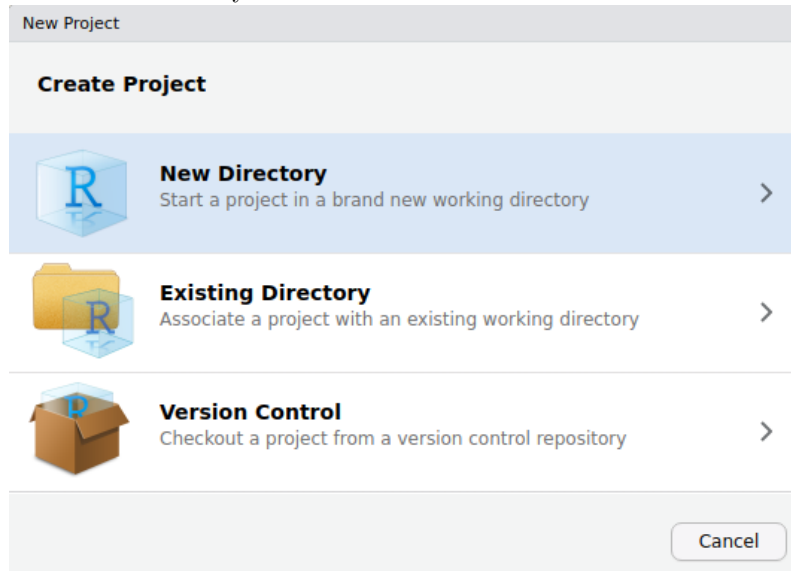
A: Download R and Rstudio onto your computer (recommended)

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

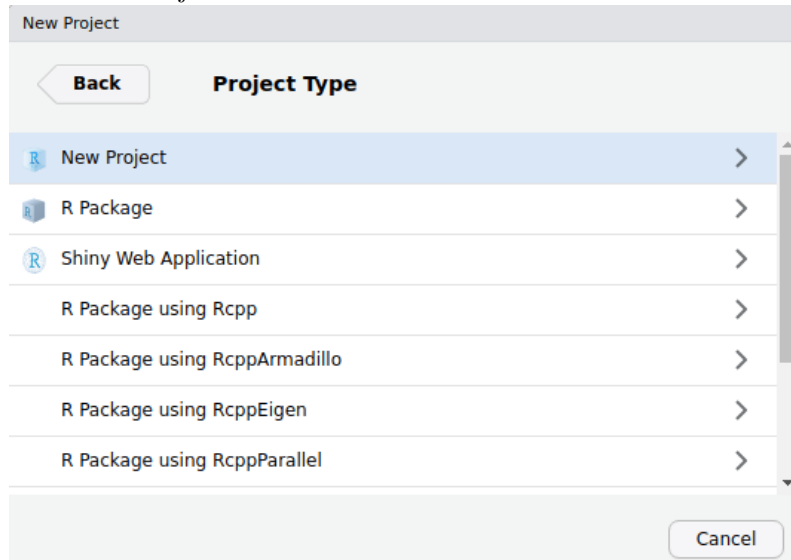
Option A: Installing R and Rstudio (recommended)

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:

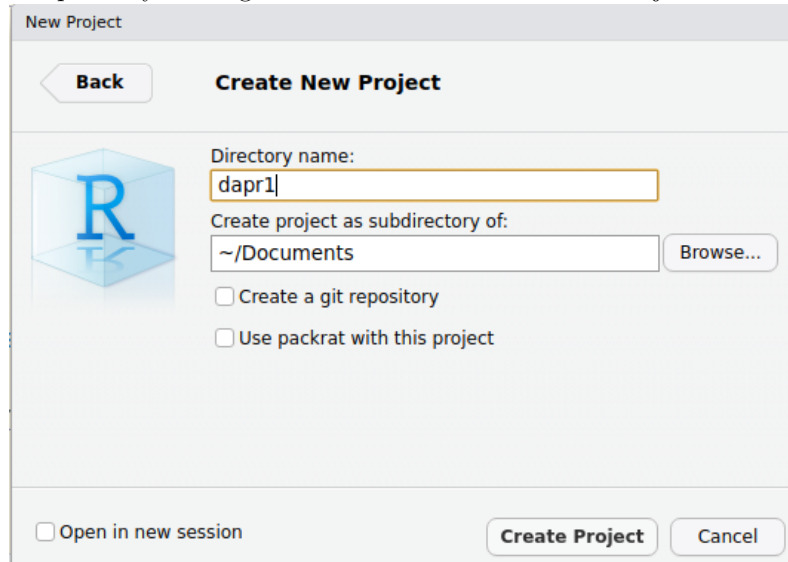


- Click New Project:



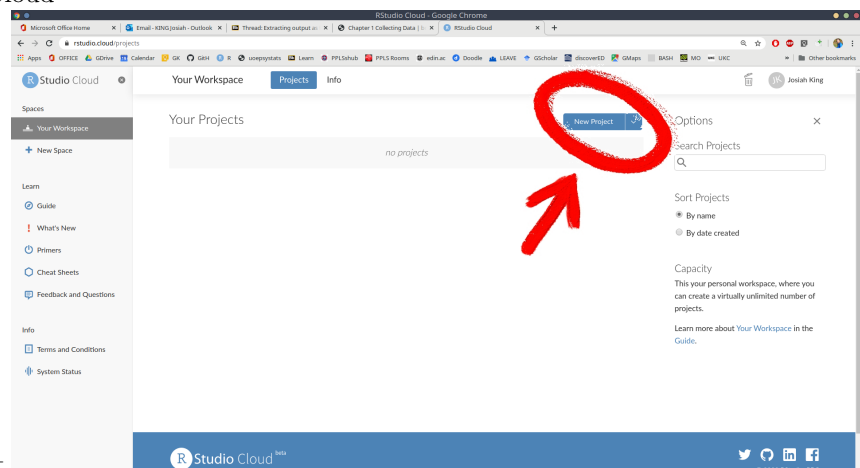
- Name the project, and decide where you want to save it on your

computer by clicking on browse. Then click Create Project:

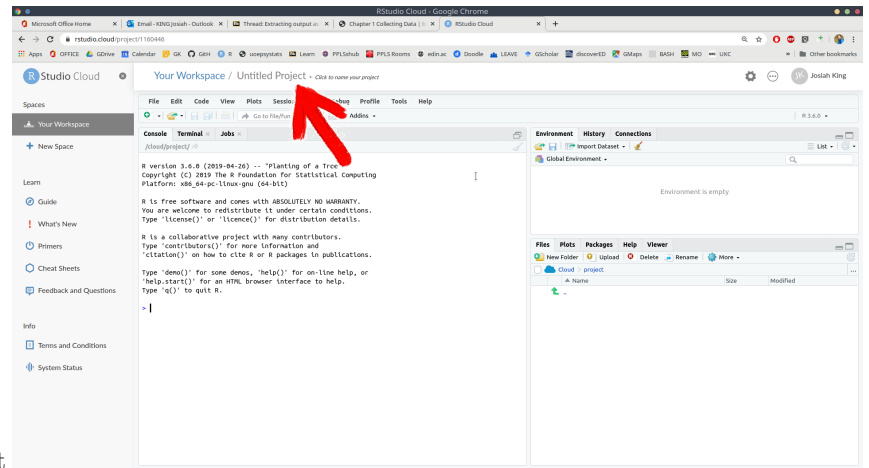


Option B: Rstudio Cloud (for chromebook users)

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



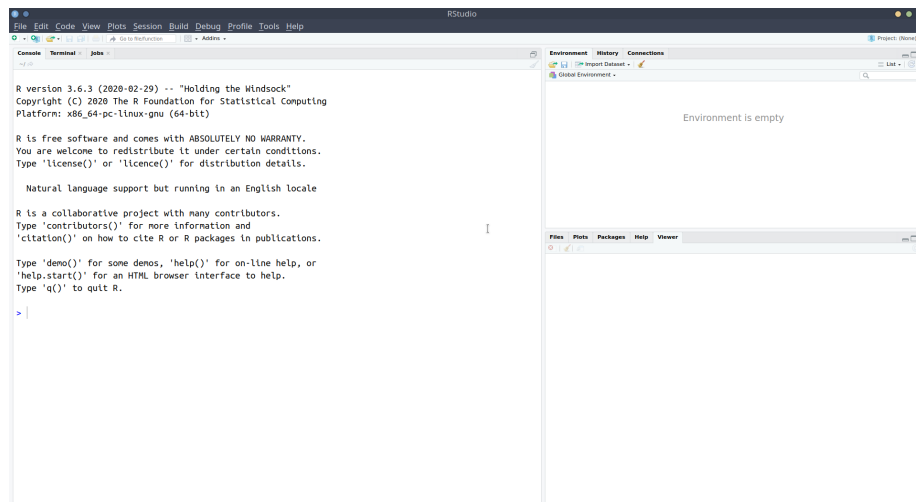
3. Create a new project



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

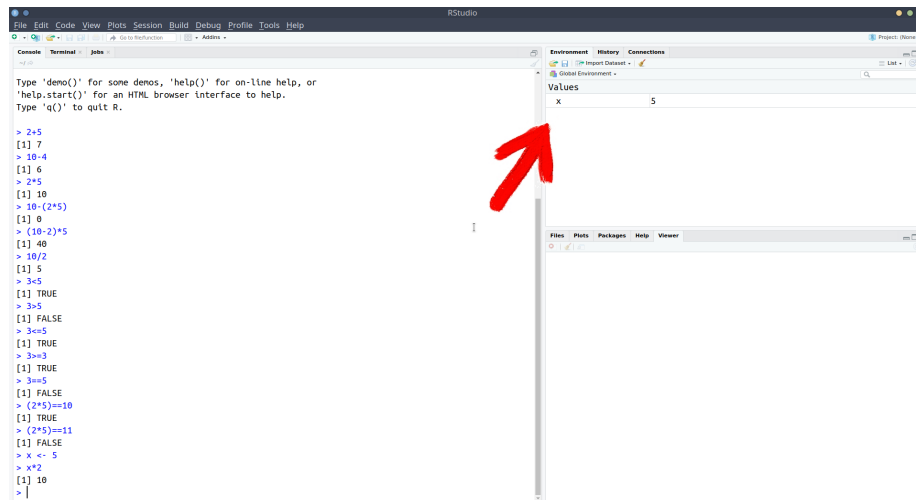
The `<-` symbol is used to *assign* a value to a named object.

`[name] <- [value]`

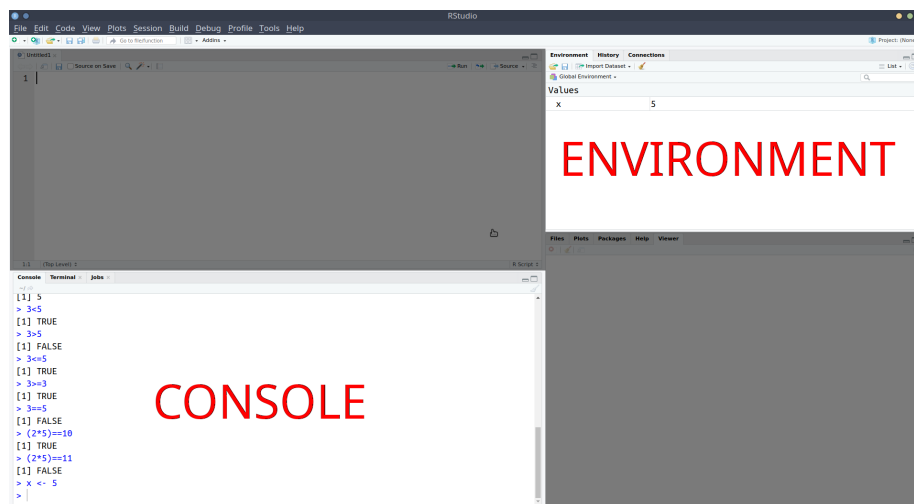
Note, there are a few rules about names in R:

- 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

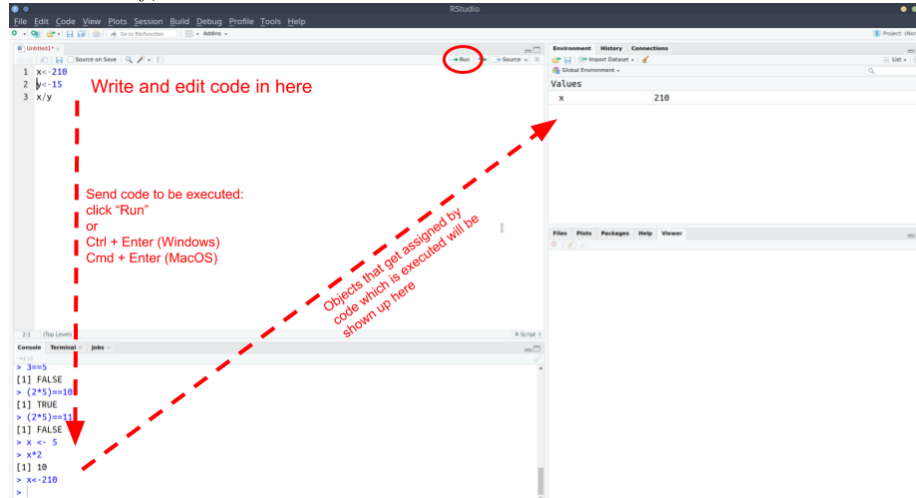
Task 1. Open an R script + **File > New File > R script 1**. Copy and paste the following into the R script

```
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

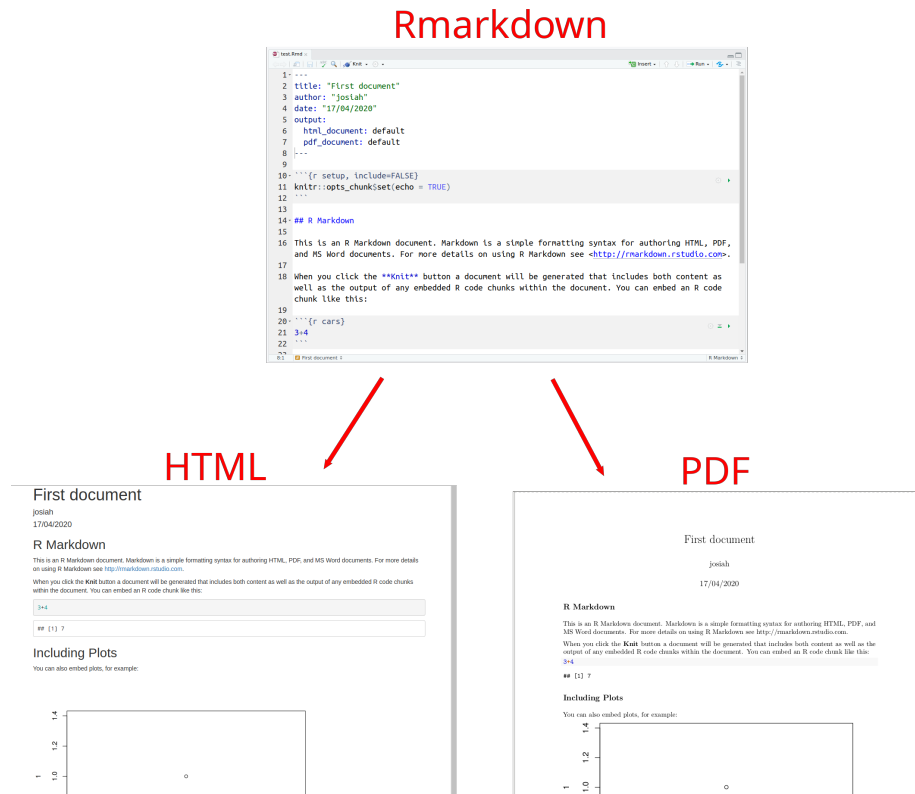
In addition to R scripts, there is another type of document we can create, known as an “Rmarkdown”.

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.

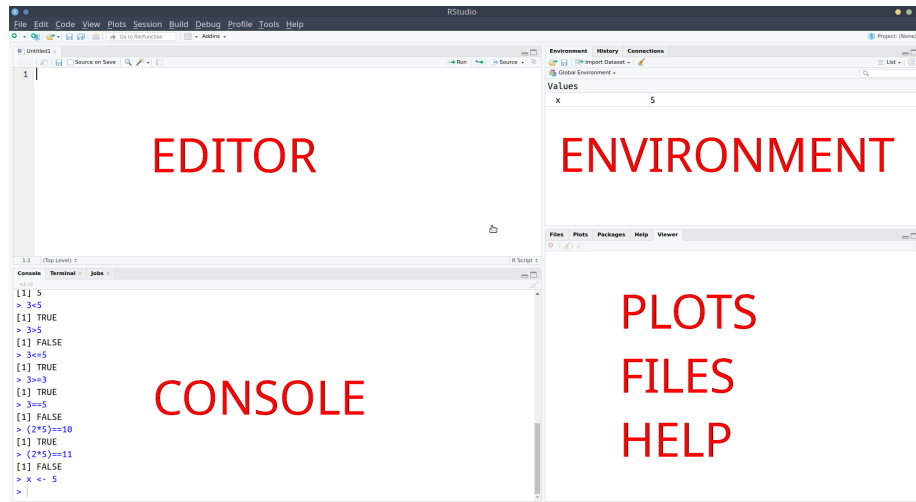
We can then compile the document into either a **.pdf** or a **.html**.



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).



1.2 Take a breather

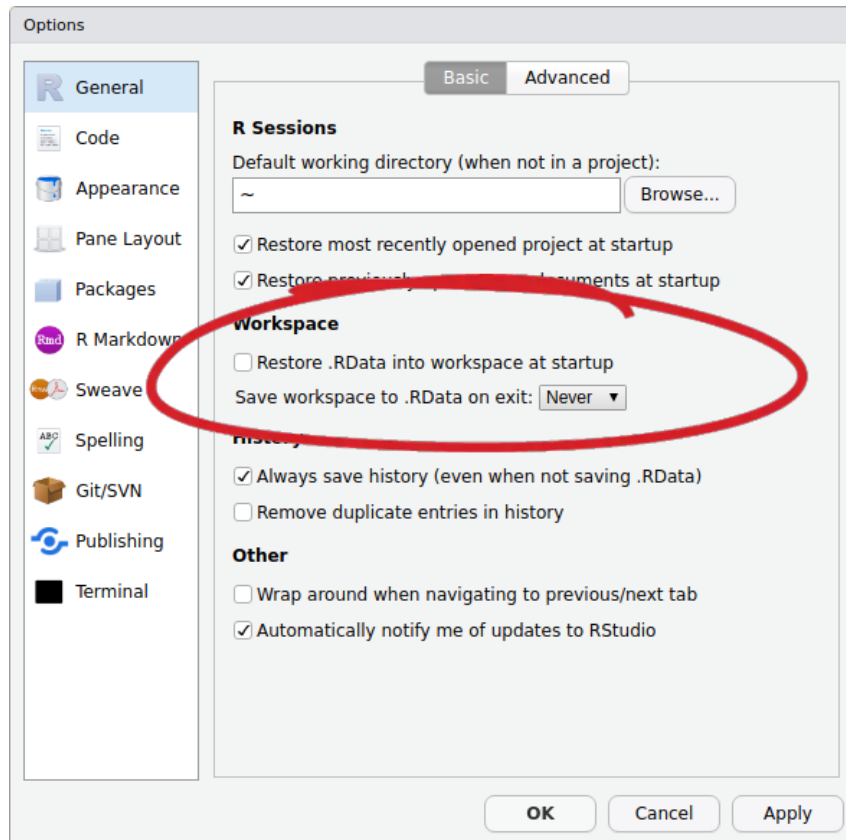
Below are a couple of our recommended settings for you to change as you begin your journey in R. After you've changed them, take a 5 minute break before moving on to learning about how we store data in R.

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”

1.3 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.

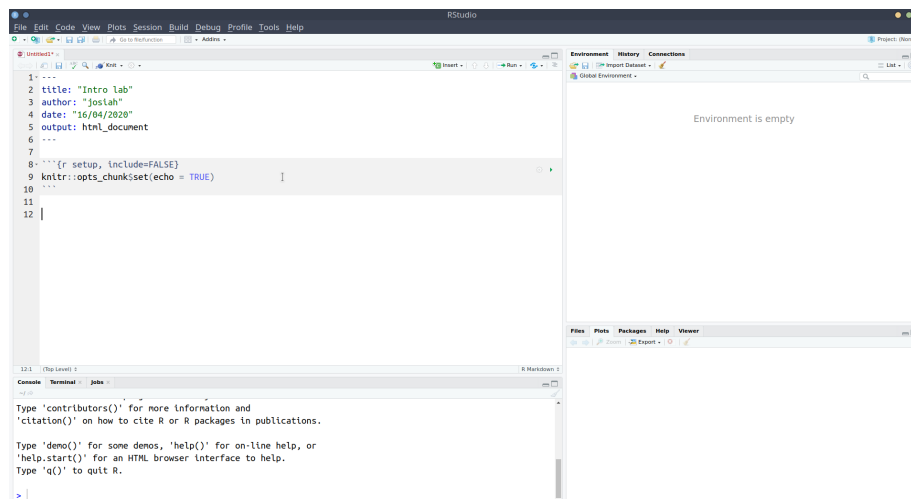
When it has finished, and R is ready for you to use again, you will see the little blue arrow `>`.

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:



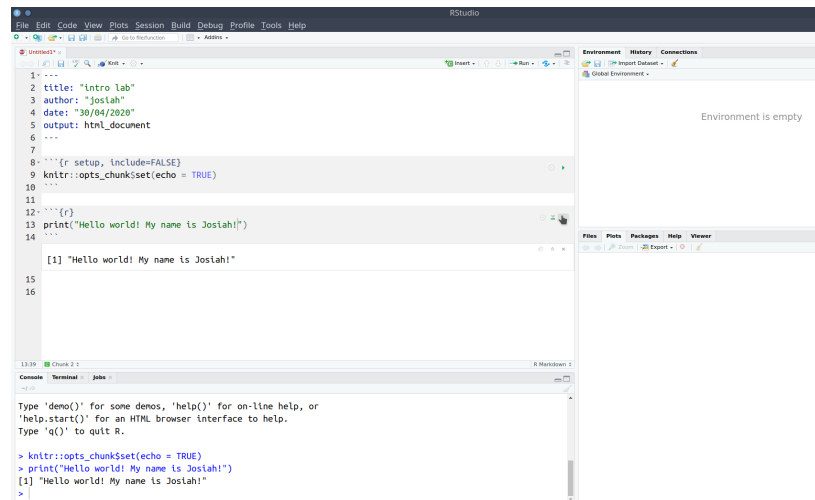
Task Insert a new code chunk by either using the Insert button in the top right of the document, and clicking R, or typing `Ctrl + Alt + i`

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!"

Environment: History: Connections: Global Environment

Files: Plots: Packages: Help: Viewer

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

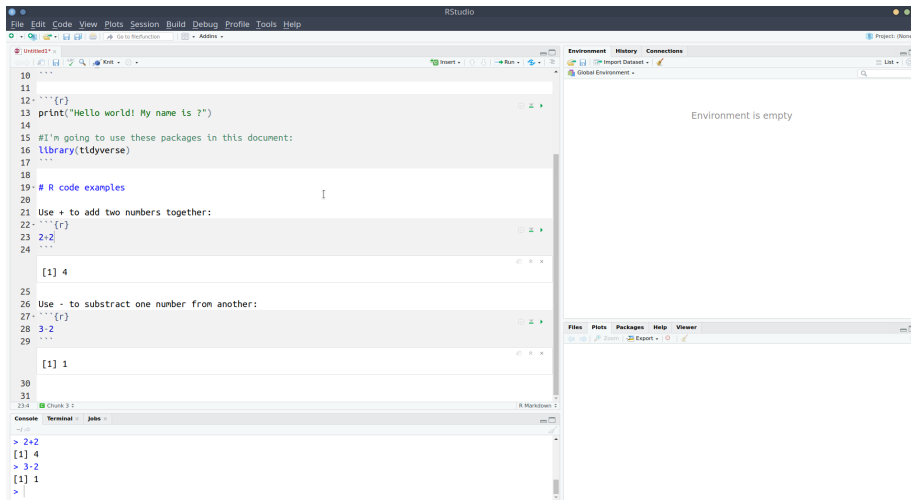
Header 5

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.

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

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



1.4 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:

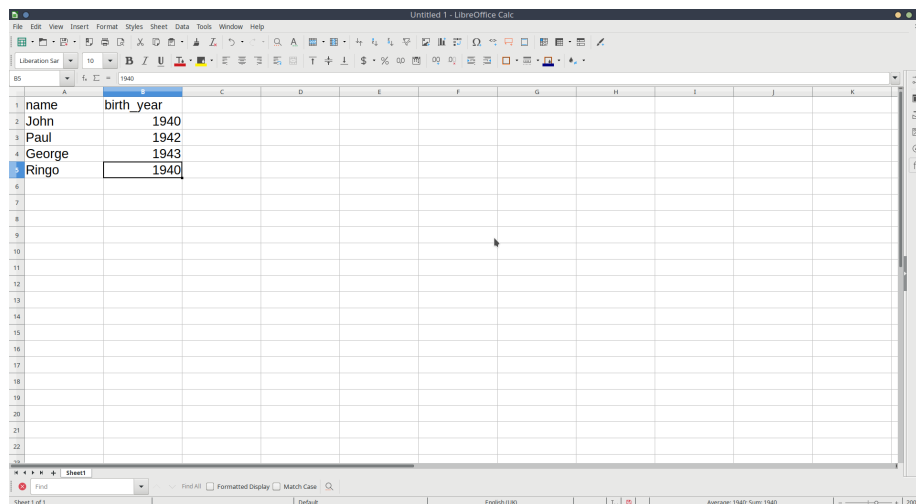
```
# Reading and storing data
```

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.

1.5 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**.

1.6 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)

Chapter 2

Types of data

Learning Objectives

- LO1: Learn how to access variables in a dataframe in R
- LO2: Understand the distinction between types of variables
- LO3: Learn how to code data as different types (classes) in R

So we've got R and Rstudio installed, and we know how to read in data from an external source, and do some basic arithmetic operations.

We're going to now take a look at how to pull out specific pieces of data, before we move on to talking about different types of data.

2.1 Accessing subsections of data

Suppose we read some data in to R:

```
some_data<-dplyr::starwars %>% select(-mass, -skin_color, -films, -vehicles,-starships, -gender,-
some_data[87,]<-c("Homer Simpson",180,NA,NA,"Springfield","unknown")
some_data[2,"species"]<-"Human"
some_data[7,"height"]<-9420
```

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:

```
some_data
```

```
# A tibble: 87 x 6
  name          height hair_color eye_color homeworld species
<chr>         <chr>   <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 77 more rows
```

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(some_data)
```

```
[1] 87 6
```

There's a reasonable amount of data in there - 87 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:

```
some_data$eye_color
```

```
[1] "blue"      "yellow"    "red"       "yellow"    "brown"     "1"
[8] "red"       "brown"     "blue-gray" "blue"      "blue"      "1"
[15] "black"     "orange"    "hazel"     "blue"      "brown"     "3"
[22] "red"       "red"       "brown"     "blue"      "orange"    "1"
```

[29]	"brown"	"black"	"blue"	"red"	"blue"	"orange"
[36]	"orange"	"blue"	"yellow"	"orange"	"brown"	"brown"
[43]	"pink"	"hazel"	"yellow"	"black"	"orange"	"brown"
[50]	"black"	"brown"	"blue"	"orange"	"yellow"	"black"
[57]	"brown"	"brown"	"blue"	"yellow"	"blue"	"blue"
[64]	"brown"	"brown"	"brown"	"yellow"	"yellow"	"black"
[71]	"blue"	"unknown"	"red, blue"	"unknown"	"gold"	"black"
[78]	"blue"	"brown"	"white"	"black"	"dark"	"hazel"
[85]	"black"	"unknown"	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:
some_data[1,4]
```

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

```
# tenth row, first column:
some_data[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:
```

```
some_data[10, ]
```

```
# A tibble: 1 x 6
```

```
  name          height hair_color   eye_color homeworld species
  <chr>         <chr>  <chr>      <chr>    <chr>    <chr>
1 Obi-Wan Kenobi 182    auburn, white blue-gray Stewjon   Human
```

```
# all rows, 2nd column:
```

```
some_data[, 2]
```

```
# A tibble: 87 x 1
```

```
  height
  <chr>
```

```
1 172
2 167
3 96
4 202
5 150
6 178
7 9420
8 97
9 183
10 182
```

```
# ... with 77 more rows
```

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

```
# first row, "species" column
```

```
some_data[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:
```

```
some_data[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:
some_data[1:6, ]
```

```
# A tibble: 6 x 6
  name      height hair_color eye_color homeworld species
<chr>      <chr>  <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
some_data$name[20]
```

```
[1] "Palpatine"
```

Note: When we do this, we don't have the comma inside the square brackets. When we use the `$` to pull out a variable, such as `some_data$name`, we no longer have a dataframe - `some_data$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]`
-

2.2 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

single entries `[]<-`

Changing a whole column

change heights from centimeters to meters `$ <`

Removing rows/columns

editing the whole data - e.g., removing a row

think of it as “overwriting”

2.3 Types of data

table - descriptions from lectures

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> , ...

Type	Description	Example
Count	Variables which can only take non-negative integer values (0,1,2,3 etc.)	Number_of_limbs_lost: 1, 0, 0, 1, 4, ...

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.

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>

```
distinct(some_data, species)
```

```
# A tibble: 39 x 1
```

```
  species
```

```
<chr>
```

```
1 Human
```

```
2 Droid
```

```
3 Wookiee
```

```
4 Rodian
```

```
5 Hutt
```

```
6 Yoda's species
```

```
7 Trandoshan
```

```
8 Mon Calamari
```

```
9 Ewok
```

```
10 Sullustan
```

```
# ... with 29 more rows
```

```
some_data <- read_csv("https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/
  rename(cracked = "value")
```

```
tibble(
```

```
  variable = names(some_data),
```

```
  description = c("popularity in their database of released passwords",
```

```
                  "password", "category", "Time to crack by online guessing", "Strength = quality
```

```
) %>% knitr::kable()
```

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

Task In a new code chunk, do the following:

1. store the following numbers as an object in R:
4,7,3,1,8,9,5,2,2,6,9,9,5,20
2. Try using the function `sum()`, with the name of your object inside the brackets.

Solution Because this is just a set of numbers, we store it as a **vector**, using `c()`.

We have named it `myvec`, but you can call yours whatever you like.

The `sum()` function will add all of the numbers together!

```
myvec <- c(4,7,3,1,8,9,5,2,2,6,9,9,5,20)
sum(myvec)
```

```
[1] 90
```

Task Using the square brackets - `[]` - pull out the 2nd, 4th and 6th values in the object you just created.

Hint: You will need to put inside the square brackets a *sequence of* numbers. How do we combine numbers in to a sequence in R? using `c()`!

Solution

```
myvec[c(2,4,6)]
```

```
[1] 7 1 9
```

Task Using the square brackets, show the 167th row, with all columns.

Remember: When you are using `[]` with a dataframe, you specify `data[rows, columns]`. If you leave either rows or columns blank it will give all of them - for instance, `data[, columns]` will give you all rows for some specified columns.

Solution

```
mydata[167,]
```

Matrices

We will often have several vectors in which the 1st value of each vector corresponds to the same observation, for instance:

```
age<-c(1,2,3,4,5)
height<-c(20,70,80,90,95)
```

But these would be better stored as a “Matrix”

```
matrix(c(1,2,3,4,5,20,70,80,90,95), nrow=5)
```

	[,1]	[,2]
[1,]	1	20
[2,]	2	70
[3,]	3	80
[4,]	4	90
[5,]	5	95

2.4 Glossary

- data cleaning

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

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