R: A first look

Course notes

Alex Homer, Srinivasa Rao

Michaelmas Term 2021

Introduction and basics

These are course notes from the course *R: A first look* as delivered in Michaelmas Term 2021. They mostly duplicate the exercise files, but are hopefully easier to read as a reference in this format. They are still, however, designed to be used in conjunction with the slides, so will not duplicate content from there.

If you're using this file without the worksheets, you'll need to type any code you see into a blank document in the editor panel. More on the editor in a moment!

RStudio

In this section we'll talk about the four panels of the standard RStudio environment. You'll only see three panels if you haven't yet opened an R code file: you can do that from the Files tab of the pane at bottom-right.

The editor

The top-left panel is for editing code, and in general, when you write R code, you'll want to write it there. You can run a line of code by positioning the cursor in that line, and pressing Ctrl + Enter (on Windows) or Cmd + Enter (on Mac) on your keyboard. When you do, you should see the line of code appear in blue in the pane below, followed by (possibly) some more text in black. We'll come back to that pane in a moment.

Any line that starts with a hash (#) symbol is a "comment"; the hash tells the computer not to interpret that line as R code. In the worksheets for this course, most lines are comments, but in a normal R document, most lines would be lines of code. You would mainly use comments to annotate your code, like this:

```r
1+2 #Add one and two together
```

```
## [1] 3
```

```r
3-4 #Subtract four from three
```

```
## [1] -1
```

(When we see R code in this document, we'll also see the output of that code immediately afterwards, unless otherwise stated.) On these lines, the computer ignores everything after the # symbol.

If you want to run multiple lines of code at once, you can highlight them all by clicking and dragging as you would in any text editor, and then pressing Ctrl/Cmd + Enter. But beware! If you incompletely highlight a line, you'll get an error. The easiest way to avoid this is to drag down the numbers on the left-hand side, so you'll always get complete lines of code.

The editor panel has tabs at the top for different documents. You can switch between documents and R will remember the code that's already been run: opening a new document doesn't start a new session.

The console

When you ran code in the editor, it was sent down to the console below. Any line that starts with a `>` or a `+` is the code that you've input. These lines are usually in blue. The output from the computer is in black, and often starts with `[1]`. Don't worry too much yet about what the `[1]` means.

You can type code directly into the console, by clicking on it, typing your code, and pressing Enter. Normally you don't want to do this, because it's a pain to re-run code, and if you make a mistake when typing complex code over multiple lines, you have to start typing the whole thing again. However, for simple things like quick calculations or looking up help, you might want to do this. (We'll come on to how to do those things shortly!)

This panel has tabs at the top, but we won't be using any of the others: just "Console".

The bottom-right panel

This panel has various tabs at the top, which do quite different things. The ones we'll be using are:

- Files, which, on RStudio Cloud, shows the uploaded files. (On your computer, it replicates Windows Explorer/Finder [on Mac].)
- Plots, which shows plots that we've made.
- Help, which shows you help files.

The top-right panel

This last panel also has tabs. We won't really use any of them, but the Environment tab is most useful. However, I can't explain what it does until we've got a bit further! For now, just note that it says "Environment is empty".

Resizing panels

You can resize the panels if you need to. Hover your mouse over the division between them, and it will turn into a four-pointed arrrow. You can then click and drag to change the sizes.

Working directory

Before we go any further, we need to set the working directory. To do this, navigate in the Files pane until the course files ("1-intro.R" and so on) are visible. Then, in the menus at the top of RStudio, go to Session > Set Working Directory > To Files Pane Location.

The basics

We've already seen a couple of basic R functions, for addition and subtraction. To complete the set of R-ithmetical functions, let's do multiplication, division and raising a number to a power. Try running the following lines of code.

```r
3*5 #Multiplication
```

```
## [1] 15
```

```r
6/3 #Division
```

```
## [1] 2
```

```r
2^3 #Powers
```

```
## [1] 8
```

We can also run mathematical functions: the syntax for these is

```
functionname(input)
```

Here are a few examples

```r
sqrt(2) #Square root
```

```
## [1] 1.414214
```

```r
sum(1,4,5,100) #Sum
```

```
## [1] 110
```

```r
prod(2,3,5,6) #Product (multiply together)
```

```
## [1] 180
```

So a function is written as a line of text, then round brackets, then zero or more "arguments", separated by commas. (Yes, I said "zero or more"! Some functions don't have arguments, or have arguments all of which are optional. But you still have to put the brackets in, or they don't work. For instance:

```r
citation()
```

```
##
## To cite R in publications use:
##
##   R Core Team (2021). R: A language and environment for statistical
##   computing. R Foundation for Statistical Computing, Vienna, Austria.
##   URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2021},
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please cite it
## when using it for data analysis. See also 'citation("pkgname")' for
## citing R packages.
```

which tells you how to cite R in your work. Please do: the authors of R were mostly volunteers, so citations are the only reward they get!)

If you don't know what a function does, you can type `?`, then it's name. This is the sort of line you would normally just type into the console: you don't want to keep a permanent record of your request for help! Try typing `?sqrt` into the console, and seeing what happens.

So far we've not seen much that R can do that a scientific calculator can't. Let's start now, by running the following lines of code.

```r
a <- 3 #Define a
a^2 #Square it
```

```
## [1] 9
```

What's going on here? The `<-` is the "assignment operator". (You can also use `=` for this purpose, but `<-` is more conventional in R.) It creates a variable called `a`, to which the value 3 is assigned. Notice that the top-right pane isn't empty any more: it shows `a`, and its value. Then in the next line, we squared it, and returned the result: 9.

There's no special key for `<-`, by the way. You just type it as two symbols: a left angle bracket (usually Shift+comma, on UK Windows and Mac keyboards), then a hyphen (on UK keyboards, usually the key to the right of 0).

The name `a` was arbitrary: we can choose almost anything we like as a variable name. But it's best to avoid things that are (or might be) names of functions, like `mean` or `sum`. It's also best to avoid things like `if` and `for`: we're not going to talk about what these do in this course, but they also have special meanings.

When R is assigning a value, its default behaviour is not to print anything (by "print", we mean return an output in the console: R will never send anything to your physical printer unless you tell it to!). For instance

```
bee <- a^3
```

assigns the value of a^3 (27) to the variable bee, but doesn't print anything. To assign and print at the same time, you can put brackets around the whole expression, like so:

```
(bee <- a^3)
```

```
## [1] 27
```

You can also find out the value of a variable by typing its name into the console, and pressing Enter. Try that now with `a` and `bee`.

Exercise 1

Write R code that defines a variable called "lucky.number" whose value is your lucky number, then prints the value of half your lucky number.

Vectors

We said R was a "statistical programming language". That means it's built to deal well with data. But we've just dealt with individual numbers so far.

The most basic data structure is a vector. We can input vectors to R using a special function, `c` (which stands for "concatenate"). Let's try it.

```
vec.1 <- c(1,4,3)
vec.2 <- c(4,2,2)
```

You'll see them appear top-right.

When you do any of the arithmetic operations on two vectors together, R does the operation element-by-element. Try:

```
vec.1 + vec.2 #Element-wise addition
```

```
## [1] 5 6 5
```

```
vec.2^vec.1 #Element-wise powers
```

```
## [1]  4 16  8
```

(Note that R ignores spaces when processing code.) This is convenient for dealing with data. For instance, if your data were heights and weights of people:

```
heights <- c(1.56, 1.62, 1.8, 1.75) #heights in metres
weights <- c(52, 65, 80, 78) #weights in kg
```

we could calculate BMI using

```
weights/heights^2
```

```
## [1] 21.36752 24.76757 24.69136 25.46939
```

However, if your vectors have different lengths, R might exhibit some strange behaviour, and it won't necessarily give an error, so be careful!

You can also do operations with vectors and constants:

```
a <- 3
vec.1 <- c(1,4,3)
a*vec.1
```

```
## [1]  3 12  9
```

We can have vectors of any type of data: for instance, by using quotation marks, we can make a vector of text strings:

```
text <- c("This", "is", "a", 'vector')
```

(There's no difference between using single and double quotes, as long as you close the text with the same type that you used to open it!)

A special kind of vector you might use is a list of numbers in sequence. We can get that using a colon, like so:

```
(naturals <- 1:50)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

This vector doesn't fit on one line in the output. Notice that the second line starts with a number other than 1 in the square brackets (here, `[26]`, but it will depend on the width of your screen): that's telling me that the first entry on the second line is the 26th entry overall. This explains why you kept seeing `[1]` before: R technically treats numbers as vectors of length 1, and is telling you that it's displaying the first (and only!) entry.

One last note on colons: be careful when operations are involved. For instance, notice the difference between the second and third lines below:

```
n <- 5
1:n-1
```

```
## [1] 0 1 2 3 4
```

```
1:(n-1)
```

```
## [1] 1 2 3 4
```

In the first line, R generates the sequence "1, 2, 3, 4, 5", and then subtracts the constant 1 from the whole sequence. In the second, R does the operation `n-1` first, and then generates the sequence from 1 to 4.

> **Exercise 2**
>
> Write R code that outputs the sequence from square numbers from $1^2 = 1$ to $20^2 = 400$. (Hint: first generate the sequence `1:20`.)
>
> **Exercise 2A** (optional, involves maths)
>
> Do the same with the first 20 triangle numbers: the kth triangle number is $k(k+1)/2$.

One final thing: you can select part of a vector by using square brackets. For instance,

```
vec.1[2]
```

```
## [1] 4
```

returns the second entry in `vec.1`. To select a range of entries, we can put another vector inside the square brackets. For instance:

```
heights[2:4]
```

```
## [1] 1.62 1.80 1.75
```

selects the second, third and fourth entries of `heights`, and

```
weights[c(1,4)]
```

```
## [1] 52 78
```

selects the first and fourth entries of weights. We'll see more of this sort of thing in the next section.

Data management

Data frames

Apologies if the first worksheet seemed like a load of abstract nonsense. Partly, that's because it was! But we needed some theoretical grounding so that the rest of this made sense. From now on, we're going to be dealing exclusively with data.

We're going to use a dataset called `mtcars`, which is a standard sample dataset about cars.

```
cars_dataset <- read.csv("Data/mtcars.csv")
```

(This line will only work if you correctly set the working directory earlier.) We're loading a CSV file, where "CSV" stands for "comma-separated value". You can look at it if you like by clicking on the Data folder in the Files pane at bottom-right, then going to "mtcars.csv". If you do that, you'll see that it's made of of data values separated by commas, as the name suggests. CSV files are a common way to share data so multiple apps and languages can use it. However, you have to be a little bit careful when you import them that they import correctly. There are some notes about this on the slides.

We've given the imported data the name `cars_dataset`, as a "data frame", which essentially means "table"—but in the context of R a "table" is something else (that we won't see much of today). `mtcars` is an example of a data frame, and we've stored it under the name `cars_dataset`. As ever, `cars_dataset` is arbitrary: we could have called it `sheep` if we'd wanted to, but that wouldn't be very wise… your future self would be really confused when they see a cars dataset named 'sheep'!

> **Exercise 2B** (optional)
>
> Why can't we just run the following to import the data? What does it do?

```
read.csv("Data/mtcars.csv")
```

> No need to write down your answer! (And don't worry too much if you're not sure, just move on; the answer is on the slides.)

Data frames are typically quite long, so we don't usually want to display all of them at once. To see the top six lines, we can use the function "head". This is also sensible after importing data, to check that it's imported correctly.

```
head(cars_dataset)
```

```
##                 Car  mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1         Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2     Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3        Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## 4    Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## 5 Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## 6           Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Now we can see the structure of a data frame. It has columns corresponding to variables, and rows corresponding to observations. They should *always* be this way around! (It's good practice to do this in other applications too, like Excel, though it's less-strictly enforced there.) One reason they have to be this way around in R is that a data frame must (apart from possibly its header) contain the same type of data all the way down: so all integers (whole numbers), or "characters" (text) or "doubles" (numbers with decimal points in them), or whatever. Rows, meanwhile, can contain different types of data, as shown in our data frame.

(You can skip this paragraph if you like, but to explain where the names come from: "integer" is a term from maths, from the Latin word "integer" meaning "whole". "Character" can normally mean a letter or number, so the term is reused to represent strings of text. "Double" is short for "double-precision floating-point number", and refers to how R stores numbers like this "under the hood"—there is also a single-precision version, but it's rare that you'd need to use those in R.)

In fact, mtcars is a standard sample dataset—you can access it by using the command `mtcars`, but for slightly technical reasons (as well as to demonstrate the technique) we're loading it from a CSV file. The fact that it's standard sample data means that, if you want to find out more about it, you can type `?mtcars` into the console. This doesn't work for just any dataset, though!

Subsetting

There are various basic operations you may like to do with data. For instance, you may like to take only a certain row, or a certain column. You can do this with square brackets, to select the number(s) of the row(s) or column(s) you want. Let's take a few examples.

```
cars_dataset[3,]
```

```
##          Car  mpg cyl disp hp drat   wt  qsec vs am gear carb
## 3 Datsun 710 22.8   4  108 93 3.85 2.32 18.61  1  1    4    1
```

```
cars_dataset[,4]
```

```
##  [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8
## [13] 275.8 275.8 472.0 460.0 440.0  78.7  75.7  71.1 120.1 318.0 304.0 350.0
## [25] 400.0  79.0 120.3  95.1 351.0 145.0 301.0 121.0
```

```
cars_dataset[3,4]
```

```
## [1] 108
```

So `cars_dataset[3,]` selects the third row, `cars_dataset[,4]` selects the fourth column, and `cars_dataset[3,4]` selects the fourth entry in the third row (or, equivalently, the third entry in the fourth column). Notice that `cars_dataset[,4]` doesn't appear as a column: when you select a single column from a data frame, R converts it into a vector for you.

We can also select multiple rows, by giving R a vector of the row numbers we want. For instance

```
cars_dataset[c(1,4,6),]
```

```
##              Car  mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1      Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 4 Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## 6        Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

will return the 1st, 4th and 6th rows. (Notice that you still need to type the comma, even if you're not selecting any columns—otherwise R doesn't know whether you're trying to select columns or rows, and will give an error.)

Exercise 3

Write R code to select only the 3rd, 9th and 10th rows, and only the 4th, 5th and 6th columns, and store it as a new data frame called `my.cars`.

You can also select columns by name, using the headers. To do this we use the "dollar operator".

```
cars_dataset$cyl
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
cars_dataset$disp[16]
```

```
## [1] 460
```

```
cars_dataset[16,]$disp
```

```
## [1] 460
```

The first operation returns the `cyl` column; the second returns the 16th entry in the `disp` column. The third is another way of writing the second.

Exercise 4 (slightly trickier!)

Write R code to select the 10th to 12th, and 14th, entries from the column `mpg`.

Hopefully, if you type `cars_dataset$` into the editor, it should have provided you with a list of columns to choose from. This is a feature of RStudio that you don't get by using R alone.

What if you wanted to select multiple columns by name? R can do this too.

```
cars_dataset[, c("mpg", "disp")]
```

```
##     mpg  disp
## 1  21.0 160.0
## 2  21.0 160.0
## 3  22.8 108.0
## 4  21.4 258.0
## 5  18.7 360.0
## 6  18.1 225.0
## 7  14.3 360.0
## 8  24.4 146.7
## 9  22.8 140.8
## 10 19.2 167.6
## 11 17.8 167.6
## 12 16.4 275.8
## 13 17.3 275.8
## 14 15.2 275.8
## 15 10.4 472.0
## 16 10.4 460.0
## 17 14.7 440.0
## 18 32.4  78.7
## 19 30.4  75.7
## 20 33.9  71.1
## 21 21.5 120.1
## 22 15.5 318.0
## 23 15.2 304.0
## 24 13.3 350.0
## 25 19.2 400.0
## 26 27.3  79.0
## 27 26.0 120.3
## 28 30.4  95.1
## 29 15.8 351.0
## 30 19.7 145.0
## 31 15.0 301.0
## 32 21.4 121.0
```

How about if you wanted to select everything *except* a certain row or column? You can do that using the minus sign, but only with the numbers method:

```
cars_dataset[-4,]
```

```
##                    Car  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1            Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## 2        Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## 3           Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## 5    Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## 6              Valiant 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## 7           Duster 360 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## 8            Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## 9             Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## 10            Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## 11           Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## 12          Merc 450SE 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## 13          Merc 450SL 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## 14         Merc 450SLC 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## 15  Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## 16 Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## 17   Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## 18            Fiat 128 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## 19         Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## 20      Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## 21       Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## 22    Dodge Challenger 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## 23         AMC Javelin 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## 24          Camaro Z28 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## 25    Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## 26           Fiat X1-9 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## 27       Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## 28        Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## 29      Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## 30        Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## 31       Maserati Bora 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## 32          Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
cars_dataset[,-c(1,5)]
```

```
##     mpg cyl  disp drat    wt  qsec vs am gear carb
## 1  21.0   6 160.0 3.90 2.620 16.46  0  1    4    4
## 2  21.0   6 160.0 3.90 2.875 17.02  0  1    4    4
## 3  22.8   4 108.0 3.85 2.320 18.61  1  1    4    1
## 4  21.4   6 258.0 3.08 3.215 19.44  1  0    3    1
## 5  18.7   8 360.0 3.15 3.440 17.02  0  0    3    2
## 6  18.1   6 225.0 2.76 3.460 20.22  1  0    3    1
## 7  14.3   8 360.0 3.21 3.570 15.84  0  0    3    4
## 8  24.4   4 146.7 3.69 3.190 20.00  1  0    4    2
## 9  22.8   4 140.8 3.92 3.150 22.90  1  0    4    2
## 10 19.2   6 167.6 3.92 3.440 18.30  1  0    4    4
## 11 17.8   6 167.6 3.92 3.440 18.90  1  0    4    4
## 12 16.4   8 275.8 3.07 4.070 17.40  0  0    3    3
## 13 17.3   8 275.8 3.07 3.730 17.60  0  0    3    3
## 14 15.2   8 275.8 3.07 3.780 18.00  0  0    3    3
## 15 10.4   8 472.0 2.93 5.250 17.98  0  0    3    4
## 16 10.4   8 460.0 3.00 5.424 17.82  0  0    3    4
## 17 14.7   8 440.0 3.23 5.345 17.42  0  0    3    4
## 18 32.4   4  78.7 4.08 2.200 19.47  1  1    4    1
## 19 30.4   4  75.7 4.93 1.615 18.52  1  1    4    2
## 20 33.9   4  71.1 4.22 1.835 19.90  1  1    4    1
## 21 21.5   4 120.1 3.70 2.465 20.01  1  0    3    1
## 22 15.5   8 318.0 2.76 3.520 16.87  0  0    3    2
## 23 15.2   8 304.0 3.15 3.435 17.30  0  0    3    2
## 24 13.3   8 350.0 3.73 3.840 15.41  0  0    3    4
## 25 19.2   8 400.0 3.08 3.845 17.05  0  0    3    2
## 26 27.3   4  79.0 4.08 1.935 18.90  1  1    4    1
## 27 26.0   4 120.3 4.43 2.140 16.70  0  1    5    2
## 28 30.4   4  95.1 3.77 1.513 16.90  1  1    5    2
## 29 15.8   8 351.0 4.22 3.170 14.50  0  1    5    4
## 30 19.7   6 145.0 3.62 2.770 15.50  0  1    5    6
## 31 15.0   8 301.0 3.54 3.570 14.60  0  1    5    8
## 32 21.4   4 121.0 4.11 2.780 18.60  1  1    4    2
```

Exercise 5

Write R code to select all but the 16th row, and all but the first three columns, of `cars_dataset`. (Hint: `-1:3` represents the vector $(-1, 0, 1, 2, 3)$. `-(1:3)` represents the vector $(-1, -2, -3)$. Which one do you need?

Note that you can use all of these subsetting operations on vectors instead of data frames. In that case, you omit the comma that separates the row selection from the column selection, and just select one collection of entries—since vectors only have one dimension.

Filtering data

A useful thing to be able to do with R is to select data of a particular type. This is typically referred to as "filtering" the data. For instance, we might like to select all the cars with four gears in our `mtcars` dataset. To do this, we will first need to be able to answer a slightly more fundamental question: which are the numbers of the rows where `gear` equals 4?

```
which(cars_dataset$gear == 4)
```

```
##  [1]  1  2  3  8  9 10 11 18 19 20 26 32
```

```
cars_dataset$gear == 4
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [25] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

Note the `==`. Using only a single equals sign for this is one of the easiest mistakes to make in R (and indeed many other programming languages). The single `=` is used for two things: firstly, as an alternative to `<-`, and secondly, as a way to specify arguments of functions (we haven't seen any examples of this latter case yet). The doubled version, `==`, means "equal to". I apologise for this!

Anyway, both of the above lines answer the question in slightly different ways. The first tells you the numbers of the rows; the second gives a vector of "Boolean" values, `TRUE` or `FALSE`, where (say) the sixth value in the vector is `FALSE` because in the sixth row the car does not have four gears. ("Boolean" is from George Boole

(1815-1864)—he developed Boolean algebra, a way of dealing with True/False values that underpins how computers work.)

Anyway, we can use these to select the rows where the condition is true.

```
cars_dataset[which(cars_dataset$gear == 4),]
```

```
##                 Car  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1         Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## 2     Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## 3        Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## 8         Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## 9          Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## 10         Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## 11        Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## 18         Fiat 128 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## 19      Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## 20 Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## 26        Fiat X1-9 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## 32       Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
cars_dataset[cars_dataset$gear == 4,]
```

```
##                 Car  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1         Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## 2     Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## 3        Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## 8         Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## 9          Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## 10         Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## 11        Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## 18         Fiat 128 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## 19      Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## 20 Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## 26        Fiat X1-9 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## 32       Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

These two rows do the same thing, with the latter being a little more concise. Both, though are a bit clunky! Later we'll see an alternate way of subsetting data.

There are other conditions we can use.

Exercise 6

Run each of the commands below. What do they do? Type your answer as a comment underneath each line. (Hint: type `?mtcars` into the console to see what the columns represent. The column `Car` is not mentioned in the help files, however, as the dataset we loaded from `Data/mtcars.csv` is *slightly* different from the `mtcars` in-built dataset available in R.)

```
cars_dataset[cars_dataset$mpg<15,]
#
cars_dataset[cars_dataset$carb>=6,]
#
cars_dataset[cars_dataset$vs!=0,]
#
cars_dataset[startsWith(cars_dataset$Car, "Merc"),]
#
cars_dataset[cars_dataset$Car >= "T",] #Trickier!
```

```
#
```

Data analysis and visualisation

Simple analysis

This isn't a statistics course, and it doesn't assume any knowledge of statistics, so there's a limit to how much data analysis I can teach you. But we're going to cover some basic techniques, particularly of what's known as "exploratory data analysis": the initial things you do when you first have a dataset.

So what kind of analysis can we do? Let's start with getting some summary statistics.

```
summary(cars_dataset)
```

```
##      Car                 mpg             cyl             disp      
##  Length:32          Min.   :10.40   Min.   :4.000   Min.   : 71.1  
##  Class :character   1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8  
##  Mode  :character   Median :19.20   Median :6.000   Median :196.3  
##                     Mean   :20.09   Mean   :6.188   Mean   :230.7  
##                     3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0  
##                     Max.   :33.90   Max.   :8.000   Max.   :472.0  
##        hp             drat             wt             qsec      
##  Min.   : 52.0   Min.   :2.760   Min.   :1.513   Min.   :14.50  
##  1st Qu.: 96.5   1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89  
##  Median :123.0   Median :3.695   Median :3.325   Median :17.71  
##  Mean   :146.7   Mean   :3.597   Mean   :3.217   Mean   :17.85  
##  3rd Qu.:180.0   3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90  
##  Max.   :335.0   Max.   :4.930   Max.   :5.424   Max.   :22.90  
##        vs               am              gear            carb      
##  Min.   :0.0000   Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4375   Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

`summary` is an R function that can be applied to all sorts of different things. It's worth trying it when you see a new type of object: it's likely that someone's coded an interpretation for the summary function.

Here, it's giving us summary statistics for each of the columns: the minimum, lower quartile, median, mean, upper quartile and maximum. That is, except for the column of characters, where it just tells us what we already know: that it's a column of characters.

What if we wanted the standard deviation of each column? The function we need here is `sd`, and we can use it column-by-column if we wish.

```
sd(cars_dataset$mpg)
```

```
## [1] 6.026948
```

If we want to apply it to multiple columns, we can use the function `apply`.

```
apply(cars_dataset, 2, sd)
```

```
## Warning in var(if (is.vector(x) || is.factor(x)) x else as.double(x), na.rm =
## na.rm): NAs introduced by coercion
```

```
##         Car         mpg         cyl        disp          hp        drat
##          NA   6.0269481   1.7859216 123.9386938  68.5628685   0.5346787
##          wt        qsec          vs          am        gear        carb
##   0.9784574   1.7869432   0.5040161   0.4989909   0.7378041   1.6152000
```

Here `cars_dataset` is the data frame, `2` indicates that we want to apply the function to each column (it's less likely, but if you wanted to apply something to each row you'd put a `1` there instead) and `sd` is the function we want to apply.

You'll probably get a warning message when you run this: although it looks like an error, a warning is subtly different. It doesn't necessarily mean anything's gone wrong, but there is something that R wants you to pay attention to. In this case, it's telling you that it doesn't know how to calculate the standard deviation of one of the columns: the one with the names of the cars. That makes sense! So R has replaced it with `NA`, which R uses to represent a missing value. (It stands for "not available" in this context.)

An aside, while we're on the subject: two similar things you'll see in R are `NaN` and `Inf`. `NaN` means "not a number" (or your grandmother, who is indeed not a number). `NaN` is what you get when you try to calculate:

```
0/0
```

```
## [1] NaN
```

Meanwhile `Inf` means infinity, and is what you get when you try to calculate:

```
1/0
```

```
## [1] Inf
```

These cases are slightly different from just "no data", and you might want to deal with them separately, which is why R gives them alternative names to `NA`. You don't need to worry too much about this, though.

You might not want the ugly red text, though, especially if you were outputting the results of your code into a document automatically (more on that later). To get rid of it, we just need to get rid of the problematic column.

```
cars_dataset.no.names <- cars_dataset[,-1] #Remove first column
apply(cars_dataset.no.names, 2, sd)
```

```
##         mpg         cyl        disp          hp        drat          wt
##   6.0269481   1.7859216 123.9386938  68.5628685   0.5346787   0.9784574
##        qsec          vs          am        gear        carb
##   1.7869432   0.5040161   0.4989909   0.7378041   1.6152000
```

Exercise 7

Write R code to calculate the mean of each column of `cars_dataset.no.names`, using `apply`. (Sometimes it's more convenient to just have the vector of values you need, rather than the full `summary` output.) The function to calculate the mean is called `mean`, as you might expect! Check against the output of `summary` to make sure it's worked.

We can also make new columns by doing operations, just like we did with vectors. For instance, suppose we wanted a new column that recorded each car's horsepower per cylinder. (I don't know much about cars, but this probably isn't all that meaningful in real life!) We can use the dollar operator to define a new column, like this:

```
cars_dataset$hp.per.cyl <- cars_dataset$hp/cars_dataset$cyl
```

Type `head(cars_dataset)` into the console and you'll see (the start of) the new column.

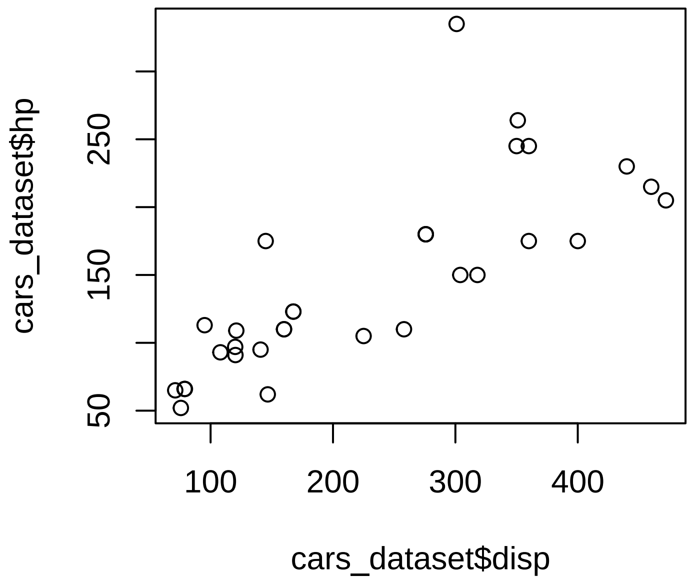
Exercise 8

Add another new column to the table that displays the total number of gears and carburettors each car has. Call it `gear.plus.carb`.

Plots

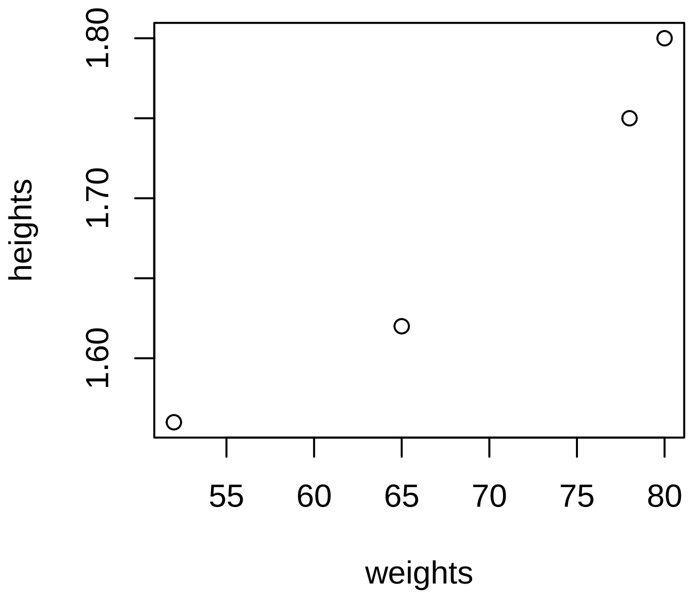
We can't hope to cover all the different types of plot that R can do in one course. But we'll cover a few of the most common types. Let's start with a scatter plot, for which the command is `plot`.

```
plot(cars_dataset$disp, cars_dataset$hp)
```



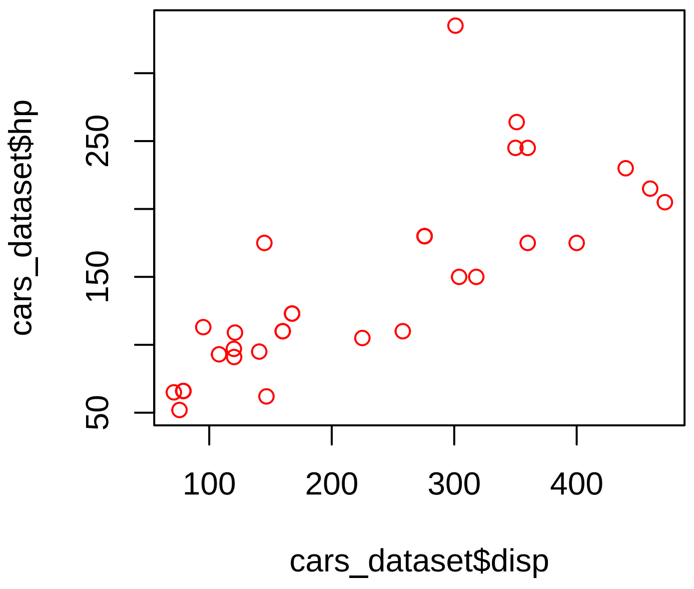
You'll usually be plotting columns of a data frame against each other, but you can use this for any two vectors, in fact:

```
heights <- c(1.56, 1.62, 1.8, 1.75)
weights <- c(52, 65, 80, 78) #The values from sheet 1
plot(weights, heights)
```



By default, R plots every point as a black circle. We can change this using additional inputs, or "arguments", to the function plot. We need to specify the names of these, like so:

```
plot(cars_dataset$disp, cars_dataset$hp, col="red") #Makes the points red
```



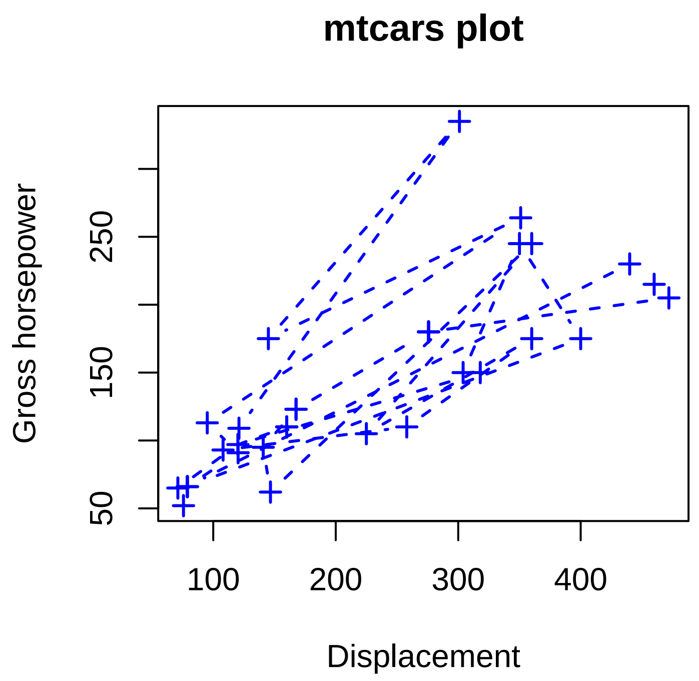
This is what we use a single `=` sign for! The reasons for this are a little bit too technical to write here without going into a long digression. For now, just note that sometimes our inputs have this `=` structure and sometimes they don't; also note that the ones without the `=` always come first and their order matters, whereas the order of inputs with `=` doesn't matter (just the names).

We can use these names to specify graphical features of the plot:

- `col`, as we've seen, takes a text string (enclosed in quotes) determining the colour of the points; it can also take a number, with the numbers being assigned to different colours
- `pch` is a number which determines the type of symbol used for each point (the default being a circle)
- `type` takes a text string which tells R what type of plot to do: `p` for points (the default), `l` for lines, `b` for both, and more (type `?base::plot` into the console to see a full list).
- `lty` takes a number that determines what kind of line to draw (e.g. a dotted line)
- `lwd` takes a number that determines how wide the lines should be
- `xlab` and `ylab` take text strings for the axis labels (the default being the names of the vectors/columns that you're plotting)
- `main` takes a text string for the chart title (the default being to have no title)

So, putting these together:

```
plot(cars_dataset$disp, cars_dataset$hp, col="blue", type="b", pch = 3, lty = 2, lwd = 1.5,
     xlab = "Displacement", ylab = 'Gross horsepower', main = "mtcars plot")
```



Notice that we've split this command across multiple lines of code. That's fine, because the open bracket, not matched by a close bracket, tells R after seeing the first line that there's more to come, and R acknowledges this in the console by displaying a `+` instead of a `>` at the start of the line. Something similar happens if you open a character string with a quotation mark and don't close it. This can lead to an error if you forget to close your brackets or your quotes!

Exercise 8A (optional, trickier)

The lines in the plot above create a jumbled mess. What's determining the order in which the lines are drawn? (Hint: try plotting the second and third points with

```
plot(cars_dataset$disp[2:3], cars_dataset$hp[2:3], type="b")
```

and then add the fourth using

```
plot(cars_dataset$disp[2:4], cars_dataset$hp[2:4], type="b")
```

and observe the difference.)

Exercise 9

Make a new plot of `wt` against `drat` from the data frame `cars_dataset`, using green, and using the symbol encoded by `2` for each point. Label the axes appropriately. (Remember you can type `?mtcars` in the console to find out what the values represent.)

What about other types of plot?

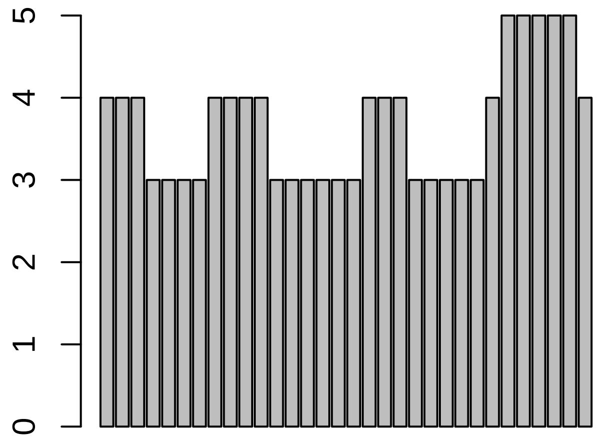
Exercise 10

Run each of the following lines of code. Underneath each, write a comment saying what it does.

```
boxplot(cars_dataset$mpg, main = "Plot (a)", ylab= "MPG")
# (a)
boxplot(cars_dataset$drat, cars_dataset$wt, ylab = "Value", names = c("Displacement", "Weight"),
        main = "Plot (b)")
# (b)
hist(cars_dataset$disp, main = "Plot (c)", xlab= "Displacement")
# (c)
```

Bar charts are a little harder. Suppose we wanted to make a bar chart of the values of `gear`. If you try

```
barplot(cars_dataset$gear)
```

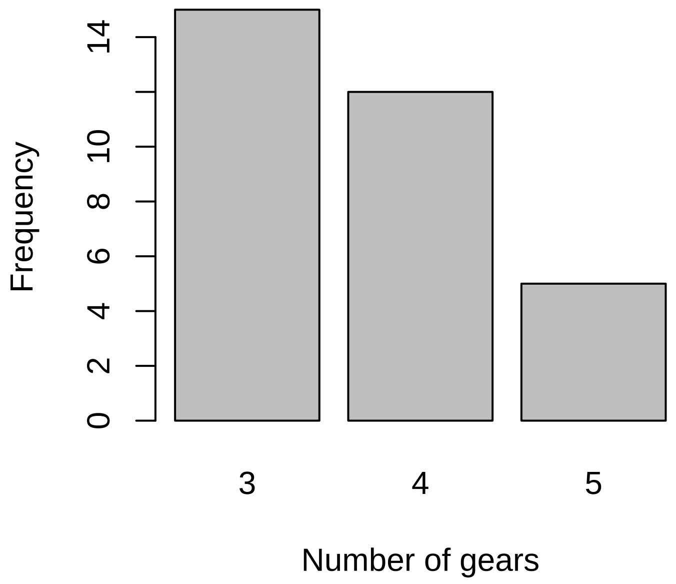


you'll notice it doesn't do what you want. That's because the input to barplot has to be the heights of the bars—that is, the number of values of each type. To do this we use

```
(gear.tab <- table(cars_dataset$gear))
```

```
##
##  3  4  5
## 15 12  5
```

```
barplot(gear.tab, xlab = "Number of gears", ylab = "Frequency")
```



The brackets around the first line force R to display the table, so you can see what the function `table` does. You don't need to put the brackets around to make the chart.

Exercise 11

Statisticians hate pie charts with a passion, but you can make one using the function `pie` (if you really must) which works in the same way as `barplot`. Make a pie chart of the values of `carb`, the number of carburettors of each car.

(Optional) Linear models

This bit is a bit statsy, but is a useful thing to know about. In simple terms, a linear model between two "continuous" variables is a way of fitting a line of best fit on the graph of those variables. A continuous variable is one where the values can be any number, including decimals, within a given range: like `disp`, `hp`, and `drat` in mtcars, but not `cyl` or `vs`. (`disp` and `hp` only have whole-number values as recorded: but that's because they've been rounded to the nearest whole number, not because of anything intrinsic.)

You can fit a linear model using the function `lm`, and the symbol `~` (Shift+# on Windows; Shift+', the key to the left of Z, on Mac):

```
mod.1 <- lm(cars_dataset$hp~cars_dataset$drat)
mod.2 <- lm(hp~drat, data = cars_dataset)
```

`mod.1` and `mod.2` are the same thing, but the latter way of writing it is a little more elegant when you come to dealing with more complex models with more than two parameters.

The convention here is the opposite way around from the one for plots: the thing you want on the y-axis comes *first*, before the tilde, and the x-axis variable comes after the `~`.

There are two simple things we can do with a linear model, starting with

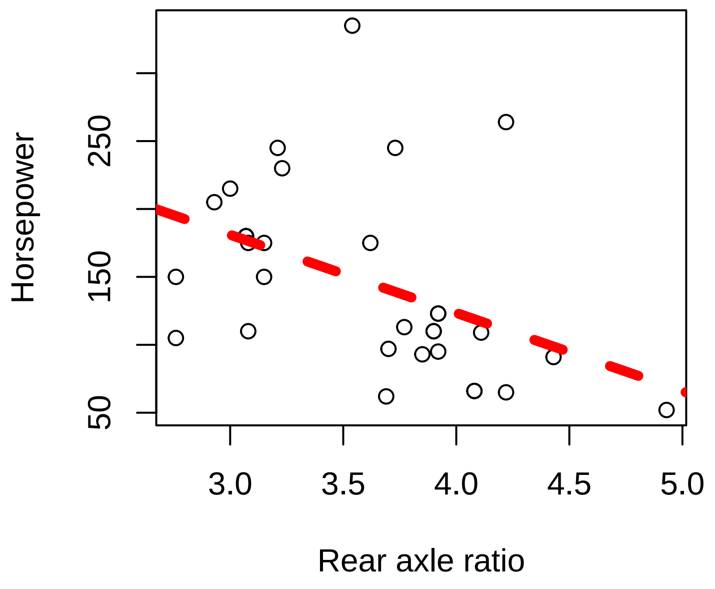
```
summary(mod.1)
```

```
##
## Call:
## lm(formula = cars_dataset$hp ~ cars_dataset$drat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -89.828 -40.261  -7.934   7.247 185.058
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         353.65      76.05    4.65 6.24e-05 ***
## cars_dataset$drat   -57.55      20.92   -2.75  0.00999 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 62.28 on 30 degrees of freedom
## Multiple R-squared:  0.2014, Adjusted R-squared:  0.1748
## F-statistic: 7.565 on 1 and 30 DF,  p-value: 0.009989
```

which gives summary statistics for the fit of the line of best fit, including the "R-squared", which is a measure of correlation between the variables. (R-squared is between zero and one, with higher numbers representing better correlation.) I won't explain all the other things in the summary output, except to say that the two values under the word `Estimate` are the intercept and slope, in that order, of the line of best fit.

We can also plot the line of best fit using the function `abline`, which is a whole separate line of code after the one for the plot. It will add the line of best fit to whatever plot you made most recently, even if it was a completely different plot! So we want to run:

```
plot(cars_dataset$drat, cars_dataset$hp, xlab="Rear axle ratio", ylab = "Horsepower")
abline(mod.1, col = "red", lty = 2, lwd= 5)
```



Notice that we can (optionally) use `col`, `lty` and `lwd` to specify the colour, line type and width of the line—the other graphical parameters are for points, so don't make sense here.

> **Exercise 11A**
>
> Make a plot of `qsec` against `wt`, with `qsec` on the y-axis, showing a coral-coloured line of best fit. (You can get a full list of the permitted colour names by entering `colours()` into the console.)

One last thing: RStudio lets you easily see past plots in the plots tab of the bottom-right panel, by clicking on the arrow buttons below the tabs: this is something you can't do in base R. You can also click on the "Export" button to save your plots for use elsewhere.

Data reporting

Commenting code

As I said at the start, the worksheet files use a lot of comments. In real life, you'd have much less in the way of comments in a file, but you would be expected to write *some* comments to say what your code is doing.

Why? Code is very abstract, and it isn't always easy to see what's going on just by looking at it, especially if it's using features you haven't used for a while. One way to tell is by running the code line-by-line yourself, but that's potentially time-consuming (sometimes, if you're working with a large dataset doing something complex, it can take hours to run, even on a powerful computer).

> **Exercise 12**
>
> This is code using a dataset called `iris`, which is another of the sample datasets. This time we are just going to load it from the sample version, instead of a CSV file. Type `?iris` into the console to learn more about it.[1]
>
> The code does several things using the techniques you've learnt in the previous sheets, as well as some things you haven't seen. Run the code and write a comment, starting with `#`, about each line saying what that line does. (The first line, and the lines that do something novel, are completed for you.)

[1] My botanist friend who reviewed these notes for me before the course would like me to inform you that, despite what the dataset suggests, irises do not have petals or sepals, instead having things that defy categorisation, known as "tepals". It's not of particular consequence to these exercises, but she's very passionate about this.

```r
my.iris <- iris #Loads the dataset "iris"
head(my.iris)
summary(my.iris)
plot(my.iris[,-5]) #Creates a "pairs plot" across the continuous variables, omitting the
                   #"species" column
spec.tab <- table(my.iris$Species)
barplot(spec.tab, xlab="Species", ylab = "Frequency")
apply(my.iris[,-5], 2, sd)
my.iris$Sepal.Ratio <- my.iris$Sepal.Length/my.iris$Sepal.Width
my.iris$Petal.Ratio <- my.iris$Petal.Length/my.iris$Petal.Width
boxplot(my.iris$Sepal.Ratio, my.iris$Petal.Ratio,
        names = c("Sepal ratio", "Petal ratio"))
plot(my.iris$Sepal.Ratio, my.iris$Petal.Ratio, xlab= "Sepal ratio",
     ylab = "Petal ratio")
plot(my.iris$Sepal.Ratio, my.iris$Petal.Ratio, xlab= "Sepal ratio",
     ylab = "Petal ratio", col = my.iris$Species)
#Changes the previous plot so that points are coloured according to their species
legend("topright", legend = c("setosa", "versicolor", "virginica"), col = 1:3, pch=1)
#Adds a legend for the colours: the colours used are colours 1, 2 and 3, colouring the
#species in the order that they first appear in the data frame
my.iris.vir <- my.iris[my.iris$Species == "virginica",]
plot(my.iris.vir$Sepal.Ratio, my.iris.vir$Petal.Ratio, xlab= "Sepal ratio",
     ylab = "Petal ratio", main = "Virginica only")
#For people who looked at optional section 3.3 on linear models:
mod.rat <- lm(Petal.Ratio~Sepal.Ratio, data = my.iris.vir)
summary(mod.rat)
abline(mod.rat, col="hotpink", lty=4)
```

(Optional) R Markdown

R Markdown is a way of writing R code and text together in one document, so that it can be output into a nice format. This course isn't going to attempt to teach you R Markdown, but it's a useful thing to know exists! (I knew R for many years before I'd heard of it, and it would have made my life much easier in several settings.) In fact, these notes, and the slides, were made using R Markdown

> EXERCISE 12A
>
> Open the slides.Rmd file. Have a look at the code and see if you can understand what's going on. (You don't have to write anything.)

An introduction to the tidyverse

Packages

The tidyverse is not supplied with base R, and you have to install it separately, as what's known as a "package" or "library". Fortunately, you can do this within R.

If you're using RStudio Cloud, and you open the file "5-tidyverse.R", there is probably a yellow bar at the top of this pane asking if you want to install the tidyverse. If so, click "Yes", and then wait for some red text to appear and disappear in the pane below (this can take a little while).

If you're running this on your own computer, the command you need to run is

```r
install.packages("tidyverse")
```

You only need to do this once per computer (unless you completely reinstall R or your operating system).

Each time you start a new R session, you then need to tell R that you want to have access to the tidyverse functions. There is a reason installed packages aren't available automatically: sometimes they redefine functions from base R, and sometimes they conflict with each other. So R leaves it to you to choose which packages you want on any given occasion. The command is:

```
library(tidyverse)
```

(You should run this even if you're using RStudio Cloud.) R might print some things to the console: don't worry if so. I've not included it here because it may differ slightly by computer.

Pipes

Before we look at the tidyverse methods themselves, we need to look at the pipe operator. This looks like %>%, and is made by typing the symbols separately: percent (Shift+5 on both Windows and Mac in the UK), then right angle bracket (Shift+., the full stop key), then percent again.

It means "take what's before the pipe, and use it as the first argument of the function after the pipe". So the following are equivalent:

```
sum(1:10)
```

```
## [1] 55
```

```
1:10 %>% sum
```

```
## [1] 55
```

You might ask what the point of this is. The idea is that the pipes clearly show from left to right the order of operations, which is convenient when you have long chains of operations.

For instance, let's reload the iris dataset.

```
my.iris <- iris
my.iris.num <- my.iris[,-5] #Get rid of the species column
```

Then the following are equivalent, but the latter is (I would argue) clearer.

```
round(mean(apply(my.iris.num, 2, sd)), digits=2)
```

```
## [1] 0.95
```

```
my.iris.num %>% apply(2, sd) %>% mean %>% round(digits=2)
```

```
## [1] 0.95
```

We haven't seen the `round` function yet, but it does what it says on the tin: it rounds things, by default to the nearest whole number or, when using the `digits` parameter, to that number of decimal places. You can use `signif` to round to a given number of significant figures.

We can store the output of a pipe using the assignment operator, as before.

```
val <- my.iris.num %>% apply(2, sd) %>% mean %>% round(digits=2)
```

If you prefer, you can also type the assignment operator the other way around, and put it at the end, like this.

```
my.iris.num %>% apply(2, sd) %>% mean %>% round(digits=2) -> val
```

This preserves the left-to-right ordering more neatly, but for some reason isn't conventional.

The disadvantage of piping is that it only really works when the thing you want to pass between various functions always occurs as the functions' first argument. The functions in the tidyverse are built so that the data frame (or rather, the equivalent to the data frame, as we will shortly see) is almost always the first argument.

One last thing about pipes: if you end a line with a pipe, it's akin to ending a line withbrackets still open: R will assume your instructions aren't complete. This lets you break piped instructions across multiple lines of code, which is necessary for readability if they get really long.

ggplot: prettier plots

The package "ggplot2" is also part of the tidyverse, though it's also common to see this one used independently. The main reason ggplot sees use is that the plots it makes are prettier, though in some ways (particularly choosing colours) they're harder to customise. "gg", by the way, stands for "grammar of graphics".

In ggplot, plots are defined as objects, like vectors and data frames. This means it's easy to build up plots out of different elements, and return to and modify them. (If you did the optional exercise on linear models, remember how you had to make sure that you'd just plotted the right graph, before using `abline`? No more!)

First, let's load the cars dataset again.

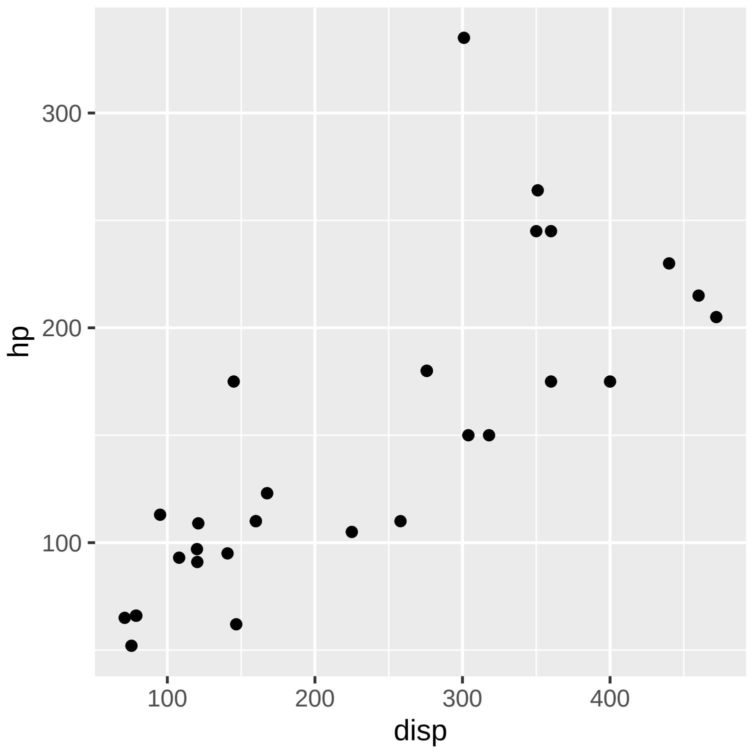
```r
cars_dataset <- read.csv("Data/mtcars.csv")
```

The basic function is `ggplot` itself. Let's initialise a plot object.

```r
plot1 <- ggplot(data = cars_dataset, aes(x=disp, y=hp))
```

Here we're telling R that any plots we make on that plot object will use data from `cars_dataset`, that the x-axis will be `disp` and the y will be `hp`. The `aes` stands for "aesthetics": i.e. the visual characteristics of the plot.

```r
plot1 + geom_point()
```

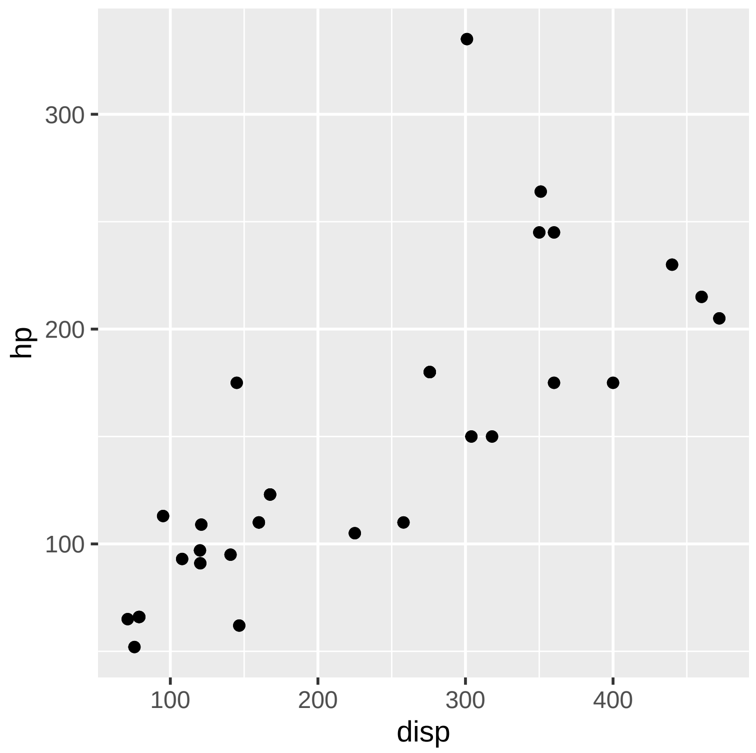


Here we've added the points to `plot1`, and displayed it. But notice that this hasn't stored the resulting plot anywhere: if you enter `plot1` now into the console, it will just display some empty axes. To store it back under the name `plot1` for later use, we would write:

```r
plot1 <- plot1 + geom_point()
```

and then you could enter `plot1` into the console to see it. Alternatively, you could write

```
(plot1 <- plot1 + geom_point())
```



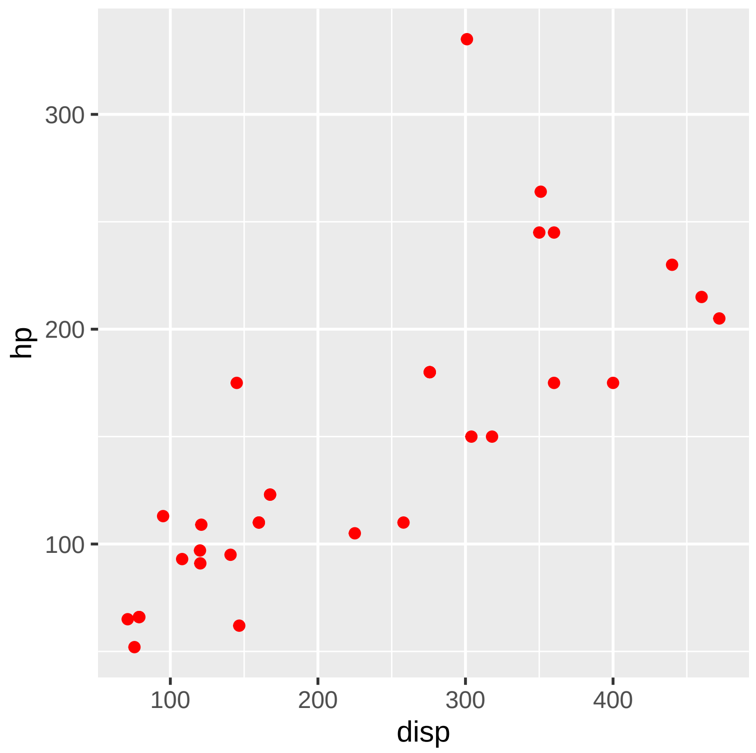
because brackets around an assignment tell R to then print whatever you've just defined.

Let's set up the empty axes again, so we can try different types of plot.

```
plot2 <- ggplot(data = cars_dataset, aes(x=disp, y=hp))
```

What if we wanted the points to be red, like before?

```
(plot2 <- plot2 + geom_point(colour="red"))
```



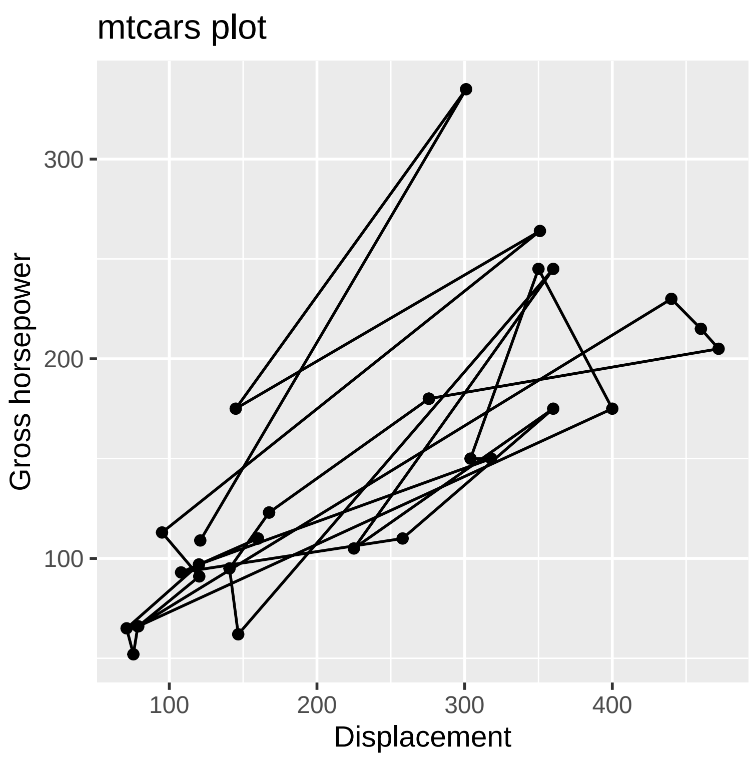
Notice we could have made this plot using one long set of instructions:

```
plot2 <- ggplot(data = cars_dataset, aes(x=disp, y=hp)) +
            geom_point(colour="red")
```

Here we've ended a line with a `+`, which is much like ending a line with a pipe: R knows to expect further instructions on the next line.

What if we wanted axis labels, and maybe also lines between the points?

```
(plot3 <- ggplot(data = cars_dataset, aes(x=disp, y=hp)) +
            geom_point() + geom_path() + #Add points, then lines
            labs(x = "Displacement", y= "Gross horsepower", title = "mtcars plot"))
```



ggplot actually has a function which automatically joins the points in left-to-right order: it's called `geom_line`.

Let's round up the other types of plot. We'll just provide the sample code here, without much commentary. Have a look at https://www.r-graph-gallery.com/ for examples of visualisation that others have done with R (with the necessary code).

Exercise 13

Run the code below to see what it does. Add any comments you feel are necessary.

```
(box_plot <- ggplot(data = cars_dataset, aes(y=mpg)) + geom_boxplot() +
              labs(y = "Miles per gallon", title = "Box plot") +
              theme(axis.text.x=element_blank(), axis.ticks.x = element_blank())
                                  #Remove meaningless x-axis
)
(hist <- ggplot(data= cars_dataset, aes(x=disp)) + geom_histogram() +
              labs(x = "Displacement", title = "Histogram") )

(bar_chart <- ggplot(data = cars_dataset, aes(x=gear)) + geom_bar() +
              labs(y = "Number of gears", title = "Bar chart"))
      #Notice we don't have to make a frequency table first any more!
```

I've not provided the sample code for making a pie chart because ggplot *really* doesn't want you to make pie charts: you have to make a stacked bar chart first, and then wrap one of the axes around into a circle. I told you statisticians hated pie charts with a passion!

More about the Tidyverse

Tibbles

Instead of data frames, the tidyverse uses "tibbles". The details between them are complex and technical; the upshot is that you use a different function to load your data from a CSV file: instead of read.csv, we use read_csv. (There is a convention that tidyverse functions use underscores—obtained by pressing Shift+-, the hyphen key to the right of 0—where base R would use dots.)

Let's try it with our mtcars CSV file.

```
cars_tib <- read_csv("Data/mtcars.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   Car = col_character(),
##   mpg = col_double(),
##   cyl = col_double(),
##   disp = col_double(),
##   hp = col_double(),
##   drat = col_double(),
##   wt = col_double(),
##   qsec = col_double(),
##   vs = col_double(),
##   am = col_double(),
##   gear = col_double(),
##   carb = col_double()
## )
```

Notice that we do actually get printed output this time, which tells us about the columns.

We can check this worked properly by viewing the top few rows, as before:

```
head(cars_tib)
```

```
## # A tibble: 6 x 12
##   Car             mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <chr>         <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Mazda RX4      21       6   160   110  3.9   2.62  16.5     0     1     4     4
## 2 Mazda RX4 W~   21       6   160   110  3.9   2.88  17.0     0     1     4     4
## 3 Datsun 710     22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
## 4 Hornet 4 Dr~   21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
## 5 Hornet Spor~   18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
## 6 Valiant        18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

The function has correctly parsed the column headers, and doesn't seem to have incorrectly parsed any data-separating commas, so we can go on.

(Note we could have written the above as

```
cars_tib <- "Data/mtcars.csv" %>% read_csv
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   Car = col_character(),
##   mpg = col_double(),
##   cyl = col_double(),
##   disp = col_double(),
##   hp = col_double(),
##   drat = col_double(),
##   wt = col_double(),
##   qsec = col_double(),
##   vs = col_double(),
##   am = col_double(),
##   gear = col_double(),
##   carb = col_double()
## )
```

```
cars_tib %>% head
```

```
## # A tibble: 6 x 12
##   Car              mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <chr>          <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Mazda RX4       21       6   160   110  3.9   2.62  16.5     0     1     4     4
## 2 Mazda RX4 W~    21       6   160   110  3.9   2.88  17.0     0     1     4     4
## 3 Datsun 710      22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
## 4 Hornet 4 Dr~    21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
## 5 Hornet Spor~    18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
## 6 Valiant         18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

if we'd wanted to.)

dplyr: manipulating data

We're now going to go through the commands from sheets 2 and 3, but using the tidyverse. We'll start with the portions concerning manipulating data, where we use the part of the tidyverse called dplyr, pronounced "dee-plier". (The different parts can be loaded as separate packages, instead of all together.)

select

We can select rows and columns of a tibble by number in the same way as for a data frame.

```
cars_tib[3,4]
```

```
## # A tibble: 1 x 1
##    disp
##   <dbl>
## 1   108
```

```
cars_tib[c(1,4,6),]
```

```
## # A tibble: 3 x 12
##   Car              mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <chr>          <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Mazda RX4       21       6   160   110  3.9   2.62  16.5     0     1     4     4
## 2 Hornet 4 Dr~    21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
## 3 Valiant         18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

```
cars_tib[,-c(1,5)]
```

```
## # A tibble: 32 x 10
##      mpg   cyl  disp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160   3.9   2.62  16.5     0     1     4     4
##  2  21       6  160   3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108   3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258   3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360   3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225   2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360   3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.  3.92  3.15  22.9     1     0     4     2
## 10  19.2     6  168.  3.92  3.44  18.3     1     0     4     4
## # ... with 22 more rows
```

To choose columns by name, we use the `select` function.

```
cars_tib %>% select(cyl)
```

```
## # A tibble: 32 x 1
##      cyl
##    <dbl>
##  1     6
##  2     6
##  3     4
##  4     6
##  5     8
##  6     6
##  7     8
##  8     4
##  9     4
## 10     6
## # ... with 22 more rows
```

```
cars_tib %>% select(mpg, disp)
```

```
## # A tibble: 32 x 2
##      mpg  disp
##    <dbl> <dbl>
##  1  21    160
##  2  21    160
##  3  22.8  108
##  4  21.4  258
##  5  18.7  360
##  6  18.1  225
##  7  14.3  360
##  8  24.4  147.
##  9  22.8  141.
## 10  19.2  168.
## # ... with 22 more rows
```

(These are the same as:

```
select(cars_tib, cyl)
```

```
## # A tibble: 32 x 1
##      cyl
##    <dbl>
##  1     6
##  2     6
##  3     4
##  4     6
##  5     8
##  6     6
##  7     8
##  8     4
##  9     4
## 10     6
## # ... with 22 more rows
```

```
select(cars_tib, mpg, disp)
```

```
## # A tibble: 32 x 2
##      mpg  disp
##    <dbl> <dbl>
##  1  21    160
##  2  21    160
##  3  22.8  108
##  4  21.4  258
##  5  18.7  360
##  6  18.1  225
##  7  14.3  360
##  8  24.4  147.
##  9  22.8  141.
## 10  19.2  168.
## # ... with 22 more rows
```

From now on, we will only show the piped method in the cases where that's conventional.)

There are a few things we can do easily with `select` that we didn't do with base R, because they're trickier to achieve in base R. For instance, we can easily select that we don't want a particular column, by name.

```
cars_no_name <- cars_tib %>% select(!Car) #Remove the car column
head(cars_no_name)
```

```
## # A tibble: 6 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
## 2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
## 3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
## 4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
## 5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
## 6  18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

Here the `!` means "not", like how we used `!=` for "not equal to".

We can also select columns by properties of the column names:

Exercise 14

Run the following code. In a comment after each pair of lines, write what the code does.

```
cars_starts_d <- cars_tib %>% select(starts_with("d"))
head(cars_starts_d)
#

cars_contains_p <- cars_tib %>% select(contains("p"))
head(cars_contains_p)
#
```

`filter`

`filter` is the function for selecting rows by their properties. The syntax for this is a little cleaner than the base-R syntax, in some cases!. We write the code that does the same as the filtrations on sheet 2.

Exercise 15

Check that these do what you'd expect. (There's no need to write anything!) Compare with the code on Sheet 2. Which method do you prefer?

```
cars_tib %>% filter(gear == 4)
cars_tib %>% filter(mpg < 15)
cars_tib %>% filter(carb >= 6)
cars_tib %>% filter(am != 1)
cars_tib %>% filter(str_detect(Car, "^Merc"))
# For "ends with 0", say, you use 'str_detect(Car, "0$")' instead
cars_tib %>% filter(Car >= "T")
```

`summarise`

We can still run `summary` from base R on tibbles.

```
cars_tib %>% summary
```

```
##      Car                 mpg             cyl             disp      
##  Length:32          Min.   :10.40   Min.   :4.000   Min.   : 71.1  
##  Class :character   1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8  
##  Mode  :character   Median :19.20   Median :6.000   Median :196.3  
##                     Mean   :20.09   Mean   :6.188   Mean   :230.7  
##                     3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0  
##                     Max.   :33.90   Max.   :8.000   Max.   :472.0  
##        hp             drat             wt             qsec      
##  Min.   : 52.0   Min.   :2.760   Min.   :1.513   Min.   :14.50  
##  1st Qu.: 96.5   1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89  
##  Median :123.0   Median :3.695   Median :3.325   Median :17.71  
##  Mean   :146.7   Mean   :3.597   Mean   :3.217   Mean   :17.85  
##  3rd Qu.:180.0   3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90  
##  Max.   :335.0   Max.   :4.930   Max.   :5.424   Max.   :22.90  
##        vs               am              gear            carb      
##  Min.   :0.0000   Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4375   Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :1.0000   Max.   :5.000   Max.   :8.000  
```

What if we wanted the standard deviation of a column, or of several? We'd use the function `summarise` for this. In the second example, we make sure we remove the column with text first—this lets us chain up some pipes.

```
cars_tib %>% summarise(sd(mpg))
```

```
## # A tibble: 1 x 1
##   `sd(mpg)`
##       <dbl>
## 1      6.03
```

```
cars_tib %>% select(!Car) %>% summarise(across(everything(),sd)) #(*)
```

```
## # A tibble: 1 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  6.03  1.79  124.  68.6 0.535 0.978  1.79 0.504 0.499 0.738  1.62
```

Compare the second of these to

```r
apply(cars_tib[,-1], 2, sd)
```

```
##         mpg         cyl        disp          hp        drat          wt 
##   6.0269481   1.7859216 123.9386938  68.5628685   0.5346787   0.9784574 
##        qsec          vs          am        gear        carb 
##   1.7869432   0.5040161   0.4989909   0.7378041   1.6152000
```

the non-tidyverse way of doing the same thing. Sometimes the tidyverse is less tidy!

(If you prefer the American/Oxford spelling of this word, `summarize` does the same thing.)

Exercise 16

How would you change the second line marked with (*) above to get the mean of each numeric column?

`mutate`

If we want to create new columns as functions of other columns, we use `mutate`. So if we wanted the horsepower per cylinder, like before, we'd do:

```r
cars_tib <- cars_tib %>% mutate(hp_per_cyl = hp/cyl)
```

(This sort of statement is common in programming, where the same object, in this case `cars_tib`, appears on both sides of an assignment: it means "take `cars_tib`, do the appropriate operation on it, and then store the result back as `cars_tib` again, overwriting the previous value". If we don't do the assignment, the new column gets calculated and the result printed, but it doesn't get stored anywhere for later use.)

Exercise 17

Create a column called `gear_plus_carb` that is the sum of the gear and carb columns, like on Exercise 8.

Exercise 18 (for the bold)

This is the code from Exercise 12, but rewritten to use tidyverse methods. I've skipped many of the lines, replacing them with a lettered comment saying what they should do. Can you fill in the blanks?

```r
my_iris <- as_tibble(iris) #Loads the dataset "iris"
head(my_iris) #Displays the first few lines of the data
summary(my_iris) #Displays summary statistics for the columns.
# We'll skip the pairs plot entirely: it's a little tricky to do in ggplot

#(a) Make a bar chart of Species
#(b) Find the standard deviations of the numerical columns
#(c) Create new columns called "Sepal.Ratio", which is Sepal.Length/Sepal.Width, and
#    "Petal.Ratio", the equivalent for petals.

long_form <- my_iris %>% pivot_longer(cols = contains("Ratio"), names_to = "type") %>%
  select(Species, type, value)
(box_plot <- ggplot(data = long_form, aes(x=type, y=value)) + geom_boxplot())
#Makes a box plot of the two ratios on one plot

#(d) Make a scatter plot with Sepal.Ratio on the x-axis and Petal.Ratio on the y-axis,
#    with axes labelled

(with_colour <- ggplot(data = my_iris, aes(x=Sepal.Ratio, y=Petal.Ratio)) +
    geom_point(aes(colour = Species)) +
    labs(x="Sepal ratio", y="Petal ratio"))
#Remakes previous plot but with points colour-coded by species

#(e) Create a new tibble with just the virginica irises
#(f) Make a scatter plot of just the virginica ratios, with axes labelled and with the
#    main title "Virginica only"; store it as "pvs_vir".

pvs_vir + geom_smooth(method = 'lm')
#Adds trend line using a linear model (as discussed in optional Section 3.3)
```

Glossaries

Throughout these glossaries, terms that only apply to the tidyverse section are marked with *(tidyverse)*.

Glossary of terms

- **argument**: an input to a function.
- **assign**: to store a value as a variable
- **call**: to run, in the context of a function.
- **character**: a data type that consists of text. Often used for categorical variables.
- **comma-separated value** (CSV): a file format that stores data as text, separated by (usually) commas.
- **comment**: a section of a script that contains text for people to read, and not code. Entered using `#`.
- **Comprehensive R Archive Network** (CRAN): a website where you can download R, and on which packages are stored.
- **concatenate**: to run together. Source of the name of the function `c`, which concatenates multiple values or vectors into one long vector.
- **console**: the pane in RStudio where code is run.
- **CRAN**: see *Comprehensive R Archive Network*
- **CSV**: see *comma-separated value*
- **data frame**: a structure for storing data, in which the columns are different variables, and the rows represent different observations.
- **dollar operator**: an operator in R that lets us access, and store, columns of a data frame (among other things we haven't looked at in this course). Typed as `$` (hence the name!).
- **double**: a data type that acts somewhat like a number written in scientific notation (e.g. 1.4×10^{11}). Used for continuous variables. Short for "double-precision floating point number".
- **dplyr**: *(tidyverse)* a part of the tidyverse, used for manipulating data.
- **editor**: the pane in RStudio for writing scripts.
- **function**: a piece of R code that does a particular task. Called using the syntax `function.name(arguments)`.
- **ggplot**: *(tidyverse)* a part of the tidyverse, used for making plots.
- **IDE**: see *integrated development environment*
- **integer**: a data type that stores whole numbers.
- **integrated development environment** (IDE): a program which lets you edit code, run code and perform auxiliary tasks.
- **linear model**: in its most basic form, a way of fitting a line of best fit to data. Also known as a *regression*.
- **list**: a kind of data structure that we haven't studied in this course (but the reason why we can't refer to vectors as "lists"!).
- **package**: a collection of functions that can be loaded into R.
- **pipe operator**: *(tidyverse)* an operator, represented by `%>%`, which tells R to put the output of whatever is on the left as the first argument of the function on the right, and run that function. Sometimes shortened to "pipe".
- **print**: usually, to display the output in the console (not to send it to a printer!).
- **regression**: see *linear model*

- **RStudio**: an IDE for R. Free to use.
- **RStudio Cloud**: the browser-based version of RStudio. Free for limited use.
- **script**: a collection of lines of code.
- **subset**: a portion of data, or to take a portion of data.
- **table**: usually, a frequency table, counting the number of observations of each value.
- **tibble**: *(tidyverse)* the tidyverse equivalent of a data frame.
- **tidyverse**: *(tidyverse)* a package that provides a different way of working with R.
- **variable**: a name, under which data frames, vectors, values etc. can be stored.
- **vector**: a data structure that contains a sequence of values, all of the same type.
- **working directory**: the folder, or "directory", in which our files are stored. It is normal to tell R which is the working directory at the start of a session.

Glossary of functions

- `abline`: add a line given by the output of the function `lm` to the most recent plot—options include:
  - `col`: the colour of the line
  - `lty`: a number determining what kind of line to draw (e.g. dotted)
  - `lwd`: a number giving line width
- `across`: *(tidyverse)* a function used within `summarise` to select columns to summarise
- `aes`: *(tidyverse)* a function used within `ggplot` or other plotting functions provided by the ggplot package to specify aesthetic properties of a plot—options include: ** `colour`: the colour of the points in a plot (usually)
  - `x`: the column of the tibble to go on the x-axis
  - `y`: the column of the tibble to go on the y-axis
- `apply`: apply a function (third argument) to a data frame (first argument), by columns (if the second argument is 2) or rows (if the second argument is 1)
- `barplot`: make a bar chart from a frequency table—options include:
  - `xlab`: the label on the x-axis
  - `ylab`: the label on the y-axis
- `boxplot`: make a box plot of the data in a vector, or several on the same axis if given multiple vectors—options include:
  - `main`: the overall graph title
  - `names`: a vector of text strings giving the words to put under each box plot
  - `ylab`: the label on the y-axis
- `c`: concatenate: make a vector of values, or run vectors together into a longer one
- `citation`: return the citation for R itself
- `contains`: *(tidyverse)* select columns that contain a given text string; used within `select`
- `element_blank`: *(tidyverse)* used within `theme` to specify that an element should not appear
- `ends_with`: *(tidyverse)* select columns that end with a given text string; used within `select`
- `everything`: *(tidyverse)* a function used within `across` (in turn within `summarise`) to specify that we wanted to apply a function to every column
- `filter`: *(tidyverse)* select rows by some condition; usually used with pipes, in which case the condition is the only argument in brackets
- `geom_bar`: *(tidyverse)* a function added to an object created by `ggplot`, which makes a bar chart of whatever is specified by `aes`, with no need to make a frequency table first; it may have a call to `aes` as one of its arguments, if the `ggplot` object didn't
- `geom_boxplot`: *(tidyverse)* a function added to an object created by `ggplot`, which makes a box plot of whatever is specified by `aes`; it may have a call to `aes` as one of its arguments, if the `ggplot` object didn't
- `geom_histogram`: *(tidyverse)* a function added to an object created by `ggplot`, which makes a histogram of whatever is specified by `aes`; it may have a call to `aes` as one of its arguments, if the `ggplot` object didn't
- `geom_line`: *(tidyverse)* a function added to an object created by `ggplot`, which makes a line plot joining points in the order in left-to-right order along the x-axis; it may have a call to `aes` as one of its

arguments, if the `ggplot` object didn't

- `geom_path`: *(tidyverse)* a function added to an object created by `ggplot`, which makes a line plot joining points in the order in which they appear in the data; it may have a call to `aes` as one of its arguments, if the `ggplot` object didn't
- `geom_point`: *(tidyverse)* a function added to an object created by `ggplot`, which makes a scatter plot; it may have a call to `aes` as one of its arguments
- `geom_smooth`: *(tidyverse)* a function added to an object created by `ggplot`, which adds a trend line—options include:
  - `method`: the method to use to fit the line (e.g. `'lm'`, for a linear model fit)
- `ggplot`: *(tidyverse)* a function to set up a `ggplot`; the first argument is a data frame (possibly supplied by pipes), and one of the arguments may be the function `aes`
- `head`: display the first few values in a vector, or the first few lines of a data frame or tibble
- `hist`: make a histogram—options include:
  - `main`: the overall graph title
  - `xlab`: the label on the x-axis
- `install.packages`: a text string giving the name of a package to install from CRAN
- `labs`: *(tidyverse)* a function added to an object created by `ggplot`, which adds labels—options include:
  - `title`: the overall graph title
  - `x`: the label on the x-axis
  - `y`: the label on the y-axis
- `library`: load a package (specified with its name not in quotes), so that its functions are available to use
- `lm`: fit a linear model to a formula of the form `y.data ~ x.data`, where `x.data` is a vector of the data on the x-axis, and `y.data` a vector of the data on the y-axis—options include:
  - `data`: tells the function that `y.data` and `x.data` are columns of a data frame or tibble
- `mean`: calculate mean of a vector
- `mtcars`: access the sample dataset mtcars
- `mutate`: *(tidyverse)* create a new column that is a function of old columns; usually used with pipes, in which the arguments are of the form `new_column_name = function(old_column_name)`, none of the column names being enclosed in quotes
- `read.csv`: read a CSV file into a data frame
- `pie` make a pie chart from a frequency table
- `pivot_longer`: *(tidyverse)* transform a tibble into a longer form, where values that were in several columns are placed in the same column, a new column being created to store which column name they came from (for instance, from a tibble with three columns named `observation_number`, `length` and `width`, we can create a tibble with three columns for `observation_number` (with each number now appearing twice), `measurement` (storing a value from either the old `length` or the old `width` column) and `type`, where `type` stores whether the measurement was the length or the width); we didn't really use this one
- `plot`: (usually) make a scatter plot (first argument on the x-axis, second on the y-axis)—options include:
  - `col`: the colour of the points
  - `main`: the overall graph title
  - `lty`: a number determining what kind of line to draw (e.g. dotted)
  - `lwd`: a number giving line width
  - `pch`: shape of each point
  - `type`: what kind of plot (`p` for points, `l` for lines, `b` for both)
  - `xlab`: the label of the x-axis
  - `ylab`: the label of the y-axis
- `plot`: (if applied to a data frame) make a pairs plot of the columns in the data frame against each other
- `prod`: product (multiply together)
- `read.csv`: read a CSV file into a data frame
- `read_csv`: *(tidyverse)* read a CSV file into a tibble
- `round`: round the value, or the values in a vector—options include:

    - `digits`: the number of decimal places to which to round
- `sd`: calculate standard deviation of a vector
- `select`: *(tidyverse)* select columns of a dataframe by name; usually used with pipes, in which case all arguments in brackets are the column names to select, not enclosed by quotation marks (or a column name preceded by `!`, to mean "not this column")
- `sqrt`: square root
- `startsWith`: test whether values in a vector (first argument) start with a text string (second argument)
- `starts_with`: *(tidyverse)* select columns that start with a given text string; used within `select`
- `str_detect`: *(tidyverse)* test whether a vector (first argument) matches a given regular expression, though we didn't talk about what regular expressions are; the notes give an example of using this to match values that start with or end with a given text string
- `sum`: sum
- `summarise`: *(tidyverse)* apply a function to calculate a summary statistic of a particular column, like the mean or standard deviation; usually used with pipes, in which case the only argument(s) in brackets are the summary statistics to calculate
- `summarize`: *(tidyverse)* an alternate spelling for the function `summarise`
- `summary`: display summary statistics for a dataset
- `table`: make a frequency table from a vector
- `theme`: *(tidyverse)* a function added to a `ggplot` object which changes some aspect of the plot—options include:
    - `axis.text.x`: sets text on x-axis tickmarks, or `element_blank()` for none
    - `axis.ticks.x`: sets x-axis tickmarks, or `element_blank()` for none +`legend.position`: a text string specifiying where the legend should be, or `"none"` for no legend
- `which`: display which values in a vector satisfy a certain condition

Glossary of symbols

- `+`: add
- `-`: subtract
- `*`: multiply
- `/`: divide
- `^`: to the power of
- `~`: used within `lm` to specify the linear relation being modelled
- `$`: dollar operator (see above)
- `#`: begins a comment
- `( )`: brackets, used:
    - to call functions (the arguments go inside the brackets, separated by commas)
    - around assignments, to tell R also to print the result
    - as mathematical brackets
- `[ ]`: square brackets, used for subsetting
- `<-`: assignment operator: stores the result of the calculation on the right to the variable name on the left
- `->`: as above, but with "left" and "right" interchanged; rarely used
- `=`: used for specifying function arguments by name, rather than order
- `==`: is equal to
- `>`: is greater than
- `>=`: is greater than or equal to
- `<`: is less than
- `<=`: is less then or equal to
- `!=`: is not equal to
- `%>%`: *(tidyverse)* the pipe operator (see above)

Copyright