Importing and Cleaning Data in the R tidyverse

Matt Denwood, University of Copenhagen

R has existed for over 25 years, and is itself based on software that was invented in 1976. Over that time, many different styles of R coding have evolved, and consequently there are a large number of different ways of accomplishing the same tasks. The most modern way of using R is within the tidyverse (http://tidyverse.org) - this is a collection of extensions to base R that are focussed on making it as easy as possible to do data science in R: i.e. import, process and analyse the types of datasets that are typical to epidemiology. Central to this are three key concepts:

1) Tidy datasets are much easier to work with than messy datasets
2) Always check for data entry mistakes in your data - never just assume that it is all correct
3) R code should be as clear and easy to read as possible even for users that are not expert programmers

R code written using the tidyverse looks a bit different to the code using base R that you might have used before, but it makes it much easier to create code that does relatively compex things with your data. This tutorial will explain some of the key features of importing, cleaning and checking data using the calf fetuses dataset as an example. The code starts off very simple in Section 1, but Sections 2 and 3 are a bit more advanced, so take your time and understand each step before moving onto the next. For a more complete (and even longer) introduction to tidyverse I thoroughly recommend the book 'R for Data Science': a free online version is available from https://r4ds.had.co.nz/

For this tutorial we will assume that you have installed both R (http://cran.r-project.org) and RStudio (http://rstudio.org), but you do not need to have any prior experience with R. Start by opening RStudio and creating a new R script by clicking 'File' then 'New File' then 'R Script'. Save it and give it a name, like 'tidy_fetuses.R'. You should see a blank text file appear within R studio - you can then copy code given here in 'R code chunks' like this one:

```
2+4
```

...and paste them into your tidy_fetuses.R file. To run the code, place the cursor somewhere on the line you want to run, and click 'Run' (or use the keyboard shortcut control/command + enter). This document also displays the result of running these code chunks, which should match what you see when you run the code in R. For example, the code above should give the following output:

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

To run multiple lines at once, highlight them all and click 'Run'. To run the entire R script file click 'Source'.

It is a good idea to write comments for yourself in the R script file to remind you what the R code is doing when you come to look at it later. Any lines starting with # are ignored by R, for example:

```
# This is a comment - it will be ignored by R
1+4
```

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

```
# 8+9
```

It can also be useful to 'comment out' R code that you don't want to run but do want to leave in the R script for later reference: for example the 8+9 calculation in the code chunk above is ignored by R because that line starts with a #. If you want to run the calculation, you can either delete the # and run the whole line, or highlight just the 8+9 and click run.

Section 1

Installing tidyverse

You need to have the tidyverse package installed within R. If you have not already installed the package, click 'Tools' then 'Install Packages' then type 'tidyverse' and click on 'Install'. You only need to install a package once, but you need to load it every time you restart R. Do this using the library() function:

```
library('tidyverse')
```

```
## -- Attaching packages ----------------------------------------------------------- tidyverse 1.2.1 -
```

```
## v ggplot2 3.1.0     v purrr   0.2.5
## v tibble  2.0.1     v dplyr   0.7.8
## v tidyr   0.8.2     v stringr 1.3.1
## v readr   1.3.1     v forcats 0.3.0
```

```
## -- Conflicts -------------------------------------------------------------- tidyverse_conflicts() -
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

When you run this code, you should see similar output to that shown above. If you have any problems then make sure that R and RStudio are updated to the latest versions. If you still have problems, then let us know ASAP so we can try to resolve them (the most common problems are due to old operating system versions or restricted access on institutional computers - if you are using an admin account on a relatively up-to-date computer then you should be OK).

Preparing your data

Before reading an Excel file into R, you will need to make sure that the relevant sheet is laid out in an appropriate way. The first row should be column names, and all columns must have a unique name. You will make life much easier for yourself if the column names are concise, and do not use spaces, special characters (e.g. + - / * & % € , etc), dates, or non-English characters (e.g. å æ ø). You can use an underscore (_) to create two-word names e.g. birth_date. The second row should be the start of the data: do not include any blank rows below the column names.

Each column can either be a number, date/time, or text - but this must be consistent within a column. It is a good idea to ensure that date/time columns are stored correctly by selecting the column and going to 'Format' -> 'Cells' and selecting either date or time as the format. As a general rule it is a much better idea to record categorical data (e.g. gender) as character rather than number codes e.g. male/female rather than 1/2. Remember that all cell comments and formatting (e.g. bold, colours etc) will be completely ignored by R when reading the data, so DO NOT use any type of comment or format to represent important information: create a new column instead. If you have an observation that is missing then jut leave the cell blank - do not use any special codes or text to indicate missing observations as this will make your life more difficult than it needs to be.

Finally, ensure that you do not have any stray text or numbers entered outside of the columns of data, or rows that do not form part of the data (e.g. formulas to calculate means as the last row) placed within the same sheet. If you want to do this in Excel, you can create a second sheet specifically for any formulas / notes etc. It is a good idea to have a separate 'data key' sheet where you record the meaning of the varibles, where the data has come from, and any other relevant meta-data such as the collection date or any other

important notes that you might need to remember when you come back to the data later on. This will also make it much easier for you to share the data with others.

Reading data

The first thing you are going to want to do is to read your data file. This can be in a number of formats, but the most common is an Excel file. For this we need to load the readxl package (this is installed as part of tidyverse, but not loaded by default):

```
library('readxl')
```

You will then need to make sure that your working directory is set correctly so that R can find the data file on your hard drive. Go to the 'Session' menu, 'Set Working Directory', and 'Choose Directory' and then select the folder that contains the calf foetuses dataset that you downloaded from Absalon as part of the previous task.

You should now be able to load the data from a specified sheet within the Excel file using:

```
fetuses <- read_excel('calf_fetuses.xlsx', sheet='calf_fetuses')
```

When you run this code, nothing much seems to happen (unless you get an error saying that the file could not be found, in which case check that you have included the .xlsx or .xlx file extension!). But in fact R has read the data from the Excel file, and assigned the resulting 'data frame' (which is like an Excel worksheet) called 'fetuses' (using the assignment operator <-) in which the data is now being stored within your R envirnoment. If you look closely you should see the text 'fetuses 262 obs. of 18 variables' appear under 'Data' in the 'Environment' tab within R studio. Click on the name 'fetuses' to open a new tab with the data in table form.

Looking at your data

There are several methods of examining and visualising your data, but here we will stick with the tidyverse way of doing things. You may already know how to do some of this in base R - it is OK to use methods you are already familiar with, but try not to mix different styles too much. That leads to messy code, and we want everything to be tidy...

The tidyverse revolves around data frames - these allow us to keep all related data together, in a consistent format where we know that all columns have the same number of rows and each row corresponds to a specific set of observations. When working with data in the tidyverse, we will therefore nearly always be working within these data frames. To do something within a data frame, we first type the name of the data frame, then the special 'pipe' or 'chaining' operator %>%, and then a function that we want to apply to the data frame. For example:

```
fetuses %>% nrow()
```

```
## [1] 262
```

This takes the data frame fetuses, and applies the function nrow() to the data - the output is the number of rows of the dataset. In base R we would achieve the same thing using:

```
nrow(fetuses)
```

```
## [1] 262
```

These two bits of code are in fact equivalent - the %>% operator automatically takes the data frame input from the left hand side and passes it (as the first argument) to the function on the right hand side. The

advantage to using the %>% operator is that we can build up a more complex chain of commands more easily - you will see this later on. There is even a handy keyboard shortcut to type the operator: command/control + shift + m.

A useful function to find out even more about the structure of the data is str() - this tells us how R is storing each of the columns (called 'variables') within the data frame:

```
fetuses %>% str()
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    262 obs. of  18 variables:
##  $ parity            : num  3 2 1 1 2 3 2 2 6 4 ...
##  $ age_days          : num  274 196 259 249 221 221 177 82 193 142 ...
##  $ weight_kg         : num  35.5 10.2 27.1 29.1 16.6 ...
##  $ crl_cm            : num  86.8 61.8 89.3 83 75.4 77.7 55.4 10.4 62.4 35.4 ...
##  $ head_width_mm     : num  128 83.8 121 118 100.6 ...
##  $ head_length_mm    : num  239 154 220 215 183 ...
##  $ hair_coronary_band: chr  "Y" "Y" "Y" "Y" ...
##  $ hair_ear          : chr  "Y" "Y" "Y" "Y" ...
##  $ hair_eyelid       : chr  "Y" "Y" "Y" "Y" ...
##  $ hair_tail         : chr  "Y" "Y" "Y" "Y" ...
##  $ hair_hornbud      : chr  "Y" "Y" "Y" "Y" ...
##  $ tactile_muzzle    : chr  "Visible" "Visible" "Visible" "Visible" ...
##  $ tactile_eyebrow   : chr  "Visible" "Visible" "Visible" "Visible" ...
##  $ tactile_eyelash   : chr  "Visible" "Visible" "Visible" "Visible" ...
##  $ eye_op_close      : chr  "Open" "Open" "Open" "Open" ...
##  $ papillae_tongue   : chr  "Whole tongue" "Whole tongue" "Whole tongue" "Whole tongue" ...
##  $ eyelid            : chr  "Y" "Y" "Y" "Y" ...
##  $ sex               : chr  "Female" "Female" "Male" "Male" ...
```

This gives several bits of information: how many rows (262) and columns (18) are in the data, what each of these columns are called (parity, age_days and so on), and the values corresponding to the first few rows of each variable. But the most important information is how these variables are being stored - each variable will have one of the following after the name:

- num: This variable is numeric, so we can do things like take the mean, range etc. This can also be called integer (int) or double (dbl) but you can think of these as being the same thing as numeric.
- chr: This variable is a character, or text string. Variables are often read in as text, but this isn't actually a very useful format so we usually want to convert these into something else (see the next section).
- Factor: This variable is a factor, which is what R uses to represent categorical data. A factor is a very useful format that we will be using a lot. A closely related format is Ord.factor, which means that the order of the factor levels is meaningful: this is how R represents ordinal data.
- POSIXct: This variable is a date time object, although the format may be shown as simply the date (in which case the time is implicitly 00:00). This is also sometimes called 'dttm'. If we have a column formatted either as a date or a time in Excel, then the variable will end up as POSIXct in R. Dealing with date times can be difficult as we can be caught out by time zones.
- Date: This variable is a simpler representation of a Date, without an associated time. These are typically much easier to deal with than POSIXct variables.
- logi: This variable takes the values FALSE or TRUE (logical).

Note that we only have numeric and character variables when reading in the fetuses dataset. As a general rule, the output of read_excel (or read_csv for a CSV file) will give us variables that are either numbers if all of the values can be interpreted as a number, or text if any of the variables can't be interpreted as a number (the main exception to this is that read_excel will automatically read columns formatted as date or time in Excel into POSIXct). Blank cells don't count when determining variable types, so if you have a

column of numbers with some empty cells then the variable will still be a number, just with some missing values corresponding to the empty rows. Missing values in R are represented by NA - but you should not use the text 'NA' in Excel (unless of course you want to represent sodium or Nebraska). If you have values that are missing, just leave the cell blank.

Another way of looking at the data is by typing the name of the data frame:

```
fetuses
```

```
## # A tibble: 262 x 18
##    parity age_days weight_kg crl_cm head_width_mm head_length_mm hair_coronary_b~ hair_ear
##     <dbl>    <dbl>     <dbl>  <dbl>         <dbl>          <dbl> <chr>            <chr>
##  1      3      274    35.5     86.8          128            239   Y                Y
##  2      2      196    10.2     61.8           83.8          154   Y                Y
##  3      1      259    27.2     89.3          121            220.  Y                Y
##  4      1      249    29.1     83            118            215   Y                Y
##  5      2      221    16.6     75.4          101.           183   Y                Y
##  6      3      221    19.0     77.7           97            189   Y                Y
##  7      2      177     6.35    55.4           74.5          142.  Y                N
##  8      2       82     0.046   10.4           24.3           36.8 N                N
##  9      6      193    11.2     62.4           85.4          153.  Y                N
## 10      4      142     1.77    35.4           62.4           99.8 N                N
## # ... with 252 more rows, and 10 more variables: hair_eyelid <chr>, hair_tail <chr>,
## #   hair_hornbud <chr>, tactile_muzzle <chr>, tactile_eyebrow <chr>, tactile_eyelash <chr>,
## #   eye_op_close <chr>, papillae_tongue <chr>, eyelid <chr>, sex <chr>
```

This shows the first few rows of data, but only for the first few variables, so it is usually better to use the str() function. Notice also that this method shows 'dbl' (and sometimes 'int') rather than 'num' for the numbers - but these are basically all the same thing.

Finally, we can use the summary() function to see summary statistics for each variable:

```
fetuses %>% summary()
```

```
##      parity         age_days        weight_kg          crl_cm       head_width_mm
##  Min.   :1.000   Min.   : 25.00   Min.   : 0.0010   Min.   :  2.00   Min.   :  3.00
##  1st Qu.:2.000   1st Qu.: 95.25   1st Qu.: 0.2537   1st Qu.: 18.30   1st Qu.: 39.24
##  Median :3.000   Median :136.00   Median : 1.7805   Median : 35.65   Median : 63.33
##  Mean   :2.752   Mean   :136.06   Mean   : 5.0029   Mean   : 37.90   Mean   : 62.34
##  3rd Qu.:4.000   3rd Qu.:175.00   3rd Qu.: 6.9250   3rd Qu.: 54.60   3rd Qu.: 84.02
##  Max.   :6.000   Max.   :274.00   Max.   :41.6000   Max.   :101.30   Max.   :152.00
##  head_length_mm   hair_coronary_band   hair_ear         hair_eyelid         hair_tail
##  Min.   :  5.00   Length:262         Length:262         Length:262         Length:262
##  1st Qu.: 55.52   Class :character   Class :character   Class :character   Class :character
##  Median :101.28   Mode  :character   Mode  :character   Mode  :character   Mode  :character
##  Mean   :103.29
##  3rd Qu.:145.54
##  Max.   :239.00
##  hair_hornbud       tactile_muzzle     tactile_eyebrow    tactile_eyelash    eye_op_close
##  Length:262         Length:262         Length:262         Length:262         Length:262
##  Class :character   Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##  papillae_tongue       eyelid              sex
##  Length:262         Length:262         Length:262
```

```
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
```

For the numeric variables (and any date/time variables), we get the range (min and max), as well as median/mean and 1st and 3rd quantiles of the variable. This is useful to check for extreme values that probably indicate a data entry error. For factors, we get a count of how many observations fall within each category. For the text variables, we don't get any useful information: this is because there is no general way to summarise text variables.

All this discussion of variable types is quite boring, but it is also very important. There is a big difference in R between the text '12.4' and the number 12.4, and between the text '2010-04-12' and the date this represents, and between the text 'male' and a factor with observed level 'male'. The format of the variable will affect the way in which R does almost everything with that variable, from stratifying in summaries and plots to handling explanatory variables in models. If the formats are correct then everything will work the way we intended. If the formats are incorrect then we will either get weird error messages that we don't understand, or even worse: no error messages but totally misleading results. So it is essential to spend the time making sure that ALL formats of variables in our data frame are correct before trying to analyse the data. Therefore, always follow the same procedure when working with data in R:

1) Load tidyverse
2) Set your working directory in R
3) Read the data into a data frame in R (typically using either read_excel or read_csv)
4) Examine the formats of the variables within your data frame using str() and decide which variables can be discarded, which are already in the correct format, and which need to be converted into a different format
5) Modify your data frame so that all variables you want to keep are in the correct format (we will disuss how to do this below)
6) Re-examine the structure of your data frame using str() to ensure that everything has worked as expected
7) Examine summary statistics of the data using summary() and basic plots, to ensure that you don't have any strange values that might indicate data entry errors
8) Carry on with your analysis safe in the knowledge that your data is correct!

Steps 4-7 can be quite boring, but they are also very important ... and it is always better to spend time checking that your data is correct now rather than wasting time on incorrect analyses later on.

Creating new variables

In order to be able to format our variables correctly, we need to modify the data frame we have created. Note that we are do NOT want to modify the original Excel file, only the data frame within R. There are several ways to do this, but here we will stick with the tidyverse approach.

The function mutate() allows us to create a new variable based on one or more existing variables. For example, the following code creates a new numeric variable called crl_mm based on the existing numeric variable crl_cm:

```
fetuses %>%
    mutate(crl_mm = crl_cm*10)
```

```
## # A tibble: 262 x 19
##    parity age_days weight_kg crl_cm head_width_mm head_length_mm hair_coronary_b~ hair_ear
##     <dbl>    <dbl>     <dbl>  <dbl>         <dbl>          <dbl> <chr>            <chr>
```

```
##  1      3      274    35.5     86.8         128            239   Y                Y
##  2      2      196    10.2     61.8          83.8          154   Y                Y
##  3      1      259    27.2     89.3         121            220.  Y                Y
##  4      1      249    29.1     83           118            215   Y                Y
##  5      2      221    16.6     75.4         101.           183   Y                Y
##  6      3      221    19.0     77.7          97            189   Y                Y
##  7      2      177     6.35    55.4          74.5          142.  Y                N
##  8      2       82     0.046   10.4          24.3           36.8 N                N
##  9      6      193    11.2     62.4          85.4          153.  Y                N
## 10      4      142     1.77    35.4          62.4           99.8 N                N
## # ... with 252 more rows, and 11 more variables: hair_eyelid <chr>, hair_tail <chr>,
## #   hair_hornbud <chr>, tactile_muzzle <chr>, tactile_eyebrow <chr>, tactile_eyelash <chr>,
## #   eye_op_close <chr>, papillae_tongue <chr>, eyelid <chr>, sex <chr>, crl_mm <dbl>
```

There are three things to note here:

- I entered a new line and tab between the chaining operator %>% and the function I am using: this is just to make the code easier to read by separating each part of the code onto a new line rather than having a single, very long line of code
- This is a very simple mutate where we create a new variable called crl_mm and calculate its values by multiplying the existing variable crl_cm by 10. This creates a new column within the data frame because we don't already have a variable called 'crl_mm' - if you reuse an existing variable name (e.g. crl_mm) then this will replace the previous column within the data frame.
- The mutate function creates a new data frame, but we have not assigned it to anything - so R just shows us the new data frame (which is identical to the old one except for having 19 rather than 18 variables) and then immediately forgets it.

To get R to remember the changes we are making to the data frame we need to assign it to something using the assignment operator. For example, this code creates a new data frame called fetuses_full:

```
fetuses_full <- fetuses %>%
    mutate(crl_mm = crl_cm*10)
```

You should now see a new data frame called fetuses_full appear in your R environment as '262 obs. of 19 variables'. To over-write the existing data frame, we can just assign the new data frame to have the same name as the old one:

```
fetuses <- fetuses %>%
    mutate(crl_mm = crl_cm*10)

fetuses %>%
    str()
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    262 obs. of  19 variables:
##  $ parity            : num  3 2 1 1 2 3 2 2 6 4 ...
##  $ age_days          : num  274 196 259 249 221 221 177 82 193 142 ...
##  $ weight_kg         : num  35.5 10.2 27.1 29.1 16.6 ...
##  $ crl_cm            : num  86.8 61.8 89.3 83 75.4 77.7 55.4 10.4 62.4 35.4 ...
##  $ head_width_mm     : num  128 83.8 121 118 100.6 ...
##  $ head_length_mm    : num  239 154 220 215 183 ...
##  $ hair_coronary_band: chr  "Y" "Y" "Y" "Y" ...
##  $ hair_ear          : chr  "Y" "Y" "Y" "Y" ...
##  $ hair_eyelid       : chr  "Y" "Y" "Y" "Y" ...
##  $ hair_tail         : chr  "Y" "Y" "Y" "Y" ...
##  $ hair_hornbud      : chr  "Y" "Y" "Y" "Y" ...
##  $ tactile_muzzle    : chr  "Visible" "Visible" "Visible" "Visible" ...
```

```
##  $ tactile_eyebrow   : chr  "Visible" "Visible" "Visible" "Visible" ...
##  $ tactile_eyelash   : chr  "Visible" "Visible" "Visible" "Visible" ...
##  $ eye_op_close      : chr  "Open" "Open" "Open" "Open" ...
##  $ papillae_tongue   : chr  "Whole tongue" "Whole tongue" "Whole tongue" "Whole tongue" ...
##  $ eyelid            : chr  "Y" "Y" "Y" "Y" ...
##  $ sex               : chr  "Female" "Female" "Male" "Male" ...
##  $ crl_mm            : num  868 618 893 830 754 777 554 104 624 354 ...
```

The first line of this code is a very common pattern that we will use repeatedly - it tells R to save a data frame called 'fetuses' which is based on an existing data frame called 'fetuses' but with some function(s) (given on additional lines) applied to it using the chaining operator.

Selecting and filtering columns or rows

Sometimes we might be reading in a large Excel dataset (with many columns and/or rows), but we don't want to work with the whole dataset. For example, let's say we are only interested in the parity, age and hair_coronary_band variables. Rather than working on the large dataset, it might be a good idea to create a second data frame with a more restricted dataset - this will be easier (and faster) to work with, but does not remove any information from the original dataset that we have already copied to fetuses_full. We can select columns from fetuses (and over-write this data frame) using the select() function:

```
fetuses <- fetuses %>%
    select(parity, age_days, hair_coronary_band)
```

You should now see that fetuses is 262 obs. or 3 variables (but fetuses_full is still 19 variables so we still have the original data). This makes it easier to look only at what we want:

```
fetuses %>%
    summary()
```

```
##      parity         age_days      hair_coronary_band
##  Min.   :1.000   Min.   : 25.00   Length:262
##  1st Qu.:2.000   1st Qu.: 95.25   Class :character
##  Median :3.000   Median :136.00   Mode  :character
##  Mean   :2.752   Mean   :136.06
##  3rd Qu.:4.000   3rd Qu.:175.00
##  Max.   :6.000   Max.   :274.00
```

We might even go further, and want to exclude the very early fetuses from the dataset we are working with. The filter() function extracts only rows that meet one or more conditions, e.g.:

```
fetuses <- fetuses %>%
    filter(age_days >= 50)
```

Now we have only the 243 obs. from fetuses with a gestational age of 50 days or more.

The great thing about the chaining operator is that it is easy to do multiple functions in a row without stopping in between, for example the two bits of R code above is the same as the following single piece of R code:

```
fetuses <- fetuses_full %>%
    select(parity, age_days, hair_coronary_band) %>%
    filter(age_days >= 50)
```

Note that we can add as many functions to the chain as we want to, as long as we remember to put %>% at the end of every function (except the last function - this terminates the chain). Let's add another function

near the top of this chain using mutate to create a new variable that tells us the row number from the original data that each row in the filtered data corresponds to (just in case we need it later):

```
fetuses <- fetuses_full %>%
    mutate(ID = row_number()) %>%
    select(ID, parity, age_days, hair_coronary_band) %>%
    filter(age_days >= 50)
```

You can see now why it is a good idea to put each function on a new line - otherwise the code would quickly get hard to read! As long as the %>% operator comes at the end of the line (rather than at the start of the new line), then it all works as expected. Remember that all the code within the chain (even if on multiple lines) is a single R command, so it won't work if you try to just run the last line (for example). Fortunately, RStudio is smart enough to know this and will run the entire command if the curser is anywhere on any of the lines within the whole chain of functions.

Converting text variables into factors

When reading Excel files, R will usually store numbers (and date/time variables) into the correct format automatically (unless these columns include some cells with text entries in Excel). But it has no way of reliably formatting our categorical data automatically, so we need to manually convert them into factor variables using 'parse_factor' within a mutate function to convert a character/text variable to a factor. For example:

```
fetuses <- fetuses %>%
    mutate(hair_coronary_band_fct = parse_factor(hair_coronary_band))
```

This creates a new variable by turning the text variable hair_coronary_band into a factor. We can now summarise this more usefully than we could with the text variable:

```
fetuses %>%
    summary()
```

```
##        ID            parity         age_days     hair_coronary_band hair_coronary_band_fct
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   Length:243         Y: 60
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Class :character   N:183
##  Median :122.0   Median :3.000   Median :144.0   Mode  :character
##  Mean   :125.6   Mean   :2.774   Mean   :143.3
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5
##  Max.   :262.0   Max.   :6.000   Max.   :274.0
```

Now we get the number of Y and the number of N for hair_coronary_band_fct. In this case, that works OK - but what would happen if the data had been entered inconsistently, for example if 'n' had sometimes been used instead of 'N'? Let's deliberately introduce an error (don't worry about the code itself - you won't do this in real life!):

```
fetuses$hair_coronary_band[1] <- 'n'
```

We can see the first observation in hair_coronary_band is now 'n':

```
fetuses %>% str()
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    243 obs. of  5 variables:
##  $ ID                  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ parity              : num  3 2 1 1 2 3 2 2 6 4 ...
##  $ age_days            : num  274 196 259 249 221 221 177 82 193 142 ...
##  $ hair_coronary_band  : chr  "n" "Y" "Y" "Y" ...
```

```
##  $ hair_coronary_band_fct: Factor w/ 2 levels "Y","N": 1 1 1 1 1 1 1 2 1 2 ...
```

What happens when this is automatically turned into a factor?

```
fetuses <- fetuses %>%
    mutate(hair_coronary_band_fct = parse_factor(hair_coronary_band))

fetuses %>%
    summary()
```

```
##        ID            parity         age_days     hair_coronary_band hair_coronary_band_fct
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   Length:243         n:  1
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Class :character   Y: 59
##  Median :122.0   Median :3.000   Median :144.0   Mode  :character   N:183
##  Mean   :125.6   Mean   :2.774   Mean   :143.3
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5
##  Max.   :262.0   Max.   :6.000   Max.   :274.0
```

There are now 3 factor levels in hair_coronary_band_fct: Y, N and n. Note that N and n are different, but we don't really want them to be! We can deal with this by first creating the factor as before, and chaining the data frame into a second mutate where we use fct_collapse to manually recode the factor levels so that Y becomes Yes and both n and N become No, like so:

```
fetuses <- fetuses %>%
    mutate(hair_coronary_band_fct = parse_factor(hair_coronary_band)) %>%
    mutate(hair_coronary_band_fct = fct_collapse(hair_coronary_band_fct, No = c('N', 'n'), Yes='Y'))

fetuses %>%
    summary()
```

```
##        ID            parity         age_days     hair_coronary_band hair_coronary_band_fct
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   Length:243         No :184
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Class :character   Yes: 59
##  Median :122.0   Median :3.000   Median :144.0   Mode  :character
##  Mean   :125.6   Mean   :2.774   Mean   :143.3
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5
##  Max.   :262.0   Max.   :6.000   Max.   :274.0
```

Notice also that the factor levels have swapped around; No is now the reference category because this came first in the arguments to the fct_collapse function.

Specifying factor levels

In the real world, we won't want to spend time examining all of the factors to make sure that there aren't any typos - particularly when we think the data should be correct to start with! This can lead to errors, because we might not check as carefully as we should and therefore miss problems with the data. So it is much better practice to tell parse_factor what to expect by using the levels argument:

```
fetuses <- fetuses %>%
    mutate(hair_coronary_band_fct = parse_factor(hair_coronary_band, levels=c('N','Y')))
```

```
## Warning: 1 parsing failure.
## row col           expected actual
##   1  -- value in level set      n
```

When we run this code, we get a warning to tell us that an unexpected text entry 'n' was observed in row 1, and the resulting variable now has 1 missing value (NA):

```
fetuses %>%
    summary()
```

```
##        ID            parity         age_days     hair_coronary_band hair_coronary_band_fct
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   Length:243         N   :183
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Class :character   Y   : 59
##  Median :122.0   Median :3.000   Median :144.0   Mode  :character   NA's:  1
##  Mean   :125.6   Mean   :2.774   Mean   :143.3
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5
##  Max.   :262.0   Max.   :6.000   Max.   :274.0
```

This is good because R has automatically alerted us to a problem that we need to fix. So we can go back and adjust our code so it looks like this:

```
fetuses <- fetuses %>%
    mutate(hair_coronary_band_fct = parse_factor(hair_coronary_band, levels=c('N','n','Y'))) %>%
    mutate(hair_coronary_band_fct = fct_collapse(hair_coronary_band_fct, No = c('N', 'n'), Yes='Y'))
```

This tells R to expect the 'n' entry, but then immediately re-codes it afterwards - this approach will fix known problems in the data, but still automatically alert us to any new problems that might occur if e.g. we update the data to include more observations (and different mistakes!).

There is one additional thing to remember with parse_factor: if there were observations in the original data that were missing before making them into a factor, then these will be treated differently to observations that were present but corresponded to unknown factor levels. For example, let's deliberately make the second observation of hair_coronary_band blank (i.e. missing) and then re-run the code with only N and Y as specified levels:

```
fetuses$hair_coronary_band[2] <- ''

fetuses <- fetuses %>%
    mutate(hair_coronary_band_fct = parse_factor(hair_coronary_band, levels=c('N','Y')))
```

```
## Warning: 1 parsing failure.
## row col           expected actual
##   1  -- value in level set      n
```

We get the red text again (telling us about the 'n' on row 1), but no mention of the missing entry on row 2. This is because parse_factor has created an explicit factor level for missing data:

```
fetuses %>%
    summary()
```

```
##        ID            parity         age_days     hair_coronary_band hair_coronary_band_fct
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   Length:243         N   :183
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Class :character   Y   : 58
##  Median :122.0   Median :3.000   Median :144.0   Mode  :character   NA  :  1
##  Mean   :125.6   Mean   :2.774   Mean   :143.3                      NA's:  1
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5
##  Max.   :262.0   Max.   :6.000   Max.   :274.0
```

We have both the factor level NA (the value that was missing to start with) and NA's (the value of 'n' that became missing because it wasn't included in the allowed levels). But this subtle distinction is not visible in the data itself:

```
fetuses
```

```
## # A tibble: 243 x 5
##       ID parity age_days hair_coronary_band hair_coronary_band_fct
##    <int>  <dbl>    <dbl> <chr>              <fct>
##  1     1      3      274 n                  <NA>
##  2     2      2      196 ""                 <NA>
##  3     3      1      259 Y                  Y
##  4     4      1      249 Y                  Y
##  5     5      2      221 Y                  Y
##  6     6      3      221 Y                  Y
##  7     7      2      177 Y                  Y
##  8     8      2       82 N                  N
##  9     9      6      193 Y                  Y
## 10    10      4      142 N                  N
## # ... with 233 more rows
```

Having this distinction can sometimes be useful, but if we want to remove the NA factor level then we can specify the include_na = FALSE argument e.g.:

```
fetuses <- fetuses %>%
    mutate(hair_coronary_band_fct = parse_factor(hair_coronary_band, levels=c('N','Y'), include_na = FA
```

```
## Warning: 1 parsing failure.
## row col           expected actual
##   1  -- value in level set      n
```

```
fetuses %>%
    summary()
```

```
##        ID            parity         age_days     hair_coronary_band hair_coronary_band_fct
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   Length:243         N   :183
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Class :character   Y   : 58
##  Median :122.0   Median :3.000   Median :144.0   Mode  :character   NA's:  2
##  Mean   :125.6   Mean   :2.774   Mean   :143.3
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5
##  Max.   :262.0   Max.   :6.000   Max.   :274.0
```

Now there is no distinction between the blank entry and the entry that was originally 'n' - they are both NA's. Alternatively, you can remove the NA factor level (or any other factor level that you want to) by setting it to NULL within fct_collapse:

```
fetuses <- fetuses %>%
    mutate(hair_coronary_band = parse_factor(hair_coronary_band, levels=c('N','n','Y'))) %>%
    mutate(hair_coronary_band = fct_collapse(hair_coronary_band, NULL = 'NA', No = c('N', 'n'), Yes='Y'

fetuses %>%
    summary()
```

```
##        ID            parity         age_days     hair_coronary_band hair_coronary_band_fct
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   No  :184          N   :183
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Yes : 58          Y   : 58
##  Median :122.0   Median :3.000   Median :144.0   NA's:  1          NA's:  2
##  Mean   :125.6   Mean   :2.774   Mean   :143.3
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5
##  Max.   :262.0   Max.   :6.000   Max.   :274.0
```

Notice this time that I have replaced the existing variable hair_coronary_band rather than creating a new

variable - generally we will want to do this when creating factors, as the original text entries are no longer needed. This has destroyed the original text variable though, so we will not be able to re-run this line of code without running the entire R code from the beginning: the easiest way to do this is by clicking on source.

The only difference between the factors hair_coronary_band and hair_coronary_band_fct is that 'n' was made into 'No' in hair_coronary_band, but left as missing in hair_coronary_band_fct. Other than for demonstration purposes we don't really need hair_coronary_band_fct, so it is best to remove it - we can do this using select and a minus symbol in front of the variable name to indicate it is to be removed rather than retained:

```
fetuses <- fetuses %>%
    select(-hair_coronary_band_fct)

fetuses %>%
    str()
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    243 obs. of  4 variables:
##  $ ID                : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ parity            : num  3 2 1 1 2 3 2 2 6 4 ...
##  $ age_days          : num  274 196 259 249 221 221 177 82 193 142 ...
##  $ hair_coronary_band: Factor w/ 2 levels "No","Yes": 1 NA 2 2 2 2 2 1 2 1 ...
```

Now we have 4 variables: ID, parity, age_days and hair_coronary_band as a factor with No/Yes recordings (and one missing observation).

Section 2

Converting numbers into factors

Most of the time we will be creating factors from text entries, but occasionally we will want to create factors from numbers. The best approach to use depends on if the number is a discrete number (i.e. whole numbers only), or a continuous number. Let's take them one at a time.

For discrete numbers, we can use the parse_factor() function like before - but we first need to turn the integer into a character (text) variable. It is also a good idea to add some text to the number so that the resulting factor is more easily distinguishable from a 'true number'. We can accomplish both tasks simultaneously using the str_c() function, for example:

```
fetuses %>%
    mutate(parity_text = str_c('Parity_', parity)) %>%
    mutate(parity_fct = parse_factor(parity_text)) %>%
    str()
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    243 obs. of  6 variables:
##  $ ID                : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ parity            : num  3 2 1 1 2 3 2 2 6 4 ...
##  $ age_days          : num  274 196 259 249 221 221 177 82 193 142 ...
##  $ hair_coronary_band: Factor w/ 2 levels "No","Yes": 1 NA 2 2 2 2 2 1 2 1 ...
##  $ parity_text       : chr  "Parity_3" "Parity_2" "Parity_1" "Parity_1" ...
##  $ parity_fct        : Factor w/ 6 levels "Parity_3","Parity_2",..: 1 2 3 3 2 1 2 2 4 5 ...
```

I've broken this down into stages so that you can see what is going on. The first step uses str_c() to create a new character variable by sticking the text 'Parity_' before the parity number, and the second step creates a

factor from the character variable as we did before. The only problem with this is that we haven't specified levels to parse_factor(), so if we have any errors in the data (e.g. a negative number for parity) then we won't detect these. A good trick is to automatically create all possible parity levels without having to type them all out:

```
str_c('Parity_', seq(from=1, to=10, by=1))
```

```
##  [1] "Parity_1"  "Parity_2"  "Parity_3"  "Parity_4"  "Parity_5"  "Parity_6"  "Parity_7"  "Parity_8"
##  [9] "Parity_9"  "Parity_10"
```

The seq() function creates all possible numbers between from (1) and to (10), with increments of by (1). We can then use this in the parse_factor function, and combine all steps of the code together:

```
fetuses <- fetuses %>%
    mutate(parity_fct = parse_factor(str_c('Parity_', parity), levels=str_c('Parity_', seq(from=1, to=1
    mutate(parity_group = fct_collapse(parity_fct, First='Parity_1', Second='Parity_2', Older=str_c('Pa

fetuses %>%
    summary()
```

```
##        ID             parity         age_days     hair_coronary_band    parity_fct  parity_group
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   No  :184           Parity_2:79   First  : 40
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Yes : 58           Parity_3:56   Second : 79
##  Median :122.0   Median :3.000   Median :144.0   NA's:  1           Parity_4:45   Older  :124
##  Mean   :125.6   Mean   :2.774   Mean   :143.3                      Parity_1:40   Ancient:  0
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5                      Parity_6:13
##  Max.   :262.0   Max.   :6.000   Max.   :274.0                      Parity_5:10
##                                                                     (Other) : 0
```

This code looks quite complicated, but if you break it down into steps you should be able to see what is going on. The first mutate function just combines the separate steps we had before - first using str_c() to add the text 'Parity_' to the parity number, then parsing this into a factor using specified levels. The second mutate function uses fct_collapse to create a second factor of parity_group, where we combine parities 3-9 into a single group.

Notice here that we specified a factor level that doesn't exist in the data - there are no Parity_10 cows (and therefore no Ancient cows). In some situations this is helpful: we can include a possible factor level and R will explictly tell us that there are no observations within that category. But if we want to remove the unused factor levels, we can do this using the fct_drop function:

```
fetuses <- fetuses %>%
    mutate(parity_fct = fct_drop(parity_fct), parity_group = fct_drop(parity_group))

fetuses %>%
    summary()
```

```
##        ID             parity         age_days     hair_coronary_band    parity_fct parity_group
##  Min.   :  1.0   Min.   :1.000   Min.   : 51.0   No  :184           Parity_1:40   First : 40
##  1st Qu.: 61.5   1st Qu.:2.000   1st Qu.:107.5   Yes : 58           Parity_2:79   Second: 79
##  Median :122.0   Median :3.000   Median :144.0   NA's:  1           Parity_3:56   Older :124
##  Mean   :125.6   Mean   :2.774   Mean   :143.3                      Parity_4:45
##  3rd Qu.:188.5   3rd Qu.:4.000   3rd Qu.:176.5                      Parity_5:10
##  Max.   :262.0   Max.   :6.000   Max.   :274.0                      Parity_6:13
```

There is now no more mention of 'Ancient' cows in parity_group. Notice here that rather than using two separate mutate() functions, I mutated two variables within the same function separated by a comma. The two approaches are the same.

There is a second discrete number that makes sense to convert into a factor: our ID variable is currently

stored as a number, but it makes more sense to think of it as a grouping variable i.e. a factor. Sometimes this doesn't make any difference, but if we want to use ID either to merge datasets, or as a stratifying variable in a plot or in a model, then we might get the wrong result if ID is a number. So we had better turn it into a factor:

```
fetuses <- fetuses %>%
    mutate(ID_simple = parse_factor(str_c('ID_', ID)))  %>%
    mutate(ID = parse_factor(str_c('ID_', str_replace_all(format(ID),' ','0'))))

fetuses
```

```
## # A tibble: 243 x 7
##    ID     parity age_days hair_coronary_band parity_fct parity_group ID_simple
##    <fct>   <dbl>    <dbl> <fct>              <fct>      <fct>        <fct>
##  1 ID_001      3      274 No                 Parity_3   Older        ID_1
##  2 ID_002      2      196 <NA>               Parity_2   Second       ID_2
##  3 ID_003      1      259 Yes                Parity_1   First        ID_3
##  4 ID_004      1      249 Yes                Parity_1   First        ID_4
##  5 ID_005      2      221 Yes                Parity_2   Second       ID_5
##  6 ID_006      3      221 Yes                Parity_3   Older        ID_6
##  7 ID_007      2      177 Yes                Parity_2   Second       ID_7
##  8 ID_008      2       82 No                 Parity_2   Second       ID_8
##  9 ID_009      6      193 Yes                Parity_6   Older        ID_9
## 10 ID_010      4      142 No                 Parity_4   Older        ID_10
## # ... with 233 more rows
```

Two different methods are used here - the first simple method works fine, but the resulting IDs are ID_1, ID_2, ... ID_10, ID_11, ... ID_99, ID_100 etc. So when these are sorted alphabetically, they end up in a different order to the numbers themselves. The second method is slightly more complex but gives us ID numbers ID_001, ID_002, ... ID_010, ID_011, ... ID_099, ID_100 etc so that the order is preserved. If you are interested in how the format() and str_replace_all() functions work, look at their help files by typing ?format and ?str_replace_all in R.

The method for turning continous numbers into factors is a little different: it uses the cut() function rather than parse_string. The cut function allows you to break a continuous variable into segments using specified breakpoints and corresponding labels for the factor levels. For example we can turn the age into a factor to reflect an early, mid or late stage foetus:

```
fetuses <- fetuses %>%
    mutate(age_group = cut(age_days, breaks=c(0, 100, 200, 300), labels=c('Early', 'Middle', 'Late')))

fetuses %>%
    summary()
```

```
##        ID          parity         age_days     hair_coronary_band    parity_fct parity_group
##  ID_001 :  1   Min.   :1.000   Min.   : 51.0   No  :184           Parity_1:40   First : 40
##  ID_002 :  1   1st Qu.:2.000   1st Qu.:107.5   Yes : 58           Parity_2:79   Second: 79
##  ID_003 :  1   Median :3.000   Median :144.0   NA's:  1           Parity_3:56   Older :124
##  ID_004 :  1   Mean   :2.774   Mean   :143.3                      Parity_4:45
##  ID_005 :  1   3rd Qu.:4.000   3rd Qu.:176.5                      Parity_5:10
##  ID_006 :  1   Max.   :6.000   Max.   :274.0                      Parity_6:13
##  (Other):237
##    ID_simple    age_group
##  ID_1   :  1   Early : 54
##  ID_2   :  1   Middle:159
```

```
##  ID_3    :  1   Late  : 30
##  ID_4    :  1
##  ID_5    :  1
##  ID_6    :  1
##  (Other):237
```

The breaks argument must always specify the lowest and highest possible values as the first and last numbers, so there is always one more group label than there are breaks. Here we used 0 and 300 as the lower and upper limits, so a value of <0 or >300 would be converted to a missing value in the factor. If you want to include all possible values, then you can specify the limits as -Inf and Inf - meaning negative infinity and infinity.

We can also use the cut function with discrete numbers, although there is a catch: we have to remember that if a data value is exactly equal to a break value then it goes into the category below. Alternatively, you can specify the breaks as being mid-way between the discrete values to make the code more obvious, for example:

```
fetuses <- fetuses %>%
    mutate(parity_fct_cut = cut(parity, breaks=c(0.5, 1.5, 2.5, 10.5), labels=c('First','Second','Older

fetuses %>%
    summary()
```

```
##        ID          parity         age_days     hair_coronary_band    parity_fct parity_group
##  ID_001 :  1   Min.   :1.000   Min.   : 51.0   No  :184           Parity_1:40   First : 40
##  ID_002 :  1   1st Qu.:2.000   1st Qu.:107.5   Yes : 58           Parity_2:79   Second: 79
##  ID_003 :  1   Median :3.000   Median :144.0   NA's:  1           Parity_3:56   Older :124
##  ID_004 :  1   Mean   :2.774   Mean   :143.3                      Parity_4:45
##  ID_005 :  1   3rd Qu.:4.000   3rd Qu.:176.5                      Parity_5:10
##  ID_006 :  1   Max.   :6.000   Max.   :274.0                      Parity_6:13
##  (Other):237
##    ID_simple    age_group   parity_fct_cut
##  ID_1    :  1   Early : 54   First : 40
##  ID_2    :  1   Middle:159   Second: 79
##  ID_3    :  1   Late  : 30   Older :124
##  ID_4    :  1
##  ID_5    :  1
##  ID_6    :  1
##  (Other):237
```

This parity_fct_cut variable is identical to the parity_fct variable that we created above.

More variable types

Although our fetuses data only has numeric and factor variable types, we will sometimes encounter other types of data. The most common of these are dates, date/times, ordinal, logical, and numbers with specialised formats (e.g. currency symbols).

Dates

Dates are a very specific type of variable that we can do special things with, such as calculating the number of days between two dates, and extracting parts of the date e.g. the year, day of the week, month name etc. But in order to be able to do this, R must first store the variable as a date. When reading an Excel file using read_excel this is usually done automatically, but sometimes we can end up with a text variable that contains text representing the date (this will always be the case when reading data from a CSV file using read_csv). In this case, we can use the parse_date function if the date is written in the ISO standard format

of YYYY-MM-DD. If it is in a non-standard format (which is unfortunately more common than the standard format), then you will need to tell R what format it is in. The lubridate package has some helpful functions to do this, for example:

```
library('lubridate')
```

```
##
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:base':
##
##     date
```

```
ymd('2010/10/21')
```

```
## [1] "2010-10-21"
```

```
make_date(year=2010, month=10, day=21)
```

```
## [1] "2010-10-21"
```

```
dmy('21-10-2010') + weeks(1)
```

```
## [1] "2010-10-28"
```

```
mdy('10-21-2010') - ymd('2010/9/21')
```

```
## Time difference of 30 days
```

```
as.numeric(mdy('10-21-2010') - ymd('2010/10/14'), units='weeks')
```

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

Date/times

Dates are relatively simple to work with, but date/times have the additional complexity of time zones. Unless you specifically need the time to be represented, you are much better off with simple dates. Unfortunately, Excel will save dates as date/time objects so we end up with these when we import them to R. For this reason, it is often a good idea to convert these to a simple date, for example:

```
dt <- as.POSIXct('2010-02-15 05:00')
dt
```

```
## [1] "2010-02-15 05:00:00 CET"
```

```
as.Date(dt)
```

```
## [1] "2010-02-15"
```

Ordinal

There is an important theoretical difference between ordinal data (where there is an order to the categories observed, e.g. body condition score) and categorical data (where there is no logical order, e.g. breed). This distinction is less important in R, so we tend to use factors to represent both. However, there is a specific type of factor that it is occasionally useful to use, called an ordered factor. These are created using the same parse_factor and cut functions that we explored earlier, but with the additional argument: ordered=TRUE. This will make a subtle difference to the modelling functions, but will make no difference to plotting or stratifying by the variable. Just temember that the order of the factor levels is more important for ordered factors than it is for standard (non-ordered) factors.

Logical

We can represent binary data as logical (FALSE/TRUE), and R has a special type called logical that can be used for this. It is possible to use this type for e.g. results of diagnostic tests, where there is a logical FALSE/TRUE outcome. However, the same thing can also be represented using a factor with two levels Negative/Positive, so for the sake of consistency it is generally a good idea to just stick to using factors.

Numbers with special formats

Sometimes we will want to read numbers from text, if for example there are non-numeric characters such as £ or € in the text. We can use the parse_number function to do this:

```
price <- "€103,401"
parse_number(price)
```

```
## [1] 103401
```

Notice that R is english-centric by default, so a danish-formatted number will be parsed incorrectly:

```
danish_price <- "DKK 103.401,00"
parse_number(danish_price)
```

```
## [1] 103.401
```

We get DKK 103,40 rather than DKK 103.401 - but this can be overcome by setting the locale:

```
danish_price <- "DKK 103.401,00"
parse_number(danish_price, locale=locale(decimal_mark = ",", grouping_mark = "."))
```

```
## [1] 103401
```

Converting multiple columns at once

Let's go back to the original dataset fetuses_full:

```
fetuses_full %>%
    summary()
```

```
##      parity         age_days        weight_kg          crl_cm       head_width_mm
##  Min.   :1.000   Min.   : 25.00   Min.   : 0.0010   Min.   :  2.00   Min.   :  3.00
##  1st Qu.:2.000   1st Qu.: 95.25   1st Qu.: 0.2537   1st Qu.: 18.30   1st Qu.: 39.24
##  Median :3.000   Median :136.00   Median : 1.7805   Median : 35.65   Median : 63.33
##  Mean   :2.752   Mean   :136.06   Mean   : 5.0029   Mean   : 37.90   Mean   : 62.34
##  3rd Qu.:4.000   3rd Qu.:175.00   3rd Qu.: 6.9250   3rd Qu.: 54.60   3rd Qu.: 84.02
##  Max.   :6.000   Max.   :274.00   Max.   :41.6000   Max.   :101.30   Max.   :152.00
##  head_length_mm  hair_coronary_band   hair_ear         hair_eyelid         hair_tail
##  Min.   :  5.00   Length:262         Length:262         Length:262         Length:262
##  1st Qu.: 55.52   Class :character   Class :character   Class :character   Class :character
##  Median :101.28   Mode  :character   Mode  :character   Mode  :character   Mode  :character
##  Mean   :103.29
##  3rd Qu.:145.54
##  Max.   :239.00
```

```
##  hair_hornbud       tactile_muzzle     tactile_eyebrow    tactile_eyelash    eye_op_close
##  Length:262         Length:262         Length:262         Length:262         Length:262
##  Class :character   Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##  papillae_tongue       eyelid              sex                crl_mm
##  Length:262         Length:262         Length:262         Min.   :  20.0
##  Class :character   Class :character   Class :character   1st Qu.: 183.0
##  Mode  :character   Mode  :character   Mode  :character   Median : 356.5
##                                                           Mean   : 379.0
##                                                           3rd Qu.: 546.0
##                                                           Max.   :1013.0
```

We now know how to format each of these variables as either a factor or number one by one - for example:

```
fetuses <- fetuses_full %>%
    mutate(hair_coronary_band = parse_factor(hair_coronary_band, levels=c('N','Y'))) %>%
    mutate(hair_ear = parse_factor(hair_ear, levels=c('N','Y'))) %>%
    mutate(hair_eyelid = parse_factor(hair_eyelid, levels=c('N','Y'))) %>%
    mutate(hair_tail = parse_factor(hair_tail, levels=c('N','Y'))) %>%
    mutate(hair_hornbud = parse_factor(hair_hornbud, levels=c('N','Y')))
```

And so on. The trick is to copy and paste the code and just change the name of the variable each time - it is a bit tedious but it isn't difficult. With a bit of practice, you will be surprised how quickly you can get this done (unless there are A LOT of variables).

A good tip is to quickly change all character (text) variables into factors and then immediately use summary on the result: this allows us to see what different values the data contains for each of the text variables. We can do this using mutate_if and the condition is_character:

```
fetuses %>%
    mutate_if(is_character, parse_factor) %>%
    summary()
```

```
##      parity         age_days        weight_kg           crl_cm       head_width_mm
##  Min.   :1.000   Min.   : 25.00   Min.   : 0.0010   Min.   :  2.00   Min.   :  3.00
##  1st Qu.:2.000   1st Qu.: 95.25   1st Qu.: 0.2537   1st Qu.: 18.30   1st Qu.: 39.24
##  Median :3.000   Median :136.00   Median : 1.7805   Median : 35.65   Median : 63.33
##  Mean   :2.752   Mean   :136.06   Mean   : 5.0029   Mean   : 37.90   Mean   : 62.34
##  3rd Qu.:4.000   3rd Qu.:175.00   3rd Qu.: 6.9250   3rd Qu.: 54.60   3rd Qu.: 84.02
##  Max.   :6.000   Max.   :274.00   Max.   :41.6000   Max.   :101.30   Max.   :152.00
##  head_length_mm   hair_coronary_band hair_ear hair_eyelid hair_tail hair_hornbud     tactile_muzzle
##  Min.   :  5.00   N:202              N:221    N:197       N:179     N:175        Visible    :169
##  1st Qu.: 55.52   Y: 60              Y: 41    Y: 65       Y: 83     Y: 87        Not visible: 34
##  Median :101.28                                                                 Hairsack   : 59
##  Mean   :103.29
##  3rd Qu.:145.54
##  Max.   :239.00
##     tactile_eyebrow    tactile_eyelash eye_op_close      papillae_tongue eyelid        sex
##  Visible    :169    Visible    :109    Open  : 37   Whole tongue :164    Y:241   Female  :110
##  Not visible: 43    Not visible:130    Closed:225   None         : 35    N: 21   Male    :128
##  Hairsack   : 50    Hairsack   : 23                 Large front  : 35            Non Diff: 24
##                                                    Furthest back: 28
##
```

```
##
##      crl_mm
##  Min.   :  20.0
##  1st Qu.: 183.0
##  Median : 356.5
##  Mean   : 379.0
##  3rd Qu.: 546.0
##  Max.   :1013.0
```

We don't specify the levels manually, but that's OK because all we wanted to do is quickly see what is in there - we are not saving the result (or any potential issues generated by data entry mistakes).

A more advanced trick if we do have a lot of variables (and they are either grouped together in the data frame or have a similar naming convention) is to use mutate_at to save us some work:

```
fetuses <- fetuses_full %>%
    mutate_at(vars(starts_with('hair')), parse_factor, levels=c('N','Y')) %>%
    mutate_at(vars(hair_coronary_band:hair_hornbud), fct_collapse, No='N', Yes='Y')
```

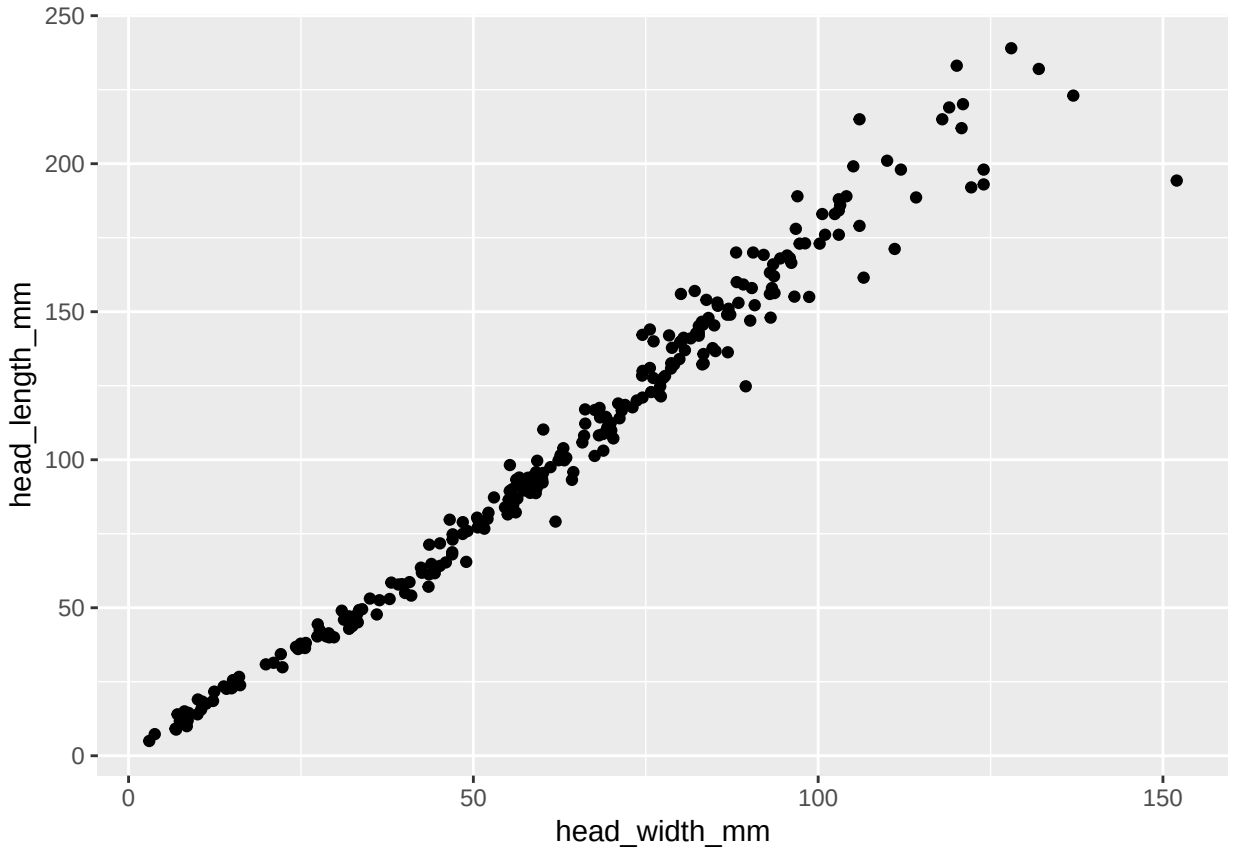
I used both options here for illustration - first using parse_factor on any variable starting with 'hair', and second using fct_collapse to rename N to No and Y to Yes on all variables between hair_coronary_band and hair_hornbud. The most appropriate choice depends on how your variables are named, and their order within the data frame. One thing to notice is that the levels arguments are given to the mutate_at function rather than parse_factor and fct_collapse - variations like this are another reason why it is such a good idea to copy and paste code rather than trying to remember how to write it from scratch every time. But don't worry if you find this example confusing - it is more advanced code, and you can achieve the same result using the simpler methods above.

Section 3

Basic plots

We can get a lot of information using the summarise() variable, but sometimes it is better to visualise data graphically to check that the observations are consistent with what we expect. There are several ways to make plots in R, but the tidyverse way is to use ggplot. These plots all start with the same pattern - first we define the data frame we want to use, then we specify which variables should go on the x and y axes (as well as any variables that can be used to group by colour etc) using the aes() function, then we add the 'layers' that we want to the plot. For example:

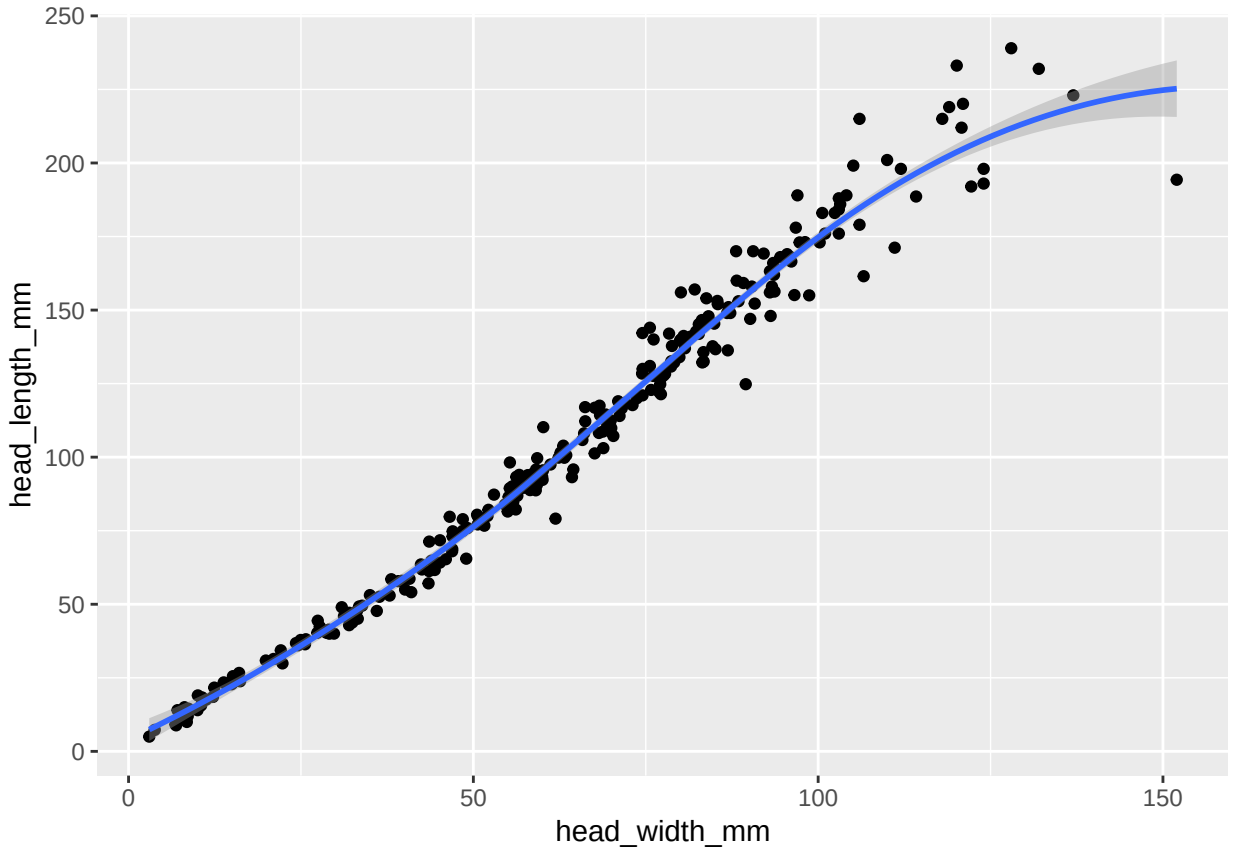
```
ggplot(fetuses, aes(x=head_width_mm, y=head_length_mm)) +
    geom_point()
```



The first line says that we want to use thw fetuses dataset, and that we want head_width_mm on the x-axis and head_length_mm on the y-axis. The first line doesn't tell R what type of plot we want, but it ends with a + symbol, which is a lot like the %>% operator in that it tells R that we haven't completed the command. The second line tells R that we want to add points to the plot, so we end up with a simple scatter-type plot. We can also add a smoothed line of best fit as a second layer using:

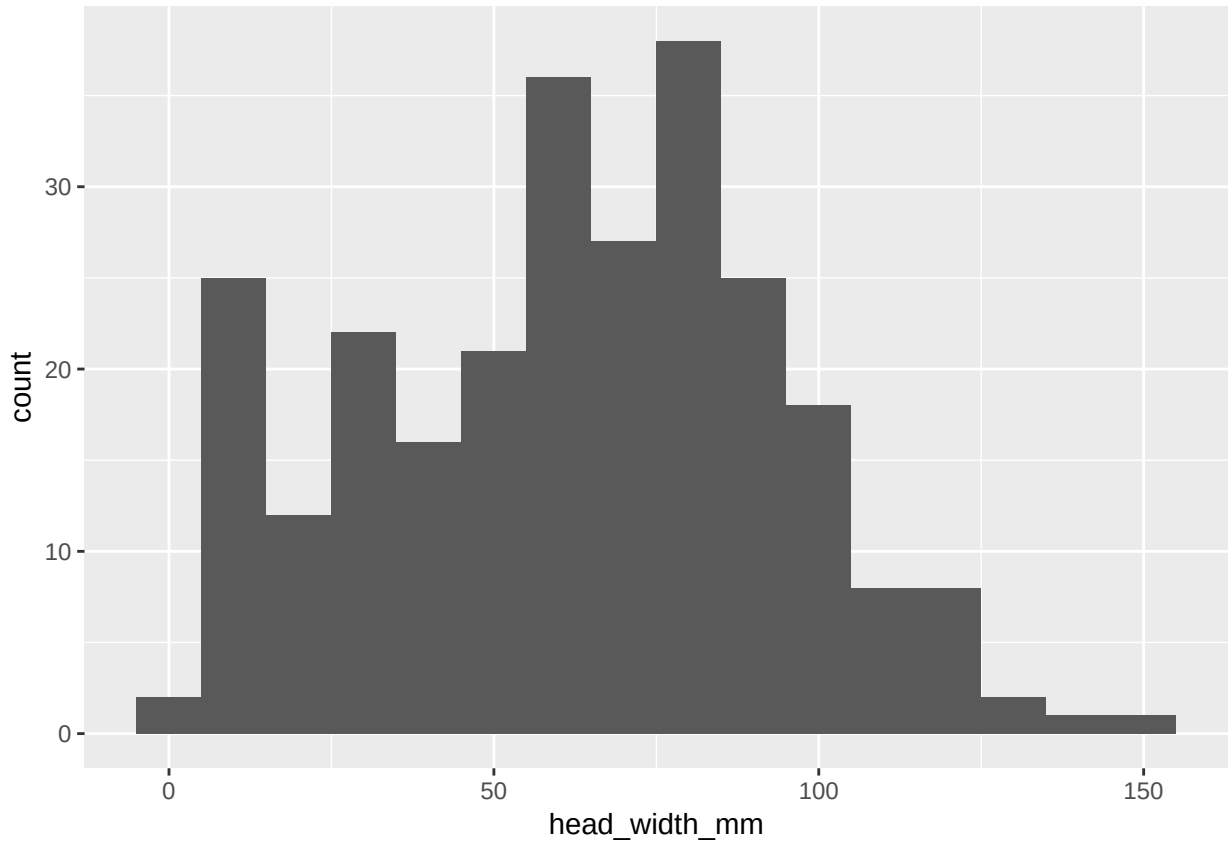
```
ggplot(fetuses, aes(x=head_width_mm, y=head_length_mm)) +
    geom_point() +
    stat_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

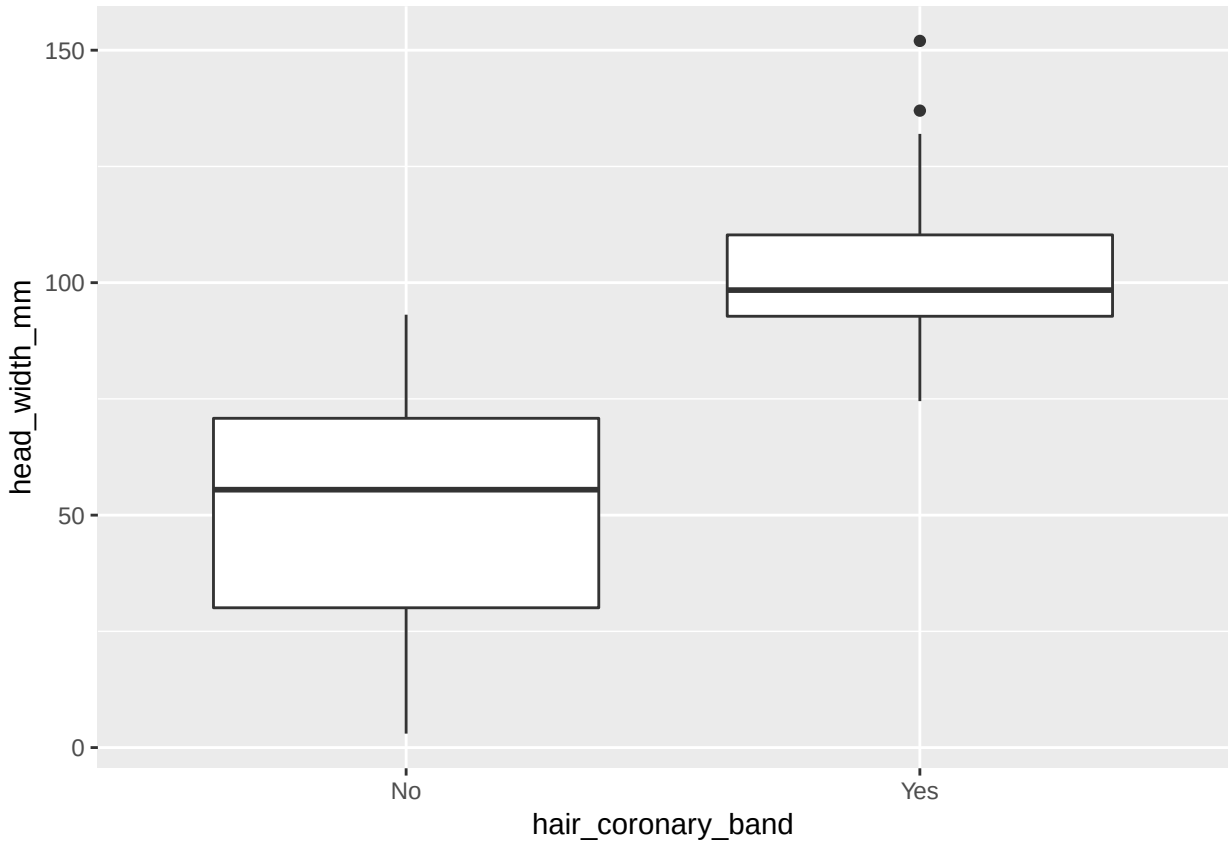


There are two types of plot that are particularly useful for summarising data: histograms and box plots. We can create these using similar code:

```
ggplot(fetuses, aes(x=head_width_mm)) +
    geom_histogram(binwidth=10)
```



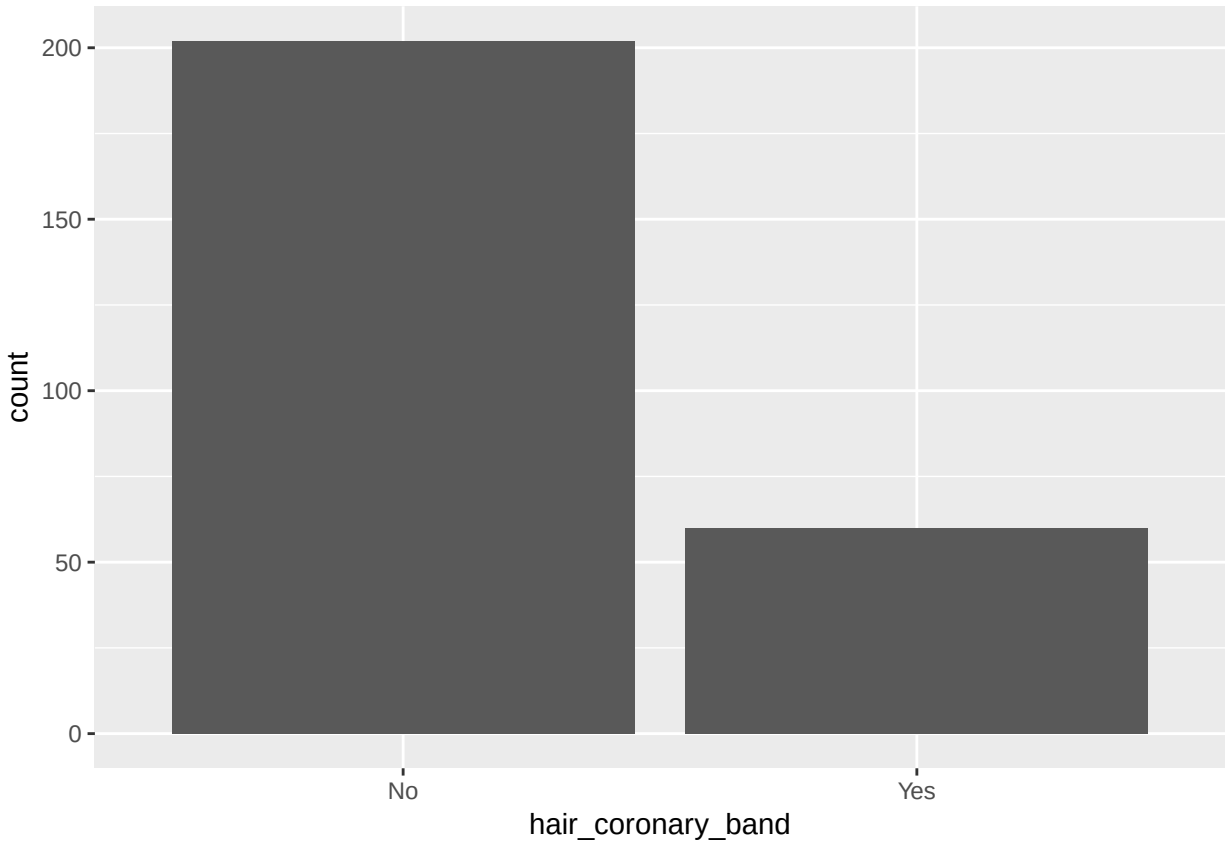

```
ggplot(fetuses, aes(y=head_width_mm, x=hair_coronary_band)) +
    geom_boxplot()
```



Notice that the histogram does not need us to specify anythig for the y-axis: the frequencies are automatically calculated for us. These two plots tell us that head_width_mm is approximately normally distributed, and that it is strongly associated with the presence of hair on the coronary band. There don't seem to be any extreme values, so there is no evidence of data entry mistakes for this variable based on these plots.

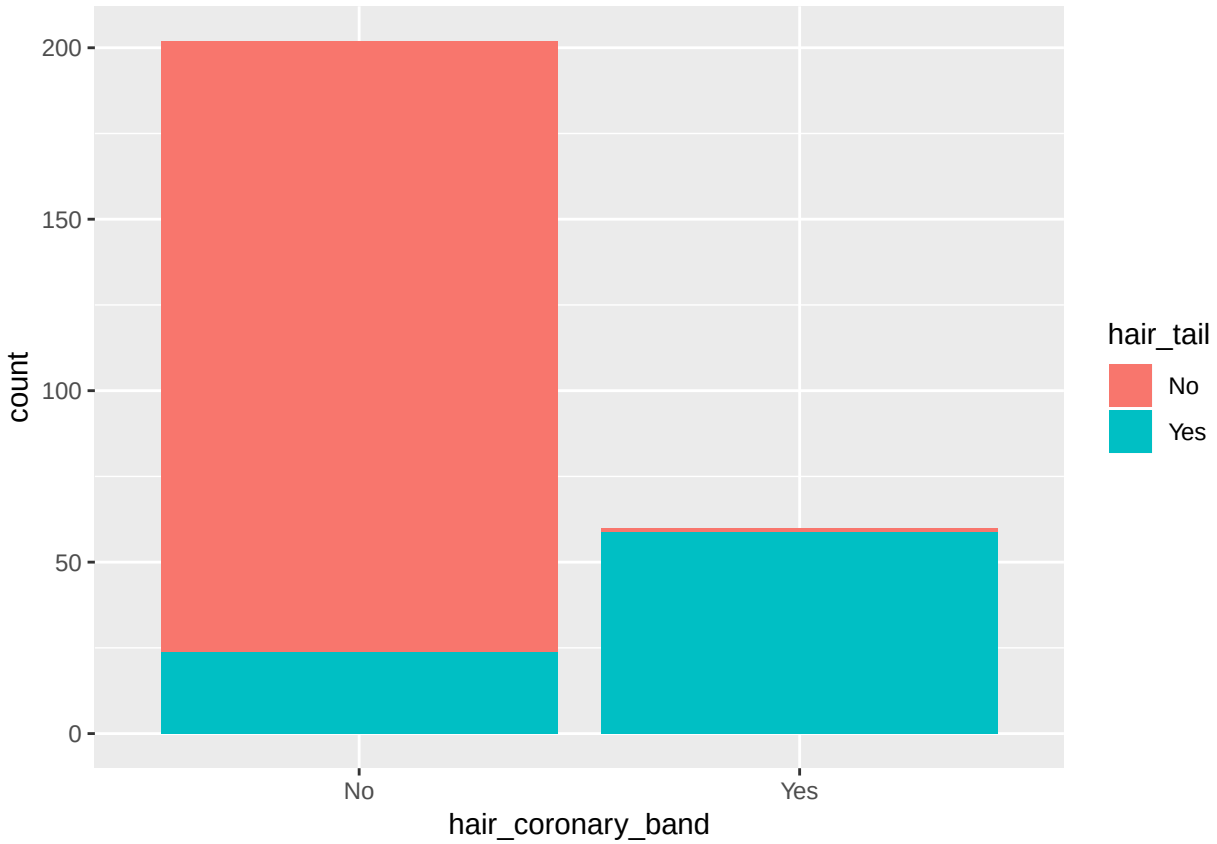
We might also want to make a bar plot to check how categorical variables relate to each other. For example:

```
ggplot(fetuses, aes(x=hair_coronary_band)) +
    geom_bar()
```



This tells us that most of the fetuses do not have hair on the corony band. If we wanted to see how this related to another categorical variable, then we can create a stacked bar chart by using the fill argument to aes:

```
ggplot(fetuses, aes(x=hair_coronary_band, fill=hair_tail)) +
    geom_bar()
```



We will do a lot more with plots later in the course, but if you want to read more about ggplot now then see the Data visualisation chapter of the R for Data Science book: https://r4ds.had.co.nz/data-visualisation.html There is also a cookbook for common plot recipes using ggplot at: http://www.cookbook-r.com/Graphs/

Long and wide data

Sometimes we encounter datasets that are structured in a way that might be good for data entry but is bad for data analysis. For example, we might have repeated observations of the same variable over time within the same individual represented as multiple columns within the worksheet: this 'wide format' is a common way to enter the data, but makes it difficult to summarise all the observations from the same individual using R. For example we might have the following dataset (that I will create directly in R using the tribble() function just for illustration):

```
wide <- tribble(~ID, ~Time1, ~Time2, ~Time3,
                                'Ben', 78, 75, 73,
                                'Geoff', 91, 87, 89)
wide
```

```
## # A tibble: 2 x 4
##   ID    Time1 Time2 Time3
##   <chr> <dbl> <dbl> <dbl>
## 1 Ben      78    75    73
## 2 Geoff    91    87    89
```

Here we have 3 repeated observations of the same variable (body weight), at 3 different time points for 2 individuals. Now let's say we want to plot these weights over time - how do we do that? For data analysis we

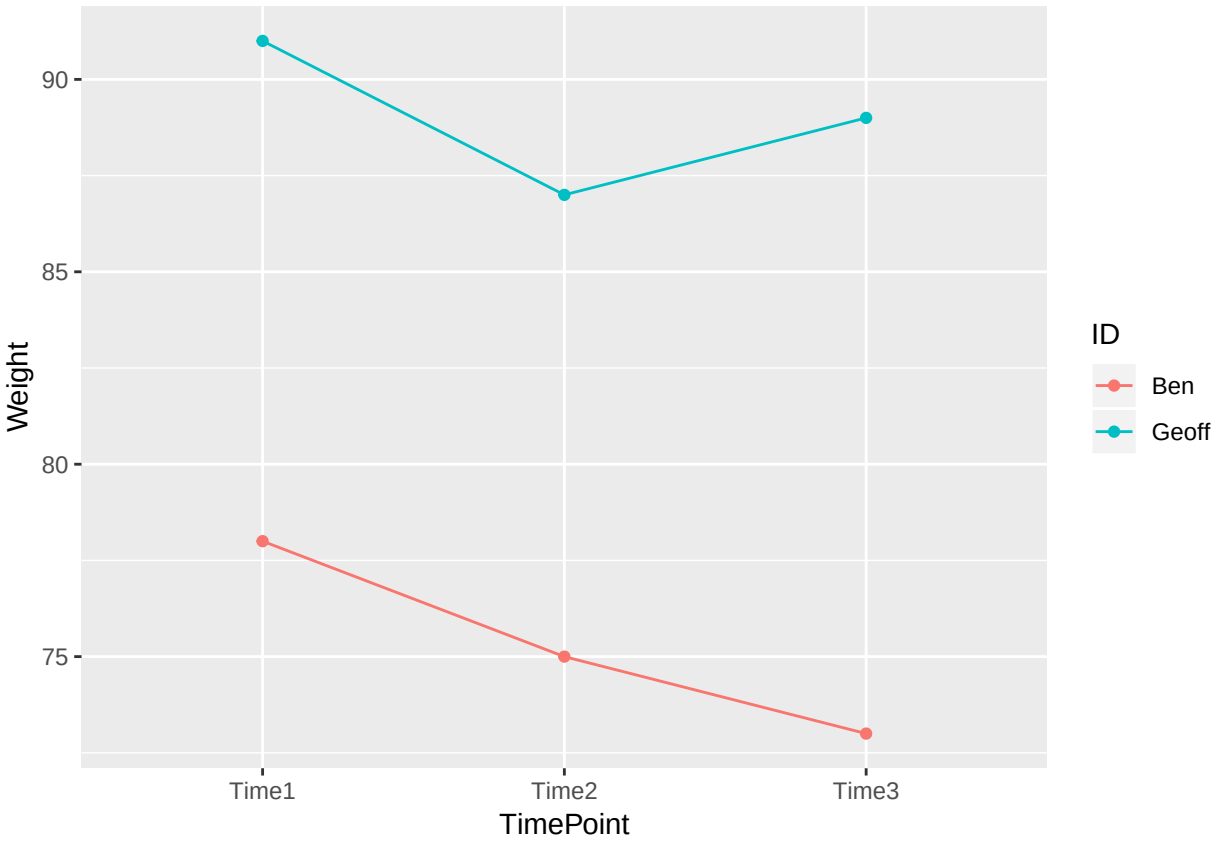
really want the data in 'long format', where we have a column for the ID of the individual, a column for the time point, and a single column for all of the observations. So to create the plot, we first convert the data to long format using the gather() function:

```
long <- wide %>%
    gather(TimePoint, Weight, -ID) %>%
    mutate(TimePoint = parse_factor(TimePoint)) %>%
    mutate(ID = parse_factor(ID, levels=c('Ben','Geoff')))
long
```

```
## # A tibble: 6 x 3
##   ID    TimePoint Weight
##   <fct> <fct>      <dbl>
## 1 Ben   Time1         78
## 2 Geoff Time1         91
## 3 Ben   Time2         75
## 4 Geoff Time2         87
## 5 Ben   Time3         73
## 6 Geoff Time3         89
```

The first 2 arguments of the gather function allows us to specify the new column name for the grouping variable (the previous column names), and the new column name for the observation variable. The remaining arguments indicate variables (with a minus in front of the name) that we want to exclude from the gather process: these variables (in our case just one variable: ID) will be added to the resulting long data in the same format, but with rows repeated as necessary. We then use the parse_factor function to convert the text variables TimePoint and ID into factors. Notice that we don't bother to specify the levels for the new TimePoint variable: this is because the variable is taken directly from the old column names, so there is no risk of any data entry mistakes! The result is exactly the same data, but represented differently: the wide data has the 6 observations in 3 columns of 2 rows, and the long data has the same observations in a single variable with 6 rows. Now we can make the plot we wanted:

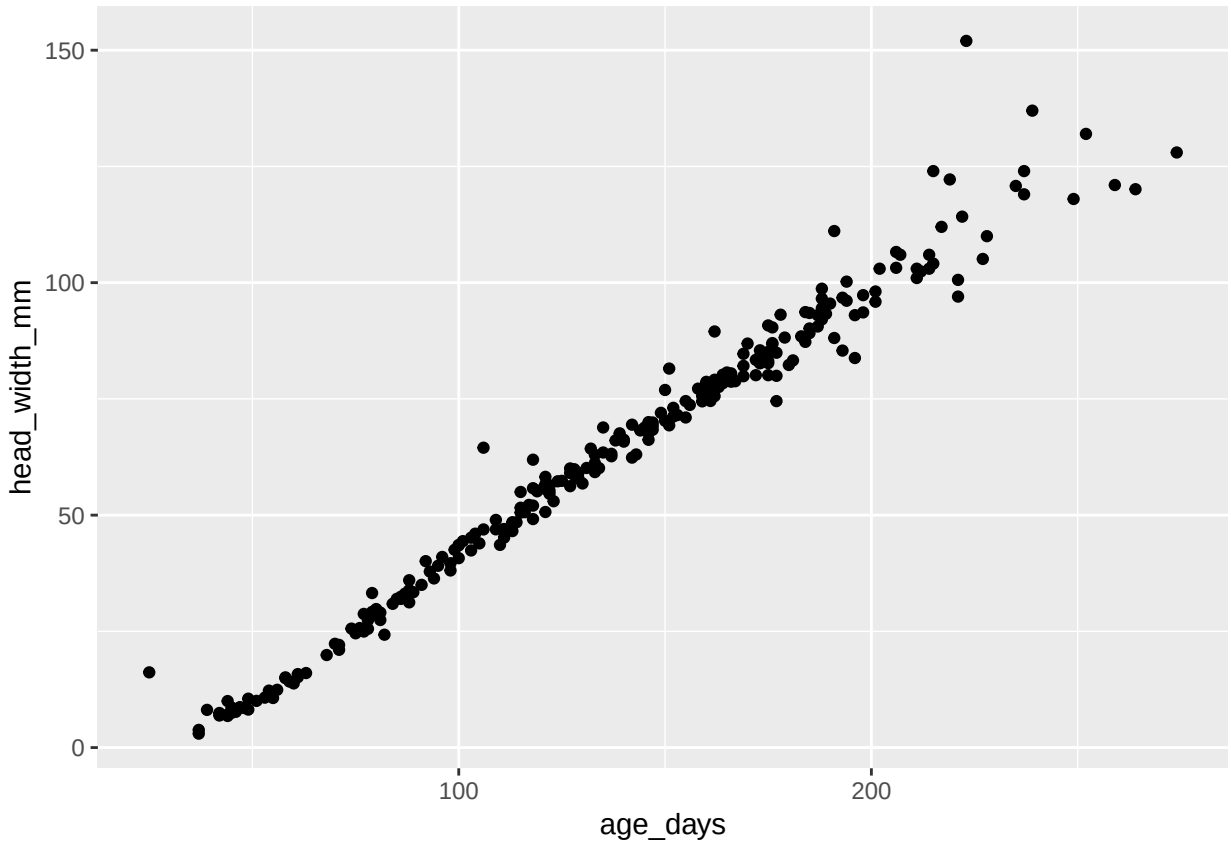
```
ggplot(long, aes(x=TimePoint, y=Weight, col=ID, group=ID)) +
    geom_line() +
    geom_point()
```



I've added col=ID to the aes() function so we can see which observations belong to which person, and also a new layer geom_line() which connects the observations within the group specified as group=ID within the aes() function.

The gather function also comes in handy for making similar types of plots where we want to put different variables side-by-side. For example, we might want to look at how the head_width_mm, head_length_mm and crl_cm variables change with age_days. We can do that one at a time using the existing data frame, for example:

```
ggplot(fetuses, aes(x=age_days, y=head_width_mm)) +
    geom_point()
```



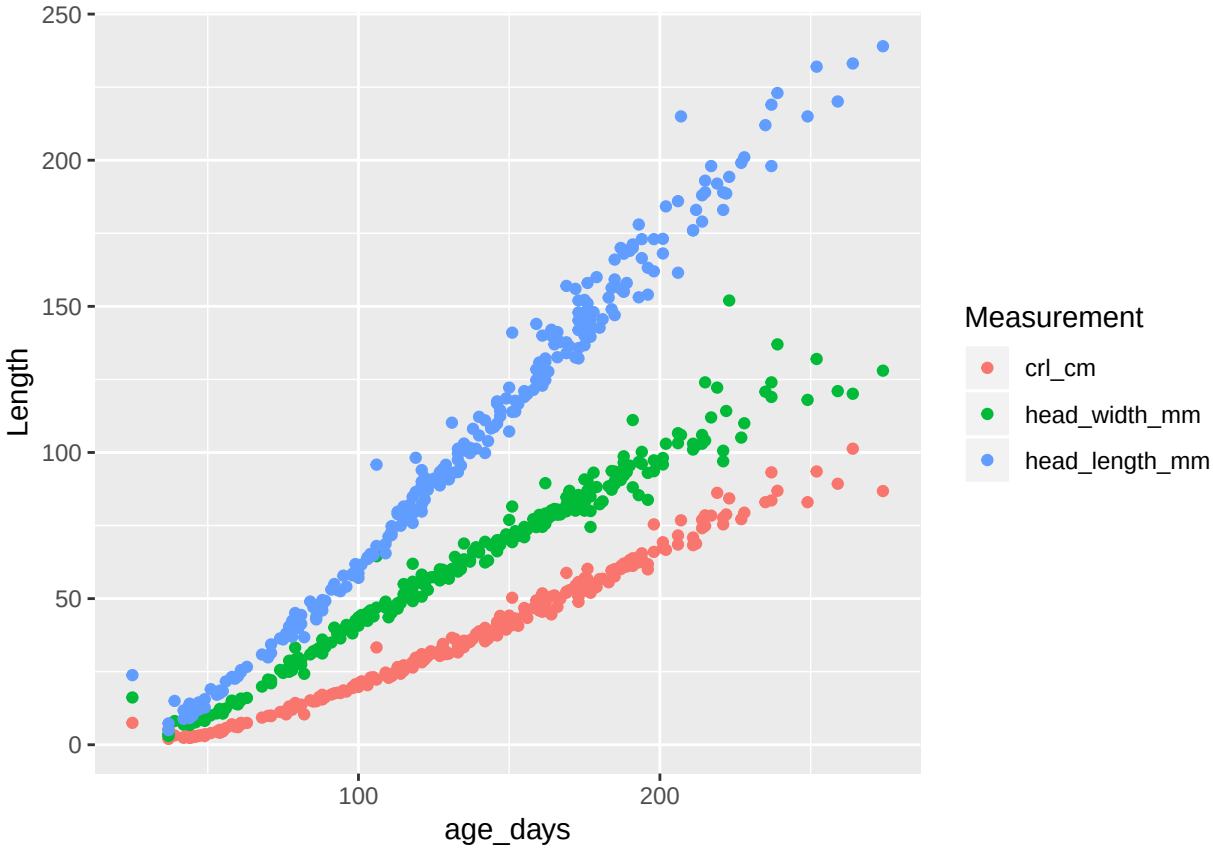
But how do we add head_length_mm and crl_cm to this plot? We need to convert the data into long format, and then make the plot:

```
plotdata <- fetuses %>%
    mutate(ID = row_number()) %>%
    select(ID, age_days, crl_cm, head_width_mm, head_length_mm) %>%
    gather(Measurement, Length, -ID, -age_days) %>%
    mutate(Measurement = parse_factor(Measurement))

str(plotdata)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    786 obs. of  4 variables:
##  $ ID         : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ age_days   : num  274 196 259 249 221 221 177 82 193 142 ...
##  $ Measurement: Factor w/ 3 levels "crl_cm","head_width_mm",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Length     : num  86.8 61.8 89.3 83 75.4 77.7 55.4 10.4 62.4 35.4 ...
```

```
ggplot(plotdata, aes(x=age_days, y=Length, col=Measurement)) +
    geom_point()
```

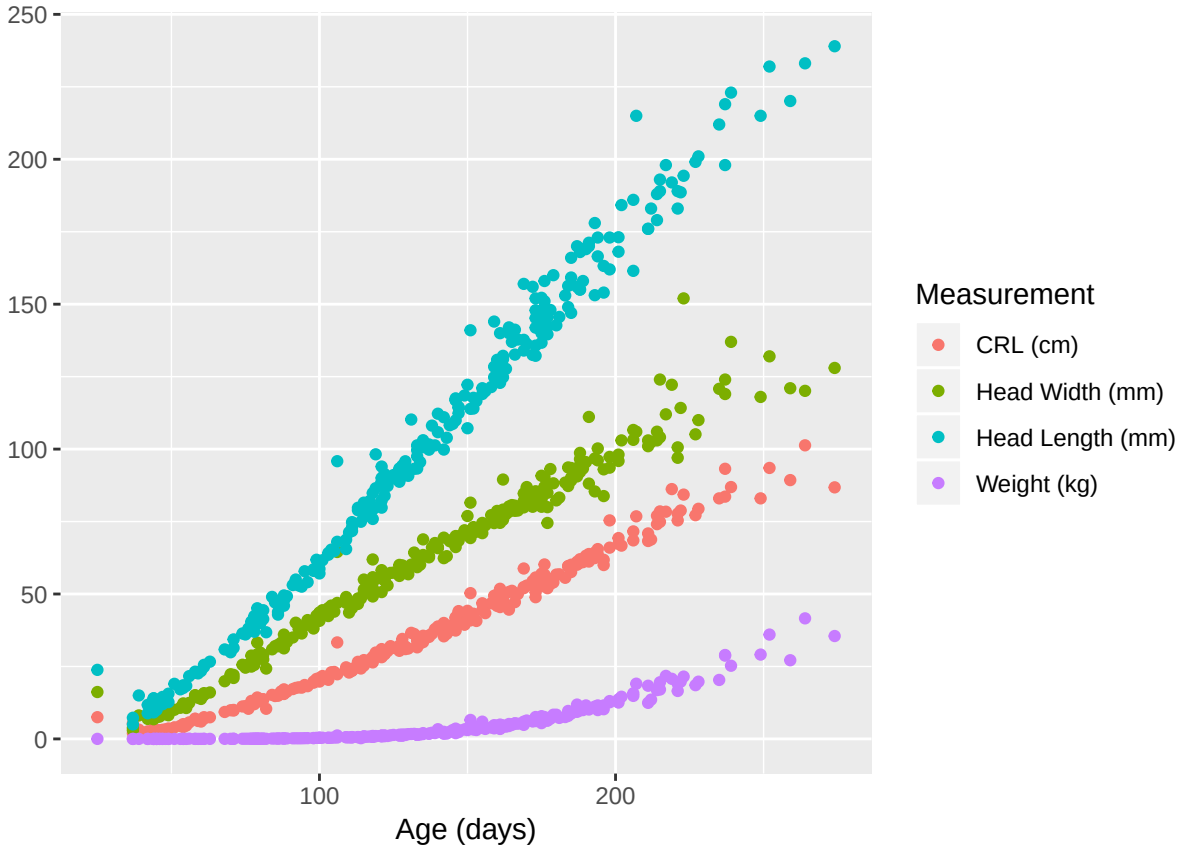


This code has several steps: first we extract the variables we are interested in using select(), then we add an ID variable so that we know how the rows in the new data frame relate to the rows in the fetuses data frame, then we use gather to convert all columns in the data into long format with the exception of ID and age_days, then we make the plot. You can see from the structure of plotdata that we now have 786 observations, which is exactly three times the original 262 observations because each ID is repeated three times. This process of creating a new data frame containing the specific data that we want to plot (and in the correct format) is something we will do a lot.

Now let's say we want to add weight to the plot as well. All we have to do is copy and paste the code above, and add weight_kg in the relevant place:

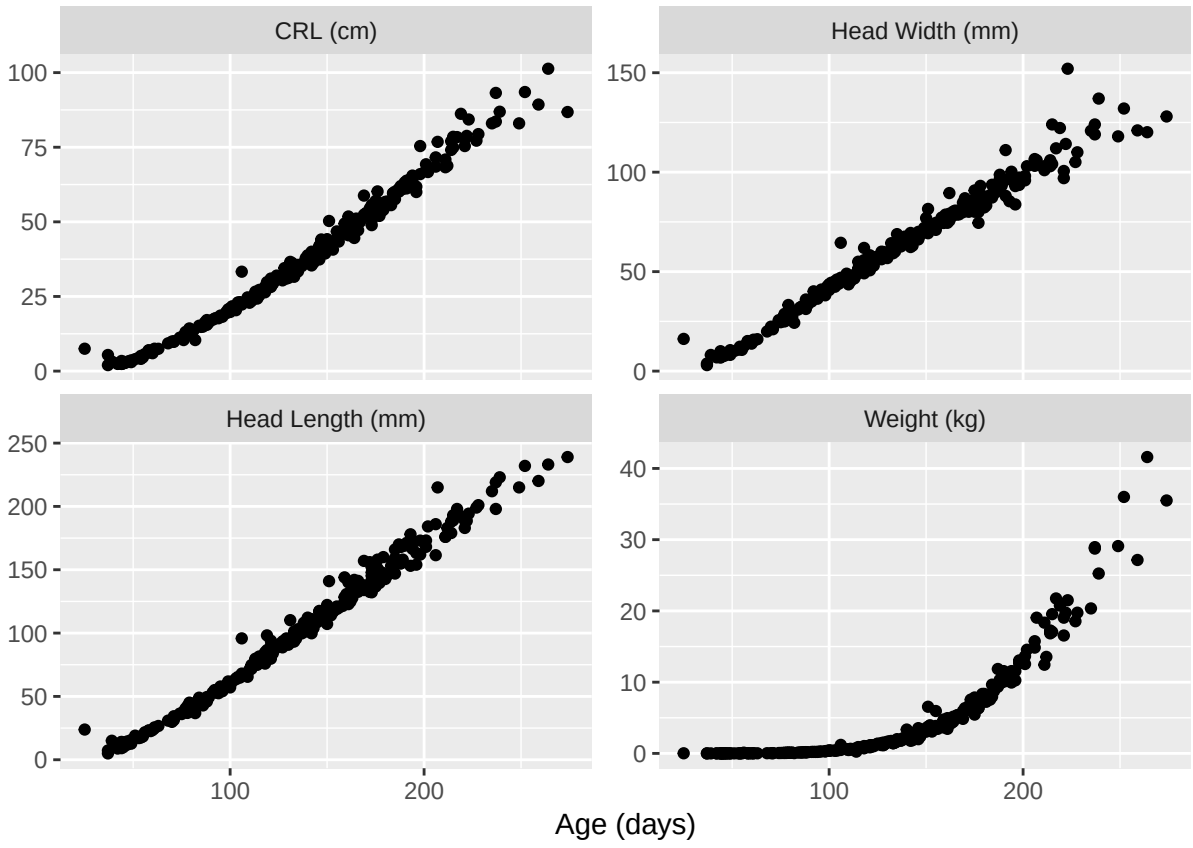
```
plotdata <- fetuses %>%
    mutate(ID = row_number()) %>%
    select(ID, age_days, crl_cm, head_width_mm, head_length_mm, weight_kg) %>%
    gather(Measurement, Length, -ID, -age_days) %>%
    mutate(Measurement = parse_factor(Measurement)) %>%
    mutate(Measurement = fct_collapse(Measurement, "CRL (cm)"="crl_cm", "Head Width (mm)"="head_width_mm

ggplot(plotdata, aes(x=age_days, y=Length, col=Measurement)) +
    geom_point() +
    xlab('Age (days)') +
    ylab('')
```



The only addition is that I have changed the names of the factor levels and x-axis label to a more readable format, and removed the y-axis label as weight isn't a length! But the weight is on a different scale to the other variables so is hard to see at the bottom of the graph. There are two ways to do this: we could change the units of weight so that 1 y-axis value represented 100g rather than 1kg, but it is probably a better idea to put the different variables into different facets using the facet_wrap() function:

```
ggplot(plotdata, aes(x=age_days, y=Length)) +
    geom_point() +
    xlab('Age (days)') +
    ylab('') +
    facet_wrap(~Measurement, scales='free_y')
```



The facet_wrap (and related facet_grid) funtion allow us to specify one or more categorical variable on which we can stratify the plot into different sub-plots. The scales='free_y' argument tells facet_wrap to use different y-axis scales for each variable type.

If you want to read more about tidy data structure (including long vs wide formats) then see the Tidy data chapter of the R for Data Science book: https://r4ds.had.co.nz/tidy-data.html

---

Additional notes

Here are a couple of additional tips:

- Never write code from scratch when you can copy and paste working code from an example you already have. The process for importing and formatting data is nearly always the same, and as a result the code will be largely identical with just a few changes to variable names. If you write code from scratch then (1) it will be slower because you will have to remember how to write everything, and (2) it will be more frustrating because you will make more mistakes. It is a good idea to keep a specific R script file handy with examples of how to do different things - use this document as a starting point.
- If you need to do something that you haven't done before, then try looking at the cheat sheets for either data import (https://github.com/rstudio/cheatsheets/raw/master/data-import.pdf), data wrangling (https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf), data visualisation (https://github.com/rstudio/cheatsheets/raw/master/data-visualization-2.1.pdf), or one of the other cheat sheets at https://www.rstudio.com/resources/cheatsheets/ If you still can't find what you are looking for then you can try googling what you want to do, but be aware that you might get lots of different solutions for the same problem, and some of them will look extremely complex

and/or unfamiliar. If you include the word 'tidyverse' in your search then you are more likely to find a familiar-looking solution.

- R script files are really just text files, so it is a good idea to stick to standard ASCII text characters - these do not include å æ or ø. You can use these in comments with only one small drawback: you will then get asked for an encoding to use when saving the file (pick either latin1 or UTF-8). It is important to write lots of useful comments so if you prefer to stick to danish for your comments then feel free. But you should definitely not use å æ or ø in the R code itself, including in variable names: this can lead to problems with your code spontaneously breaking if the encoding is changed later on (this can happen just by sending the R script file over email).
- When you are writing R scripts, try to make sure that the entire script file contains valid code all the way though, and that code at the start of the R script does not depend on code written further down in the script. You should be able to clean out your environment by clicking the sweeping brush to the right of 'Import Dataset', and then re-create all of your work just by clicking on 'source'. This ability to re-create an entire analysis from scratch at the click of a button (if for example your data changes) is a major advantage of using R. This also means that if you re-open RStudio you are guaranteed to be able to get back to where you were just before you closed it (or it crashed, which can happen) - so it is a good idea to test this periodically to make sure that you don't get any errors. You will also need to do this if you make a mistake and either delete something that you didn't mean to, or break a variable by e.g. using mutate to convert it to a factor but mis-typing or forgetting about one of the levels.
- As a general rule, R ignores white space (including spaces and line breaks), so it doesn't really matter in terms of the code if we use them or not. But it is MUCH easier for humans to read and understand your code if it is laid out in a consistent and clear way. So get into the habit of putting spaces between functions and arguments, and breaking large chains of functions over separate lines. You might still end up with some long lines, and by default you need to scroll to see these if they disappear off the right hand side of the screen. If that annoys you, then go to the Tools menu, then Global Options, then click on the Code tab and make sure that 'Soft-wrap R source files' is checked.

R Markdown environment:

```
sessionInfo()
```

```
## R version 3.5.2 (2018-12-20)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_GB.UTF-8/en_GB.UTF-8/en_GB.UTF-8/C/en_GB.UTF-8/en_GB.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] lubridate_1.7.4 bindrcpp_0.2.2  readxl_1.2.0    forcats_0.3.0   stringr_1.3.1   dplyr_0.7.8
##  [7] purrr_0.2.5     readr_1.3.1     tidyr_0.8.2     tibble_2.0.1    ggplot2_3.1.0   tidyverse_1.2.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.0       cellranger_1.1.0 pillar_1.3.1     compiler_3.5.2   plyr_1.8.4
##  [6] bindr_0.1.1      tools_3.5.2      digest_0.6.18    jsonlite_1.6     evaluate_0.12
```

```
## [11] nlme_3.1-137     gtable_0.2.0     lattice_0.20-38  pkgconfig_2.0.2  rlang_0.3.1
## [16] cli_1.0.1        rstudioapi_0.9.0 yaml_2.2.0       haven_2.0.0      withr_2.1.2
## [21] xml2_1.2.0       httr_1.4.0       knitr_1.20       hms_0.4.2        generics_0.0.2
## [26] grid_3.5.2       tidyselect_0.2.5 glue_1.3.0       R6_2.3.0         fansi_0.4.0
## [31] rmarkdown_1.11   modelr_0.1.2     magrittr_1.5     backports_1.1.3  scales_1.0.0
## [36] htmltools_0.3.6  rvest_0.3.2      assertthat_0.2.0 colorspace_1.4-0 labeling_0.3
## [41] utf8_1.1.4       stringi_1.2.4    lazyeval_0.2.1   munsell_0.5.0    broom_0.5.1
## [46] crayon_1.3.4
```