R for Psych

Glenn Williams

2018-08-31 (last updated: 2022-10-07)

Contents

Preface & Overview

In this course you'll learn how to use R for data analysis and presentation. This course has a particular focus on using R for psychology (hence the name), but it should be applicable to most cases in the social sciences. Here, we'll primarily take a tidyverse first approach to R. This means we'll be relying primarily on a collection of packages designed to get things done quickly, with highly readable syntax. We will still cover some of the base R functions and the basics of programming, but our aim is to quickly and efficiently use R for managing and processing data. Along the way, we'll look into how R encourages open and reproducible science, and how R can be useful for managing your research projects as a whole.

By the time you're finished, you will be able to tell R how (and where) to read input files (e.g. raw data from an experiment), how to perform operations on your data (e.g. data wrangling and aggregation), and how to produce and save ouputs based on your data (e.g. graphs and test statistics). You'll also be able to produce documents that incorporate your R code with formatted text so every time you update your code, your written statistics, tables, and graphs update automatically. For this, we'll explore R-markdown, specifically using R notebooks.

We will rely on the R for Data Science (R4DS) textbook by Garrett Grolemund and Hadley Wickham as a core text. Follow the link for a free online version of the book uploaded by the authors.

R4DS by Wickham & Grolemund

The above book assumes some familiarity with programming and/or R at the outset, but covers the basics in Chapter 4. As in this course, the aim here is to get you doing productive things (e.g. producing a graph) as quickly as possible. If, however, you feel like you'd prefer a basic grounding in R prior to doing so, you can check out R for Beginners by Emmanuel Paradis. This is a short introduction to all of the core concepts required for working with your data in R (including reading, manipulating, and saving data).

As this course focuses on using R for psychologists, we'll cover a range of traditional parameteric and non-parametric analyses, such as:

- Correlations
- *t*-tests
- ANOVA

We will also cover topics such as *power analysis*, particularly using simulation based methods (which are scalable for any set of tests), before we move on to more advanced methods, such as *hierarchical mixed effects modelling* (or linear mixed effects models; LMM). Throughout, we will use examples and assignments to entrench the concepts taught in this module.

Finally, we will focus on creating reproducible analyses and write-ups using R markdown.

To join the rest of your class in discussing this course, please join the R @ Abertay Slack channel. We will use this channel for all communications about the course, including help, hints, and tips about the course content.

To download the course content, get the lesson materials from my GitHub repo. Download all of the content from this course using the *Clone or Download* button, save the files as a zip, unzip them, and go to the lesson_materials folder. You can follow along with the slides using the R notebooks.

Chapter 1

Introduction

To begin with, we'll focus on getting you started in R. We'll look into installing our two key programs, R and RStudio. After that, we'll look into how R can communicate with files on your system. Finally, we'll look at the core packages for this course, how to do basic data manipulation using base R (R Core Team, 2022), and tips for good practice when working with R. We'll complete some exercises at the end so you can get a handle of the key concepts introduced here.

1.1 Installation and Setup

To get started, at the very least you'll have to download R from CRAN. Select the correct distribution for your operating system and then click through to *install R for the first time.*

For Windows users, you'll see a new page; at the top click on "Download R [**version number**] for Windows." Click on that and follow the prompts in the installer. If you have a 64 bit system, install the 64 bit version of R as you'll be able to take advantage of having more than 4Gb RAM. (This is useful in instances where you're working with very large data sets.)

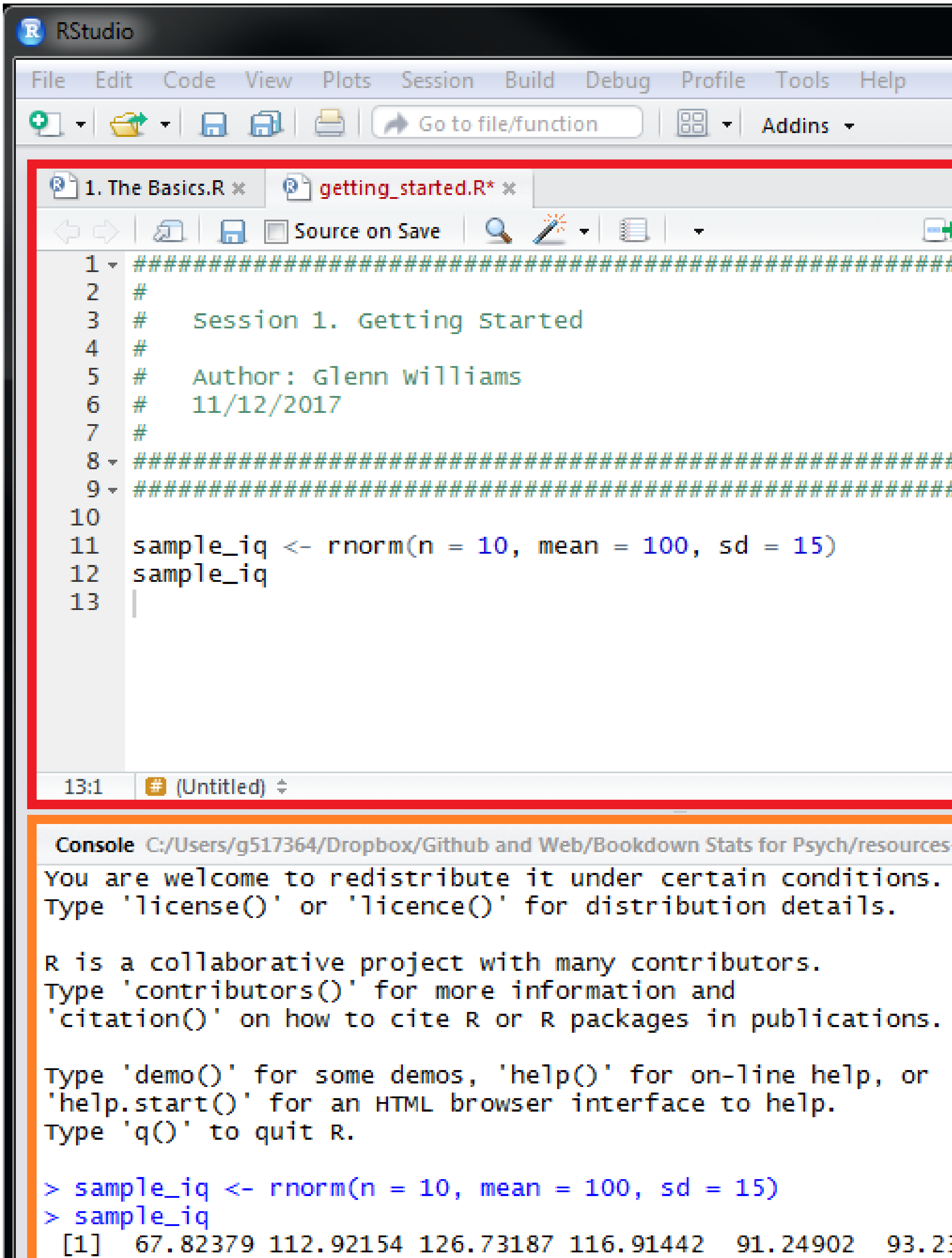
In R, you don't want to type commands directly to the console. Instead, you should use a text editor to write your script and send it to the console as and when you want. This way, you'll have a document of what you did, and any minor changes you have to make are very easy to implement (vs. writing it all out from scratch again).

One of the most popular ways to work on your R scripts is to use an Integrated Development Environment (IDE). The most popular IDE for R is RStudio, which you can download from the RStudio website. In the navigation panel, select **products** and choose **RStudio**. Scroll down until you see **Download**

RStudio Desktop. Finally, for the Free tier, click the **Download** button and select the correct installer for your operating system. For Windows users, you should select RStudio [**version number**] - Windows Vista/7/8/10. Follow the prompts on the installer, and your system should automatically pick up that you already have R on your system. From there, be sure to open the RStudio program to get started. This is the logo that looks like this:

The RStudio logo

Once open, you'll see something that should look like this. Only, you won't have the top pane unless you choose to start a new script (File, New File, R script) or if you've opened an existing script already.


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
1. The Basics.R
getting_started.R*
Source on Save
13:1
(Untitled)
Console C:/Users/g517364/Dropbox/Github and Web/Bookdown Stats for Psych/resources

```
##################################################################
#
#   Session 1. Getting Started
#
#   Author: Glenn Williams
#   11/12/2017
#
##################################################################
##################################################################

sample_iq <- rnorm(n = 10, mean = 100, sd = 15)
sample_iq

```

```
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

> sample_iq <- rnorm(n = 10, mean = 100, sd = 15)
> sample_iq
 [1]  67.82379 112.92154 126.73187 116.91442  91.24902  93.2
```

RStudio can be split into 4 main panes:

1. The **editor**: Type, edit, and save scripts as you would in any text editor.
2. The **console**: Execute scripts by typing and pressing enter.
3. The **environment and history**: View objects (e.g. values, variables, and user-defined functions etc.) stored in memory for this working session. You can also see a history of your commands in the *History* tab.
4. The **viewer**: view any *files* in your working directory, see your last *plot* from this session, view installed *packages* on your machine (you can also load them here), view the *help* documentation for any R commands, in the *viewer* you can view (surprise, surprise) markdown/other documents.

All of these panes are very useful when developing your R scripts, but RStudio has some other features such as syntax highlighting, code completion, and an easy interface for compiling your reproducible R-markdown notebooks. For advanced users, you can also set up projects that play nicely with Github for version control of your work.

Finally, RStudio defaults to giving you the option to save your workspace (e.g. all of the data you've created) when you close the program and to restore this workspace once you restart RStudio. I'd advise you to change RStudio's defaults to never do this as this could cause unforseen errors when, for example, you want to change parts of script and you accidentally delete the stage to create a variable that is crucial later on. This deletion may not be obvious if you reload your workspace, but you won't have a record of how you created said variable. To make these changes, go to **Tools, Global Options** then deselect **Restore .RData into workspace at startup** and from the dropdown menu on **Save workspace to .RData on exit:** to **Never**.

1.2 Working Directory and File Paths

When you create an R file (File, New File, R Script), where that file sits is its **working directory**. This is where R interprets any commands that go outside of your R session. For example, if you want to read data into R, or output a graph, R will do so in respect to the working directory. You can *get* your *working directory* like so:

```
getwd()
```

You should see something along the lines of "C:/Users/*YOUR_NAME*/FOLDER/*RFILE*.R"

Now, you can set your working directory to any folder on your computer with an absolute file path. Often, people use absolute file paths in order read inputs and send outputs to different folders on their computer, telling R exactly where to look.

However, I recommend against this. That's because if you change computers or pass your code over to someone else R will kick out an error saying it can't find that folder; other computers are highly unlikely to have the same folder structure or even username as you, and fixing this can be frustrating. Also, it can be a pain having to type out your full file path, so let's avoid that. Below, I've outlined one method for working that will allow you to use relative filepaths (relatively) easily.

If you want to keep your folder from getting cluttered, keep your R script(s) in a folder, with an Inputs and Outputs folder at that same level, like so:

Potential Folder Structure for Your Projects

This way, when you want to read in raw data, or output a graph (or something else) you can use a **relative file path** rather than an absolute file path to define where your files are taken from/should go. This saves you a lot of typing of file paths, and it means your R scripts will work on someone else's computer if you just send the entire folder containing the R script, Inputs, and Outputs folders.

To read in or save data using a relative file path, do this:

```r
# reads a CSV file from the Inputs folder
my_data <- read.csv("Inputs/my_data.csv")

# writes a CSV file to the Outputs folder
my_data <- write.csv(my_data, "Outputs/my_data_again.csv")
```

You just have to type in the folder name and a slash (to indicate to go below that folder) and the name of your data. This saves you from all of the hassle of using an abosolute file path described above.

1.3 Packages

Next, while you can do a lot in base R, if you can think of some task that is reasonably laborious there's probably a package out there that can help you to achieve a certain goal. For us, the whole data processing, analysis, and presentation workflow can be made so much simpler by using a set of packages from the tidyverse library (Wickham, 2017). This package is required for what we're about to do next, so install tidyverse (using `install.packages("tidyverse")`). Once installed, you needn't do it again. But on each new session you have to define which packages you want to load into R using the `library()` command. Make sure to uncomment the `install.packages("tidyverse")` line if you haven't installed this package yet.

```
# install.packages("tidyverse") # do this only once to install the package
library(tidyverse) # do this to load your package every time you open R
```

```
## -- Attaching packages ------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6      v purrr   0.3.4
## v tibble  3.1.8      v dplyr   1.0.10
## v tidyr   1.2.1      v stringr 1.4.1
## v readr   2.1.2      v forcats 0.5.2
## -- Conflicts ---------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

By default, R installs packages from CRAN, which is essentially a centralised network for all things R. However, some useful packages may not have been submitted here, and you can install them from other places, like GitHub. We won't install packages from github or other repositories in this course. But it's worth knowing that you can install packages from just about anywhere!

Most of the time, you won't have any trouble using functions from a loaded package. However, there can be cases when you have two packages installed that use the same function name. To tell R exactly which version of a function to use, we can specify both the package and function name in the form `package::function_name()`. For example, we can use the `group_by()` function from the package `dplyr` by typing `dplyr::group_by()`. You won't come across this in this course, as we'll be using packages that have functions with unique names, but it's worth bearing in mind if you come across problems with functions you *know* should work in the future.

Note: You may notice that in my code chunks I have some green text following a #. These are comments in R and are not read when you execute your code. Comments are very useful for telling you what each section of your code does, and how. I tend to comment quite heavily, as there's always a chance you'll forget what you've done when you go back to it after a few months away from the code.

1.4 Objects and Functions

You can use R like a calculator, e.g.

```
1 + 4
```

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

You can execute this directly in the console, or run it from your R script by selecting that line of code (or highlighting several lines) and pressing Ctrl + Enter (for mac, this is probably Cmd + Enter). While I said you shouldn't use the console when writing scripts, you can use it to test out a bit of code, or to quickly use R as a calculator when you're in a meeting and feeling like mental arithmetic is a step too far. You can see that once the code is run, the R console returns the result back to you. You see a history of what you asked, and what was returned.

Top tip: use the up arrow key from the console and R will automatically fill in the last line/block of code you ran. Press up again to cycle back to older inputs, and down to back to the most recent ones.

R always waits for you to finish an expression before it runs your code. So, if you ended your line with a +, it'll wait for the next number. This is useful for complex expressions that can take up multiple lines, e.g.

```
1 + 2 + 3 + 4 + 5 + 6 + 7 +
  8 + 9
```

```
## [1] 45
```

Watch the console when you type this out. You'll notice that if you press enter after typing `7 +` on the new line you will no longer see `>` but you'll see +. The same happens even if you pass a different mathematical operator (e.g. `-`, `%`, `^`). This is there to tell you that R is not waiting for a new statement, but is waiting for you to finish off the current statement. If you see `>` it means that whatever you can start a new statement.

R also parses text if included in quotes. The same rule applies about finishing expressions here; if you don't close your quote, then R will wait for you to do so. This means you can spread your text over several lines (by pressing Enter) and R will parse that as one expression. Note with our output we get \n which indicates that a new line follows the comma.

```
"Hello, world!"
```

```
## [1] "Hello, world!"
```

```
"How are you doing today?
Things certainly have been interesting these past few years."
```

```
## [1] "How are you doing today? \nThings certainly have been interesting these past few years."
```

Crucially, you can store this sort of information in a variable. Variables are useful because you may want to use the results of a calculation and do some further operations on that result. This is especially useful if you're not sure what that first result could be. We assign values to a variable using the assignment operator `<-` (read this as create from).

```
summed_numbers <- 5 + 8
```

We can then output this variable later on, like so: (This is often useful for checking that your variables store what you think they store.)

```
summed_numbers
```

```
## [1] 13
```

Or we can perform operations on the variable.

```
summed_numbers*5
```

```
## [1] 65
```

Note: You cannot start variables with a number, you cannot use special characters (e.g. %!*) and you cannot include spaces in your variable name. Also, note that your capitalisation matters, so Summed_numbers is not summed_numbers. (I often get errors due to this issue…)

R has the simple arithmetic operations you'd expect from any program. For example, you can:

- add `x + y`,
- subtract `x - y`,
- multiply `x * y`
- divide `x/y`,
- exponentiate `x^y`
- find the modulus `x %% y`, (e.g. 5 mod 2 = 1; i.e. the remainder from how many times 2 goes into 5)
- and conduct integer division `x %/% y` (e.g. 5 int div 2 = 2)

R also has some logical operations built in. For example:

- less than `x < y`
- less than or equal to `x <= y`
- greater than `x > y`

- greater than or equal to `x >= y`
- exactly equal to `x == y`
- not equal to `x != y`
- not x `!x`
- x OR y `x | y`
- x AND y `x & y`
- test if X is true `isTRUE(x)`

These come in pretty handy for performing most operations on our data. If you're unfamiliar with these, don't worry. We'll cover how you might use some of these in a staggered format as you progress through this course. Nicely, R also has a number of functions built in. This means you don't need to write your own function if you want to sum a sequence of numbers. I'm sure I couldn't provide an exhaustive list here, but as stated above we'll cover most of the common functions as you get to grips with some data. Still, lets get an idea of how these functions work.

Below, we will sum the numbers 1 to 4. This function is built into R already, so we don't have to write a whole lot of code for this.

```r
sum(1, 2, 3, 4)
```

```
## [1] 10
```

What if we want to calculate the mean score from this set of numbers? That's also built into R.

```r
mean(1, 2, 3, 4)
```

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

Notice that whenever we want to run a function, these functions always have a name and are followed by parentheses (e.g. `mean()`, `sum()`). What goes in the parentheses? The argument you want to pass to the function. These can have default values, or not (requiring you to specify the argument). Above, we passed the values 1 through 4 as the arguments to the `mean()` function. Later, we'll look at functions that ask for arguments from separate data types (e.g. numbers and characters). If you're unsure what an argument does, you can always ask R what it does, how it does it, and what to pass to it by using `?`, e.g. `?mean()`. This will bring up a document in the Help window of RStudio.

Typing out all of these numbers each time we want to perform some operation is a little tedious. To fix this, we can use what we learned above and store these numbers into a variable. In order to save these into their own variable, we use the `c` function. Think of this as **c**oncatenation. When we combine values into a variable, this variable is stored in our global environment. This means that we can perform operations on the variable later on, without the worry of typing our the values again. This is particularly useful if you want to store values from one function (say a statistical test) that you cannot pre-define but that you want to use later on.

Let's give that a go. First, we'll concatenate our values into a variable using the `c()` function described above. Here, we simply define that we want to concatenate some values, and list each value separated by a comma.

```
stored_values <- c(1, 2, 3, 4)
```

If we then call a function that works with a set of numbers, it should also work if we call that function on the variable storing those numbers. Let's see how this works with the `mean()` function.

```
mean(stored_values)
```

```
## [1] 2.5
```

Great! That'll save is a lot of time writing and rewriting code later on. This also allows our code to be flexible, in that we can write a script that performs operations on variables that can take any range of values. This, to me, is one of the nicest things about doing your analyses in R. While you may spend more time getting your script up and running in the first place when compared to using point-and-click methods (e.g. in SPSS), if you gain new data or run a new experiment, it's likely that your script can simply be re-run with no (or few) changes at very little cost to your time.

Now, this part is pretty important but may only be obvious if you've programmed in other languages. R is a **vectorised** language, which means that, as with the `sum()` function above, R can perform operations on the entire variable So, if you want to increment all values in your variable by 1, you can simply tell R to do so in one line of code, without the need for loops or other complex methods.

```
stored_values + 1
```

```
## [1] 2 3 4 5
```

1.5 Creating and Accessing Data

So, you've made it through all of the boring stuff. Now we can look at manipulating some data in R.

Let's pretend we've administered IQ tests to 100 participants. For some reason, we're interested in whether dog and cat owners have different IQs. I'm sure some of you can come up with some incredibly biased hypotheses here, but we'll leave that out for now.

We'll create this data in R, then take a look at it using some of the packages from the `tidyverse` library that you installed and loaded above. If the functions below don't work, be sure to load the tidyverse library (using `library(tidyverse)`) before running the code below.

First, we'll create a variable containing the numbers 1 to 100, using the `seq()` function; this can be our participant ID variable. Next, we'll use the `sample()` function to sample randomly (and with replacement) from the two labels "cat" and "dog" which acts as our factor for pet ownership. (Note that here I use the `sample()` function on the concatenated values of "cat" and "dog", but this would work equally well with a variable containing these labels.) Finally, we'll use the `rnorm()` function to generate some random numbers (sampled from the normal distribution) with a mean of 150, and a standard deviation of 15 to act as our IQ scores.

Remember how I said we'd look at functions that take several arguments? Well, thats what all of these functions below do. We define each parameter of the argument within the function call. So, if we want to generate a sequence of numbers from 1 to 100 , in increments of 1s (e.g. `seq(from = 1, to = 100, by = 1)`), then we define each argument with its name (e.g. from/to/by) and tell R which values to set for these arguments using `=`.

```r
# create participant IDs
participant <- seq(from = 1, to = 100, by = 1)

# create pet ownership codes
set.seed(88)
pet <- sample(c("cat", "dog"), 100, replace = TRUE)

# create IQ scores
set.seed(88)
IQ_score <- rnorm(n = 100, mean = 150, sd = 15)
IQ_score <- round(IQ_score) # round IQ scores
```

Note: I've overspecified the functions above. You can simply run `seq(1: 100)` and `rnorm(100, 150, 15)` to get the same results, but it's a little less readable. Here, and throughout, I'll use the most readable version so you have the best idea of what you're doing. Because we're using random sampling, I've also set a seed, so that you'll sample the same values as me. If you want people to be able to reproduce your random sampling so they get the same data, set a seed!

Finally, it's important to note that you can call a function on the result of another function in R.

Right now, each number in the participant variable is unique. We can see how many participants are in the study by asking R “how long is the variable, participant?”. We do this like so:

```
length(participant)
```

```
## [1] 100
```

Cool, we have 100 participants! But what if we have several observations for one participant? I’ll define that (very simply) in the code below:

```
participant[2] <- 1 # assign the second value of participant the integer 1
head(participant) # return the first 6 values of the participant variable
```

```
## [1] 1 1 3 4 5 6
```

Now, using `length(participant)` will return 100, because there’s still 100 values in the `participant` variable. How do we find how many **unique** numbers are in the `participant` variable? We simply nest the `unique()` function within the `length()` function:

```
length(unique(participant))
```

```
## [1] 99
```

This asks R how many unique values are in the variable participant. That’s 99!

1.5.0.1 Data Frames and Tibbles

Now, in the real world, if you tested IQs you’d typically have this data stored in a table somewhere prior to reading it into R. So lets pair the data together into a table in R. The most useful way to do this is to create a `data.frame`. But, since we’ve already loaded the tidyverse, we may as well use the built in **tibbles**. These are like data frames, but they have some different defaults that make working in the tidyverse easier.

```
IQ_data <- tibble::tibble(
  participant_number = participant,
  pet_id = pet,
  IQ = IQ_score
  )
```

What's the main advantage of using a tibble over a `data.frame` here? Tibbles don't automatically convert certain data types. With `data.frame`, character data is automatically converted to a factor. This can be useful, but not always. The thing I like most is that tibbles show the data type under the column name. This is taken from `str()` which tells you the **str**ucture of your data. Also, tibbles default to printing the first 10 rows of data when you type their name in the console. This will stop you from getting overwhelmed by a massive printout if you have a lot of data. Let's see all of this in action.

```
IQ_data
```

```
## # A tibble: 100 x 3
##    participant_number pet_id    IQ
##                 <dbl> <chr>  <dbl>
##  1                  1 cat      147
##  2                  1 cat      160
##  3                  3 dog      185
##  4                  4 cat      122
##  5                  5 cat      157
##  6                  6 dog      152
##  7                  7 cat      152
##  8                  8 cat      119
##  9                  9 cat      131
## 10                 10 dog      160
## # ... with 90 more rows
```

If you want more than 10 rows, or a specific number of columns, tell R that's what you'd like. Here, we've asked for the first 12 rows, and the width of the table to be infinity so that R prints out all columns even if they don't fit on 1 row in the console. However, with only 3 columns, that isn't an issue here.

```
print(IQ_data, n = 12, width = Inf)
```

```
## # A tibble: 100 x 3
##    participant_number pet_id    IQ
##                 <dbl> <chr>  <dbl>
##  1                  1 cat      147
##  2                  1 cat      160
##  3                  3 dog      185
##  4                  4 cat      122
##  5                  5 cat      157
##  6                  6 dog      152
##  7                  7 cat      152
##  8                  8 cat      119
##  9                  9 cat      131
## 10                 10 dog      160
## 11                 11 cat      138
## 12                 12 dog      165
## # ... with 88 more rows
```

Unfortunately, some older functions in R won't allow you to use a tibble. If this is the case, simply convert your tibble to a `data.frame` using the `as.data.frame()` function. Note, we use `head()` to see the head of our data frame, or the first 6 values. This is necessary here to avoid printing out each row, as we're not in using a tibble any more. Notice that We've assigned the `data.frame` version of our IQ data to a new variable, rather than overwriting the previous variable. This is good practice when testing your code, as you never know what might break, resulting in data loss. (Although this wansn't strictly necessary here.)

```
IQ_data_df <- as.data.frame(IQ_data)
head(IQ_data_df)
```

```
##   participant_number pet_id  IQ
## 1                  1    cat 147
## 2                  1    cat 160
## 3                  3    dog 185
## 4                  4    cat 122
## 5                  5    cat 157
## 6                  6    dog 152
```

1.5.0.2 Accessing Tibble Information

With tibbles and `data.frames`, you often want to access a specific column in order to perform some calculation. Let's say we want to calculate the mean IQ score in our data set. Why doesn't the code below work?

```
mean(IQ_data)
```

```
## Warning in mean.default(IQ_data): argument is not numeric or logical: returning
## NA
```

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

R gives us a warning, saying that the argument to our function (i.e. the tibble, IQ_data) is not numeric or logical. Therefore, R cannot calculate the mean on something that isn't a number (or set of numbers). Instead, we have to point

R to the correct column within the tibble. This is often done by name using `$` , or by name/position in the tibble using `[[` and the name/position followed by `]]`.

First, let's see that we're accessing everything correctly here.

```
# by name
IQ_data$IQ
IQ_data[["IQ"]]

# by position
IQ_data[[3]]
```

```
## [1] 147 160 185 122 157 152
```

Tibbles are stricter than data frames, in that they'll always return a tibble. For example, with a `data.frame` if you ask for a value in a specific row of a specific column under the format `data_frame[row, column]`, R will return a vector of the value at that position in the `data.frame`. With tibbles, your single value will be returned as a tibble.

```
# tibble output
IQ_data[1, 3] # row 1, col 3
```

```
## # A tibble: 1 x 1
##      IQ
##   <dbl>
## 1   147
```

```
# data.frame output
IQ_data_df[1, 3]
```

```
## [1] 147
```

Now, lets calculate the mean on the correct column in the tibble. Personally, I prefer using `$` for cases like this. But, feel free to use any of the above methods.

```
mean(IQ_data$IQ)
```

```
## [1] 150.17
```

We can combine what we learned above about `c()` to access multiple columns at once:

```
IQ_data[, c("participant_number", "IQ")]
```

```
## # A tibble: 100 x 2
##    participant_number    IQ
##                 <dbl> <dbl>
##  1                  1   147
##  2                  1   160
##  3                  3   185
##  4                  4   122
##  5                  5   157
##  6                  6   152
##  7                  7   152
##  8                  8   119
##  9                  9   131
## 10                 10   160
## # ... with 90 more rows
```

Or we can access a range of values by specifying the rows we'd like to access:

```
IQ_data[c(1, 2, 3, 4, 5), c("participant_number", "IQ")]
```

```
## # A tibble: 5 x 2
##   participant_number    IQ
##                <dbl> <dbl>
## 1                  1   147
## 2                  1   160
## 3                  3   185
## 4                  4   122
## 5                  5   157
```

or, to save typing, we can just define this from a starting to an ending value for the rows:

```
IQ_data[1:5, c("participant_number", "IQ")]
```

```
## # A tibble: 5 x 2
##   participant_number    IQ
##                <dbl> <dbl>
## 1                  1   147
## 2                  1   160
## 3                  3   185
## 4                  4   122
## 5                  5   157
```

In **Session 4** we'll look at more intuitive ways of accessing data from columns and rows in your data frames. However, for now we'll quickly look at ways to manipulate data using the base R functionality so you have some idea how indexing works in R.

1.5.0.3 Manipulating Tibble Information

1.5.0.3.1 Changing Values We can use these same principles to edit the information within our data. Say, we'd like to change participant number in row 2 back to 2, we just need to do this:

```
# specifying row and column by index
IQ_data[2, 1] <- 2

# specifying row by index and column by name
IQ_data[2, "participant_number"] <- 2

# specifying index within a column
IQ_data$participant_number[2] <- 2
```

You can see how all of this combined can result in some pretty powerful flexibility in how you access and manipulate your data in R.

1.5.0.3.2 Adding and Removing Columns To add a row to a data frame, we simply need to specify what we want to add and assign it a new name. Let's say that we want to add a column that indicates the operating system used by each participant.

We may have this because we made assumptions that people who use Windows, macOS, or the Linux families of operating systems differ in their IQ. This is a silly example for several reasons, not only because you can use more than one system; but we'll stick with this for now.

Imagine we already have a sample of operating systems to draw from. You don't need to understand how this works, but briefly I've used the inbuilt `sample()` function to pick from the three names with replacement, skewing the probabilities to select windows most often, followed by mac, then linux. All that matters is that we're assigning 100 names to a variable.

```
set.seed(1000) # make the sampling procedure the same for us all

operating_system <- sample(c("windows", "mac", "linux"),
                           size = 100,
                           replace = TRUE,
                           prob = c(0.5, 0.3, 0.2)
                           )
```

In the `IQ_data` set, we can add a new column for the operating systems used by the participants like so:

```
IQ_data$operating_system <- operating_system # add new column

head(IQ_data) # view first 6 rows
```

```
## # A tibble: 6 x 4
##   participant_number pet_id    IQ operating_system
##                <dbl> <chr>  <dbl> <chr>
## 1                  1 cat      147 windows
## 2                  2 cat      160 mac
## 3                  3 dog      185 windows
## 4                  4 cat      122 mac
## 5                  5 cat      157 mac
## 6                  6 dog      152 windows
```

Note that you can rename the column to anything you like. But, for consistency, I like to keep the same name as the variable which acts as the data source.

Finally, we can remove the new column (and any column) by setting the entire column to nothing (`NULL`), like so:

```
IQ_data$operating_system <- NULL # remove new column

head(IQ_data) # view first 6 rows
```

```
## # A tibble: 6 x 3
##   participant_number pet_id    IQ
##                <dbl> <chr>  <dbl>
## 1                  1 cat      147
## 2                  2 cat      160
## 3                  3 dog      185
## 4                  4 cat      122
## 5                  5 cat      157
## 6                  6 dog      152
```

Now the data is back to its original format.

We'll look at another way to remove one or several columns from your data frame in **Session 4**, but for now this a quick way to get things done using base R.

1.5.0.3.3 Adding and Removing Rows What if we want to add a new row to our data? This may be less common than adding a new column for data processing purposes, but it's good to know anyway.

First, we need to know what should go in each cell. Remember that we have to keep the data square, so you can't have missing values when you add a row. If you don't have any data, you can just put `NA` (with **no quotations**) to keep the data square but to show that you don't have any value for a given cell.

Let's assume we want to add a new participant, 101, who has a dog but an unknown IQ. We must define a list of data where we assign values to the columns that match up with our IQ data column headings. We do this like so:

```
# add new row with values
IQ_data[101, ] <- list(participant_number = 101, pet_id = "dog", IQ = NA)

tail(IQ_data) # see last 6 rows
```

```
## # A tibble: 6 x 3
##   participant_number pet_id    IQ
##                <dbl> <chr>  <dbl>
## 1                 96 dog      148
## 2                 97 dog      178
## 3                 98 cat      156
## 4                 99 dog      165
## 5                100 cat      150
## 6                101 dog       NA
```

Here, we have to define all our values to be added in parentheses, using the `list()` function:

- participant number is `101`
- pet_id is `"dog"`
- IQ is `NA` (i.e. unknown)

We then assign this to our data frame using the assignment operator (`<-`).

We have to tell R where these values should go in our data frame. Because we're adding a new row, we specify the data frame, with row 101, and all columns (remember, an empty value after the comma = all columns).

Note, this is a little more complicated than assigning a new row to our data in a data.frame, but makes us be more specific about what we are adding and where. For more details on lists, keep reading this section.

We can then remove this row again setting our data frame to itself, minus the 101st row:

```
IQ_data <- IQ_data[-101, ] # remove the new row

tail(IQ_data) # see last 6 rows
```

```
## # A tibble: 6 x 3
##   participant_number pet_id    IQ
##                <dbl> <chr>  <dbl>
## 1                 95 dog      188
## 2                 96 dog      148
## 3                 97 dog      178
## 4                 98 cat      156
## 5                 99 dog      165
## 6                100 cat      150
```

This is a pain as it isn't consistent with removing the rows. But, in **Session 4** we'll look at removing rows in a consistent way to removing columns. Still, it's useful to know how this process works.

1.5.0.4 Other Data Types

For most purposes, you'll only need vectors and data frames/tibbles when you want to work with your data. But it's worth being aware of other data types, such as lists and matrices.

With lists, you can define a vector that contains other objects. Think of this as nesting vectors within vectors (very meta).

```
person_quality <- list(glenn = c("handsome", "smart", "modest"),
                       not_glenn = c("less_handsome", "less_smart", "less_modest")
                       )
```

We can then access the elements of a list similarly to how we access vectors, but the name associated with the list will also be returned:

```
person_quality[1]
```

```
## $glenn
## [1] "handsome" "smart"    "modest"
```

If, instead, we just want the values associated with that entry, we can use `[[your list name]]`:

```
person_quality[[1]]
```

```
## [1] "handsome" "smart"    "modest"
```

We can edit these values in a similar way to how we did with a data frame. This time, we just need to tell R which vector to access first (here, it's the first vector, using double brackets so we can access the values stored there, and not the name (as would happen if we just had 1 bracket)), and then specify which location in the vector you'd like to assign a new value using a separate bracket here:

```
person_quality[[1]][4] <- "liar"
person_quality[1]
```

```
## $glenn
## [1] "handsome" "smart"    "modest"   "liar"
```

An advantage to using lists over data frames or tibbles is that the data need not be square. That is, you can have uneven lengths for your entries. Notice how glenn and not_glenn have different numbers of elements in the list. With a data frame, this is problematic. Let's try adding another participant number to our IQ_data tibble.

```
IQ_data[101, 1] <- 101
tail(IQ_data)
```

```
## # A tibble: 6 x 3
##   participant_number pet_id    IQ
##                <dbl> <chr>  <dbl>
## 1                 96 dog      148
## 2                 97 dog      178
## 3                 98 cat      156
## 4                 99 dog      165
## 5                100 cat      150
## 6                101 <NA>      NA
```

Did you notice that R automatically introduced NA values in the final cells for the pet_id and IQ columns?

Matrices work very similarly to data frames and tibbles, but they're even stricter. They can only contain the same data type throughout, so we can't mix columns containing characters and numbers without converting them all to the same data type (hint: it's a character!). I'll show you how to make a matrix, but we won't linger on that as I haven't found them all that useful in my own work.

```
matrix_example <- matrix(rep(1: 25),
                         nrow = 5,
                         ncol = 5
                         )
matrix_example
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
```

1.6 Good Practice

1.6.1 Data Checking Tips

Finally, a few tips on checking your data before you manipulate your data:

1. If you're unsure what you have in your working directory, use either check your environment (the panel in light blue in the RStudio screenshot above) or type `ls()` in the console. This will list everything currently in the global environment.
2. If you want to know the data class for some object, use the `class()` function (e.g. `class(IQ_data)`).
3. If you want to know the structure (including object classes) for some object, use the `str()` function (e.g. `str(IQ_data)`. Nicely, `str()` also tells you how many variables are in the object, and how many observations you have in total.

Most importantly with anything in R, if you aren't sure how a function works, or what it's done to your data, check how that function works! You can do this with any function by typing `?` followed by the function name into the console (e.g. `?str()`).

1.6.2 Style

I strongly recommend that you choose a style guide and stick to it throughout when you write your R code. This will make it easier to notice any errors in your code, and increases readability for you and others. Consistency is key here. Since we're using a tidyverse first approach to teaching R in this course, I recommend this one by the creator of the tidyverse:

R Style Guide by Hadley Wickham

The important things to take home are that:

- Use sensible variable names: if a column shows, e.g. participant weight, call it participant_weight
- Use verbs to describe user-defined functions: if you write a function to make all the descriptive statistics you could ever want, call it something like make_descriptives()
- use a consistent style, like snake_case_here, or even camelCase, but don't mix_snake_and_camelCase.
- comment your code with descriptions of why you've done something: you can often work out how you did it by following your code, but the why is easily lost!

1.7 Exercises

Try out the exercises below, we'll cover these in the class with the solutions uploaded at the beginning of each session in a separate downloadable link. Try to solve these questions before resorting to the solutions. I'll be there to help out where necessary. First, I want you to figure out why some code doesn't work, then we'll move on to you manipulating data.

For the opener, we'll look at the basics of solving issues with your code.

1.7.1 Question 1

What's wrong with the following code?

```r
# create a sequence from 0 to 40 in intervals of 2
sequence < - seq(from = 0, to = 40, by = 2)
```

1.7.2 Question 2

Why doesn't this return anything?

```r
# draw 100 times from a uniform distribution between 1 and 10
uniform_distribution <- runif(n = 100, min = 1, max = 10)
uniform_distrbiution
```

Now, we'll work with a data set. But I'd like you to produce it. We'll break this down into steps for now.

1.7.3 Question 3

Lets pretend we have 100 participants. Create a variable that will store participant IDs. Their IDs can be anything you like, but some sort of logical numbering looks best.

Hint: `seq()` is your friend here!

1.7.4 Question 4

Create a variable that will store their scores on some test. Let's assume participant scores are drawn from a **normal distribution** with a **mean** of 10 and an **SD** of 0.75.

Hint: `rnorm()` is a wonderful thing.

1.7.5 Question 5

Finally, create some ages for your participants. Here, you'll have to use a new command called `sample()`. See if you can figure out how to use this command. If you're lost, remember to use `?` on the function.

1.7.6 Question 6

Create a data frame consisting of your participant IDs, their ages, and test scores.

1.7.7 Question 7

Take a look at the start and the end of your data frame. Do the values look reasonable? (Don't worry about outliers or missing values etc., we'll check those in later classes.)

1.7.8 Question 8

Access the row (and all columns) for participant 20. What do the scores look like? Are they the same as the person sitting next to you? Why or why not, might this be the case?

1.7.9 Question 9

Access just the test score for participant 73.

1.7.10 Question 10

Output some simple descriptive statistics from your sample. We want to know:

- Mean age
- Mean score
- SD score (*Hint*: Use `sd()`)

It may be useful to store these descriptives in another data frame for further use later on. Can you do that?

1.7.11 Question 11

Access all rows and columns where the test score is greater than 11.

Hint: define the column as `data$column` in your subsetting command, and perform a logical operation on this.

1.7.12 Question 12

Access all rows from the participant_number and IQ columns where the test score is greater than 11.

Hint: use the `c()` function to select multiple columns.

Chapter 2

Data Visualisation 1

In this session we'll look at data visualisation using the **ggplot2** package (Wickham, 2009) from the tidyverse (Wickham, 2017). As with most R stats courses, we're focusing on data visualisation early on as this allows you to get a good grasp of your data and any general patterns within those data prior running any inferential tests.

We'll follow Hadley Wickham's approach in R for Data Science by getting you working on producing some pretty graphs from the outset to see how `ggplot()` works. After that, we'll look at how ggplot can handle several data types. Along the way, we'll add some customisation to our graphs so you can see the flexibility of this package.

2.1 Getting Started

First, we'll load the packages necessary for this class. Nicely, ggplot2 is part of the tidyverse family, so we don't need to load this separately to the other packages in our library.

```
library(tidyverse)
```

Sorry to have made you create your own data frames before, but R and it's packages often come with in-built data sets. We'll use the `starwars` data set from `dplyr()` which loaded with the `tidyverse` package. Why star wars? It's convenient, and I'm a big nerd, so indulge me. Because this is built into R, you won't see it in your Data pane in the **Global Environment**. That doesn't matter for us, but rest assured it is there. Let's get a sense of how this data looks. How about printing the first 10 entries?

2.1.1 The Star Wars Tibble

```
# look at first 10 entries
starwars
```

```
## # A tibble: 87 x 14
##    name         height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex   gender homew~5
##    <chr>         <int> <dbl> <chr>   <chr>   <chr>     <dbl> <chr> <chr>  <chr>
##  1 Luke Skywa~     172    77 blond   fair    blue       19   male  mascu~ Tatooi~
##  2 C-3PO           167    75 <NA>    gold    yellow    112   none  mascu~ Tatooi~
##  3 R2-D2            96    32 <NA>    white,~ red        33   none  mascu~ Naboo
##  4 Darth Vader     202   136 none    white   yellow     41.9 male  mascu~ Tatooi~
##  5 Leia Organa     150    49 brown   light   brown      19   fema~ femin~ Aldera~
##  6 Owen Lars       178   120 brown,~ light   blue       52   male  mascu~ Tatooi~
##  7 Beru White~     165    75 brown   light   blue       47   fema~ femin~ Tatooi~
##  8 R5-D4            97    32 <NA>    white,~ red        NA   none  mascu~ Tatooi~
##  9 Biggs Dark~     183    84 black   light   brown      24   male  mascu~ Tatooi~
## 10 Obi-Wan Ke~     182    77 auburn~ fair    blue-g~    57   male  mascu~ Stewjon
## # ... with 77 more rows, 4 more variables: species <chr>, films <list>,
## #   vehicles <list>, starships <list>, and abbreviated variable names
## #   1: hair_color, 2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

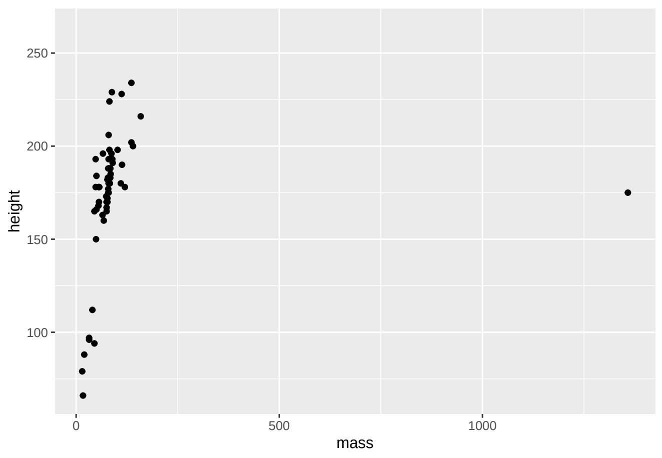
I know that the starwars data set is saved as a tibble. That allowed me to just print its name to see the first 10 entries. But be wary of this with large data sets where you don't know how it's stored. You don't want to flood your console if your data is stored as a `data.frame`!

Let's plot the mass and height of our characters against each other to see if there's a trend.

2.1.2 Plotting in ggplot2

```
ggplot(data = starwars) +
  geom_point(mapping = aes(x = mass, y = height))
```

```
## Warning: Removed 28 rows containing missing values (geom_point).
```



So, we can see just how easy it is to create a plot of points in ggplot. Well done! There seems to be a positive relationship between mass and height in the starwars data set. We also got a couple of surprises:

1. A warning about 28 rows that contain missing values;
2. A really big outlier.

First, we'll explore how ggplot works, then we'll look into these surprises.

The `ggplot()` function always needs to take a data set. This data set should hold everything you want to plot. Crucially, ggplot builds up the plots in layers. So making the first call `ggplot(data = starwars)` tells ggplot where it should look for data, but it doesn't do much else aside from making a grey background.

After this, you need to add some layers to your plot in the form of geometric objects (or geoms, for short). We decided that because we want to look at the link between mass and height, two continuous variables, that adding some points to the plot will be most useful for getting an idea of how the data are related. To do this, we used the `geom_point()` function. There are other geoms we could add, but for now we'll focus on points.

Crucially, geom functions take as an argument the mapping in your data. That is, how the visuals of the plot are mapped to your data. This mapping is always defined in terms of the aesthetics of your plot `aes()`, e.g. which variables to map onto the x and y axis, in this case.

You can see how this makes ggplot so flexible:

1. Your data argument is flexible, so you can pass different data sets to the same chunk of code by changing out what you pass to the `data =` argument.

2. Your aesthetics are flexible, so you can pass different columns to your x and y axis
3. Your aesthetics are even more flexible because they can take aesthetics other than just what to plot on the x and y axes

Let's see the flexibility of your aesthetics in action.

2.1.3 Cleaning Before Plotting

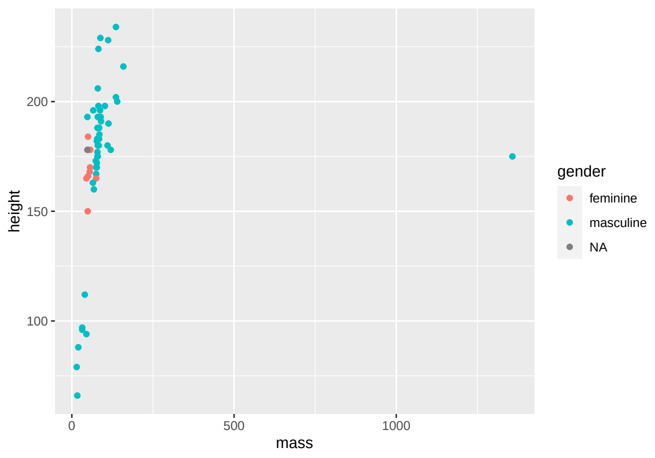
We saw before that R gave us a warning that we have rows containing missing values. In this instance, this just means that 28 people weren't plotted because they didn't have height and/or mass values. I'll save filtering data properly (and what the `%>%` (read pipe) symbol does) for another lesson, but we'll get rid of them for now by running this code, below:

```
filtered_starwars <- starwars %>%
  drop_na(height, mass)
```

2.1.4 Changing Your Aesthetics

Let's say we're interested in showing the relationship between mass and height within genders. How can we do this? One way would be to add colour to our the points on the plot in order to highlight the different genders.

```
ggplot(data = filtered_starwars) +
  geom_point(mapping = aes(x = mass,
                           y = height,
                           colour = gender
                           )
             )
```



We don't get that warning now that we removed the NAs and passed the filtered data as an argument. That wasn't really necessary as ggplot can handle that for us, but it stops the horrible warning from popping up!

Warnings in R are there to tell you that the output of your code may not contain what you wanted it to contain. In the previous plot, ggplot dropped those with missing heights and masses, even though we didn't explictly tell it to do so. Here, because we filtered those entried with missing values before plotting, we don't get a warning.

Warnings are different to errors in that your code will still work, but you need to check out whether it did what you wanted it to do. On the other hand, errors indicate you did something wrong and your code will fail to run.

Now, we now have colour to see the relationship between mass and height across the genders in the data set. But it's a little difficult to see this relationship given the outlier. Let's see what that outlier is, and whether we should remove it from our data.

Again, you'll learn how this filtering works in later lessons. But for now, I just want to show you how it's useful to understand your data prior to creating a final plot of your data.

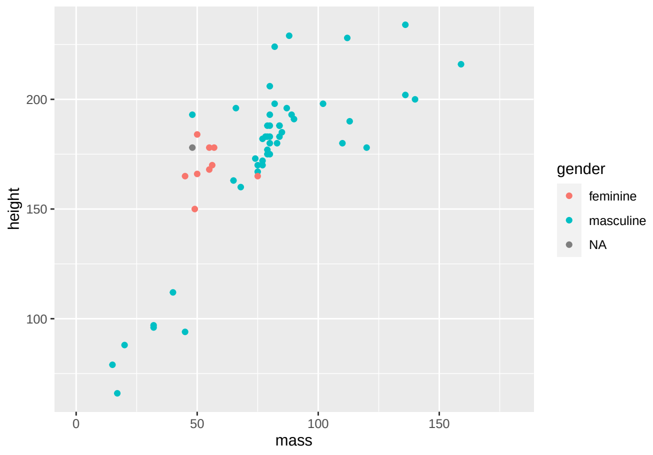
```
filtered_starwars %>% filter(mass > 1000)
```

```
## # A tibble: 1 x 14
##   name          height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex   gender homew~5
##   <chr>          <int> <dbl> <chr>   <chr>   <chr>     <dbl> <chr> <chr>  <chr>
## 1 Jabba Desil~    175  1358 <NA>    green-~ orange      600 herm~ mascu~ Nal Hu~
## # ... with 4 more variables: species <chr>, films <list>, vehicles <list>,
## #   starships <list>, and abbreviated variable names 1: hair_color,
## #   2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

```
# overwrite data to remove outlier (keep all with mass below 1000)
# filtered_starwars <- filtered_starwars %>% filter(mass < 1000) # not run
```

Of course, it's Jabba the Hutt. We could choose to throw Jabba out of our data by using the code above to overwrite the data (commented out), but for now, we'll see just how powerful ggplot is without throwing away our data.

```
ggplot(data = filtered_starwars) +
  geom_point(mapping = aes(x = mass,
                           y = height,
                           colour = gender
                           )
             ) +
  coord_cartesian(xlim = c(0, 180))
```



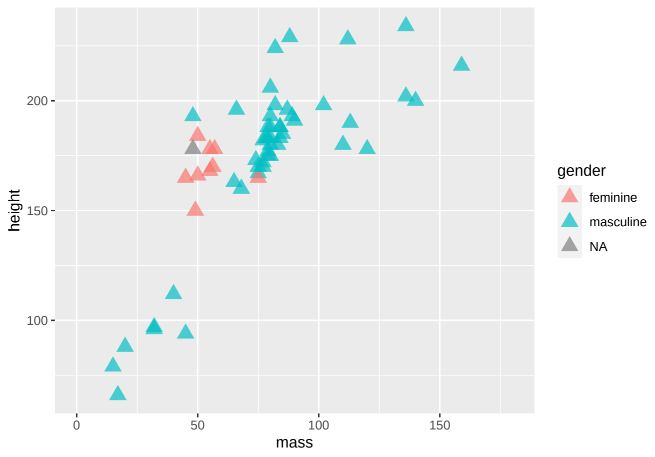
This time we added a function `coord_cartesian()` to the end of our `ggplot()` call. We defined the limits of the x axis to be between 0 and 180. This way, we can get a better look at the trends in our data.

Why do we define limits inside this function? Well, we could have also manually defined the scale with `scale_x_continuous(limits = c(0, 180))`. This may seem more inuitive, but it throws out the data points outside the limits prior to plotting. Why is this a problem? ggplot has some nice functionalities such as drawing lines of best fit for you based on the data in the plot. If you throw data away while plotting, your line of best fit will shift. So, if you decide that you want to change your scale but keep your model fits for all data, use `coord_cartesian()`.

Given this is an outlier, your choice doesn't matter if you're just trying to show trends like this, but your choice is important if you want to show any inferential statistics associated with the data.

Next up, we'll look at changing a few components of the points on the plot.

```
ggplot(data = filtered_starwars) +
  geom_point(mapping = aes(x = mass,
                           y = height,
                           colour = gender
                           ),
             alpha = 0.7, # opacity
             shape = 17, # triangles
             size = 4) + # bigger points
  coord_cartesian(xlim = c(0, 180))
```



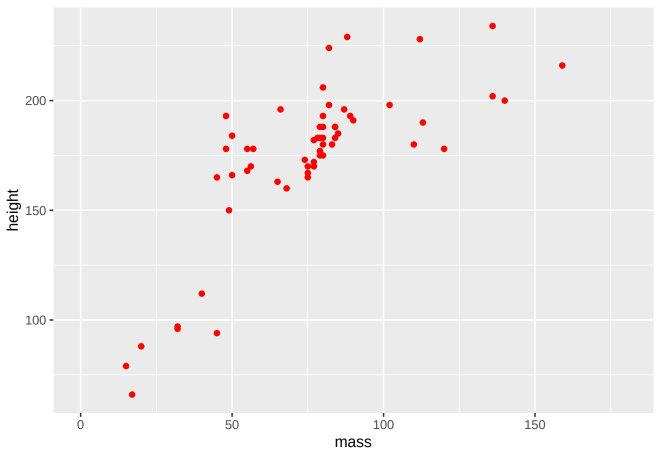
We've added 3 variables outside of the `aes()` mapping. This means that all of the points within the plot are changed in the same way:

1. They all become a little bit transparent through the `alpha` variable definition;
2. They all take a triangle shape through the `shape` variable definition (and the number associated)
3. They are all the same size due to the `size` variable definition.

If we put these variables within the `aes()` mappings and associated them with a variable within the data set, such as `gender`, then each point would be affected differently depending on which level of the gender factor the individual data points belong to. It's important to remember that everything within `aes()` is mapped onto variables with which to display your data. So, the x location, y location, and anything else that you define within `aes()` can vary by your data set. Everything outside of it will affect all levels of your data.

Here, we'll define colour both within and outside the `aes()` mapping, causing a clash. Try this plot below to see how clashes are resolved in ggplot:

```
ggplot(data = filtered_starwars) +
  geom_point(mapping = aes(x = mass,
                           y = height,
                           colour = gender
                           ),
             colour = "red") +
  coord_cartesian(xlim = c(0, 180))
```



The different colours for each level of gender are now gone, along with the legend! Our variable definition for `colour` outside of the `aes()` mappings overrides that within `aes()`. This is because we've manually set the aesthetic properties of our plot by defining colour as an argument of the geom function, rather than of the aethetic mapping.

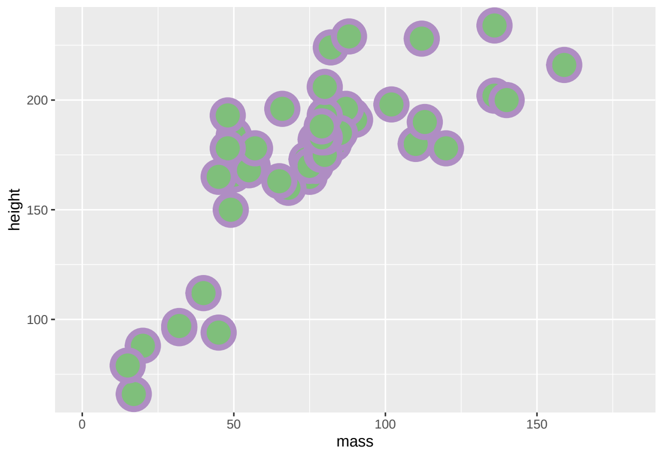
R has some built in shapes that we can define within our plots. For shapes, these are divided into three categories:

- Colour border with hollow fill (0 - 14)
- Colourless border with colour fill (15 - 18)
- Colour border with colour fill (21 - 24)

Bear in mind that colour and fill are different properties that we can control within our plots. Below, I've used hex values to specify the exact colours that I'd like for the colour (around the border) and the fill for our points. You can find a nice hex selector at htmlcolorcodes.com which will allow you to customise your plot colours to your liking. Just change the letters and numbers after the `#` in the `colour` call, and you can change the colours to your liking.

In the example below I chose my colours based on the diverging colourblind safe selection of colours from Color Brewer. I'd recommend that you use this if you're going to include colour in any plots for papers/presentations.

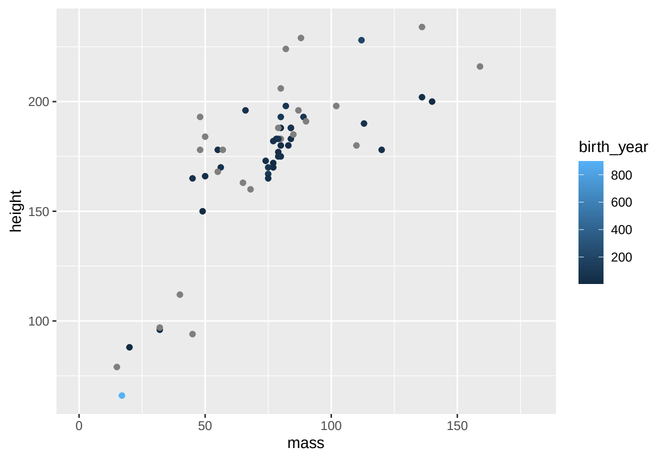
```
ggplot(data = filtered_starwars, na.rm = T) +
  geom_point(mapping = aes(x = mass,
                           y = height,
                           colour = gender
                           ),
             colour = "#af8dc3",
             fill = "#7fbf7b",
             shape = 21,
             size = 8,
             stroke = 3
             ) +
  coord_cartesian(xlim = c(0, 180))
```



Try to mess about with the different definitions that we provided above. Change the colour, size, and stroke values to see how these variables work.

We've used the aesthetics above to define categorical data by colour. But what happens if we use continuous data?

```
ggplot(data = filtered_starwars, na.rm = T) +
  geom_point(mapping = aes(x = mass,
                           y = height,
                           colour = birth_year
                           )
             ) +
  coord_cartesian(xlim = c(0, 180))
```



You can see that we get a sliding scale for the hue of the points. Pretty neat, but also quite difficult to get a real idea of where on the scale the points lie.

There are many inbuilt plots that you can create with ggplot2 by mapping your data to different geoms. To get an idea of all of different types of geoms, type `??geom`. This should give you an index of all of the geom types available to ggplot2 in your Help pane of RStudio.

2.2 Exploring Different Geoms

You've already learned about `geom_point()` above, but now we'll explore the other geom types.

For this exploration, we'll use a different data set with a limited number of groups for ease of visualisation. I've simulated some data for this exercise. You can find this data in the inputs folder for this lesson. (If you're interested, you can also find the script for generating this data in the data_generation folder).

Let's load this data into R. Here, we use the function `read_csv` rather than the base R `read.csv`. This is because `read_csv` is faster, typically does a better job at guessing the data types for your columns of data, and saves the output as a tibble.

Note: `read_csv` will try to guess the types of data in each column for a data set loaded into R, but sometimes it can fail. We'll cover instances where this fails, and how to remedy it, in the next lesson.

```r
rt_data <- read_csv("inputs/rt_data.csv")
```

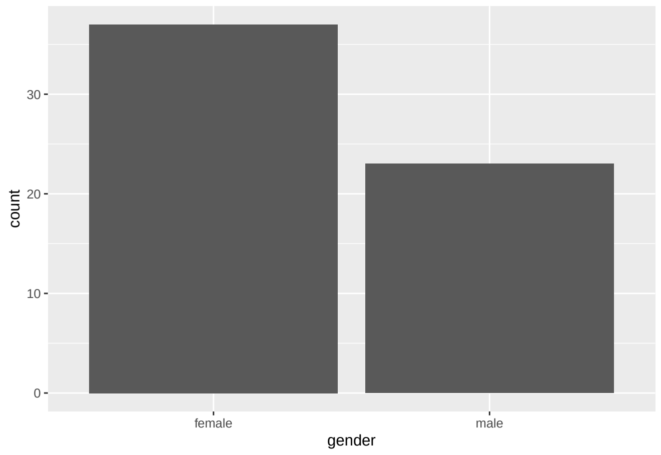
How should we interpret this data? Imagine that we ran a lexical decision task on two groups of participants. In one condition, participants responded to sentences such as *"Eric turned down the volume"*, and had to indicate whether this sentence made sense by turning a knob. In the match condition, agreement matched the motion indicated in the sentence (e.g. left = yes, it makes sense), and in the mismatch condition agreement did not match the motion indicated in the sentence (e.g. right = yes, it makes sense). Zwaan and Taylor (2006, Experiment 4) hypothesised that when the action and response are matched, resposne times should be quicker than if they do not match.

The `participant` column indicates individual participants, the `gender` column indicates the gender of our participants, `response_condition` indicates whether participants took part in the match or mismatch conditions, and `reaction_time`, our dependent variable, represents the average reaction time in milliseconds for each participant to respond to whether or not the sentences made sense.

2.2.1 Bar Plots

Bar plots are one of the most common plots you'll come across in Psychology. They're most useful for representing counts of data that are divided into categories. For example, you could use a bar plot to show the number of male and female participants in your study. Using our simulated data set, we can use a bar plot to show the counts of males and females in our data set.

```r
ggplot(data = rt_data,
       mapping = aes(x = gender)
       ) +
  geom_bar()
```



You'll often find that psychologists like to plot continuous data in a bar plot. I've done this myself, but we can do better. One alternative would be to use a boxplot, a violin plot, or even a pirate plot which we'll explore in the next lesson (well, we are plotting with Rrrrrrrr…).

Since we've got data which represents mean scores for each participant, this will be suitable for plotting straight away. If, however, you have raw data, remember to aggregate by participants or items prior to plotting. We'll cover this in later classes.

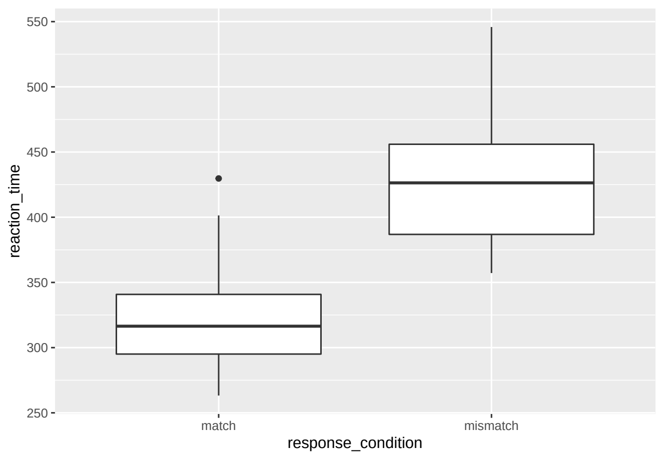
2.2.2 Box Plots

Box plots provide a better way to represent continuous outcomes (such as reaction times) than bar plots as they give you more information about the variance within your data set.

2.2.2.1 How to read a box plot

- The middle line represents the median
- The upper white section of the box the upper quartile: 75% of scores fall below this.
- The lower white section the lower quartile: 25% of scores fall below this.
- Together the quartiles represent the interquartile range: The middle 50% of scores.
- The limits of the whiskers (black lines) for the upper and lower parts of the graph represent the smallest and largest observations that are equal to the upper or lower quartiles minus or plus 1.5 times the interquartile range. Effectively, this is most of the rest of the data, apart from outliers.
- The dots represent outliers (i.e. those values outside of the whiskers)

```
ggplot(data = rt_data,
       mapping = aes(x = response_condition,
                     y = reaction_time
                     )
       ) +
  geom_boxplot()
```



We can see that the median reaction time is highest for those in the mismatch condition. Notice also that the interquartile range is larger, and so is the upper limit of the whiskers. This all suggests that those in the mismatch group vary more from each other than those in the match condition.

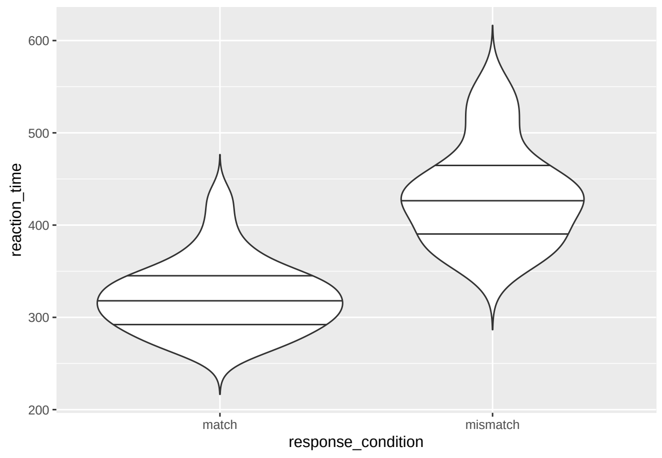
Also, notice the outlier in the match group. Someone obviously has a very slow reaction time compared to the rest of the cohort. I wonder why that could be? Perhaps it's because they misinterpreted the instructions, accidentally putting themselves into the mismatch condition. Or, it could be because I simulated this data and didn't control for how outliers were distributed.

2.2.3 Violin Plots

Additionally, we have violin plots which show you the density of the mean scores. The wider the section of the violin, the more scores around that area. We set `trim` to `FALSE` within the violin plot so that we see the full tails of the data. If we set this to `TRUE`, then the tails are trimmed to the range of the data.

It's also useful to draw quantiles on the violin plot, so we can see the quantiles as with the box plot using `draw_quantiles` and by specifying where we want these quantiles. Here, we chose the upper and lower 25% and the interquartile range.

```r
ggplot(data = rt_data,
       mapping = aes(x = response_condition,
                     y = reaction_time
                     )
       ) +
  geom_violin(
    trim = FALSE,
    draw_quantiles = c(0.25, 0.5, 0.75)
    )
```

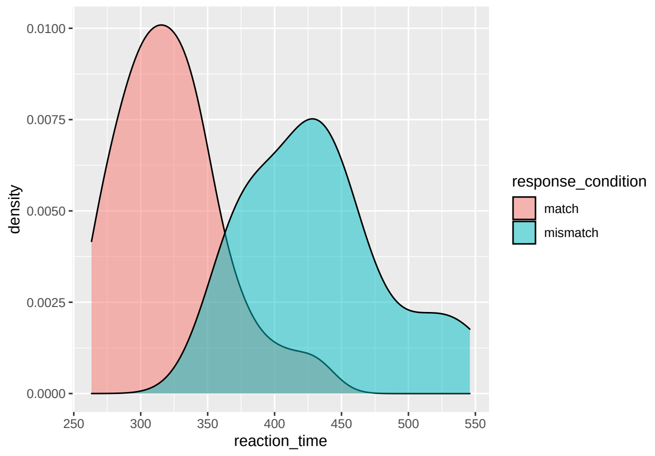


2.2.4 Density Plots

Now we'll look at ways to check for the distribution of your continuous variables. This is a good way to get an eye for whether or not your data are skewed and require a transformation on the data, or a non-parametric test when it comes to inferential statistics. Again, we'll return to what this means in regards to statistics later in this course, but for now it's a good idea to understand how you can check the distribution of your data prior to computing any statistics.

We've added the additional argument `fill` to the aethetic mapping, to specify that we wanted different colours for our groups. We also specified the additional argument `alpha` in our density plot to specify that we wanted some opacity to the densities.

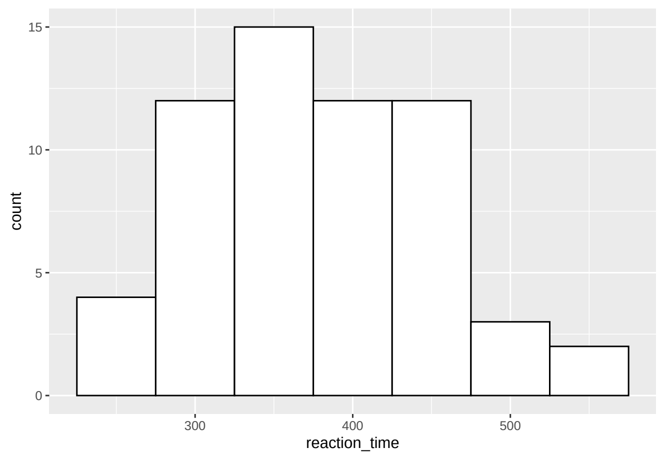
```
ggplot(data = rt_data,
       mapping = aes(x = reaction_time,
                     fill = response_condition
                     )
       ) +
  geom_density(alpha = 0.5) # alpha = opacity
```



2.2.5 Histograms

If you have relatively few observations, a histogram may be more appropriate than a density plot. But, we'll just stick to the same data as before to see how this works. Here, it's often useful to set the `binwidth` prior to plotting. ggplot will try to set it to a default if it can, but this may not be what you want. Here, we've set the `binwidth` to 50, so we count how many observations we have for reaction times in 50ms bins, i.e. from 225 to 275ms, from 275 to 325ms etc. We also set the `fill` of the bars to white, and the `colour` to black so we get white bars with a black outline.

```
ggplot(data = rt_data, mapping = aes(x = reaction_time)) +
  geom_histogram(binwidth = 50,
                 fill = "white",
                 colour = "black"
                 )
```



2.3 Exercises

If that's not enough to make you fall asleep, try out the exercises below. We'll cover these in the class with the solutions uploaded at the beginning of each session in a separate downloadable link. Try to solve these questions before resorting to the solutions. I'll be there to help out where necessary.

In this section, we'll use some new data from the pre-packaged data sets in R. First off, we'll load the `chickwts` data set, which looks at chicken weights by different feed types.

```
data(chickwts)
```

2.3.1 Main Exercises

2.3.2 Question 1

Take a look at the first 6 rows of the data set. How does the data look? Is this appropriate for plotting?

2.3.3 Question 2

Calculate the overall means for the chick weights.

2.3.4 Question 3

Calculate the overall SD for the chick weights.

2.3.5 Question 4

Create the basis of a ggplot by defining the `chickwts` data as the data argument. Assign this all to the object `chick_plot`.

2.3.6 Question 5

Make a box plot of the chick weights by feed. To do this, use your `chick_plot` object and add the function for creating a boxplot.

2.3.7 Question 6

Add colour to your box plot by the feed type.

2.3.8 Question 7

Create a density distribution of the chick weights by feed type. Set different colours and fills by the feed type. To make all densities visible, set the transparency for all distributions to 0.4.

2.3.9 Question 8

Make a bar plot to show the counts of each feed type.

2.3.10 Question 9

Pick 6 hex colours from the Color Brewer website. Put these together in a variable, `bar_colours`. Create your bar plot again, but this time make the fill of the bars the same as those stored in `bar_colours`.

Why do we not get a legend when we specify colours this way, but we do if we specify colours as in **Question 7**?

We'll return to adding colour to bars while keeping the legend labels in the next lesson.

2.3.11 Question 10

Make a histogram showing the overall distribution of the chick weights. Set the bin width to 50, and make the bars have a white fill and black border.

2.3.12 Additional Exercise

Make a point plot from a data set of your choosing. Check out the inbuilt data sets in R by typing `data()` in the console. Customise this plot to your liking.

Chapter 3

Data Visualisation 2

In this section we'll cover more advanced plotting methods in ggplot2. We'll look at customising plots, making **pirate plots**, installing packages from GitHub, faceting, and stitching plots together.

To get started, as always we'll load our packages and saved data from the previous lesson.

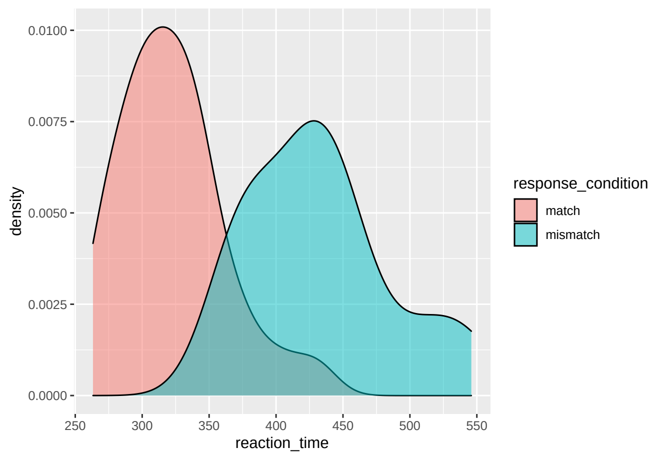
```
library(tidyverse)
```

```
rt_data <- read_csv("inputs/rt_data.csv")
```

3.1 Customising Your Plots

Take the desnity plot below, this is functional, but it's pretty ugly.

```
ggplot(data = rt_data,
       mapping = aes(x = reaction_time,
                     fill = response_condition
                     )
       ) +
  geom_density(alpha = 0.5)
```



You already know how to change the colours of your aesthetics across all conditions and split by condition, so now we'll look at other ways to improve the plot.

In this version of the plot, we use the same code as before, but we add labels to the axes using `labs()`, assigning a nicer looking version of our variables to the x and y axes.

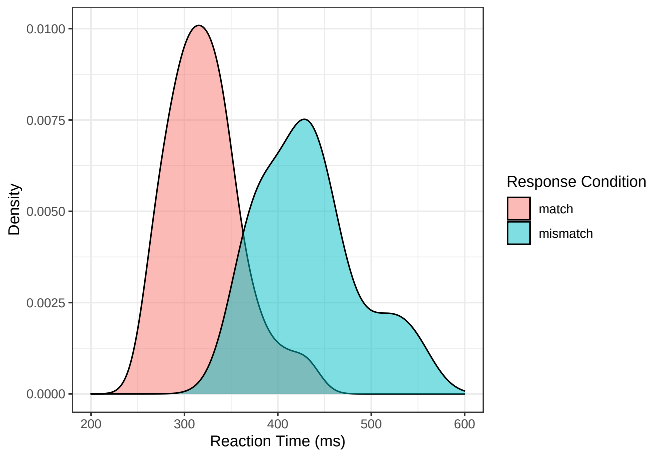
Additionally, we change the scale of the x-axis using the `scale_x_continuous` function. To this function, we pass the **limits** that we want for our axis (between 200 and 600ms), and we identify the breaks, or where we want the ticks along the axis. We pass another function that we learned in the Lesson One, called `seq()`. This sets up a sequence of numbers for us without us having to type them all out. Here, it goes `from` 200 `to` 600 `by` ticks every 100ms; as a result, our axis gets labels of 200, 300, 400, 500, and 600.

On top of this, we also improved the label for our legend using the `guides()` function. As our legend is only there to identify different parts of the graph with a different colour (from `fill = response_condition` in `aes()`), then we have to tell the guide to change the legend that pops up because of the differnt coloured parts of the plot. So, we change our guide, and change the legend that comes up because of the change in colour (`guides(fill = guide_legend())`) and within the guide legend, we change the title (`title = "Response Condition"`. I know this sounds like a lot to take in, and you're very likely to forget how this works (I do all the time), but hopefully you can get a grip of it by seeing it in action.

Finally, we've changed the theme of our plot to `theme_bw()`. This is one of many inbuilt themes in ggplot2, but I find it one of the cleanest.

```
ggplot(data = rt_data,
       mapping = aes(x = reaction_time,
                     fill = response_condition
```

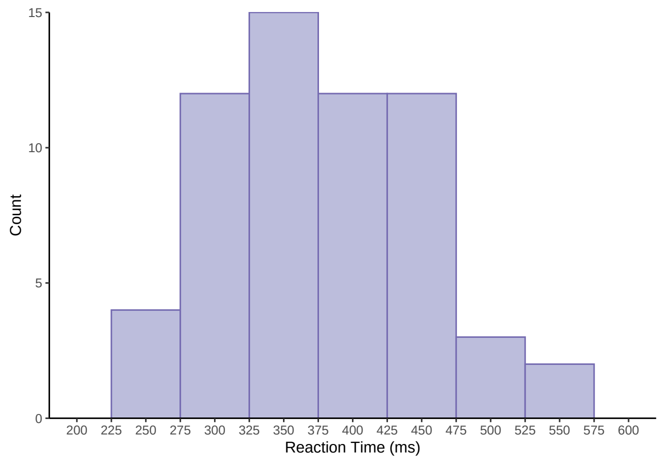
```
                  )
      ) +
  geom_density(alpha = 0.5) +
  labs(x = "Reaction Time (ms)",
       y = "Density"
      ) +
  scale_x_continuous(limits = c(200, 600),
                     breaks = seq(from = 200,
                                  to = 600,
                                  by = 100
                                  )
                     ) +
  guides(fill = guide_legend(title = "Response Condition")) +
  theme_bw()
```



With the histrogram below, we don't have much new to introduce, expect this time we use `theme_classic()` instead of `theme_bw()`. This gets rid of the major and minor grid lines from the previous plot, and also keeps only the lines for the axes. However, we added the argument `expand` to the `scale_y_continuous()` function, and pass this the values 0 and 0. This makes removes the spacing between the plotted elements and the axes. These values simply indicate how much extra space should be added to the top and bottom of the plot.

```
ggplot(data = rt_data,
       mapping = aes(x = reaction_time)) +
  geom_histogram(binwidth = 50,
                 fill = "#bcbddc",
                 colour = "#756bb1"
                 ) +
  scale_x_continuous(limits = c(200, 600),
                     breaks = seq(from = 200,
                                  to = 600,
                                  by = 25
                                  )
                     ) +
  scale_y_continuous(expand = c(0, 0)) +
  labs(x = "Reaction Time (ms)", y = "Count") +
  theme_classic()
```

```
## Warning: Removed 2 rows containing missing values (geom_bar).
```



3.2 Pirate Plots

Now we'll look at a new plot type that takes a bit of extra work to generate; pirate plots. Pirate plots are a great choice of plot for the indecisive. They are essentially the individual points of data, a bar plot, a violin plot, and a confidence interval interval all in one. This way, you get to see the raw, descriptive, and inferential data on one plot!

This is a nice way to show data that are grouped by categories but with a continuous dependent variable.

We could make these ourselves by doing some calculations and combining `geoms` in ggplot2. Or, we could just install a package from Github which will do that for us. First, we'll have to install `devtools` in R which will allow us to install packages from Github. (For packages on CRAN, this isn't necessary, but unfortunately ggpirate isn't on CRAN at the time of writing.)

To install `devtools` and the `ggpirate` package, uncomment and run the code below. Then as always, use `library(ggpirate)` to load the package.

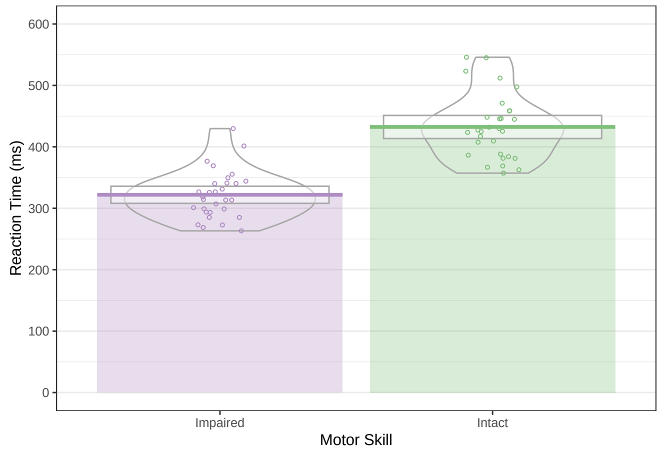
```r
# install.packages("devtools")
# devtools::install_github("mikabr/ggpirate")
library(ggpirate)
```

Below, we'll make a pirate plot. Note that you only need the first two calls (lines 1-6) to create the plot: The first to set up how you're going to map your data (and the source of your data), and the second to add the `geom` for the pirate plot itself. We added the aethetics of colour and fill to match our conditions within this call, so the two levels of `response_condition` have different colours.

For the additional lines:

- `labs` allows you to manually change the x and y axis labels
- `scale_x_discrete` allows you to manipulate your x scale. Within this, we change the labels of the columns using `labels`. We changed them to Impaired (vs. impaired) and Intact (vs. intact) for stylisitc reasons.
- `scale_y_continuous` allows you to manipulate your y scale. Here, we set the `limits`, i.e. how far to extend our y-axis to between 0 and 600ms. Additionally, we set our `breaks` to increment in the sequence (`seq`) from 0 to 600 by 100ms. This way, we've added more granularity to our axis ticks.
- We use `theme_bw()` to change to a black and white theme. There are other themes in ggplot2, but this one is nice and clean.
- Next, `theme` allows you to manually specify other aspects of how your plot should look. Here, we used `panel.grid.major.x` and set this to nothing (`element_blank()`) because we don't need vertical lines in our plot.
- Finally, we define the colour and fill for our plot using manual hex values using the `scale_colour_manual()` and `scale_fill_manual()` functions.

```r
ggplot(data = rt_data,
       mapping = aes(x = response_condition,
                     y = reaction_time,
                     colour = response_condition,
                     fill = response_condition)
       ) +
  geom_pirate() +
  labs(x = "Motor Skill", y = "Reaction Time (ms)") +
  scale_x_discrete(labels = c("Impaired", "Intact")) +
  scale_y_continuous(limits = c(0, 600),
                     breaks = seq(from = 0, to = 600, by = 100)
                     ) +
  theme_bw() +
  theme(panel.grid.major.x = element_blank()) +
  scale_colour_manual(values = c("#af8dc3", "#7fbf7b")) +
  scale_fill_manual(values = c("#af8dc3", "#7fbf7b"))
```



In this plot, we have coloured bars and lines indicating the mean scores, a box representing the 95% confidence interval assuming a normal sampling distribution, violins indicating density, and the raw data points.

If the 95% confidence interval between the two groups doesn't overlap, we can be fairly sure there is a significant difference between the groups. So, here we can be fairly certain the two groups differ in reaction times.

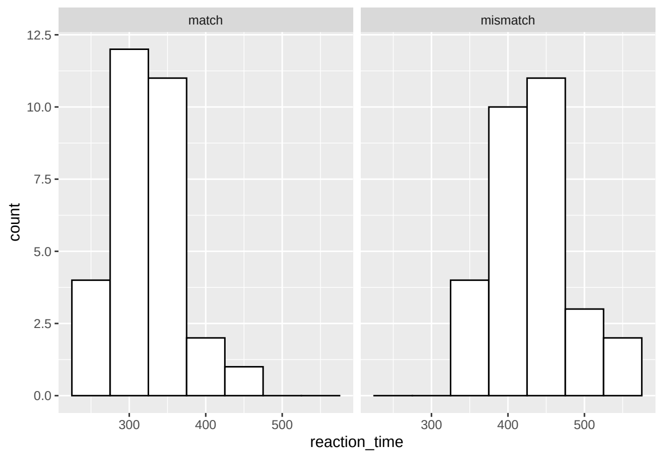
3.3 Faceting

Another useful part of plotting in ggplot2 is that you can make facets of plots, or subplots. This is a good way to display your data if you have multiple categorical variables. Essentially, you'll get a plot for each category in your data.

3.3.1 Facet Wrap

If you want to create facets from one variable then use `facet_wrap()`.

```
ggplot(data = rt_data, mapping = aes(x = reaction_time)) +
  geom_histogram(binwidth = 50,
                 fill = "white",
                 colour = "black"
                 ) +
  facet_wrap(~ response_condition)
```



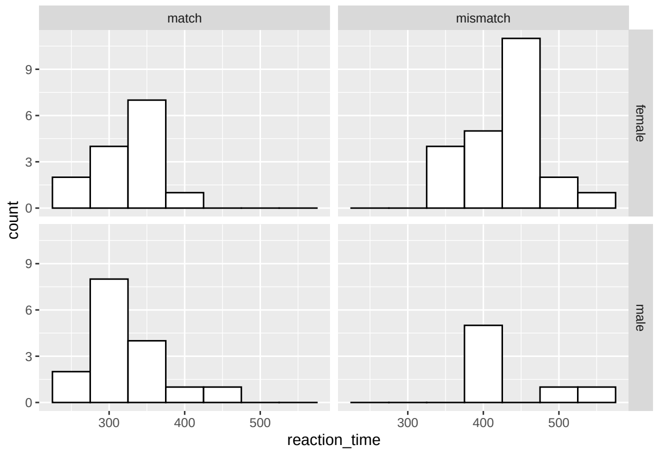
In this plot, we've specified a histogram as we normally would. However, we use `facet_wrap()` and a tilde (`~`) to create a formula for how to display our plots. We define our variable with which to split our plots to the right of the `~`, and ggplot automatically plots the two separately at the same time. Notice that we get useful labels at the top of these plots, too.

3.3.2 Facet Grid

If we wanted to make a facet by two variables, then we would use `facet_grid()` instead. In this case, we just add each variable to either side of the `~` and ggplot will do the splitting for us.

Let's see how this works if we split our data by gender and response condition.

```
ggplot(data = rt_data, mapping = aes(x = reaction_time)) +
  geom_histogram(binwidth = 50,
                 fill = "white",
                 colour = "black"
                 ) +
  facet_grid(gender ~ response_condition)
```



The order in which you specify the two variables matters. Try swapping around between `facet_wrap(gender ~ response_condition)` and `facet_wrap(response_condition ~ gender)` to see how this works.

3.4 Calculating Statisitcs in ggplot2

Sometimes, plotting just the means isn't enough. We saw how useful the 95% confidence interval from `ggpirate` is for making inferences about the differences between groups. Nicely, we can get standard errors or confidence intervals around our data points within ggplot for other geoms.

3.4.1 Means and Error Bars

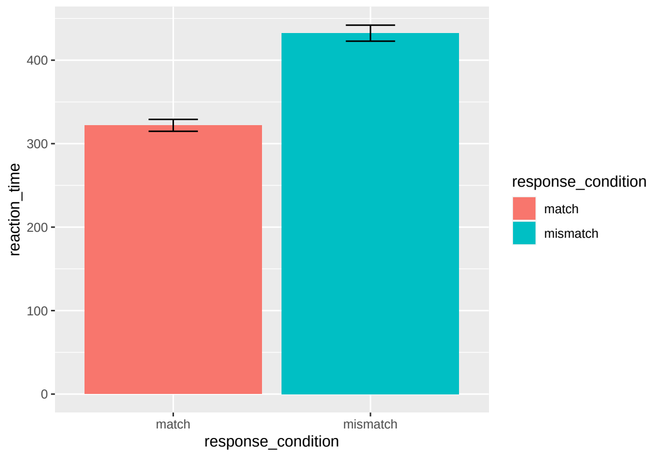
Let's say you wanted a bar plot with error bars showing the standard error. You can create this in ggplot using the `stat_summary()` function. In the first instance here, we tell it that we want to run the function `mean` over our data that make up the y-axis; hence `fun.y = mean`. We also need to specify which geom we want to return from this.

Try changing the geom to `point` in the first `stat_summary()` call to see what happens when you run this plot with `geom = "point"`.

Finally, we ask for another summary, but this time we want an error bar. So, for the geom call we request an error bar. Crucially, the function we require to get this errorbar is `fun.data = "mean_se"`. That's because we need the mean to know where the centre the errorbar, and the standard error to get the limits of the bar. We manually changed the width of the bar to a quarter of the bar size using the `width` argument to stop ggplot returning super wide error bars.

```
ggplot(data = rt_data,
       mapping = aes(
         x = response_condition,
         y = reaction_time,
         fill = response_condition
         )
       ) +
  stat_summary(fun.y = "mean", geom = "bar") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = 0.25)
```

```
## Warning: `fun.y` is deprecated. Use `fun` instead.
```



I don't often use `stat_summary` in my own plots, as I often want to know exactly how I've calculated things like the standard error. Doing things by hand allows you to change error bar sizes appropriately for within- and between-subject designs. But, this is useful if you want to create a plot rapidly, or want to avoid the hassle of writing extra code.

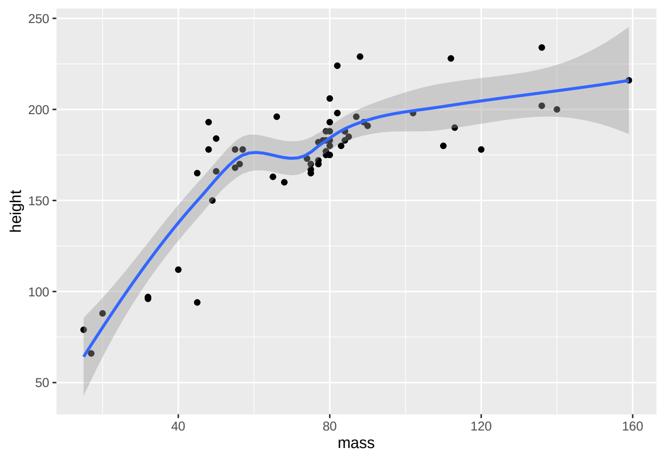
3.4.2 Model Fits

Here' we'll switch again to a different data set that has two continuous variables. The `starwars` data set is useful for this exercise.

We can use the `geom_smooth()` function to fit a model to our data. This defaults to a loess method, but we can change this to a linear model (or other alternatives) as in the second plot below. By default, these smoothed plots display a ribbon around the fit which indicates the confidence interval (95% by default).

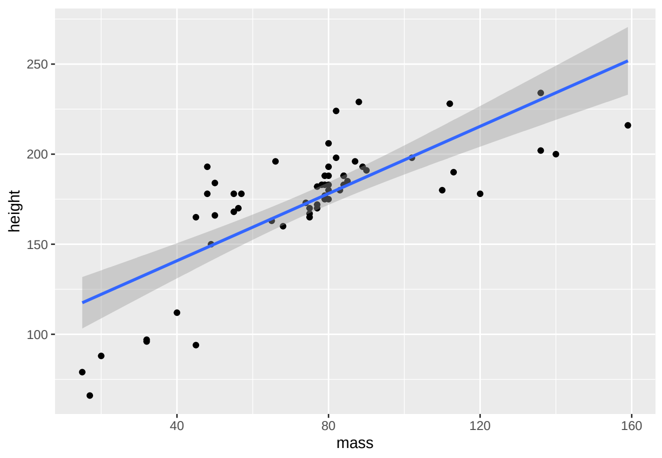
```
# remove NA and mass above 1000
filtered_starwars <- starwars %>%
  drop_na(height, mass) %>%
  filter(mass < 1000)

# plot
ggplot(data = filtered_starwars,
       mapping = aes(x = mass, y = height)
       ) +
  geom_point() +
  geom_smooth()
```

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



Next, we'll change the defaults in order to fit a linear model. We do this in the `geom_smooth` function, `method = "lm"`. We can additionally specify a function, so we can change how we fit the data. Currently, our formula is `y ~ x`, which is a regular linear model. We could, however, fit a quadratic function to the data by using `y ~ poly(x, 2)`. The function `poly(x, 2)` calculates orthogonal polynomials to a certain degree. The first being linear, second quadratic (think curved lines with 1 inflection point), the third cubic (think curved lines with 2 inflection points), etc.. For now, we'll just fit the linear model.

```
ggplot(data = filtered_starwars,
       mapping = aes(x = mass, y = height)
       ) +
  geom_point() +
  geom_smooth(method = 'lm', formula = y ~ x)
```



Alternatively, if we have a fitted model saved from our analyses, we can add the fitted line directly using the `stat_summary` function. However, this requires some insight into the models fitted, so we'll save this for later classes.

Finally, we'll look at how we can combine these smoothed fits with averages of scores. This is often useful if you're looking at timecourse data and you want to summarise performance across a whole range of participants. We'll also look at adding a few of the graphical flourishes from above to make the plot look really nice.

Next, we'll look at making the point and line points look a little better. For this, we'll use the inbuilt `ChickWeight` data set from R. This is similar to the `chickwts` data set from the previous session, but crucially this data is not aggregaged across time. Instead, we have several measurements for the chicks at different time points. This makes this data especially nice for plotting changes over time in our sample.

First, load the data.

```
data(ChickWeight)
```

Then we can see how the data looks. This data is stored as a data.frame, so we have to be careful when printing it out. We'll use the `head()` function to get only the first 6 rows of data.

```
head(ChickWeight)
```

```
## Grouped Data: weight ~ Time | Chick
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
## 6     93   10     1    1
```

We have 4 columns:

- weight: out dependent variable, the weight of the chicks
- Time: an independent variable indicating the time point at which a measurement was taken
- Chick: an ID column for each chick
- Diet: an independent variable indicating the diet of the chick

Unfortuantely, we only know that there are different diets from the data set, but not what those diets are. We also don't know the units by which weight is measured or by which the measurements were taken.

This is a good lesson in using sensible variable names. If you want yourself and others to understand your data years down the line with little background knowledge of your study, then use informative labels for your data!

We'd expect that as time goes on, chick weights should increase. Also, we might expect that the rate of growth should differ for each diet. As such, we'll plot this together using our data set.

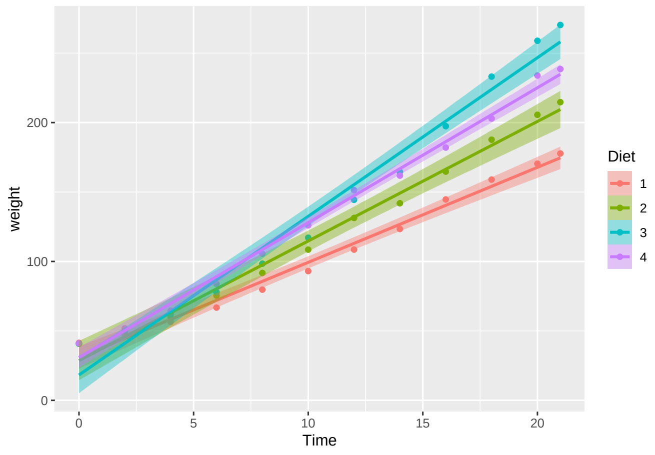
First, we'll plot our chick data, using the `stat_summary()` function to plot some mean points for the chick weights, and the `geom_smooth()` function to fit a linear model to the data. We'll make sure that the points, lines, and ribbons (indicating the confidence interval) are different colours for each level of the diet factor.

We're going to build up our plot bit by bit, so this time we can same our plot as an object, before returning the object to see the plot.

```
chick_time_plot <- ggplot(data = ChickWeight, # data
                          mapping = aes(x = Time, # x-axis
                                        y = weight, # y-axis
                                        colour = Diet, # colour ID
                                        fill = Diet # fill ID
                                        )
                          ) +
  stat_summary(fun.y = mean, geom = "point") + # point means
  geom_smooth(method = 'lm', formula = y ~ x) # linear fit
```

```
## Warning: `fun.y` is deprecated. Use `fun` instead.
```

```r
# return the plot
chick_time_plot
```

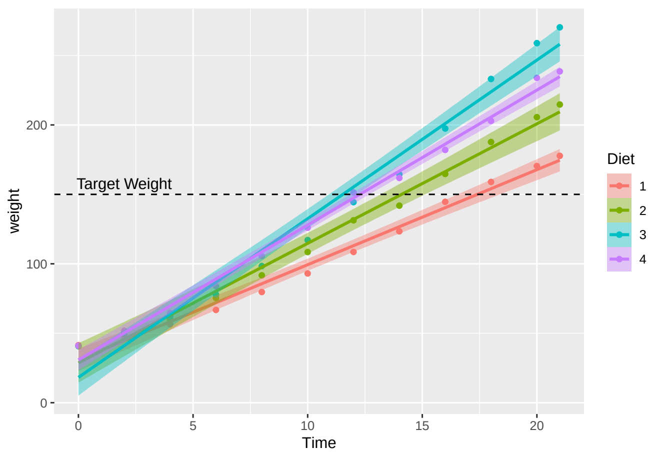


We can improve this plot. So, let's add some further information. What if we had a target weight we'd like our chickens to reach over time, and we want to see when each diet pushes chicks over this line?

We can add this information by including a horizontal line using `geom_hline()`. We just need to specify where on the y-axis this line should go (`yintercept`), and which style we want for the line (`linetype`). We should also add an annotation to the line indicating what it represents. We can do this by using the `annotate` function, and specifying that we want a text geom. We need to specify the x and y values for where the text lies, and what the text should say.

To add this to our original plot, we'll overwrite the original plot name with itself, plus our additional arguments.

```r
chick_time_plot <- chick_time_plot +
  geom_hline(yintercept = 150, linetype = 2) + # horizontal line
  annotate(geom = "text", x = 2, y = 158, label = "Target Weight") # line label

# return the plot
chick_time_plot
```



Finally, we can improve the general look of the plot by adding some additional arguments. We'll specify the labels for the x and y axes using the `labs()` function.

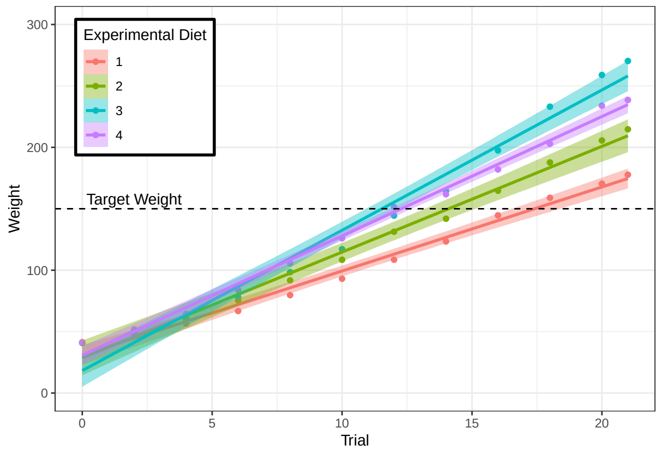
We'll also specify the limits of the y-axis using the `coord_cartesian()` function and specifying the minimum and maximum values for the limits of the y-axis.

Next, we'll specify a general basic theme for the plot using the `theme_bw()` argument. We'll change a minor detail of this theme, namely the position and style of the legend by specifying additional `theme()` arguments. This has to come after `theme_bw()` so `theme_bw()` doesn't overwrite our additional arguments. First, we simply state the x and y locations for the legend by defining the `legend.position`. Next, we want a black border around our legend, so we do this by changing the `legend.background`. Here, we set colour (the outline) to black, with a relatively small line size (1), and a solid linetype.

Finally, we can change the title of our legend if we'd like. The original title is quite informative, but it's useful to know how to do this. Here, we use the `guides` function, and specify both the guide legend title using `guide_legend()` for both the `colour` and `fill` properties. It's important to do this for both properties as our data is identified by colours on both of these properties. If we just change the title of one of the properties (e.g. fill), then we'll have two legends!

```r
chick_time_plot <- chick_time_plot +
  labs(x = "Trial", y = "Weight") + # axis labels
  coord_cartesian(ylim = c(0, 300)) + # range of y-axis
  theme_bw() + # general theme
  theme(legend.position = c(x = 0.15, y = 0.8), # position of legend
        legend.background = element_rect(color = "black",
                                         size = 1,
                                         linetype = "solid"
```

```
                                               ) # legend styling
  ) +
  guides(colour = guide_legend("Experimental Diet"),
         fill = guide_legend("Experimental Diet")
         ) # legend title

# return the plot
chick_time_plot
```



3.5 Combining Plots

Finally, while it's all well and good plotting one model at a time, often if you're writing a paper you want several plots together to save space. You could do this by hand in certain image editing packages, but luckily there's a package for this on CRAN; `cowplot`. First, we need to install and load the package. Intall this by running the commented out code below. After that, load `cowplot` each time you want to use it.

```
# install.packages("cowplot")
library(cowplot) # run this each time
```

In order to stitch two plots together, we need to save our plots as objects so they're available to the R environment. This is the same process we used above for the `ChickWeight` plots.

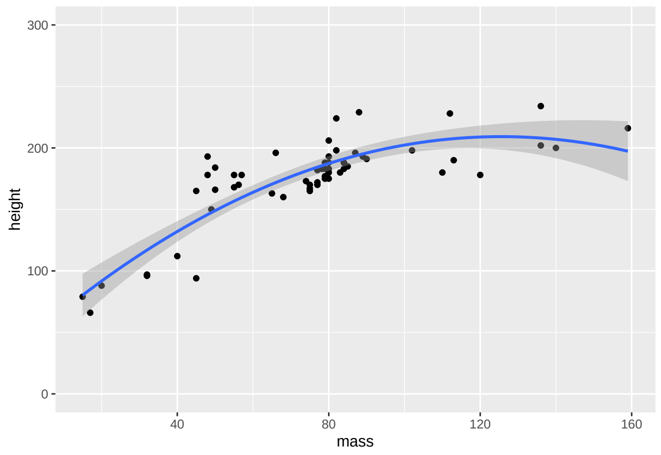
If we assign our plotting code to an object, then every time we run the name of the object, we'll get the output of that code back.

For keeping several plots in my environemnt at once, I often save them to an object. In this instance, we'll save a plot of just the points to the name `point_plot`. That's because we want to overlay a linear fit and a quadratic fit on this plot separately, but we don't want to type the full code out twice. Instead, we fit the plot of points to `point_plot`, then add a linear or quadratic fit by adding the `geom_smooth` argument to `point_plot` and saving both under new names. We can do this like so:

```
# create a plot of points
point_plot <- ggplot(data = filtered_starwars,
                     mapping = aes(x = mass, y = height)
                     ) +
  geom_point() +
  coord_cartesian(ylim = c(0, 300))

# create a linear plot by adding the fit to the plot of points
linear_plot <-  point_plot +
  geom_smooth(method = 'lm', formula = y ~ x)
```

When you do this, you won't automatically see the plot once you run the code unless you run the object assigned to the plot. Let's try this for a quadratic fit of the same data.

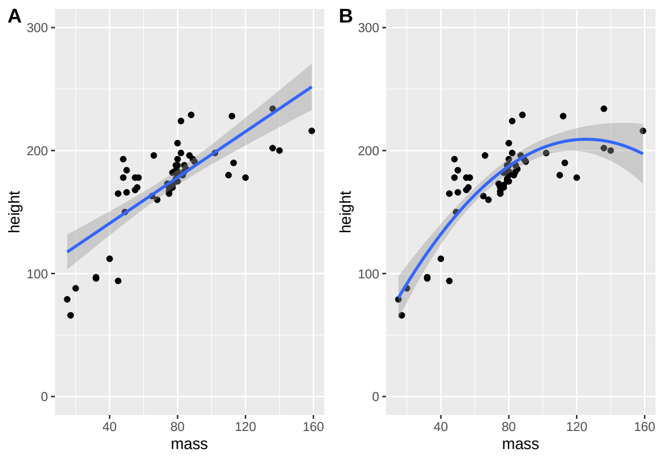
```
# create a quadratic plot by adding the fit to the plot of points
quadratic_plot <- point_plot +
  geom_smooth(method = 'lm', formula = y ~ poly(x, 2)) # fit quadratic

# return the plot
quadratic_plot
```



You can see that we've got the quadratic fit and the 95% confidence interval around this fit from the above code. Why does the plot look different to the base plots in ggplot2? `cowplot` loads some defaults for all plots outputted by `ggplot` to save you on typing out your own theme arguments.

Now, we can combine these two plots into a single plot using the new functionalities from `cowplot`.

```
combined_plots <- plot_grid(linear_plot,
                            quadratic_plot,
                            labels = c("A", "B"), # label plots
                            align = "h" # align axes
                            )
combined_plots
```

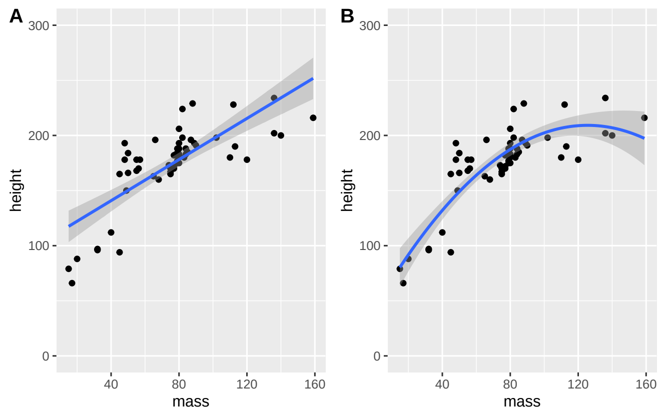


What if we want a title for our plots? We first have to define our title with a combination of `ggdraw()` and `draw_label()`. Inside `draw_label()` we used a new function, `paste()` to paste two strings of text together (with a space between the two strings). We could simply input the entire string, but we broke it down into two bits so we don't exceed the 80 character width limit from our style guide!

We further specify a bold `fontface` in the `draw_label()` command outside our pasted title.

Finally, we display the plot by creating a `plot_grid()` of our plot and the title in this order so the title displays underneath the plot. We specify that we just want 1 column so the plot and title are stacked together, and we specify the relative heights of the title and plot separately so that the title is smaller than the plot so it doesn't take up an equal amount of space as the plot.

```
# create title
title <- ggdraw() +
  draw_label(paste("Linear (A) and Quadratic",
                   "(B) fits of Height and Weight"
                   ),
             fontface = "bold"
             )

# print plot as a grid
combined_plots <- plot_grid(combined_plots,
                            title,
                            ncol = 1,
                            rel_heights = c(1, 0.1)
                            )

# return the combined plots
combined_plots
```



Linear (A) and Quadratic (B) fits of Height and Weight

You can find further information on adding joint titles, annotations, etc. in this article by Claus O. Wilke.

Note that you can reset the the default styles inherited by cowplot so that your plots return to their defaults with the following code:

```
theme_set(theme_gray())
```

3.6 Saving Plots

Finally, if you've went to the work of producing a nice plot, then it'll be useful to know how to save it somewhere. To save our combined plot of the linear and quadratic fits, we'll use `ggsave()`. You can name your plot anything you like, but remember to add a file extension at the end. I've used .png as this format suffers from fewer artifacts when comapred to JPEG, and it's a pretty common filetype.

```
ggsave(filename = "outputs/starwars_mass_by_height.png",
       plot = combined_plots
       )
```

You can do a whole bunch of other things with ggplot, like adding vertical and horizontal lines (often useful for identifying chance performance in participants), and annotating plots directly (useful for adding statistics, or commenting on interesting sections of the data). We'll cover these in the exercises, however, as these are just additional flavour to our plots!

3.7 Exercises

3.7.1 Main Exercises

If all this complexity hasn't given you nausea, try out the exercises below.

You may have to install the `languageR` library if you don't have it installed already. We're going to use the `lexdec` data set from this package. This contains lexical decision times (is this a word or not?) for a range of subjects with different native languages over different words. We also get information on the class of the words (animal or plant), and on the frequency of the words (i.e. how often they occur in most texts). Finally, our dependent variable is reaction time in the lexical decision task.

Execute the following code to load the data set, clean it, and create an aggregated version of the data set along with some fictional demographics.

```
# install.packages("languageR") # install if needed
library(languageR) # load the package containing the lexdec data

 # lexical decision times split by subject and item
lex_dec <- lexdec %>%
  select(Subject, NativeLanguage, Trial, Word, Class, Frequency, RT) %>%
  rename_all(tolower) %>%
  rename(native_language = nativelanguage, reaction_time = rt) %>%
  as.tibble()
```

```
## Warning: `as.tibble()` was deprecated in tibble 2.0.0.
## Please use `as_tibble()` instead.
## The signature and semantics have changed, see `?as_tibble`.
```

```
# aggregate by items to explore differences in items
lex_dec_items <- lex_dec %>%
  group_by(word, class, frequency) %>%
  summarise(mean_RT = mean(reaction_time),
            sd_RT = sd(reaction_time)
  )
```

```
## `summarise()` has grouped output by 'word', 'class'. You
## can override using the `.groups` argument.
```

```
# ensure same values for gender and age columns
set.seed(1892)

# create demographic information
# here, I've simulated gender and age data
lex_demog <- lex_dec %>%
  distinct(subject, native_language) %>%
  mutate(age = rnorm(n = nrow(.), mean = 35, sd = 7),
         gender = sample(c("male", "female"),
                      size = nrow(.),
                      replace = TRUE)
         )
```

Try to solve these questions before resorting to the solutions.

3.7.2 Question 1

Let's assume we want to know if the density of the mean reaction times differs across word class within our items.

Using the `lex_dec_items` data set, make a density plot of the word frequency by class. Save this plot as `lexical_density`, and output the plot.

3.7.3 Question 2

Add a black and white theme to your `lexical_density` plot. Next, give the axis labels and legend labels uppercase names. Finally, give your legend title the name Word Class. Assign all of this to a new object `lexical_density_improved`, and output your plot below.

Note: To change the legend labels, you need to use both `scale_fill_discrete` and `scale_colour_discrete`. Why do you think this is? Why don't we just use `scale_x_discrete` as we did in class?

3.7.4 Question 3

There's some repetition in the code for the plot above. Can you improve your code to remove that?

3.7.5 Question 4

Now we want to check the distribution of the reaction times depending on word class and the language spoken by our participants.

Using the `lex_dec` data set, create a faceted plot that looks at the density of reaction times. This should be made up of a grid of densities split by native language and word class.

Assign this to the object `rt_density` and output your plot.

3.7.6 Question 5

Now we want to explore if there's any relationship between the mean reaction time to our items and the frequency of the item.

Using the `lex_dec_items` data set, plot the relationship between word frequency and mean reaction time as a scatter plot. We want a fitted line and points for the mean reaction time.

3.7.7 Question 6

Now we want to know how many males and females took part in our experiment.

Using the `lex_demog` data set, create a count of the number of males and females who took part in the experiment. Make all text in the plot uppercase, and make the plot theme black and white.

Assign this to the object `gender_count` and output your plot.

3.7.8 Question 7

What if we want to know the mean age and distribution of ages split by language spoken and gender?

Using the `lex_demog` data set, create a pirate plot of the ages by each gender. You can set the colour and fill to gender to have more colourful plot if you'd prefer.

Additionally, we would like these plots split by the native language of the speaker, so facet your plot by native language.

Assign this all to the object `demographic_age`, and output your plot.

Can you see how a bar plot of the average ages might be misleading in this instance? Pay particular attention to the male bar for the other language group.

3.7.9 Question 8

Next, we want to see the mean (and standard error) of reaction times to words with different word frequencies.

Using the `lex_dec` data set, create a scatter plot of the reaction time by word frequency. We would like this split by word class.

Hint: Be sure to use the `stat_summary` function to get pointranges that represent the mean and standard error.

Assign this to the object `rt_pointrange`, and output your plot.

3.7.10 Question 9

Finally, we want to show a few of our graphs in one display.

Using the cowplot library, stitch together the plots for question 6, 7, 4, and 8.

Add the labels A-D to identify each plot.

Save these plots under the object `combined_plots` and return your combined plots.

3.7.11 Additional Exercise

We can improve the combined plots above.

Add a (short) simple title to your combined plots, and save this plot in the outputs folder.

These plots won't look perfect, and you may need to change the font and element sizes. That can be easily achieved, but we won't do that here.

Chapter 4

Data Manipulation 1

In this chapter we'll look at tidying up and cleaning our data with the **tidyr** package (Wickham and Henry, 2018) from the tidyverse (Wickham, 2017). We'll also use the **dplyr** package (Wickham et al., 2017a) from the tidyverse (Wickham, 2017) to join different sets of data together into a useful format for plotting and running analyses.

This data cleaning is an important first step on your way to working with your data, so it's important for you to get acquainted with the different ways your data can be formatted, and how to get it in the format we want.

This chapter, particularly the section on joins, was informed by the Glasgow University Research Cycle by Dale Barr and Lisa DeBruine. Please see this resource for further reading.

4.1 Getting Started

First, we'll load the packages necessary for this class. Nicely, tidyr and dplyr are part of the tidyverse family, so we don't need to load this separately to the other packages in our library.

```
library(tidyverse)
```

Next, we'll load some data from the `languageR` library. The data set we'll look at is the `lexdec` data set, which looks at lexical decision latencies for English nouns from native and non-native speakers of English. If you load this data set from the `languageR` library then it'll already be in the correct format for us to perform our analyses. So, I've made this data more messy; adding missing values, additional columns that represent more than one source of data, and few extra participants.

On top of this, I've produced a separate data set which stores some (fake) demographic data for the participants. This data set contains information about the participant ID, their gender, age, and any programming languages that they know. Why did I add this last column? Well, sometimes your data contains additional information that isn't important for your current analyses, so it's good to get some experience with filtering our data. (We'll cover filtering data in more detail in Lesson 5.)

4.2 Data Formats

I've saved the data in both **wide** and **long** formats.

In **wide** formats, each row represents one participant, and any information gathered from these participants is stored in a new column. So, let's say we have several items where we gather some data (e.g. reaction times), here each column will represent an item, and each cell will store a participant's score.

In **long** formats, each column represents one measurement. Here, we could have a column for participant ID, a column for item number, and a column for reaction times. In this instance, each row should be unique by its combination of our three columns, but IDs (e.g. participant ID, item ID) will be repeated several times.

4.2.1 Loading Data

Let's load the messy data sets from the csv files in the Lesson 4 **lesson_materials** folder to get a better grasp of these formats. We'll get some (scary looking, red) messages when we load the data. That's because we haven't specified how we want each column to be parsed.

```r
demo_wide <- read_csv("inputs/lexical_decision_demographic_data_wide.csv")
demo_long <- read_csv("inputs/lexical_decision_demographic_data_long.csv")
```

When we use `read_csv()` this function tries to guess the data type contained within any column. Basically, if all the data in a column are numbers, then it'll be parsed as a numeric data type. If even one cell in a column is text, the whole column will be parsed as text. This can cause problems if most of the data are numbers and you want to do some calculations with this column, as you can't add, subract, or divide with text columns!

Often, dates are parsed as datetimes, which allows for some easy calculations for differences in times. But, our completion_time column was parsed as a character. Why? Because this column contains two datetimes separated by an underscore. It looks like the researchers (me) were too lazy to actually calculate the completion times, and just threw the start and end times togerther into one column!

4.2.2 Wide and Long Data

Now we've loaded the data, and understand how R reads data, let's look back at wide and long data formats.

4.2.2.1 Wide Data

In the wide data format, each row is a participant (ID). We have columns representing all of the programming languages reported to be known by the participants. If a participant knows the language, they get a 1 in this column, otherwise they have an NA. We also have several other columns covering other information gathered.

```
demo_wide
```

```
## # A tibble: 29 x 14
##    ID    `C++` FORTRAN JavaScr~1 Python     R  Ruby LANGU~2 progr~3 gender   age
##    <chr> <dbl>   <dbl>     <dbl>  <dbl> <dbl> <dbl> <chr>   <chr>   <chr>  <dbl>
##  1 22        1       1         1     NA    NA    NA OTHER   FINISH  female    30
##  2 23        1      NA        NA     NA    NA    NA <NA>    no      male      30
##  3 24       NA       1        NA     NA    NA    NA ENGLISH END     female    30
##  4 25        1       1        NA     NA    NA    NA <NA>    ethics  male      18
##  5 26       NA      NA        NA      1    NA     1 english ethics  male      31
##  6 27       NA       1         1     NA    NA    NA ENGLISH END     female    44
##  7 28       NA      NA        NA     NA     1     1 <NA>    ethics  male      23
##  8 29       NA      NA         1     NA    NA    NA english ethics  male      34
##  9 30       NA       1        NA     NA    NA    NA <NA>    <NA>    female    32
## 10 A2        1       1         1     NA    NA    NA English END     female    33
## # ... with 19 more rows, 3 more variables: tester <chr>, funRec <chr>,
## #   completion_time <chr>, and abbreviated variable names 1: JavaScript,
## #   2: LANGUAGE, 3: progress
```

4.2.2.2 Long Data

With the long data format, we have each ID for our data in one column, and the measurements for these variables in each cell. The main difference here is that we have a column at the end called computer_language which simply lists the language each participant knows. This cuts down on the need for redundant columns for a computer language when a participant doesn't know that language. Compare the languages known for participant 22 in this data set and how it's represented in the wide data set.

In order to display this properly, I'll cut out the columns tester, funRec, and completion time, when printing for this website, but you needn't do this in R.

```
demo_long
```

```
## # A tibble: 54 x 6
##    ID    LANGUAGE progress gender   age computer_language
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>
##  1 22    OTHER    FINISH   female    30 C++
##  2 22    OTHER    FINISH   female    30 FORTRAN
##  3 22    OTHER    FINISH   female    30 JavaScript
##  4 23    <NA>     no       male      30 C++
##  5 24    ENGLISH  END      female    30 FORTRAN
##  6 25    <NA>     ethics   male      18 C++
##  7 25    <NA>     ethics   male      18 FORTRAN
##  8 26    english  ethics   male      31 Python
##  9 26    english  ethics   male      31 Ruby
## 10 27    ENGLISH  END      female    44 FORTRAN
## # ... with 44 more rows
```

We end up with some repetition here (several rows with the same ID, language, gender, etc.), but each row is unique when we consider all measurements. This is a common data format for raw data, as we'll see next.

4.2.2.3 Understanding our Loaded Data

The NA value is important in R; as Hadley Wickham says, it is the evidence of absence. Missing values however are more problematic in that they are the absence of evidence. If you want to indicate missing data, use NA, and not N/A or N_A etc. as you have to tell R to parse these as NAs.

We also have columns indicating the language known by the participant, their progress in the experiment (i.e. did they finish it or not?), their gender, age, who tested them and two final columns. The funRec column tells us whether they liked the experiment or not (on a 0-7 scale) and whether they'd recommend the experiment to others (yes/no). Unfortunately, these values are separated by a dash. This is bad practice as each cell should represent one data point, not two. The same can be said for completion time, with the dates and times for starting and ending the experiment separated by an underscore.

4.3 Reformatting Data

4.3.1 Gathering Data

Let's say we want to perform some operations to change how our data looks. What if we want to turn our data from a wide format into long format? We might do this if we want to make a bar plot which counts how many people know each programming language.

To gather data that is spread across several columns, we use the `gather()` function.

In this function, we have to specify a few things. As always, with our tidyverse functions we need to tell R which data set on which to perform the function.

We have to say what we will call our new column which contains the headings of the columns we want to gather. Here, we call it `prog_lang`, and it will contain the column names for each programming language. (Normally, I'd use a more readable name, but I want to show as many columns for this data on the website.)

Next, we need to specify a value, which will contain the numbers from the programming language columns we've gathered together. This will essentially tell us whether or not people know that language or not.

Finally, we need to give the function the columns to gather together. We can do this by name, or by number. Since our programming language columns are all together, from column number 2 to 7, we can just specify the range of `2:7`.

```
gather(data = demo_wide,
       key = prog_lang,
       value = known,
       2:7
       )
```

```
## # A tibble: 174 x 8
##    ID    LANGUAGE progress gender   age tester prog_lang known
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr>     <dbl>
##  1 22    OTHER    FINISH   female    30 GW     C++           1
##  2 23    <NA>     no       male      30 GW     C++           1
##  3 24    ENGLISH  END      female    30 GW     C++          NA
##  4 25    <NA>     ethics   male      18 GW     C++           1
##  5 26    english  ethics   male      31 GW     C++          NA
##  6 27    ENGLISH  END      female    44 GW     C++          NA
##  7 28    <NA>     ethics   male      23 GW     C++          NA
##  8 29    english  ethics   male      34 GW     C++          NA
##  9 30    <NA>     <NA>     female    32 GW     C++          NA
## 10 A2    English  END      female    33 RHB    C++              1
## # ... with 164 more rows
```

That worked nicely, but it seems that it's formatted the data so it goes through each programming language alphabetically first, so our IDs are spread all over the data set.

To fix this, we can use another function, called `arrange`, which takes a data argument and a column by which to arrange the data.

4.3.1.1 The Pipe

At this point, I'll introduce you to a new way of writing our commands which is better when we want to apply several functions. Instead of nesting it all together, we can write our commands from left to right, like how we read English text. Here, we can use the pipe `%>%` at the end of a line, which can be read as "and then do…". Below, we simply give our data.frame, demo_wide, and use the pipe to apply the gather function. This is the same code as above, just represented in a different way. Read this like, "take our data, and then, gather the columns together."

```
demo_wide %>%
  gather(
    key = prog_lang,
    value = known,
    2:7
    )
```

```
## # A tibble: 174 x 8
##    ID    LANGUAGE progress gender   age tester prog_lang known
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr>     <dbl>
##  1 22    OTHER    FINISH   female    30 GW     C++           1
##  2 23    <NA>     no       male      30 GW     C++           1
##  3 24    ENGLISH  END      female    30 GW     C++          NA
##  4 25    <NA>     ethics   male      18 GW     C++           1
##  5 26    english  ethics   male      31 GW     C++          NA
##  6 27    ENGLISH  END      female    44 GW     C++          NA
##  7 28    <NA>     ethics   male      23 GW     C++          NA
##  8 29    english  ethics   male      34 GW     C++          NA
##  9 30    <NA>     <NA>     female    32 GW     C++          NA
## 10 A2    English  END      female    33 RHB    C++           1
## # ... with 164 more rows
```

Still, we're left with the same grouping problem, so we can apply another function, `arrange()` at the end to arrange the data by ID.

```
demo_wide %>%
  gather(
    key = prog_lang,
    value = known,
    2:7
    ) %>%
  arrange(ID)
```

```
## # A tibble: 174 x 8
##    ID    LANGUAGE progress gender   age tester prog_lang  known
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr>      <dbl>
##  1 22    OTHER    FINISH   female    30 GW     C++            1
##  2 22    OTHER    FINISH   female    30 GW     FORTRAN        1
##  3 22    OTHER    FINISH   female    30 GW     JavaScript     1
##  4 22    OTHER    FINISH   female    30 GW     Python        NA
##  5 22    OTHER    FINISH   female    30 GW     R             NA
##  6 22    OTHER    FINISH   female    30 GW     Ruby          NA
##  7 23    <NA>     no       male      30 GW     C++            1
##  8 23    <NA>     no       male      30 GW     FORTRAN       NA
##  9 23    <NA>     no       male      30 GW     JavaScript    NA
## 10 23    <NA>     no       male      30 GW     Python        NA
## # ... with 164 more rows
```

This is now in a better format. However, we have a lot of rows with NA in the language, where people don't know that language. This is redundant information. Additionally, the known column is now redundant if we remove the languages people don't know, so we can remove this column too.

We'll save this data under the name demo_gathered for comparison with the long formatted data set we already loaded.

```
demo_gathered <- demo_wide %>%
  gather(
    key = prog_lang,
    value = known,
    2:7,
    na.rm = TRUE
    ) %>%
  arrange(ID) %>%
  select(-known)
```

```
demo_gathered
```

```
## # A tibble: 54 x 7
##    ID    LANGUAGE progress gender   age tester prog_lang
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr>
##  1 22    OTHER    FINISH   female    30 GW     C++
##  2 22    OTHER    FINISH   female    30 GW     FORTRAN
##  3 22    OTHER    FINISH   female    30 GW     JavaScript
##  4 23    <NA>     no       male      30 GW     C++
##  5 24    ENGLISH  END      female    30 GW     FORTRAN
##  6 25    <NA>     ethics   male      18 GW     C++
##  7 25    <NA>     ethics   male      18 GW     FORTRAN
##  8 26    english  ethics   male      31 GW     Python
##  9 26    english  ethics   male      31 GW     Ruby
## 10 27    ENGLISH  END      female    44 GW     FORTRAN
## # ... with 44 more rows
```

We used the argument, `na.rm = TRUE` to remove any rows in our value column, known, with an NA. Since we did that, the known column contains all 1s, as the only languages left are the ones people know. So, we used the `select` function from **dplyr** to remove the known column. This function is used to select the column you want to keep in your data set. If you provide a column name with the `-` prefix, this tells R to keep everything except that column; so we drop it from our data set!

4.3.2 Separating Columns

In our wide formatted data, we have two columns which store two data points in each cell: funRec has information on whether people found the experiment fun, and whether they'd recommend it to others. Let's split this into separate columns. We just need to supply the name of the column to separate `col`, and what you want it split into, as a list of the names the columns should take, `into`.

As in previous examples, I'll remove the middle rows from the data here, but feel free to print them all yourself in R. I've presented the code to do this below.

Remember, if we want to supply multiple names, we need to concatenate (`c`) these names together.

```
demo_gathered %>%
  separate(
    col = funRec,
    into = c("fun", "recommend")
    )
```

```
## # A tibble: 54 x 5
##    ID    tester fun   recommend completion_time
##    <chr> <chr>  <chr> <chr>     <chr>
##  1 22    GW     7     no        2018-03-22 23:06:11_2018-03-23 00:25:51
##  2 22    GW     7     no        2018-03-22 23:06:11_2018-03-23 00:25:51
##  3 22    GW     7     no        2018-03-22 23:06:11_2018-03-23 00:25:51
##  4 23    GW     6     yes       2018-03-26 00:30:20_2018-03-26 02:15:52
##  5 24    GW     4     yes       2018-03-21 11:09:38_2018-03-21 12:16:28
##  6 25    GW     0     no        2018-03-25 13:03:58_2018-03-25 14:45:25
##  7 25    GW     0     no        2018-03-25 13:03:58_2018-03-25 14:45:25
##  8 26    GW     4     no        2018-03-24 06:46:30_2018-03-24 08:17:29
##  9 26    GW     4     no        2018-03-24 06:46:30_2018-03-24 08:17:29
## 10 27    GW     5     no        2018-03-21 03:23:57_2018-03-21 04:28:01
## # ... with 44 more rows
```

Take a look at the two new columns. They are both parsed as characters, even though the fun column only contains numbers. We can ask R to convert the data types for the split column during in the separate function using `convert = TRUE`.

```
demo_gathered %>%
  separate(
    col = funRec,
    into = c("fun", "recommend"),
    convert = TRUE
    )
```

```
## # A tibble: 54 x 5
##    ID    tester   fun recommend completion_time
##    <chr> <chr>  <int> <chr>     <chr>
##  1 22    GW         7 no        2018-03-22 23:06:11_2018-03-23 00:25:51
##  2 22    GW         7 no        2018-03-22 23:06:11_2018-03-23 00:25:51
##  3 22    GW         7 no        2018-03-22 23:06:11_2018-03-23 00:25:51
##  4 23    GW         6 yes       2018-03-26 00:30:20_2018-03-26 02:15:52
##  5 24    GW         4 yes       2018-03-21 11:09:38_2018-03-21 12:16:28
##  6 25    GW         0 no        2018-03-25 13:03:58_2018-03-25 14:45:25
##  7 25    GW         0 no        2018-03-25 13:03:58_2018-03-25 14:45:25
##  8 26    GW         4 no        2018-03-24 06:46:30_2018-03-24 08:17:29
##  9 26    GW         4 no        2018-03-24 06:46:30_2018-03-24 08:17:29
## 10 27    GW         5 no        2018-03-21 03:23:57_2018-03-21 04:28:01
## # ... with 44 more rows
```

That looks much better!

`separate()` is smart enough to know how to separate values if they are split by special characters. Before, we had an underscore in the funRec column, so it split the data by that. If this fails, you can directly specify how the values are separated using the `sep` argument.

We also wanted to split the completion_time column. It looks like the first value is the start time, and the second is the end time. So lets separate these together with the funRec column.

```
demo_gathered %>%
  separate(
    col = funRec,
    into = c("fun", "recommend"),
    convert = TRUE
    ) %>%
  separate(
    col = completion_time,
    into = c("start_time", "end_time")
    )
```

```
## Warning: Expected 2 pieces. Additional pieces discarded in 54 rows [1, 2, 3, 4,
## 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

```
## # A tibble: 54 x 6
##    ID    tester   fun recommend start_time end_time
##    <chr> <chr>  <int> <chr>     <chr>      <chr>
##  1 22    GW         7 no        2018       03
##  2 22    GW         7 no        2018       03
##  3 22    GW         7 no        2018       03
##  4 23    GW         6 yes       2018       03
##  5 24    GW         4 yes       2018       03
##  6 25    GW         0 no        2018       03
##  7 25    GW         0 no        2018       03
##  8 26    GW         4 no        2018       03
##  9 26    GW         4 no        2018       03
## 10 27    GW         5 no        2018       03
## # ... with 44 more rows
```

Oops, it looks like separate struggled to split our completion_time column correctly. That's because it wants to split at every dash, whitespace, colon, and underscore; pretty much the whole completion_time column! Let's be more specific and tell `separate()` to just split the columns at the underscore.

Let's also overwrite our demo_wide data (assign the new data to the old data name) to use this new format in the next section.

```
demo_gathered <- demo_gathered %>%
  separate(
    col = funRec,
    into = c("fun", "recommend"),
    convert = TRUE
    ) %>%
  separate(
    col = completion_time,
    into = c("start_time", "end_time"),
    sep = "_"
    )
```

```
# see the data
demo_gathered
```

```
## # A tibble: 54 x 6
##    ID    tester   fun recommend start_time          end_time
##    <chr> <chr>  <int> <chr>     <chr>               <chr>
##  1 22    GW         7 no        2018-03-22 23:06:11 2018-03-23 00:25:51
##  2 22    GW         7 no        2018-03-22 23:06:11 2018-03-23 00:25:51
##  3 22    GW         7 no        2018-03-22 23:06:11 2018-03-23 00:25:51
##  4 23    GW         6 yes       2018-03-26 00:30:20 2018-03-26 02:15:52
##  5 24    GW         4 yes       2018-03-21 11:09:38 2018-03-21 12:16:28
##  6 25    GW         0 no        2018-03-25 13:03:58 2018-03-25 14:45:25
##  7 25    GW         0 no        2018-03-25 13:03:58 2018-03-25 14:45:25
##  8 26    GW         4 no        2018-03-24 06:46:30 2018-03-24 08:17:29
##  9 26    GW         4 no        2018-03-24 06:46:30 2018-03-24 08:17:29
## 10 27    GW         5 no        2018-03-21 03:23:57 2018-03-21 04:28:01
## # ... with 44 more rows
```

That looks a lot better! Notice that we didn't try to convert the start_time and end_time columns as this data type doesn't play nicely with separte. We'll look into how to convert between data types in Lesson 5.

Note: If every row doesn't produce the same number of columns, you can control what happens here with the `extra` argument. We won't cover this, but it's useful to know if you get into problems with `separate()` because of this issue.

Let's compare our gathered data to the long formatted data we already loaded. It's exactly the same, only we separated the two problematic columns – funRec and completion_time – and we have a different label for the programming languages known (prog_lang vs. computer_language).

```
head(demo_gathered)
head(demo_long)
```

```
## # A tibble: 6 x 11
##   ID    LANGUAGE progress gender   age tester   fun recommend start_time end_t~1
##   <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <int> <chr>     <chr>      <chr>
## 1 22    OTHER    FINISH   female    30 GW         7 no        2018-03-2~ 2018-0~
## 2 22    OTHER    FINISH   female    30 GW         7 no        2018-03-2~ 2018-0~
## 3 22    OTHER    FINISH   female    30 GW         7 no        2018-03-2~ 2018-0~
## 4 23    <NA>     no       male      30 GW         6 yes       2018-03-2~ 2018-0~
## 5 24    ENGLISH  END      female    30 GW         4 yes       2018-03-2~ 2018-0~
## 6 25    <NA>     ethics   male      18 GW         0 no        2018-03-2~ 2018-0~
## # ... with 1 more variable: prog_lang <chr>, and abbreviated variable name
## #   1: end_time
```

```
## # A tibble: 6 x 9
##   ID    LANGUAGE progress gender   age tester funRec completion_time     compu~1
##   <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr>  <chr>               <chr>
## 1 22    OTHER    FINISH   female    30 GW     7-no   2018-03-22 23:06:1~ C++
## 2 22    OTHER    FINISH   female    30 GW     7-no   2018-03-22 23:06:1~ FORTRAN
## 3 22    OTHER    FINISH   female    30 GW     7-no   2018-03-22 23:06:1~ JavaSc~
## 4 23    <NA>     no       male      30 GW     6-yes  2018-03-26 00:30:2~ C++
## 5 24    ENGLISH  END      female    30 GW     4-yes  2018-03-21 11:09:3~ FORTRAN
## 6 25    <NA>     ethics   male      18 GW     0-no   2018-03-25 13:03:5~ C++
## # ... with abbreviated variable name 1: computer_language
```

4.3.3 Spreading Data

What if we want to go from long format to wide format? This can be useful if we want to do a paired-samples *t*-test, where we might want the first scores in one column, and the second scores in another. (We'll cover *t*-tests in Lesson 6.)

To make our long data wide, we use the `spread()` function from tidyr.

To spread our data we need a `key`, the column containing the values we'd like to make column headers. We then also need a value, indicating the column containing the scores associated with the values. Often, this would be conditions in an experiment and test results. Our problem here is a little more complex. Remember that we dropped the redundant column telling us whether or not people knew a programming language? Well, we need this back so we have a `value` column to work from. We'll **mutate** our data to create this column. To do this, we use the `mutate()` function from **dplyr**. (We'll look at this process in detail in Lesson 5.) Here, we just set everything in our new known column to 1 as we know if a language is present in a participant's row, then they know it!

```
demo_gathered %>%
  mutate(known = 1) %>%
  spread(key = prog_lang, value = known)
```

```
## # A tibble: 29 x 11
##    ID      fun recomm~1 start~2 end_t~3 `C++` FORTRAN JavaS~4 Python     R  Ruby
##    <chr> <int> <chr>    <chr>   <chr>   <dbl>   <dbl>   <dbl>  <dbl> <dbl> <dbl>
##  1 22        7 no       2018-0~ 2018-0~     1       1       1     NA    NA    NA
##  2 23        6 yes      2018-0~ 2018-0~     1      NA      NA     NA    NA    NA
##  3 24        4 yes      2018-0~ 2018-0~    NA       1      NA     NA    NA    NA
##  4 25        0 no       2018-0~ 2018-0~     1       1      NA     NA    NA    NA
##  5 26        4 no       2018-0~ 2018-0~    NA      NA      NA      1    NA     1
##  6 27        5 no       2018-0~ 2018-0~    NA       1       1     NA    NA    NA
##  7 28        0 no       2018-0~ 2018-0~    NA      NA      NA     NA     1     1
##  8 29        6 no       2018-0~ 2018-0~    NA      NA       1     NA    NA    NA
##  9 30        1 yes      2018-0~ 2018-0~    NA       1      NA     NA    NA    NA
## 10 A2        4 yes      2018-0~ 2018-0~     1       1       1     NA    NA    NA
## # ... with 19 more rows, and abbreviated variable names 1: recommend,
## #   2: start_time, 3: end_time, 4: JavaScript
```

Great, that looks exactly the same as our demo_wide data, only with our nicely split columns.

Note: If you have your values spread across several columns, spread will spread by every unique value, so be sure to collapse your values into one column before you do this. You can do this using the `unite()` function, or pasting values together with `mutate()`, but we'll cover this more in Lesson 5.

4.4 Joins

Finally, we'll look at combining data together from separate tables. This is a common problem when we store demographic information in one data set, and test scores in another. Let's say we're interested in differences in performance by age. To do this, we somehow need to join together the demographic information of age with the correct participant ID in the test data.

Let's load some raw data for the lexical decision times to see how we might join data together from separate data sets.

```
lexdec_data <- read_csv("inputs/lexical_decision_raw_data.csv")
```

To make these examples easier to digest, we'll simply look at a single trial for an individual participant. Again, to do this we'll use some subsetting techniques that we'll go into in more detail in Lesson 5.

```
# keep only trials (rows) where the word is ant
lexdec_subset <- lexdec_data %>% filter(word == "ant")
```

How does the data look? We have 26 recorded entries. For two participants, they have missing values (NA) for these trials, indicating that they didn't complete this trial, or the trial wasn't recorded.

```
lexdec_subset
```

```
## # A tibble: 25 x 9
##    subject trial native_language word  class  frequency length correct    RT
##    <chr>   <dbl> <chr>           <chr> <chr>      <dbl>  <dbl> <chr>   <dbl>
##  1 A1        157 English         ant   animal      5.35      3 correct  876.
##  2 A3         41 Other           ant   animal      5.35      3 correct  607.
##  3 C          86 English         ant   animal      5.35      3 correct  725.
##  4 D         138 Other           ant   animal      5.35      3 correct  628.
##  5 I         105 Other           ant   animal      5.35      3 correct  560.
##  6 J          43 Other           ant   animal      5.35      3 correct  516.
##  7 K         183 English         ant   animal      5.35      3 correct  442.
##  8 M1        171 English         ant   animal      5.35      3 correct  427.
##  9 M2        117 Other           ant   animal      5.35      3 correct  530.
## 10 P         115 Other           ant   animal      5.35      3 correct  533.
## # ... with 15 more rows
```

This data set contains all the information for our trials, including the data to identify each trial, and what the score was on our dependent variables (correct/incorrect response, reaction time). Here we have the data in a nice format where each column represents a variable, and each cell represents a value for that variable.

Let's look at our demographic data set.

As you can tell, our our data is in a long format. Additionally, it looks like we don't have any record of the language spoken by subject 23. On top of this, we have the cryptically named "no" entry in the progress column. I'm guessing this means that they decided to withdraw from the experiment. This means that we have more information on participants than we will have in the lexdec_subset dataset. The implications of this will become apparent as we try out different joining operations.

```
demo_gathered
```

```
## # A tibble: 54 x 11
##    ID    LANGUAGE progress gender   age tester   fun recommend start_t~1 end_t~2
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <int> <chr>     <chr>     <chr>
##  1 22    OTHER    FINISH   female    30 GW         7 no        2018-03-~ 2018-0~
##  2 22    OTHER    FINISH   female    30 GW         7 no        2018-03-~ 2018-0~
##  3 22    OTHER    FINISH   female    30 GW         7 no        2018-03-~ 2018-0~
##  4 23    <NA>     no       male      30 GW         6 yes       2018-03-~ 2018-0~
##  5 24    ENGLISH  END      female    30 GW         4 yes       2018-03-~ 2018-0~
##  6 25    <NA>     ethics   male      18 GW         0 no        2018-03-~ 2018-0~
##  7 25    <NA>     ethics   male      18 GW         0 no        2018-03-~ 2018-0~
##  8 26    english  ethics   male      31 GW         4 no        2018-03-~ 2018-0~
##  9 26    english  ethics   male      31 GW         4 no        2018-03-~ 2018-0~
## 10 27    ENGLISH  END      female    44 GW         5 no        2018-03-~ 2018-0~
## # ... with 44 more rows, 1 more variable: prog_lang <chr>, and abbreviated
## #   variable names 1: start_time, 2: end_time
```

4.4.1 Mutating Joins

There are a number of joining operations we can do that will mutate (change the look of) our data:

- `left_join(data_one, data_two)`: Keeps everything in data_one and adds everything present in both data_one and data_two
- `right_join(data_one, data_two)`: Keeps everything in data_two and adds everything present in both data_two and data_one
- `inner_join(data_one, data_two)`: Keeps everything present in both data sets.
- `full_join(data_one, data_two)`: Keeps everything from both data sets. Adds NAs if information is present in only one data set.

Don't worry about the number of different joins here, they all take a similar form, but just do slightly different things to your data.

All of these joins take a `by` argument, which asks you which columns by which you want to combine the data. If we want to make sure we match up the data, we have to make sure our columns have the same headings across the two data sets.

Take a look at the two data sets above, it looks like we identify subjects with **subject** in the lexdec_data data set, and by **ID** in the demo_gathered data set. We also have the identifier for the language spoken as **native_language** in the lexdec_data data set, and as **LANGUAGE** in the demo_gathered data set.

We can use `rename` from **dplr** to rename our columns. Here we just supply the new name and the old name. The names in demo_gathered are messy, so we'll change those to match the lexdec_data names.

```
demo_gathered <- rename(demo_gathered,
                        subject = ID,
                        native_language = LANGUAGE
                        )
demo_gathered
```

```
## # A tibble: 54 x 11
##    subject native_la~1 progr~2 gender   age tester   fun recom~3 start~4 end_t~5
##    <chr>   <chr>       <chr>   <chr>  <dbl> <chr>  <int> <chr>   <chr>   <chr>
##  1 22      OTHER       FINISH  female    30 GW         7 no      2018-0~ 2018-0~
##  2 22      OTHER       FINISH  female    30 GW         7 no      2018-0~ 2018-0~
##  3 22      OTHER       FINISH  female    30 GW         7 no      2018-0~ 2018-0~
##  4 23      <NA>        no      male      30 GW         6 yes     2018-0~ 2018-0~
##  5 24      ENGLISH     END     female    30 GW         4 yes     2018-0~ 2018-0~
##  6 25      <NA>        ethics  male      18 GW         0 no      2018-0~ 2018-0~
##  7 25      <NA>        ethics  male      18 GW         0 no      2018-0~ 2018-0~
##  8 26      english     ethics  male      31 GW         4 no      2018-0~ 2018-0~
##  9 26      english     ethics  male      31 GW         4 no      2018-0~ 2018-0~
## 10 27      ENGLISH     END     female    44 GW         5 no      2018-0~ 2018-0~
## # ... with 44 more rows, 1 more variable: prog_lang <chr>, and abbreviated
## #   variable names 1: native_language, 2: progress, 3: recommend,
## #   4: start_time, 5: end_time
```

Note: We could alternatively set `by` to `by = c("subject" = "ID", "native_language" = "LANGUAGE")` to join by variables with different names across the data sets, but I find it's good practice to be consistent with your naming.

4.4.1.1 Full Join

Now we can join the data sets together. We'll do a `full_join()` first, just to see what happens.

```
full_join(lexdec_subset, demo_gathered, by = c("subject", "native_language"))
```

```
## # A tibble: 55 x 18
##    subject trial nativ~1 word  class frequ~2 length correct    RT progr~3 gender
##    <chr>   <dbl> <chr>   <chr> <chr>   <dbl>  <dbl> <chr>   <dbl> <chr>   <chr>
##  1 A1        157 English ant   anim~    5.35      3 correct  876. <NA>    <NA>
##  2 A3         41 Other   ant   anim~    5.35      3 correct  607. END     male
##  3 C          86 English ant   anim~    5.35      3 correct  725. END     female
##  4 C          86 English ant   anim~    5.35      3 correct  725. END     female
##  5 D         138 Other   ant   anim~    5.35      3 correct  628. END     non-b~
##  6 D         138 Other   ant   anim~    5.35      3 correct  628. END     non-b~
##  7 I         105 Other   ant   anim~    5.35      3 correct  560. END     male
##  8 J          43 Other   ant   anim~    5.35      3 correct  516. END     female
##  9 J          43 Other   ant   anim~    5.35      3 correct  516. END     female
## 10 J          43 Other   ant   anim~    5.35      3 correct  516. END     female
## # ... with 45 more rows, 7 more variables: age <dbl>, tester <chr>, fun <int>,
## #   recommend <chr>, start_time <chr>, end_time <chr>, prog_lang <chr>, and
## #   abbreviated variable names 1: native_language, 2: frequency, 3: progress
```

We've successfully merged the two data sets, but we now have multiple rows for our responses because we kept the programming language column. This is problematic if we want to calculate any statistics directly on this data frame, as we'll end up with what seems like multiple observations for a single trial.

We have a couple of workarounds for this problem:

1. Merge with the demographic data in a wide format, in which case we'll have multiple columns each representing a different programming language.
2. Merge with the demographic data set in a long format, but exclude the prog_lang column and filter the leftover duplicate rows prior to merging.

For now, we'll stick with 1 as it required fewer steps. But first, we want to transform our nicely tidied demographic data set into a wide format. Just reuse the code from the spreading section to do this:

```
tidy_demo_wide <- demo_gathered %>%
  mutate(known = 1) %>% # create a value column
  spread(key = prog_lang, value = known) # data to wide format

# see the output
tidy_demo_wide
```

```
## # A tibble: 29 x 16
##    subject native_la~1 progr~2 gender   age tester   fun recom~3 start~4 end_t~5
##    <chr>   <chr>       <chr>   <chr>  <dbl> <chr>  <int> <chr>   <chr>   <chr>
##  1 22      OTHER       FINISH  female    30 GW         7 no      2018-0~ 2018-0~
##  2 23      <NA>        no      male      30 GW         6 yes     2018-0~ 2018-0~
##  3 24      ENGLISH     END     female    30 GW         4 yes     2018-0~ 2018-0~
##  4 25      <NA>        ethics  male      18 GW         0 no      2018-0~ 2018-0~
##  5 26      english     ethics  male      31 GW         4 no      2018-0~ 2018-0~
##  6 27      ENGLISH     END     female    44 GW         5 no      2018-0~ 2018-0~
##  7 28      <NA>        ethics  male      23 GW         0 no      2018-0~ 2018-0~
##  8 29      english     ethics  male      34 GW         6 no      2018-0~ 2018-0~
##  9 30      <NA>        <NA>    female    32 GW         1 yes     2018-0~ 2018-0~
## 10 A2      English     END     female    33 RHB        4 yes     2018-0~ 2018-0~
## # ... with 19 more rows, 6 more variables: `C++` <dbl>, FORTRAN <dbl>,
## #   JavaScript <dbl>, Python <dbl>, R <dbl>, Ruby <dbl>, and abbreviated
## #   variable names 1: native_language, 2: progress, 3: recommend,
## #   4: start_time, 5: end_time
```

Now, if we try the full join, we'll merge together the two data sets so we have all of the information in one place! We want to match the data sets by the subject

ID and the native language spoken by the participants, as these two columns appear in both data sets.

As before, we have so much data that I'll subset things so we can see the relevant information, but be sure to print the whole output in R yourself.

```
full_join(lexdec_subset, tidy_demo_wide, by = c("subject", "native_language"))
```

```
## # A tibble: 5 x 6
##   subject trial native_language progress correct    RT
##   <chr>   <dbl> <chr>           <chr>    <chr>   <dbl>
## 1 A1        157 English         <NA>     correct  876.
## 2 23         41 <NA>            no       correct  602.
## 3 24         43 ENGLISH         END      correct  541.
## 4 28         NA <NA>            ethics   <NA>      NA
## 5 A2         NA English         END      <NA>      NA
```

As you can see, we now have 1 row for each subject.

In cases where we don't have data on a subject, we simply have NAs in those cells.

- Look at subject 23, you can see that we don't have data on their native language, but we have their trial information. This means they are in both data sets, but they have missing data in both cases.
- Look at subject A2, they have missing data for their trial information, but we know their native language and progress. This means they are missing from the lexdec_subset data set, but they are present in the tidy_demo_wide data set.
- Look at subject 28, they have missing trial data, indicating they aren't present in the lexdec_subset data set, and they are missing a native language, which indicates this data is also missing in the tidy_demo_wide data set.

4.4.1.2 Inner Join

This keeps data only present in **both** data sets.

We have lost participants A1 and A2 because A1 wasn't present in the tidy_demo_wide data set, and A2 wasn't present in the lexdec_subset data set.

```
inner_join(lexdec_subset, tidy_demo_wide, by = c("subject", "native_language"))
```

```
## # A tibble: 24 x 6
##    subject trial native_language progress correct    RT
##    <chr>   <dbl> <chr>           <chr>    <chr>   <dbl>
##  1 A3         41 Other           END      correct  607.
##  2 C          86 English         END      correct  725.
##  3 D         138 Other           END      correct  628.
##  4 I         105 Other           END      correct  560.
##  5 J          43 Other           END      correct  516.
##  6 K         183 English         END      correct  442.
##  7 M1        171 English         END      correct  427.
##  8 M2        117 Other           END      correct  530.
##  9 P         115 Other           END      correct  533.
## 10 R1         30 English         END      correct  483.
## # ... with 14 more rows
```

4.4.1.3 Left Join

Left joins only keep the data that is present in the left data set (lexdec_subset) and adds anything that matches up from the right data set (tidy_demo_wide).

Here we have participant A1 because they are in the lexdec_subset, even if they are missing from the tidy_demo_wide data set.

```
left_join(lexdec_subset, tidy_demo_wide, by = c("subject", "native_language"))
```

```
## # A tibble: 25 x 6
##    subject trial native_language progress correct    RT
##    <chr>   <dbl> <chr>           <chr>    <chr>   <dbl>
##  1 A1        157 English         <NA>     correct  876.
##  2 A3         41 Other           END      correct  607.
##  3 C          86 English         END      correct  725.
##  4 D         138 Other           END      correct  628.
##  5 I         105 Other           END      correct  560.
##  6 J          43 Other           END      correct  516.
##  7 K         183 English         END      correct  442.
##  8 M1        171 English         END      correct  427.
##  9 M2        117 Other           END      correct  530.
## 10 P         115 Other           END      correct  533.
## # ... with 15 more rows
```

4.4.1.4 Right Join

This works like the left join, only it keeps everything present in the right data set and anything matching from the left data set.

Here, we do not have data on participant A1 because they are not present in the tidy_demo_wide data set.

```
right_join(lexdec_subset, tidy_demo_wide, by = c("subject", "native_language"))
```

```
## # A tibble: 29 x 6
##    subject trial native_language progress correct    RT
##    <chr>   <dbl> <chr>           <chr>    <chr>   <dbl>
##  1 A3         41 Other           END      correct  607.
##  2 C          86 English         END      correct  725.
##  3 D         138 Other           END      correct  628.
##  4 I         105 Other           END      correct  560.
##  5 J          43 Other           END      correct  516.
##  6 K         183 English         END      correct  442.
##  7 M1        171 English         END      correct  427.
##  8 M2        117 Other           END      correct  530.
##  9 P         115 Other           END      correct  533.
## 10 R1         30 English         END      correct  483.
## # ... with 19 more rows
```

4.4.2 Filtering Joins

We can filter data by using joins. These next joins don't merge columns, but instead allow us to just subset our data.

4.4.2.1 Semi Join

With a semi-join we keep all rows and columns from the left data set where we have matching values in the right data set. Crucially, we do not keep the columns from the right data set.

```
semi_join(lexdec_subset, tidy_demo_wide, by = c("subject", "native_language"))
```

```
## # A tibble: 24 x 9
##    subject trial native_language word  class  frequency length correct    RT
##    <chr>   <dbl> <chr>           <chr> <chr>      <dbl>  <dbl> <chr>   <dbl>
##  1 A3         41 Other           ant   animal      5.35      3 correct  607.
##  2 C          86 English         ant   animal      5.35      3 correct  725.
##  3 D           138 Other           ant   animal      5.35      3 correct  628.
##  4 I           105 Other           ant   animal      5.35      3 correct  560.
##  5 J            43 Other           ant   animal      5.35      3 correct  516.
##  6 K           183 English         ant   animal      5.35      3 correct  442.
##  7 M1          171 English         ant   animal      5.35      3 correct  427.
##  8 M2          117 Other           ant   animal      5.35      3 correct  530.
##  9 P           115 Other           ant   animal      5.35      3 correct  533.
## 10 R1           30 English         ant   animal      5.35      3 correct  483.
## # ... with 14 more rows
```

Here we only kept data in the lexdec_subset for subjects that were present in both data sets. Notice how we do not have data for subjects A1 and A2.

This works like an inner join, but does not duplicate rows.

Notice that we get the same result with the long demographic data set as with the wide demographic data set.

```
semi_join(lexdec_subset, demo_gathered, by = c("subject", "native_language"))
```

```
## # A tibble: 24 x 9
##    subject trial native_language word  class  frequency length correct    RT
##    <chr>   <dbl> <chr>           <chr> <chr>      <dbl>  <dbl> <chr>   <dbl>
##  1 A3         41 Other           ant   animal      5.35      3 correct  607.
##  2 C          86 English         ant   animal      5.35      3 correct  725.
##  3 D         138 Other           ant   animal      5.35      3 correct  628.
##  4 I         105 Other           ant   animal      5.35      3 correct  560.
##  5 J          43 Other           ant   animal      5.35      3 correct  516.
##  6 K         183 English         ant   animal      5.35      3 correct  442.
##  7 M1        171 English         ant   animal      5.35      3 correct  427.
##  8 M2        117 Other           ant   animal      5.35      3 correct  530.
##  9 P         115 Other           ant   animal      5.35      3 correct  533.
## 10 R1         30 English         ant   animal      5.35      3 correct  483.
## # ... with 14 more rows
```

4.4.2.2 Anti Join

An anti-join works like the inverse of a semi-join. Here, we get all the values from the left table that do not have a match in the right table.

```
anti_join(lexdec_subset, tidy_demo_wide, by = c("subject", "native_language"))
```

```
## # A tibble: 1 x 9
##   subject trial native_language word  class  frequency length correct    RT
##   <chr>   <dbl> <chr>           <chr> <chr>      <dbl>  <dbl> <chr>   <dbl>
## 1 A1        157 English         ant   animal      5.35      3 correct  876.
```

In this case, we only get participant A1 from the lexdec_subset data set, as we do not have any demographic information on this subject in the tidy_demo_wide data set.

4.4.3 Binding Joins

We can bind rows from separate data sets with the same number of columns using the `bind_rows()` command. This is useful if we have ran an experiment in two parts on differnet sets of subjects, and we simply want to put all of the responses in one data set.

Alternatively, we can bind columns from separate data sets with the same number of rows using the `bind_cols()` command. This is useful if we have an experiment in two parts where we want to want to add some additional information about all of participants to one data set.

4.5 Checking for Unique and Duplicate Information

Finally, we can use a number of functions to check for unique information across two different data sets.

- `intersect()` gives us all the rows in two tables that match exactly. This is useful if we have messy data stored in multiple tables and we're not sure if we have duplicates. Note that every cell has to match exactly for this to work.
- `union()` gives us all of the rows from two tables except any duplicates.
- `setdiff()` gives us rows from our first data set that aren't present in the second.

4.6 Exercises

4.6.1 Introduction and Setup

For these exercises, we will look at the core concepts from this lesson. We'll also get some hands-on experience with binding joins and checking for duplicates, two concepts that we've touched on but not went into much detail.

For these exercises we'll use some toy data sets; ex_demo_data, which has demographic information on 6 participants, and ex_test_data, which has IQ test scores for 6 participants. Crucially, the first data set has some missing values, and the second has the same participant tested twice.

```r
# load the tidyverse
library(tidyverse)

# demographic data
ex_demo_data <- tibble(
  subject = seq(1: 6),
  height = c(NA, 170, 160, 165, NA, 180),
  weight = c(70, 65, 80, NA, 77, 90),
  age = c(18, 19, 19, NA, 22, 28)
  )

# IQ test scores
ex_test_data <- tibble(
  subject = c(1, 3, 4, 4, 5, 6, 7),
  IQ = c(150, 160, 155, 155, 190, 120, 140)
  )
```

4.6.2 Long and Wide Data

4.6.2.1 Question 1

Put the `ex_demo_data` into a long format with three columns: subject, measurement_id, and measurement. The measurement column should contain the scores for the height, weight, and age of the participants. The measurement_id column should contain text specifying which measurement belongs to which variable (height, weight, or age). Assign this to the variable `long_data` and return this table of data.

4.6.2.2 Question 2

Turn your newly created `long_data` back into a wide format.

4.6.3 Uniting and Separating Columns

Here we have some messy data where we have two values for two variables in one column; `height_weight`.

```r
messy_demo_data <- unite(ex_demo_data,
                         "height_weight",
                         c("height", "weight"),
                         sep = "_"
                         )
messy_demo_data
```

```
## # A tibble: 6 x 3
##   subject height_weight   age
##     <int> <chr>         <dbl>
## 1       1 NA_70            18
## 2       2 170_65           19
## 3       3 160_80           19
## 4       4 165_NA           NA
## 5       5 NA_77            22
## 6       6 180_90           28
```

4.6.3.1 Question 3

Separate the messy columns into two tidy columns for height and weight. Should you convert the values when separating the column? If so, why?

4.6.4 Mutating Joins

4.6.4.1 Question 4

Join the `ex_demo_data` and `ex_test_data` together by subject number, keeping only data with a match in `ex_test_data`.

4.6.4.2 Question 5

Join the `ex_demo_data` and `ex_test_data` together by subject number, keeping only data with a match in `ex_demo_data`.

4.6.4.3 Question 6

Why do we get different results in question 4 and question 5?

4.6.5 Filtering Joins

4.6.5.1 Question 7

Return all of the values from `ex_demo_data` that have a match in `ex_test_data`. Look at subject 4, why do we get a different result to that from question 5? Look at the columns returned, why does this differ from question 5?

4.6.6 Binding Joins

Here we have some new data looking at the demographic scores for new subjects. We also have another rating for all of the participants from our study and we want to add this to the demographic data.

```
new_demographics <- tibble(
  subject = c(9, 10),
  height = c(170, 190),
  weight = c(76, 85),
  age = c(40, 59)
  )

eye_colour <- tibble(
  eye_colour = sample(c("blue", "brown", "green"),
                      size = 8,
                      replace = TRUE
                      )
  )
```

4.6.6.1 Question 8

Add the rows from `new_demographics` to `ex_demo_data`. Assign this to `all_demo_data` and return this table.

4.6.6.2 Question 9

Add the eye colour column to the `all_demo_data` table. Why did we not have a subject identifier in the `eye_colour` data set? Can you predict the result if we did have this information?

4.6.7 Checking for Duplicates

We have some new test data below.

```
extra_test_data <- tibble(
  subject = c(1, 9, 10),
  IQ = c(150, 156, 179)
  )
```

4.6.7.1 Question 10

Return rows with duplicates from the `ex_test_data` and `extra_test_data` data sets.

Chapter 5

Data Manipulation 2

In this section, we'll look at getting our data in the correct format for reporting your analyses. We'll cover topics such as renaming and reordering variables, creating new variables, and creating summaries of your data. To do this, we'll use the **dplyr** package (Wickham et al., 2017a) from the tidyverse (Wickham, 2017).

We'll cover the following functions that work with the `group_by` functioin:

- `arrange()`: order variables (i.e. ordering in rows)
- `select()`: pick out variables (i.e. subsetting by columns, changing column order)
- `filter()`: pick out observations (i.e. subsetting by observations)
- `mutate()`: create new variables
- `summarise()`: create a summary of variables (e.g. mean, *SD*, n)

The `group_by` function allows you to change the scope by which these functions work. For example, we can group our data by condition with `group_by(condition)` and then create summary statistics with `summarise()`.

Additionally, we'll cover renaming variables with `rename()`.

You should notice some familiar patterns with how we use and mix together these functions as we progress.

5.1 Getting Started

As always, we first need to load the `tidyverse` set of packages for this Chapter.

```
library(tidyverse)
```

Next, we'll load the raw and (wide formatted) demographic data sets from the lexical decision experiment from the previous Chapter. To do so, we'll use `read_csv()` here, for reasons explained in the previous Chapter.

```
demo <- read_csv("inputs/lexical_decision_demographic_data_wide.csv")
```

You should get a warning when loading the data, telling you how each column is parsed. We can specify during the loading process how to parse each column, for example setting strings to factors. However, we'll do this manually in `dplyr()` to give you more experience using this pacakge for mutating your data.

5.2 Understanding our Data

Let's look at the data we've just loaded to get an idea of how we want to change it. When we have many rows and we attempt to print a tibble, we won't see all columns in the console. One way to get an idea of how all columns look is to transpose these columns and take a look at the first few observations. To do this, we can use `glimpse()` to get a glimpse of our data.

```
glimpse(demo)
```

```
## Rows: 29
## Columns: 14
## $ ID              <chr> "22", "23", "24", "25", "26", "27", "28", "29", "30", ~
## $ `C++`           <dbl> 1, 1, NA, 1, NA, NA, NA, NA, NA, 1, 1, NA, NA, NA, NA,~
## $ FORTRAN         <dbl> 1, NA, 1, 1, NA, 1, NA, NA, 1, 1, NA, NA, 1, NA, 1, NA~
## $ JavaScript      <dbl> 1, NA, NA, NA, NA, 1, NA, 1, NA, 1, NA, NA, NA, NA, 1,~
## $ Python          <dbl> NA, NA, NA, NA, 1, NA, NA, NA, NA, NA, NA, 1, NA, NA, ~
## $ R               <dbl> NA, NA, NA, NA, NA, NA, 1, NA, NA, NA, NA, 1, 1, NA, N~
## $ Ruby            <dbl> NA, NA, NA, NA, 1, NA, 1, NA, NA, NA, NA, NA, NA, 1, N~
## $ LANGUAGE        <chr> "OTHER", NA, "ENGLISH", NA, "english", "ENGLISH", NA, ~
## $ progress        <chr> "FINISH", "no", "END", "ethics", "ethics", "END", "eth~
## $ gender          <chr> "female", "male", "female", "male", "male", "female", ~
## $ age             <dbl> 30, 30, 30, 18, 31, 44, 23, 34, 32, 33, 20, 31, 33, 36~
## $ tester          <chr> "GW", "GW", "GW", "GW", "GW", "GW", "GW", "GW", "GW", ~
## $ funRec          <chr> "7-no", "6-yes", "4-yes", "0-no", "4-no", "5-no", "0-n~
## $ completion_time <chr> "2018-03-22 23:06:11_2018-03-23 00:25:51", "2018-03-26~
```

As we can see, we have information on the subject's ID, columns indicating any programming languages they might know, a column specifying their language spoken, how far they got in the experiment, some demographic information, who tested them, whether they liked the experiment and would recommend it to others, and the start and end times for their part in the experiment.

We can also see how this data was parsed, with numeric columns as integers, and all other columns as characters. However, some of this data would be best represented as a different format. We'll look into changing this later in this Chapter.

5.3 Preparing our Data

Our data is quite messy as it stands. One of the biggest issues is that we have two variables contained within one column for both the `funRec` and `completion_time` columns. Let's separate these two columns using the `separate()` function from **tidyr**.

```
demo <- demo %>%
  separate(col = funRec, into = c("fun", "recommend")) %>%
  separate(col = completion_time, into = c("start", "end"), sep = "_")
glimpse(demo)
```

```
## Rows: 29
## Columns: 16
## $ ID          <chr> "22", "23", "24", "25", "26", "27", "28", "29", "30", "A2",~
## $ `C++`       <dbl> 1, 1, NA, 1, NA, NA, NA, NA, NA, 1, 1, NA, NA, NA, NA, 1, 1~
## $ FORTRAN     <dbl> 1, NA, 1, 1, NA, 1, NA, NA, 1, 1, NA, NA, 1, NA, 1, NA, NA,~
## $ JavaScript <dbl> 1, NA, NA, NA, NA, 1, NA, 1, NA, 1, NA, NA, NA, NA, 1, 1, N~
## $ Python      <dbl> NA, NA, NA, NA, 1, NA, NA, NA, NA, NA, NA, 1, NA, NA, 1, NA~
## $ R           <dbl> NA, NA, NA, NA, NA, NA, 1, NA, NA, NA, NA, 1, 1, NA, NA, NA~
## $ Ruby        <dbl> NA, NA, NA, NA, 1, NA, 1, NA, NA, NA, NA, NA, NA, 1, NA, NA~
## $ LANGUAGE    <chr> "OTHER", NA, "ENGLISH", NA, "english", "ENGLISH", NA, "engl~
## $ progress    <chr> "FINISH", "no", "END", "ethics", "ethics", "END", "ethics",~
## $ gender      <chr> "female", "male", "female", "male", "male", "female", "male~
## $ age         <dbl> 30, 30, 30, 18, 31, 44, 23, 34, 32, 33, 20, 31, 33, 36, 28,~
## $ tester      <chr> "GW", "GW", "GW", "GW", "GW", "GW", "GW", "GW", "GW", "RHB"~
## $ fun         <chr> "7", "6", "4", "0", "4", "5", "0", "6", "1", "4", "3", "1",~
## $ recommend   <chr> "no", "yes", "yes", "no", "no", "no", "no", "no", "yes", "y~
## $ start       <chr> "2018-03-22 23:06:11", "2018-03-26 00:30:20", "2018-03-21 1~
## $ end         <chr> "2018-03-23 00:25:51", "2018-03-26 02:15:52", "2018-03-21 1~
```

5.4 Selecting Columns

First off, we should probably drop any columns from the data that we don't need to use. We're not going to look at the programming languages they know, so we can drop all of these columns using `select()`. We'll create a subset of our data that doesn't contain these columns.

Here, we'll ask to keep all of the columns we want to keep. We can do this by name or column number.

```
# by name
demo %>% select(ID,
                LANGUAGE,
                progress,
                gender,
                age,
                tester,
                fun,
                recommend,
                start,
                end
                )
```

```
## # A tibble: 29 x 10
##    ID    LANGUAGE progress gender   age tester fun   recommend start       end
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>     <chr>       <chr>
##  1 22    OTHER    FINISH   female    30 GW     7     no        2018-03-22~ 2018~
##  2 23    <NA>     no       male      30 GW     6     yes       2018-03-26~ 2018~
##  3 24    ENGLISH  END      female    30 GW     4     yes       2018-03-21~ 2018~
##  4 25    <NA>     ethics   male      18 GW     0     no        2018-03-25~ 2018~
##  5 26    english  ethics   male      31 GW     4     no        2018-03-24~ 2018~
##  6 27    ENGLISH  END      female    44 GW     5     no        2018-03-21~ 2018~
##  7 28    <NA>     ethics   male      23 GW     0     no        2018-03-25~ 2018~
##  8 29    english  ethics   male      34 GW     6     no        2018-03-24~ 2018~
##  9 30    <NA>     <NA>     female    32 GW     1     yes       2018-03-22~ 2018~
## 10 A2    English  END      female    33 RHB    4     yes       2018-03-23~ 2018~
## # ... with 19 more rows
```

```
# by number
demo %>% select(c(1, 8:16))
```

```
## # A tibble: 29 x 10
##    ID    LANGUAGE progress gender   age tester fun   recommend start       end
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>     <chr>       <chr>
##  1 22    OTHER    FINISH   female    30 GW     7     no        2018-03-22~ 2018~
##  2 23    <NA>     no       male      30 GW     6     yes       2018-03-26~ 2018~
##  3 24    ENGLISH  END      female    30 GW     4     yes       2018-03-21~ 2018~
##  4 25    <NA>     ethics   male      18 GW     0     no        2018-03-25~ 2018~
##  5 26    english  ethics   male      31 GW     4     no        2018-03-24~ 2018~
##  6 27    ENGLISH  END      female    44 GW     5     no        2018-03-21~ 2018~
##  7 28    <NA>     ethics   male      23 GW     0     no        2018-03-25~ 2018~
##  8 29    english  ethics   male      34 GW     6     no        2018-03-24~ 2018~
##  9 30    <NA>     <NA>     female    32 GW     1     yes       2018-03-22~ 2018~
## 10 A2    English  END      female    33 RHB    4     yes       2018-03-23~ 2018~
## # ... with 19 more rows
```

Alternatively, we could have just specified the columns we wanted to drop. To do so, we can tell R to select all columns, except (`-`) column numbers 2 through 7. Remember, we have to concatenate these values to apply tell R to remove all of these columns, and not just the first one.

This time, we'll save our data as `demo_sub`, as we've made a subset of our original demographic data.

```
demo_sub <- demo %>% select(-c(2:7))
demo_sub
```

```
## # A tibble: 29 x 10
##    ID    LANGUAGE progress gender   age tester fun   recommend start       end
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>     <chr>       <chr>
##  1 22    OTHER    FINISH   female    30 GW     7     no        2018-03-22~ 2018~
##  2 23    <NA>     no       male      30 GW     6     yes       2018-03-26~ 2018~
##  3 24    ENGLISH  END      female    30 GW     4     yes       2018-03-21~ 2018~
##  4 25    <NA>     ethics   male      18 GW     0     no        2018-03-25~ 2018~
##  5 26    english  ethics   male      31 GW     4     no        2018-03-24~ 2018~
##  6 27    ENGLISH  END      female    44 GW     5     no        2018-03-21~ 2018~
##  7 28    <NA>     ethics   male      23 GW     0     no        2018-03-25~ 2018~
##  8 29    english  ethics   male      34 GW     6     no        2018-03-24~ 2018~
##  9 30    <NA>     <NA>     female    32 GW     1     yes       2018-03-22~ 2018~
## 10 A2    English  END      female    33 RHB    4     yes       2018-03-23~ 2018~
## # ... with 19 more rows
```

There are a number of helper functions in `select()` which can help you to stop having to explicitly mention every column you want to keep. These are:

- `starts_with("string")`: this keeps any columns with names starting with "string"
- `ends_with("string")`: this keeps any columns with names ending with "string"

- `contains("string")`: this keeps any columns with names containing "string" at any point
- `matches(regular_expression)`: this keeps any columns that match a regular expression
- `num_range("prefix", range)`: this keeps any columns with a matching prefix and a following range of numbers. For example, 'num_range("measurement", 1:3) would keep all columns called "measurement1", "measurement2", and "measurement3".

For this simple case here, these aren't necessary, but they're good to know for working with larger data sets.

5.4.1 Renaming and Reordering Columns

We can use the `rename()` function to change our columns names. We have some messy names here, so we'll look at how we can rename them now. This renaming function takes the order of our new names are equal to our old names. Let's just overwrite our old subsetted data to change the names.

```
demo_sub <- demo_sub %>% rename(language = LANGUAGE)
demo_sub
```

```
## # A tibble: 29 x 10
##    ID    language progress gender   age tester fun   recommend start       end
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>     <chr>       <chr>
##  1 22    OTHER    FINISH   female    30 GW     7     no        2018-03-22~ 2018~
##  2 23    <NA>     no       male      30 GW     6     yes       2018-03-26~ 2018~
##  3 24    ENGLISH  END      female    30 GW     4     yes       2018-03-21~ 2018~
##  4 25    <NA>     ethics   male      18 GW     0     no        2018-03-25~ 2018~
##  5 26    english  ethics   male      31 GW     4     no        2018-03-24~ 2018~
##  6 27    ENGLISH  END      female    44 GW     5     no        2018-03-21~ 2018~
##  7 28    <NA>     ethics   male      23 GW     0     no        2018-03-25~ 2018~
##  8 29    english  ethics   male      34 GW     6     no        2018-03-24~ 2018~
##  9 30    <NA>     <NA>     female    32 GW     1     yes       2018-03-22~ 2018~
## 10 A2    English  END      female    33 RHB    4     yes       2018-03-23~ 2018~
## # ... with 19 more rows
```

Finally, what if we want to put the age and gender columns next to the ID column? We can use `select()` to pick out our first two columns, then `everything()` to grab every remaining column and stick them on the end.

```
demo_sub %>% select(ID, age, gender, everything())
```

```
## # A tibble: 29 x 10
##    ID      age gender language progress tester fun   recommend start        end
##    <chr> <dbl> <chr>  <chr>    <chr>    <chr>  <chr> <chr>     <chr>        <chr>
##  1 22       30 female OTHER    FINISH   GW     7     no        2018-03-22~ 2018~
##  2 23       30 male   <NA>     no       GW     6     yes       2018-03-26~ 2018~
##  3 24       30 female ENGLISH  END      GW     4     yes       2018-03-21~ 2018~
##  4 25       18 male   <NA>     ethics   GW     0     no        2018-03-25~ 2018~
##  5 26       31 male   english  ethics   GW     4     no        2018-03-24~ 2018~
##  6 27       44 female ENGLISH  END      GW     5     no        2018-03-21~ 2018~
##  7 28       23 male   <NA>     ethics   GW     0     no        2018-03-25~ 2018~
##  8 29       34 male   english  ethics   GW     6     no        2018-03-24~ 2018~
##  9 30       32 female <NA>     <NA>     GW     1     yes       2018-03-22~ 2018~
## 10 A2       33 female English  END      RHB    4     yes       2018-03-23~ 2018~
## # ... with 19 more rows
```

5.5 Creating and Changing Columns

If we want to create a new column in our data, or it we want to change how a column is represented, we have to mutate our data using`mutate()`.

What if we want to make a new column that tells us how long people spent on the experiment? We can create a new column by subtracting the start time of each participant from their end time.

```
# this will not run

# demo_sub <- demo_sub %>%
#   mutate(time = end - start)
```

Oops, that didn't work. Why not? It looks like the start and end columns are stored as characters. We can't perform a subtraction on a character, so we need to tell R that these are numbers. Specifically, we need to tell R that these are datetimes.

One package that makes this very easy is the **lubridate** package (Grolemund and Wickham, 2011), which is designed to make working with datetimes very easy. Let's install this before we convert our start and end columns to datetimes.

```
library(lubridate)
```

```
# install.packages("lubridate") # uncomment and run this only once
library(lubridate)
```

The **lubridate** package has a number of useful functions that allow us to covert our data types to datetimes, as well as ways to run calculations on these times.

One function we'll use here is the `ymd_hms()` function, which converts data to the POSIXct format. The upshoot of this is that we can now subract one time from the other to get a duration.

Below, we'll mutate the start and end columns to be themselves, only converted to the proper datetime format. We'll also create a new column, time, made up of subtracting the start times from the end times to get a duration.

```
demo_sub <- demo_sub %>%
  mutate(start = ymd_hms(start),
         end = ymd_hms(end),
         time = end - start
         )
glimpse(demo_sub)
```

```
## Rows: 29
## Columns: 11
## $ ID        <chr> "22", "23", "24", "25", "26", "27", "28", "29", "30", "A2", ~
## $ language  <chr> "OTHER", NA, "ENGLISH", NA, "english", "ENGLISH", NA, "engli~
## $ progress  <chr> "FINISH", "no", "END", "ethics", "ethics", "END", "ethics", ~
## $ gender    <chr> "female", "male", "female", "male", "male", "female", "male"~
## $ age       <dbl> 30, 30, 30, 18, 31, 44, 23, 34, 32, 33, 20, 31, 33, 36, 28, ~
## $ tester    <chr> "GW", "GW", "GW", "GW", "GW", "GW", "GW", "GW", "GW", "RHB",~
## $ fun       <chr> "7", "6", "4", "0", "4", "5", "0", "6", "1", "4", "3", "1", ~
## $ recommend <chr> "no", "yes", "yes", "no", "no", "no", "no", "no", "yes", "ye~
## $ start     <dttm> 2018-03-22 23:06:11, 2018-03-26 00:30:20, 2018-03-21 11:09:~
## $ end       <dttm> 2018-03-23 00:25:51, 2018-03-26 02:15:52, 2018-03-21 12:16:~
## $ time      <drtn> 1.327778 hours, 1.758889 hours, 1.113889 hours, 1.690833 ho~
```

Did you notice that we can refer to columns we've asked to create within `mutate()` to create other columns? Pretty cool!

We can perform any number of operations on our columns in `mutate()`. But what if we want to do this and only keep the new columns? We can use `transmute()`.

Below, we'll ask to keep the subject ID column, but to create our new start and end columns with mutate. Since the difference from the mean will (typically) be quite small, we can convert from hours to minutes by setting the units within our time variable to minutes.

```
# calculate difference from mean completion time
transmuted_time <- demo_sub %>%
  mutate(
    start = ymd_hms(start),
    end = ymd_hms(end),
    time = end - start
  ) %>%
  transmute(
    ID,
    time_diff = time - mean(time)
  )

# define the units of time, this can be mins, hours, days, etc.
units(transmuted_time$time_diff) <- "mins"

# print the object
transmuted_time
```

```
## # A tibble: 29 x 2
##    ID    time_diff
##    <chr> <drtn>
##  1 22     -7.185632 mins
##  2 23     18.681034 mins
##  3 24    -20.018966 mins
##  4 25     14.597701 mins
##  5 26      4.131034 mins
##  6 27    -22.785632 mins
##  7 28     17.481034 mins
##  8 29      8.147701 mins
##  9 30    -13.902299 mins
## 10 A2     -5.868966 mins
## # ... with 19 more rows
```

Note that we can change a column to many different data types by simply setting the column name to itself and the data type you want. For example, `mutate(start = as.numeric(start))`. We can also mutate multiple columns at once using `mutate_all()` or `mutate_at()`, but we won't cover this here. Check out the variants on the **dplyr** verbs to see how you can improve your code!

So far, we've learned how to create new columns, including keeping these with the original data set (`mutate()`) or keeping only the new columns (`transmute()`). We've also seen how we can subset our data to the most relevant columns using `select()`. Next, we'll look at how we can subset our data by selecting certain observations (i.e. rows) within our data. For this, we'll use the `filter()` function.

5.6 Filtering to Observations

The `filter()` function allows us to filter our data to certain observations. We can use a number of logical operations to filter by certain conditions. As with the other **dplyr** functions, `filter()` must take the data as an argument, but afterwards it needs to take a rule by which to subset your data. Here, you just specify the conditons by which to keep observations. Nicely, the `filter()` argument can be piped, and used in conjunction with other **dplyr** functions.

Take a look at the `demo_sub` data set.

```
glimpse(demo_sub)
```

```
## Rows: 29
## Columns: 11
## $ ID        <chr> "22", "23", "24", "25", "26", "27", "28", "29", "30", "A2", ~
## $ language  <chr> "OTHER", NA, "ENGLISH", NA, "english", "ENGLISH", NA, "engli~
## $ progress  <chr> "FINISH", "no", "END", "ethics", "ethics", "END", "ethics", ~
## $ gender    <chr> "female", "male", "female", "male", "male", "female", "male"~
## $ age       <dbl> 30, 30, 30, 18, 31, 44, 23, 34, 32, 33, 20, 31, 33, 36, 28, ~
## $ tester    <chr> "GW", "GW", "GW", "GW", "GW", "GW", "GW", "GW", "GW", "RHB",~
## $ fun       <chr> "7", "6", "4", "0", "4", "5", "0", "6", "1", "4", "3", "1", ~
## $ recommend <chr> "no", "yes", "yes", "no", "no", "no", "no", "no", "yes", "ye~
## $ start     <dttm> 2018-03-22 23:06:11, 2018-03-26 00:30:20, 2018-03-21 11:09:~
## $ end       <dttm> 2018-03-23 00:25:51, 2018-03-26 02:15:52, 2018-03-21 12:16:~
## $ time      <drtn> 1.327778 hours, 1.758889 hours, 1.113889 hours, 1.690833 ho~
```

You can see that we have a number of columns, but crucially, we have one where we kept track of the subject's progress in our experiment. How many labels do we have for this? We can check for the unique labels in oru data using the `unique()` function. Alternatively, if our data is a factor, then it should have defined levels, so we could use the `levels()` function. To save time recoding our data types, we'll just use the `unique()` function for now.

```
unique(demo_sub$progress)
```

```
## [1] "FINISH" "no"     "END"    "ethics" NA
```

We have a few labels: FINISH, no, END, ethics, and NA. It looks like we have a slightly messy data set, as both FINISH and END indicate the same point of progress; these subjects completed the experiment. If we just want to subset our data to those who finished the experiment, we could use these labels to filter out observations.

5.6.1 Filtering with Logical Operations

```
demo_sub %>% filter(progress == "FINISH" | progress == "END")
```

```
## # A tibble: 23 x 11
##    ID    language progress gender   age tester fun   recom~1 start
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>   <dttm>
##  1 22    OTHER    FINISH   female    30 GW     7     no      2018-03-22 23:06:11
##  2 24    ENGLISH  END      female    30 GW     4     yes     2018-03-21 11:09:38
##  3 27    ENGLISH  END      female    44 GW     5     no      2018-03-21 03:23:57
##  4 A2    English  END      female    33 RHB    4     yes     2018-03-23 02:48:32
##  5 A3    Other    END      male      20 RHB    3     no      2018-03-25 23:56:13
##  6 C     English  END      female    31 RHB    1     yes     2018-03-22 21:17:22
##  7 D     Other    END      non-b~    33 RHB    5     yes     2018-03-26 18:28:33
##  8 I     Other    END      male      36 RHB    5     no      2018-03-26 01:23:54
##  9 J     Other    END      female    28 RHB    6     yes     2018-03-21 04:15:50
## 10 K     English  END      male      33 RHB    1     yes     2018-03-24 02:46:45
## # ... with 13 more rows, 2 more variables: end <dttm>, time <drtn>, and
## #   abbreviated variable name 1: recommend
```

Read the above command like, "filter to observations where the progress is equal to FINISH **or** the progress is equal to END". Remember, to check if something is equal to another thing, we have to use double equals (==) rather than a single equal (=) as a single equal is an assignment operator. Finally, the pipe between these two statements is an OR operator, so we keep any observations that are equal to FINISH **or** equal to END.

This type of filtering can get confusing if we have more than 2 conditions that we would like to check. One way around this is to use the %in% operator, which evaluates to TRUE if an observation is **in** a sequence that you provide. The best way to see how this works is to try it out yourself:

```
demo_sub %>% filter(progress %in% c("FINISH", "END"))
```

```
## # A tibble: 23 x 11
##    ID    language progress gender   age tester fun   recom~1 start
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>   <dttm>
##  1 22    OTHER    FINISH   female    30 GW     7     no      2018-03-22 23:06:11
##  2 24    ENGLISH  END      female    30 GW     4     yes     2018-03-21 11:09:38
##  3 27    ENGLISH  END      female    44 GW     5     no      2018-03-21 03:23:57
##  4 A2    English  END      female    33 RHB    4     yes     2018-03-23 02:48:32
##  5 A3    Other    END      male      20 RHB    3     no      2018-03-25 23:56:13
##  6 C     English  END      female    31 RHB    1     yes     2018-03-22 21:17:22
##  7 D     Other    END      non-b~    33 RHB    5     yes     2018-03-26 18:28:33
##  8 I     Other    END      male      36 RHB    5     no      2018-03-26 01:23:54
##  9 J     Other    END      female    28 RHB    6     yes     2018-03-21 04:15:50
## 10 K     English  END      male      33 RHB    1     yes     2018-03-24 02:46:45
## # ... with 13 more rows, 2 more variables: end <dttm>, time <drtn>, and
## #   abbreviated variable name 1: recommend
```

You can see that we get the same result here, but we can easily add more and more values inside the parentheses to be evaluated.

We can combine multiple filtering arguments by different columns, too. What if we wanted to get only those who completed the experiment, were tested by “RHB”, and are over the age of 30?

5.6.2 Combining Filtering Criteria

```
demo_sub %>% filter(progress %in% c("FINISH", "END"),
                    tester == "RHB",
                    age > 30
                    )
```

```
## # A tibble: 9 x 11
##   ID    language progress gender    age tester fun   recom~1 start
##   <chr> <chr>    <chr>    <chr>   <dbl> <chr>  <chr> <chr>   <dttm>
## 1 A2    English  END      female     33 RHB    4     yes     2018-03-23 02:48:32
## 2 C     English  END      female     31 RHB    1     yes     2018-03-22 21:17:22
## 3 D     Other    END      non-bi~    33 RHB    5     yes     2018-03-26 18:28:33
## 4 I     Other    END      male       36 RHB    5     no      2018-03-26 01:23:54
## 5 K     English  END      male       33 RHB    1     yes     2018-03-24 02:46:45
## 6 R1    English  END      female     31 RHB    6     yes     2018-03-21 19:23:38
## 7 R3    English  END      female     35 RHB    0     no      2018-03-22 16:40:33
## 8 V     Other    END      male       37 RHB    7     no      2018-03-23 19:56:16
## 9 W1    English  END      non-bi~    31 RHB    1     no      2018-03-26 11:28:20
## # ... with 2 more variables: end <dttm>, time <drtn>, and abbreviated variable
## #   name 1: recommend
```

5.6.3 Removing by Ceriteria

Similarly to keeping those that match some crtieria, we can remove those who meet some criteria. Let’s say we just want to look at those who haven’t finished the experiment. We can do this by using the `!` (read: not) operator. Here, we ask to **not** keep those who have their progress as FINISH or END.

```
demo_sub %>% filter(!progress %in% c("FINISH", "END"))
```

```
## # A tibble: 6 x 11
##   ID    language progress gender   age tester fun   recomm~1 start
##   <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>    <dttm>
## 1 23    <NA>     no       male      30 GW     6     yes      2018-03-26 00:30:20
## 2 25    <NA>     ethics   male      18 GW     0     no       2018-03-25 13:03:58
## 3 26    english  ethics   male      31 GW     4     no       2018-03-24 06:46:30
## 4 28    <NA>     ethics   male      23 GW     0     no       2018-03-25 21:07:24
## 5 29    english  ethics   male      34 GW     6     no       2018-03-24 18:03:12
## 6 30    <NA>     <NA>     female    32 GW     1     yes      2018-03-22 04:16:08
## # ... with 2 more variables: end <dttm>, time <drtn>, and abbreviated variable
## #   name 1: recommend
```

5.6.4 Handling NAs

We always need to be wary of NAs in R. If we ask R whether something is equal to an NA (an unknown value) it is very literal in that it tells us it can't know. That's because R has no information about what the NA is! Try filtering our data to only those with NAs for the language known.

```
# this will not run
demo_sub %>% filter(language == NA)
```

```
## # A tibble: 0 x 11
## # ... with 11 variables: ID <chr>, language <chr>, progress <chr>,
## #   gender <chr>, age <dbl>, tester <chr>, fun <chr>, recommend <chr>,
## #   start <dttm>, end <dttm>, time <drtn>
```

Instead, we have to surround our variable with the `is.na()` function, to check whether these values are NAs.

```
demo_sub %>% filter(is.na(language))
```

```
## # A tibble: 4 x 11
##   ID    language progress gender   age tester fun   recomm~1 start
##   <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>    <dttm>
## 1 23    <NA>     no       male      30 GW     6     yes      2018-03-26 00:30:20
## 2 25    <NA>     ethics   male      18 GW     0     no       2018-03-25 13:03:58
## 3 28    <NA>     ethics   male      23 GW     0     no       2018-03-25 21:07:24
## 4 30    <NA>     <NA>     female    32 GW     1     yes      2018-03-22 04:16:08
## # ... with 2 more variables: end <dttm>, time <drtn>, and abbreviated variable
## #   name 1: recommend
```

Also note that by default `filter()` excludes cases that don't meet your criteria, or are NAs, so if you want to keep NAs you have to ask for them explicitly. Let's look at filtering to only those who finished the experiment (FINISH or END) and those who we're unsure about (NA).

Here, we look for observations for progress that are equal to FINISH **or** equal to END **or** which are NAs.

```
demo_sub %>% filter(progress == "FINISH" | progress == "END" | is.na(progress))
```

```
## # A tibble: 24 x 11
##    ID    language progress gender   age tester fun   recom~1 start
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>   <dttm>
##  1 22    OTHER    FINISH   female    30 GW     7     no      2018-03-22 23:06:11
##  2 24    ENGLISH  END      female    30 GW     4     yes     2018-03-21 11:09:38
##  3 27    ENGLISH  END      female    44 GW     5     no      2018-03-21 03:23:57
##  4 30    <NA>     <NA>     female    32 GW     1     yes     2018-03-22 04:16:08
##  5 A2    English  END      female    33 RHB    4     yes     2018-03-23 02:48:32
##  6 A3    Other    END      male      20 RHB    3     no      2018-03-25 23:56:13
##  7 C     English  END      female    31 RHB    1     yes     2018-03-22 21:17:22
##  8 D     Other    END      non-b~    33 RHB    5     yes     2018-03-26 18:28:33
##  9 I     Other    END      male      36 RHB    5     no      2018-03-26 01:23:54
## 10 J     Other    END      female    28 RHB    6     yes     2018-03-21 04:15:50
## # ... with 14 more rows, 2 more variables: end <dttm>, time <drtn>, and
## #   abbreviated variable name 1: recommend
```

Alternatively, we can filter our data to anything that is not an NA (or that does not meet any range of crieria). To do this, we use the `!` logical operator (read this as "not").

Similarly, we can combine turn this around and ask to not keep those that match the criteria above. Here, we have to ask to throw away those with progress that isn't FINISH and isn't END, **and** we want to throw out those with an NA for their progress.

```
demo_sub %>% filter(!progress %in% c("FINISH", "END") & !is.na(progress))
```

```
## # A tibble: 5 x 11
##   ID    language progress gender   age tester fun   recomm~1 start
##   <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>    <dttm>
## 1 23    <NA>     no       male      30 GW     6     yes      2018-03-26 00:30:20
## 2 25    <NA>     ethics   male      18 GW     0     no       2018-03-25 13:03:58
## 3 26    english  ethics   male      31 GW     4     no       2018-03-24 06:46:30
## 4 28    <NA>     ethics   male      23 GW     0     no       2018-03-25 21:07:24
## 5 29    english  ethics   male      34 GW     6     no       2018-03-24 18:03:12
## # ... with 2 more variables: end <dttm>, time <drtn>, and abbreviated variable
## #   name 1: recommend
```

5.7 Arranging Data

Often, with psychological data, we want to order our columns by the subject number, then their observations. This makes it easy to think about how individuals progressed through the experiment.

We can arrange our data using `arrange()` from **dplyr**. Again, this function takes the data, and then sorts by any columns that you give it. This function defaults to having lowest mnumbers first. So, if we sorted by subject ID and trial ID, you would get the lowest value for subjects first, and their lowest number for trials first.

We have a simple case here, where we only have one observation for each participant. Here, numbers come before letters, but those with numbered IDs were tested at the end of the experiment. How, then, should we order the data?

One option is to order the data such that IDs appear in descending order (using the `desc()` function).

```
demo_sub %>% arrange(desc(ID))
```

```
## # A tibble: 29 x 11
##    ID    language progress gender   age tester fun   recom~1 start
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>   <dttm>
##  1 Z     Other    END      female    28 RHB    0     no      2018-03-21 12:33:04
##  2 W2    English  END      female    29 RHB    1     yes     2018-03-21 07:16:06
##  3 W1    English  END      non-b~    31 RHB    1     no      2018-03-26 11:28:20
##  4 V     Other    END      male      37 RHB    7     no      2018-03-23 19:56:16
##  5 T2    Other    END      male      27 RHB    0     yes     2018-03-25 10:40:49
##  6 T1    English  END      male      22 RHB    4     no      2018-03-24 12:08:05
##  7 S     English  END      female    25 RHB    4     no      2018-03-21 04:04:38
##  8 R3    English  END      female    35 RHB    0     no      2018-03-22 16:40:33
##  9 R2    English  END      non-b~    27 RHB    5     no      2018-03-26 08:15:03
## 10 R1    English  END      female    31 RHB    6     yes     2018-03-21 19:23:38
## # ... with 19 more rows, 2 more variables: end <dttm>, time <drtn>, and
## #   abbreviated variable name 1: recommend
```

Alternatively, we know that the tester GW ran the experiment last, so we could sort by tester. But let's say that we want to sort by who tested them first, and then by who started the experiment first. To do this, we simply need to provide the start and tester columns to the function.

```
demo_sub %>% arrange(tester, start)
```

```
## # A tibble: 29 x 11
##    ID    language progress gender   age tester fun   recom~1 start
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>   <dttm>
##  1 27    ENGLISH  END      female    44 GW     5     no      2018-03-21 03:23:57
##  2 24    ENGLISH  END      female    30 GW     4     yes     2018-03-21 11:09:38
##  3 30    <NA>     <NA>     female    32 GW     1     yes     2018-03-22 04:16:08
##  4 22    OTHER    FINISH   female    30 GW     7     no      2018-03-22 23:06:11
##  5 26    english  ethics   male      31 GW     4     no      2018-03-24 06:46:30
##  6 29    english  ethics   male      34 GW     6     no      2018-03-24 18:03:12
##  7 25    <NA>     ethics   male      18 GW     0     no      2018-03-25 13:03:58
##  8 28    <NA>     ethics   male      23 GW     0     no      2018-03-25 21:07:24
##  9 23    <NA>     no       male      30 GW     6     yes     2018-03-26 00:30:20
## 10 S     English  END      female    25 RHB    4     no      2018-03-21 04:04:38
## # ... with 19 more rows, 2 more variables: end <dttm>, time <drtn>, and
## #   abbreviated variable name 1: recommend
```

Again, we can flip the order for tester by using the `desc()` function.

```
demo_sub %>% arrange(desc(tester), start)
```

```
## # A tibble: 29 x 11
##    ID    language progress gender   age tester fun   recom~1 start
##    <chr> <chr>    <chr>    <chr>  <dbl> <chr>  <chr> <chr>   <dttm>
##  1 S     English  END      female    25 RHB    4     no      2018-03-21 04:04:38
##  2 J     Other    END      female    28 RHB    6     yes     2018-03-21 04:15:50
##  3 M2    Other    END      female    27 RHB    4     no      2018-03-21 06:16:32
##  4 W2    English  END      female    29 RHB    1     yes     2018-03-21 07:16:06
##  5 Z     Other    END      female    28 RHB    0     no      2018-03-21 12:33:04
##  6 R1    English  END      female    31 RHB    6     yes     2018-03-21 19:23:38
##  7 R3    English  END      female    35 RHB    0     no      2018-03-22 16:40:33
##  8 C     English  END      female    31 RHB    1     yes     2018-03-22 21:17:22
##  9 A2    English  END      female    33 RHB    4     yes     2018-03-23 02:48:32
## 10 V     Other    END      male      37 RHB    7     no      2018-03-23 19:56:16
## # ... with 19 more rows, 2 more variables: end <dttm>, time <drtn>, and
## #   abbreviated variable name 1: recommend
```

Note: If we have missing values, these always come at the end of the table.

5.8 Summarising Data

Finally, we get onto the most important section for psychologists; how to report summaries of your data.

`summarise()` collapses across all observations in your data set to produce a single row of data. Within the `summarise()` function, we have to specify what we would like to create. To do this, we just use the same functions we would use to create a new column in our original data set.

Let's say we want to calculate the average time spent on the experiment. We just need to specify what our column will be called (mean_time here), and how we should create this column. Here, we've created it by calculating the `mean()` over the time column in our data.

```
demo_sub %>% summarise(mean_time = mean(time))
```

```
## # A tibble: 1 x 1
##   mean_time
##   <drtn>
## 1 1.447538 hours
```

Now we know that the average completion time was 1.45 minutes. However, this doesn't tell us much. We might also want to know the standard deviation for the mean times, as well as the count (i.e. how many people completed the experiment). To do this, we just need to list more arguments.

To add these summaries, we use the inbuilt `sd()` function, as well as the inbuilt `n()` function. We don't pass anything to the `n()` function because it simply counts the number of observations over which we've summarised our data.

You don't need to understand the mutating part of this chain, but just know that it replaces the first cell in time with an NA.

```
demo_sub %>%
  summarise(mean_time = mean(time),
            sd_time = sd(time),
            N = n()
            )
```

```
## # A tibble: 1 x 3
##   mean_time      sd_time     N
##   <drtn>           <dbl> <int>
## 1 1.447538 hours   0.275    29
```

Finally, we have to be wary of NAs when calculating our statistics with `summarise()`. Here, I'll introduce a missing value to our data before we calculate our summary.

```
demo_sub %>%
  mutate(time = replace(time, 1, NA)) %>%
  summarise(mean_time = mean(time),
           sd_time = sd(time),
           N = n()
           )
```

```
## # A tibble: 1 x 3
##   mean_time sd_time     N
##   <drtn>      <dbl> <int>
## 1 NA hours       NA    29
```

dplyr is strict, in that it will give you all NAs for the summary of a value if that value contains NAs. This is good as you'll be aware if you have missing data before you report anything. To suppress this behaviour, we simply have to ask R to remove NAs when calculating the mean and standard deviation.

To remove NAs, we need to explicitly set `na.rm = TRUE` in the functions we use. Read this as "Should I remove NAs? Yes!".

```
demo_sub %>%
  mutate(time = replace(time, 1, NA)) %>%
  summarise(mean_time = mean(time, na.rm = TRUE),
            sd_time = sd(time, na.rm = TRUE),
            N = n()
            )
```

```
## # A tibble: 1 x 3
##   mean_time      sd_time     N
##   <drtn>           <dbl> <int>
## 1 1.451815 hours   0.280    29
```

5.9 Grouping Data

One of the most convenient functions in **dplyr** is the `group_by` function. This plays well with a number of the functions we've looked at above, but it's most useful when calculating summaries.

In this experiment, we cared about reaction times for those who are native and non-native English speakers. But did their completion times differ at all? To save on splitting our data and calculating summaries on the subsets of data, we can instead group our data by the language spoken, and pass these both to the `summary()` function. We do this using `group_by()`.

```
demo_sub %>%
  group_by(language) %>%
  summarise(mean_time = mean(time, na.rm = TRUE),
           sd_time = sd(time, na.rm = TRUE),
           N = n()
           )
```

```
## # A tibble: 6 x 4
##   language mean_time      sd_time     N
##   <chr>    <drtn>           <dbl> <int>
## 1 english  1.549861 hours  0.0473     2
## 2 English  1.417045 hours  0.266     11
## 3 ENGLISH  1.090833 hours  0.0326     2
## 4 Other    1.486389 hours  0.317      9
## 5 OTHER    1.327778 hours NA          1
## 6 <NA>     1.601111 hours  0.258      4
```

As you can see, we got a summary by all of the groups in our language column. Unfortunately for us here, R doesn't identify groups with the same name but different capitalisation as the same group. To fix this, we can chain the functions we've used above before calculating our descriptive statistics.

I'll fix the names by setting the language column values to the language column values, only with all of the text in lowercase (using `tolower()` from baseR). This effectively overwrites the old values with our new, improved naming scheme.

```
demo_sub %>%
  mutate(language = tolower(language)) %>%
  group_by(language) %>%
  summarise(mean_time = mean(time, na.rm = TRUE),
            sd_time = sd(time, na.rm = TRUE),
            N = n()
            )
```

```
## # A tibble: 3 x 4
##   language mean_time      sd_time     N
##   <chr>    <drtn>           <dbl> <int>
## 1 english  1.391259 hours   0.260    15
## 2 other    1.470528 hours   0.303    10
## 3 <NA>     1.601111 hours   0.258     4
```

At a glance, there doesn't seem to be much of a difference in the completion time for the English and Other groups. However, for the NA group, subjects seem to take a little longer than the other two groups. That's perhaps something we would look into if we wanted to analyse their data.

Finally, we can use `group_by()` along with other functions, such as filtering our data to the two participants with the lowest times from the language groups. Let's do that now.

To do this, we use another function from baseR called `rank()`. This rank orders all scores from the smallest to the largest value. Thus, if we group by language and filter to times with a rank of 1, we will filter to the lowest time in our language groups.

```
demo_sub %>%
  mutate(language = tolower(language)) %>%
  group_by(language) %>%
  filter(rank(time) == 1) %>%
  glimpse()
```

```
## Rows: 3
## Columns: 11
## Groups: language [3]
## $ ID        <chr> "27", "30", "J"
## $ language  <chr> "english", NA, "other"
## $ progress  <chr> "END", NA, "END"
## $ gender    <chr> "female", "female", "female"
## $ age       <dbl> 44, 32, 28
## $ tester    <chr> "GW", "GW", "RHB"
## $ fun       <chr> "5", "1", "6"
## $ recommend <chr> "no", "yes", "yes"
## $ start     <dttm> 2018-03-21 03:23:57, 2018-03-22 04:16:08, 2018-03-21 04:15:5~
## $ end       <dttm> 2018-03-21 04:28:01, 2018-03-22 05:29:05, 2018-03-21 05:20:1~
## $ time      <drtn> 1.067778 hours, 1.215833 hours, 1.073056 hours
```

You can see that we have times of around 1 hour for each group, which is below the means we calculated before.

5.9.1 Ungrouping Data

Finally, if you want to perform further operations on a set of data after you've initially grouped it, you can ungroup the data using `ungroup()` prior to running further calculations.

Below, we remove the quickest people in each group before calculating the grand mean.

```
demo_sub %>%
  mutate(language = tolower(language)) %>%
  group_by(language) %>%
  filter(rank(time) != 1) %>%
  ungroup() %>%
  summarise(mean_time = mean(time, na.rm = TRUE),
            sd_time = sd(time, na.rm = TRUE),
            N = n()
            )
```

```
## # A tibble: 1 x 3
##   mean_time      sd_time     N
##   <drtn>           <dbl> <int>
## 1 1.485459 hours   0.264    26
```

5.10 Chaining Many Functions

As you saw before, our data wasn't exactly tidy when we first loaded it. We can avoid a lot of intermediate steps, and make some of the processes we've went through above if we just chain our functions together. Here, we'll tidy up our data before calculating some descriptive statistics.

```
# load package
library(lubridate)

# load, clean, and transform data
demo_clean <- read_csv("inputs/lexical_decision_demographic_data_wide.csv") %>%
  separate(col = funRec, into = c("fun", "recommend")) %>%
  separate(col = completion_time, into = c("start", "end"), sep = "_") %>%
  select(c(1, 8:16)) %>%
  rename(language = LANGUAGE) %>%
  mutate(start = ymd_hms(start),
         end = ymd_hms(end),
         time = end - start,
         language = tolower(language)
         )
```

In one step, we've used several of the commands that we used above to clean our data and to get it in the correct format for what we want to do with it. We can then pass this cleaned data to our `group_by()` and `summarise()` functions to generate summaries as before.

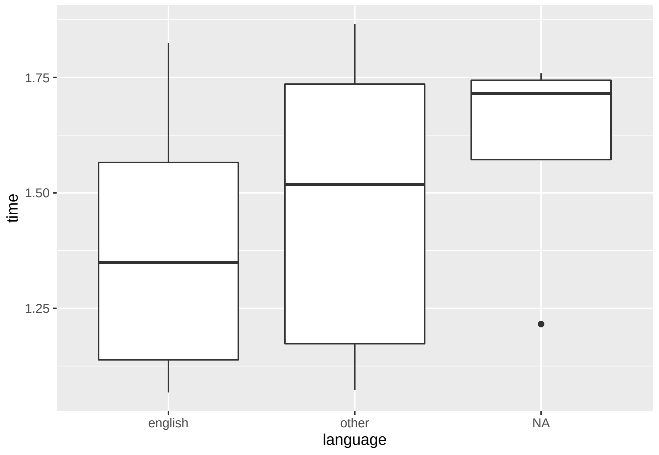
```
demo_clean %>%
  group_by(language) %>%
  summarise(mean_time = mean(time, na.rm = TRUE),
           sd_time = sd(time, na.rm = TRUE),
            N = n()
            )
```

```
## # A tibble: 3 x 4
##   language mean_time      sd_time     N
##   <chr>    <drtn>           <dbl> <int>
## 1 english  1.391259 hours   0.260    15
## 2 other    1.470528 hours   0.303    10
## 3 <NA>     1.601111 hours   0.258     4
```

Or we can even plot the data using the pipe (%>%) with **ggplot2**. Notice that the ggplot elements end the line in a + not a %>%.

```
demo_clean %>%
  ggplot(mapping = aes(x = language, y = time)) +
  geom_boxplot()
```

```
## Don't know how to automatically pick scale for object of type difftime. Defaulting t
```



There are some helper functions we haven't covered here that might help you out with **dplyr**, but you should have most of what you need now to do the majority of your data processing tasks.

5.11 Saving Data

Finally, we should look at saving our data. There are a few ways we can do this.

5.11.1 CSV Files

One way is to save your data as a .csv file, in a similar format to the data you loaded into R. However, this way we lose any customised data types (such as start and end times being POSIXct) if we are to load the data back up into R. Still, it's always good to have a copy of your data that is platform agnostic, so we should save it as a .csv so other people can load it up with any program they like. This format is also good for exploring your data in excel.

We'll save our data in the outputs folder as a csv, using the `write_csv()` function from **readr** (Wickham et al., 2017b), a part of the **tidyverse** (Wickham, 2017).

To save the data, we just need to specify the object to save (the table of your data) in the `write_csv()` function, along with the path to save the data. Here, we've specified that we want to save the data in the ouputs folder, and then given it a name. Folders are separated by a forwardslash, with the last string of text representing the file name. File names must end with .csv to specify the file type.

```
write_csv(demo_clean, "outputs/filtered_demographic_data.csv")
```

We could also save the data as an excel file, but it's generally best to keep platform agnostic for your data backups. If you want to save your data in a useful format for further manipulation, save it as an RData file! That way, you can keep using R to work with your objects.

5.11.2 R Data Files

You don't want to re-run all of your pre-analysis code every time you want to make a graph/model data, so saving your data after cleaning it up is generally a good idea. However, some of the R-specific information can get lost when saving as a .csv or other format (e.g. dates will be re-loaded as characters unless otherwise specified). Instead, we can save as an RData file which retains this information. Additionally, RData files generally take up less space on your hard drive than a .csv. Finally, we can save object types which don't play well with .csv and other formats in an RData file, such as model outputs and nested data sets (which is pretty advanced; think of a table of data within each cell of a larger table).

Here, we will save our data as an .RData file, which we can then load up again in R for further processing. To do so, we need to specify the object(s) to save (we can save several in one data file!), and we need to specify the file as above. Here, we need to be specific by using the `file` argument to define where to save our data.

```r
save(demo_clean, file = "outputs/filtered_demographic_data.RData")
```

When it comes to loading RData files back up, we just need to use the `load()` function from baseR. Bear in mind that the names of the objects you've saved in the data file will be the same ones you see on loading, so use informative names! Here, we just need to specify the path from which to load our data.

```r
load("outputs/filtered_demographic_data.RData")
```

5.12 Exercises

We'll try some exercises to get you used to using all of the functions we've discussed here.

For these exercises we'll a simulated data set I created in the **data_generation** folder which is saved to the inputs folder as `sim_data.RData`. This data was saved as a tibble in the RData file, so we just use `load()` to open this data into R.

```r
# load libraries
library(tidyverse)
library(lubridate)

# load data
load("inputs/sim_data.RData")
```

This data set looks at reaction times in spotting a target when people have had caffeine or not, and when they respond with their dominant or non-dominant hand. This data set is entirely made up, so we can't be sure how realistic these results are.

First, you should take a look at your data to understand it. A simple first step is to see what the data table itself looks like.

```r
glimpse(data)
```

```
## Rows: 100
## Columns: 5
## $ subject  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18~
## $ age      <int> 34, 56, 23, 53, 44, 21, 55, 47, 29, 31, 35, 56, 34, 62, 56, 2~
## $ caffeine <fct> no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, n~
## $ response <fct> non-dominant, non-dominant, non-dominant, non-dominant, non-d~
## $ DV       <dbl> 398.6627, 396.3824, 400.1234, 401.9182, 397.6403, 398.8435, 3~
```

After this, we should make a more detailed plot to understand the general trends in the data.

5.12.1 Question 1

Make a plot of your data which looks at the relationship beween **caffeine** and **DV**. Split this plot into two to show the differences across the two response conditions.

What are the general trends in the data?

5.12.2 Question 2

Subset your data to remove the ages from the data set.

5.12.3 Question 3

Rename the **DV** column to something more informative, like **reaction_time**

5.12.4 Question 4

What if we care about differences in the ages? Let's assume we have a prediction that caffeine only has an effect on those above 30 years of age. Subset your data to just those above 30 years of age.

5.12.5 Question 5

Rearrange the data set by age, starting from the **highest** age.

5.12.6 Question 6

Calculate mean centered scores for each subject and add these to a new column called **DV_c** (DV, centered). The formula for this is `subjects_score - mean(all_scores)`. Can you work out what the mean should (approximately) be?

5.12.7 Question 7

Let's assume we have a prediction that response times should be slower above 30 years of age. Create a new column, **age_group** that puts participants into two groups **30_or_under**, and **above_30**.

Hint: Look up the `ifelse()` function using `?ifelse()` to see how you can use logical operations to achieve this.

5.12.8 Question 8

Calculate the mean, standard deviation, and number of observations for each group.

5.12.9 Question 9

Calculate the mean, standard deviation, and number of observations for each group, excluding those with the 3 highest ages from each group.

5.12.10 Question 10

Take the data and do this all together:

1. Rename the DV column to **response_time**.
2. Remove any observations where the age of the participant is above 60.
3. Combine the two columns, age and caffeine, into one column called condition. Hint: Use `paste()` here. Use an underscore separator for the condition names.
4. Remove the caffeine and response columns and reorder the data so we have subject first, followed by age, condition, and response_time.
5. Calculate mean, standard deviation, and n for condition on the response time column. Call your new column names anything sensible.

5.12.11 Question 11

Was there any point in us combining the two factors into a single condition column? Do the same process above without making a summary of the data. Feed this data into a boxplot, with condition on the x-axis, and response_time on the y-axis.

Chapter 6

Simple Statistical Tests

In this session, we'll cover some of the most basic tests that you might use in your research. While we'll focus on selecting different tests for different scenarios, at the end of this section we'll focus on the **general linear model**, and how you can specify almost any relationship using this framework. In this session, we will cover unifactorial tests with only 2 levels in a factor. In the next session, we will cover more complicated designs.

Specifically, we'll cover:

- correlations
- one-sample, paired, and independent *t*-tests
- one-way ANOVA
- simple linear regression

Throughout, we will calculate descriptive statistics for our data.

This section assumes some familiarity with basic statistics. It is beyond the score of this section to teach the basics behind all of these methods, but we will cover some background on how these methods work, and why you might choose to use them in different scenarios.

6.1 Getting Started

As always, we first need to load the `tidyverse` set of packages for this Chapter.

```
library(tidyverse)
```

In this session, we'll use simulated data, and data from real studies using the the Open Stats Lab (OSL) resources (hosted on my GitHub account). For each section/analysis, we'll just load the data as and when we need it.

6.2 Correlation

We use correlations when we want to see if there's a significant relationship between some observed variables.

More specifically, we have to know a little bit about variance, covariance, and correlation coefficients to grasp correlation and other tests.

Variance tells you how much your data is spread out from the mean, it's the average of the squared differences from the mean. Why do we square the differences? That's so negative values don't cancel out positive values, so we can still have an idea of how much our scores vary.

No variance (i.e. all values are the same) is 0, and larger numbers represent more variance in your data. The square root of the variance is the standard deviation, which we can use to get exact distances of our data from the mean. Your mean +- your standard deviation multiplied by 1.96 represents 95% of the coverage of your data.

Covariance is similar to the variance, but rather than telling you how much one variable varies, it tells you how much two variables vary together. If we have a positive number for the covariance, we know that our variables are related to one-another. However, covariance is sensitive to the scale you have, so if you have big numbers for each variable, your covariance will also be a big number. This magnitude doesn't tell you much about how strong the relationship is between two variables. FOr that, we need the correlation coefficient.

The **correlation coefficient** is the covariance divided by the standard deviation. This just puts the covariance on a standard scale between -1 and 1, so we now have an idea of how strongly related two variables are. A correlation of 1 shows that the two are perfectly positively correlated, so an increase in one variable is followed by an increase in the other. A correlation of 0 is no relationship, and a correlation of -1 is a perfect negative relationship, where an increase in one variable is followed by a decrease in the other.

6.2.1 Analysing Real Data

In this example, we'll look at a study by Dawtry, Sutton, and Sibley (2015). In this study, they were interested in why people's attitudes towards increasing wealth inequality differs within developed nations. They hypothesised that wealthy people will be less supportive of wealth distribution if their social circle is primarily made up of wealthy people. This is because they should see society as being already wealthy, and thus not requiring of wealth distribution.

Our aim here is to test the simple hypothesis that people who see society as fair and are thus satisfied with the current state of affairs will less supportive of wealth redistribution.

We'll load and clean up the data set from the Open Stats Lab. For simplicity, we'll load this data directly from a csv file stored on GitHub. (To get a link for this, you must view the raw file on GitHub online.)

First, we'll load the data using `read_csv()`, then we'll recode the reverse-scores questions of **redist2** and **redist4** so all of the questions assessing redistribution are on the same scale.

Then we'll transmute the data, keeping only the columns we need, and calculating the mean scores for perceptions of fairness and satisfaction with distribution of wealth in society (i.e. one factor), and mean support for redistribution (our second factor). During this process, we'll also keep the participant IDs. Finally, we'll make everything lowercase, because it looks better!

```
# variables
scale_max <- 6

# load and clean data
corr_data <- read_csv(
  "https://raw.githubusercontent.com/gpwilliams/r4psych/master/lesson_materials/06_simple_statist
  ) %>%
  mutate(redist2 = (scale_max + 1) - redist2,
         redist4 = (scale_max + 1) - redist4) %>%
  transmute(PS,
            Fair_Satisfied = (fairness + satisfaction) / 2,
            Support = (redist1 + redist2 + redist3 + redist4) / 4
            ) %>%
  rename_all(tolower)
```

```
## Rows: 305 Columns: 37
## -- Column specification ------------------------------------
## Delimiter: ","
## dbl (37): PS, PD_15, PD_30, PD_45, PD_60, PD_75, PD_90, PD_105, PD_120, PD_1...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
corr_data
```

```
## # A tibble: 305 x 3
##       ps fair_satisfied support
##    <dbl>          <dbl>   <dbl>
##  1   233            1      5.5
##  2   157            3.5    3.25
##  3   275            5      3.75
##  4   111            7      2.75
## 5     52             4.5    3
## 6     11             2.5    3.75
## 7     76             3      4.25
## 8     90             4.5    3.5
## 9     93             4      3.75
## 10   104             4.5    3.5
## # ... with 295 more rows
```

Next, we'll calculate some means and standard deviations for these two factors of interest. We've used the `summarise_at()` version of the `summarise()` function, as this allows us to pass two column names, and a list of functions by which to produce our statistics. We can either just pass the functions in a list, or as here give names to these functions which will be appended to the names of the variables in the table.

```
corr_data %>% summarise_at(
  c("fair_satisfied", "support"),
  list(mean = mean, sd = sd)
)
```

```
## # A tibble: 1 x 4
##   fair_satisfied_mean support_mean fair_satisfied_sd support_sd
##                 <dbl>        <dbl>             <dbl>      <dbl>
## 1                3.54         3.91              2.02       1.15
```

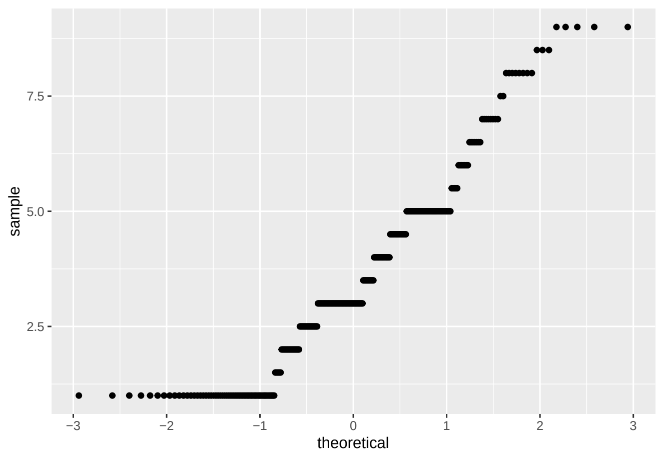
6.2.2 Checking Assumptions

To decide which correlation to run, we need to check some assumptions: are the factors linearly related, and do they follow a normal distribution (assumption of normality)? We'll leave exploring the first assumption for other tests, just to save time, but we'll look into testing the assumption of normality here.

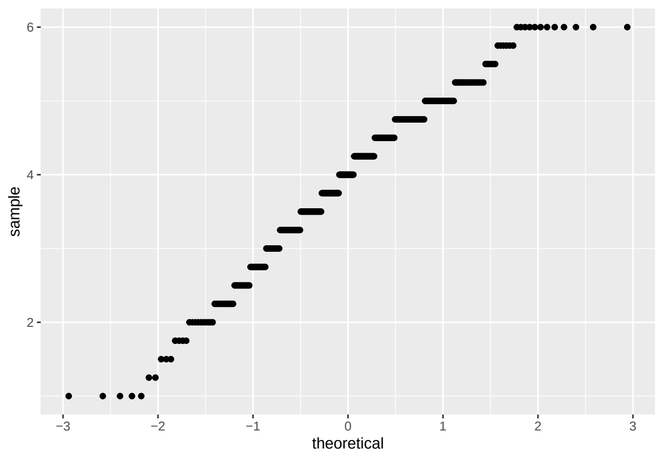
We have two methods to check for normality:

1. Visual inspection of the quantile-quantile plot which plotsthe correlation between our sample (observed values) and the normal distribution.
2. The Shapiro-Wilk test, which is a statistical test of normality. Significant results indicate that the data are **non-normal**.

```
corr_data %>%
  ggplot(aes(sample = fair_satisfied)) +
    geom_qq()
```



```
corr_data %>%
  ggplot(aes(sample = support)) +
    geom_qq()
```



```
shapiro.test(corr_data$fair_satisfied)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  corr_data$fair_satisfied
## W = 0.92602, p-value = 3.697e-11
```

```r
shapiro.test(corr_data$support)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  corr_data$support
## W = 0.97622, p-value = 5.993e-05
```

Both the plot and test show that our data are non-normal. As such, we shouldn't use a parametric test such as Pearson's R, but we should use a non-parametric alternative. One option is Kendall's Tau, which rank orders our observations, in an attempt to account for these factors being being non-normally distributed.

6.2.3 Running a Correlation

Next, we'll run a correlation looking at whether there is any relationship between whether you perceive society as fair, and you're satisfied with , and your support for wealth redistribution.

Here, we'll use `cor.test()`, which takes as an argument two columns from a data.frame or matrix, which are your two factors to compare, and produces test statistics, including *t*-values, *p*-values, and 95% confidence intervals.

This test defaults to use Pearson's R, and to a two-sided test of significance (i.e. you had no predictions for the direction of effect), but you can change this to "greater", a positive relationship, or "less", a negative relationship, which will adjust *p*-values accordingly.

We'll use Kendall's Tau here. Note that you simply have to replace "kendall" with "pearson" if your data do not violate any test assumptions. Also, we'll specify that we had a directional hypothesis, where there was a negative relationship between our factors ("less").

```r
cor.test(corr_data$fair_satisfied, corr_data$support,
         method = "kendall",
         alternative = "less"
         )
```

```
##
##  Kendall's rank correlation tau
##
## data:  corr_data$fair_satisfied and corr_data$support
## z = -13.157, p-value < 2.2e-16
## alternative hypothesis: true tau is less than 0
## sample estimates:
```

```
##        tau
## -0.5473132
```

As you can see, we have a significant negative correlation between our two factors, indicating that as perceptions of fairness and satisfaction with society (in regards to wealth) increases, support for wealth redistribution decreases.

This is represented by the *p*-value, which is below 0.05 (but see reasons why you shouldn't mindlessly pick 0.05 as your *p*-value). Simply, the *p*-value tells you how often you might observe an effect as large or larger than you one you've found in the long run (i.e. infinite sampling) assuming the null hypothesis is true. If your *p*-value is below 0.05 by convention we assume that this is an acceptable false-positve rate, as we'd only wrongly accept our results as showing an effect when there isn't one present at a rate below 5%. Yes, Neyman-Pearson statistics is weird.

Kendall's Tau is a measure of the association between two factors. For Pearson's R (again, a measure of association), you should typically interpret the R value as R^2, which shows how the percentage of variation in one variable is explained by another.

6.3 *t*-tests

t-tests allow us to see whether two means are significantly different from one another. Here, we'll get a *t*-value and *p*-value. The *t*-value is the ratio of the difference between two groups of scores and the difference within the groups of scores. A larger *t*-value tells us the difference between the groups is large, and a smaller value tells us the difference is small (i.e. they are similar to one another). Nicely, a *t*-value of 10 tells us that the groups are 10 times as different from one another as they are within each other.

For this next analysis, we'll look at the **lexdec** data set from the `languageR` library that we've used in previous sessions. We'll do this to save time on filtering the data too much, as above. The only thing we have to do is to remove the subjects we simulated for the other exercises (i.e. 22-30), so we only have real data.

Can you understand the code below? Ask yourself, why might we use `as.character(22:30)` to state which subjects we should drop?

```
lexdec <- read_csv("inputs/lexical_decision_raw_data.csv") %>%
  filter(!subject %in% (as.character(22:30)))
```

6.3.1 One-Sample *t*-tests

Here, we want to assess whether the frequencies of words in the study (i.e. how often people use these words) differed significantly from some baseline value of word frequency.

For this, we'll use a one-sample *t*-test, which compares our scores to a known average value whether the variance of the population is not known.

We can assume that word frquency is calculated from many observations, and as such the frequency is an aggregate score, so it is appropriate to submit summaires of word frquency to a *t*-test.

6.3.1.1 Preparing the Data

First, we have to subset our raw data, which contains information on the subjects and items tested, and only get scores for the word frequencies.

```
lexdec_onesamp <- lexdec %>%
  group_by(word) %>%
  summarise(freq = mean(frequency))
```

Below, we'll use the `t.test()` function to run a *t*-test. Here, we input as the first value the data or scores we want to test, and in the second part, we specify the mean score in the population (mu) by which to compare the scores.

6.3.1.2 Running the Test

```
t.test(lexdec_onesamp$freq, mu = 4)
```

```
##
##  One Sample t-test
##
## data:  lexdec_onesamp$freq
## t = 5.2044, df = 78, p-value = 1.535e-06
## alternative hypothesis: true mean is not equal to 4
## 95 percent confidence interval:
##  4.463784 5.038436
## sample estimates:
## mean of x
##   4.75111
```

As we can see, there is a significant difference between the frequencies for our sampled words and our assumption for the mean word frequency for all words. Again, we get t and p values, as well as the 95% confidence interval for our scores.

6.3.2 Independent-Samples *t*-tests

Here, we'll explore a different question, which is does the native language of the speaker, English or "other", have any impact on reaction times to the task?

First, we have to prepare our data.

6.3.2.1 Preparing the Data

We need to create scores aggregated by subject, as our raw data contains information on each subject and the items they responded to. To do this, we just need to calcualte a mean score over all items for each subject in the study. We'll use the `group_by()` and `summarise()` functions from **dplyr** to do this.

Additionally, we'll calculate log-transformed mean reaction times. This is because reaction times are typically not normally distributed, so we should transform them to make them more normal prior to analysis. The log transformation is one way to achieve this.

```r
lexdec_ind <- lexdec %>%
  group_by(subject, native_language) %>%
  summarise(log_RT = mean(log(RT), na.rm = T))
```

```
## `summarise()` has grouped output by 'subject'. You can
## override using the `.groups` argument.
```

We'll quickly calculate some descriptives for our study to see how the two groups look.

```r
lexdec_ind %>%
  group_by(native_language) %>%
  summarise(mean_log_RT = mean(log_RT, na.rm = T),
            sd_log_RT = sd(log_RT, na.rm = T),
            n = n())
```

```
## # A tibble: 2 x 4
##   native_language mean_log_RT sd_log_RT     n
##   <chr>                 <dbl>     <dbl> <int>
## 1 English                6.33    0.0982    11
## 2 Other                  6.47    0.178      9
```

6.3.2.2 Running the Test

As our data are in a long format, we have to do the *t*-test using a formula, seeing as our log_RT column contains data for both conditions of the experiment. We simply specify our formula like this:

`dependent variable ~ condition`

The `~` (read: tilde) simply says that the thing on the left is a function of the thing on the right; i.e. our scores are a function of our conditions.

Additionally, we specify where the data is stored (lexdec_ind), and whether the data are paired (i.e. paired-samples test) or not (i.e. independent-samples test). Here, we specify `FALSE`, as we want an independent samples *t*-test.

```
t.test(log_RT ~ native_language, data = lexdec_ind, paired = FALSE)
```

```
##
##  Welch Two Sample t-test
##
## data:  log_RT by native_language
## t = -2.2213, df = 11.903, p-value = 0.0465
## alternative hypothesis: true difference in means between group English and group Ot
## 95 percent confidence interval:
##  -0.29139463 -0.00268428
## sample estimates:
## mean in group English   mean in group Other
##              6.326837              6.473876
```

We can see that there is a significant difference between the two groups. Inspection of the log-transformed means shows us that reaction times are generally quicker for English speakers than non-English speakers. This is unsurprising considering the lexical decisions were made for English words.

You might notice that the degrees of freedom here are not integers (i.e. whole numbers). That's because R defaults to using Welch's t-test, which does not make the assumption of homogeneity of variance (which is generally a good thing). Due to not making this assumption, the degrees of freedom and p-value are adjusted slightly.

6.3.2.3 Checking Assumptions

As with many tests *t*-tests make a number of assumptions, some of which are easily handled (e.g. are the data scalar or ordinal and randomly sampled with a large sample size?). However, these tests also assume that the data are normally distributed. We've done this for the correlation above, so we won't do it again here. However, a final assumption is homogeneity of variance, in which the variance across our samples should be roughly equal. To test for this, we can run a Barlett test for parametric data, or a Fligner test for non-parametric data (using `fligner.test()`).

Here, we'll run the Barlett test:

```
bartlett.test(log_RT ~ native_language, data = lexdec_ind)
```

```
##
##  Bartlett test of homogeneity of variances
##
## data:  log_RT by native_language
## Bartlett's K-squared = 2.9197, df = 1, p-value = 0.08751
```

We got a non-significant result from this test, which indicates that the variances are not significantly different from one another, and so we've met this test assumption.

Note that we should check this before running any analyses, but I didn't want to introduce the formula notation in assumption checks.

6.3.3 Paired-Samples *t*-tests

Paired-samples *t*-tests are run in the same way as independent-samples tests, only we need to set `paired` to `FALSE` within the *t*-test call.

We'll quickly simulate some data to show how this might work. Here, we'll simulate data for a stroop task, where subjects name words (colour names; e.g. green, blue, red) which are in different colours. Our hypothesis is that they will say the name more quickly if the word and colour are the same (congruent), and they will be slower if they mismatch (incongruent).

We'll generate 60 subjects, with the congruent reaction time being around 400ms, and the incongruent reaction time being on average 30ms slower than this.

Finally, we'll gather the data together, assuming you'll often have your data in a long format.

```
set.seed(1000)
stroop_dat <- tibble(
  subject = seq(1:60),
  congruent = rnorm(mean = 400, sd = 30, n = 60),
  incongruent = congruent + rnorm(mean = 30, sd = 10, n = 60)
  ) %>%
  gather(key = condition, value = reaction_time, 2:3) %>%
  mutate(
    log_RT = log(reaction_time)
  )
```

Note that if your data is in a wide format, you specify the model formula like so:

```
t.test(data$dependent_variable_one, data$dependent_variable_two, data = data_name, pai
```

In this format, you simply give the column names for where your data is stored for each condition, and R will compare the first entry against the second. However, I typically keep my data in long format, so we will mainly use the formula format for specifying our models. Using the formula notation is also useful for more complex designs where wide formatted data is unrealistic, so we'll stick to this as our default way to specify our models.

6.3.3.1 Running the Test

The paired-samples *t*-test formula takes the same form as the independent sampels *t*-test, only with `paired = TRUE`.

```
t.test(log_RT ~ condition, data = stroop_dat, paired = TRUE)
```

```
##
##  Paired t-test
##
## data:  log_RT by condition
## t = -25.573, df = 59, p-value < 2.2e-16
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -0.08379794 -0.07163601
## sample estimates:
## mean difference
##     -0.07771698
```

As with the independent-samples t-test, we get all of the same statistics in the output.

Here, there is a significant difference between the two conditions, with participants responding to items more quickly in the congruent when compared to the incongruent condition. This makes sense, as the name and ink colour both match, so there's no interference during processing.

6.4 ANOVA

Like the *t*-value, the F value from an ANOVA lets you know whether two groups of scores are similar or different from one another. More specifically, it gives you the ratio of the variance between two samples.

As ANOVAs and *t*-tests are both part of the general linear model family, the outputs of both tests are similar when we have only 2 levels of a condition. Compare the results of the following analyses with those from the *t*-tests aboves.

We use the `aov()` function in R to perform an analysis of variance. With the ANOVA and linear models we will explore later, we get a lot of information from the output just by running the `aov()` function. As such, we'll assign this to an object, which can just be a name in which to store our output of the model. After this, we use the `summary()` function, to get a neat summary of the statistics that we want to look at.

6.4.1 One-Way ANOVA

6.4.1.1 Indpendent-Samples ANOVA

We'll simply refit the same data from the `lexdec` data set to get an idea of how similar *t*-tests and ANOVAs are. Again, we just need to specify our DV to the left of the condition.

```r
anova_ind <- aov(log_RT ~ native_language, data = lexdec_ind)
summary(anova_ind)
```

```
##                 Df Sum Sq Mean Sq F value Pr(>F)
## native_language  1 0.1070 0.10702   5.523 0.0304 *
## Residuals       18 0.3488 0.01938
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As before, we have a significant difference between the two language groups on their log reaction times in the lexical decision task.

6.4.1.2 Paired-Samples ANOVA

For a paired-sampels ANOVA, we just need to specify our models as before, but we also need to include an error term. This error term takes the form of:

`Error(observation_unit/factor)`

We always need to specify that this is an error term we're adding, hence the name `Error()`. We then need to specify what our error term is. Since we're using a paired-samples test, where the same people ttake part in both conditions, we need to tell R that our error term is for the condition factor within subjects, i.e. different people might react differently to the conditions. Finally, as always we need to specify where our data lies.

Below, we'll just refit the stroop data from the *t*-test example above.

```
anova_paired <- aov(log_RT ~ condition + Error(subject/condition), data = stroop_dat)
summary(anova_paired)
```

```
##
## Error: subject
##           Df   Sum Sq  Mean Sq F value Pr(>F)
## Residuals  1 0.005198 0.005198
##
## Error: subject:condition
##           Df Sum Sq Mean Sq
## condition  1 0.1359  0.1359
##
## Error: Within
##            Df Sum Sq Mean Sq F value  Pr(>F)
## condition   1 0.0453 0.04535   8.302 0.00472 **
## Residuals 116 0.6336 0.00546
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As before, we have a significant difference between the two congruency conditions on their log reaction times in the lexical decision task.

6.5 Linear Regression

In this section, we'll look at linear regression. We would typically fit a linear model when we predict a linear relationship between one or more variable(s), e.g. X, and a continuous dependent variable, e.g. Y. This relationship for any one subject (or item) can be described as:

$$Y = \beta_0 + \beta_1 X + e$$

Where Y, the outcome or dependent variable, is calcualted by the individual's intercept value β_0 (i.e. typically the score at 0 on the x-axis of a graph), their slope term β_1, or how much to increase their Y value for every tick of the x-axis, multiplied by their value along the X axis (X), and an error term e, which is the residual error from our model.

The residual error (or simply residuals) show the difference between our predicted values from our model and observed values from our data. (Remember our QQ-plot? This looks at how your residuals are distributed. If they are normally distributed, then we're OK!)

We have this error term as we do not want to fit a perfect model to our data. If we did that, our model would work very well for our exact sample, but it wouldn't work so well for more generalised cases (e.g. with other samples). As such, we do not **overfit** our model so that it keeps some explanatory power for broader cases.

Like *t*-tests and ANOVAs, linear regression is simply another way for us to analyse our data. The general linear model is very useful in that it can be applied to the simplest and most complex cases of analyses, and really, it's likely this is all you'll need to know. Nicely, our predictors can be scalar data (e.g. quantifying the success of different doses of a drug) or categorical (e.g. different conditions; drug or no drug).

6.5.1 General Linear Model

We use the general linear model when we have outcome data that is a continuous variable. However, this procedure is not appropriate when we have bounded data (e.g. Likert scales, binomial (0 or 1) responses). In these cases, we have to use a more specific form of the model, called the **generalised linear model**, which we'll look at after this section.

6.5.1.1 Independent-Samples Linear Model

As with the *t*-test and ANOVA, we'll use the `lexdec_ind` simulated data set from before.

6.5.1.1.1 Running the Test As with the ANOVA, we need to specify our model using the formula, with our dependent variable to the left of the `~`, and our conditions to the right. Here, we'll assign this to the object `lm_ind`, and we'll get a summary of the data using the `summary()` function.

```
lm_ind <- lm(log_RT ~ native_language, data = lexdec_ind)
summary(lm_ind)
```

```
##
## Call:
```

```
## lm(formula = log_RT ~ native_language, data = lexdec_ind)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.22075 -0.07276 -0.00893  0.07599  0.33469
##
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)           6.32684    0.04197  150.75   <2e-16 ***
## native_languageOther  0.14704    0.06256    2.35   0.0304 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1392 on 18 degrees of freedom
## Multiple R-squared:  0.2348, Adjusted R-squared:  0.1923
## F-statistic: 5.523 on 1 and 18 DF,  p-value: 0.03037
```

Nicely, we get a lot of information from this model. As with the ANOVA, we have a test statistic (this time a t-value) and a p-value, under the heading `Pr(>|t|)`.

The t-value is the intercept term for our condition, divided by the standard error. The p-value here is the proportion of the t distribution which is greater than the t statistic. The p-value is just the proportion of the t distribution which is greater than the value of the t statistic (hence the weird notation).

We also get some information that is more informative than the ANOVA. We get our intercept term (`(Intercept)`), which here is mean score for the English language group. Let's see how that matches our mean scores:

```
lexdec_means <- lexdec_ind %>%
  group_by(native_language) %>%
  summarise(mean = mean(log_RT))
lexdec_means
```

```
## # A tibble: 2 x 2
##   native_language  mean
##   <chr>           <dbl>
## 1 English          6.33
## 2 Other            6.47
```

Under this, we have the difference from this intercept for the Other language group (`native_languageOther`). How does this relate to the means? Below, we'll subtract the congruent scores from the incongruent scores.

```
lexdec_means$mean[2] - lexdec_means$mean[1]
```

```
## [1] 0.1470395
```

As you can see, this value relates to the score for `native_languageOther`, which shows how the Other language group differs from the baseline, English language group.

You might ask now, how did R know which condition to set as the intercept? R defaults to going alphabetically, so English comes before Other, hence it is the baseline. In simple cases like this, where we have only 2 levels, this is not a concern. But when we have more levels or factors, we need to pay attention to how we specify our contrasts in R.

Finally, in the output we have our residuals, including the residual standard error.

6.5.1.2 Paired-Samples Linear Model

As with the *t*-test and ANOVA, we'll use the `stroop_dat` simulated data set from before.

6.5.1.2.1 Running the Test As with the ANOVA, we need to specify our model using the formula, with our dependent variable to the left of the `~`, and our conditions to the right. Here, we'll assign this to the object `lm_ind', and we'll get a summary of the data using the`summary()‘ function. Unlike with *t*-tests and ANOVA, we do not need to specify an error term if our data are within-subjects (or paired-samples), so we simply use the same formula for independent- and paried-samples tests. Note that this means that our model doesn't account for how much of our outcome variable is predicted by subject-specific factors (i.e. variation between people). Some might respond more or less strongly to the effect of condition than others. As such, we can do a bit better with these models by fitting a mixed-effects model that can account for subject level effects. For now, we'll leave that for another lesson. If you want to get linear model output while accounting for by-subjects effects like in an ANOVA, head over to the next lesson!

```
lm_pair <- lm(log_RT ~ condition, data = stroop_dat)
summary(lm_pair)
```

```
##
## Call:
## lm(formula = log_RT ~ condition, data = stroop_dat)
```

```
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.153786 -0.049960  0.006172  0.039194  0.202953
##
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)          5.978613   0.009499 629.391  < 2e-16 ***
## conditionincongruent 0.077717   0.013434   5.785 6.07e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.07358 on 118 degrees of freedom
## Multiple R-squared:  0.221,  Adjusted R-squared:  0.2144
## F-statistic: 33.47 on 1 and 118 DF,  p-value: 6.066e-08
```

As before, we get our intercept term (`(Intercept)`), which here is mean score for the congruent condition. Let's see how that matches our mean scores:

```
stroop_means <- stroop_dat %>%
  group_by(condition) %>%
  summarise(mean = mean(log_RT))
stroop_means
```

```
## # A tibble: 2 x 2
##   condition    mean
##   <chr>       <dbl>
## 1 congruent    5.98
## 2 incongruent  6.06
```

Under this, we have the difference from this intercept for the congruent condition (`conditionincongruent`). How does this relate to the means? Below, we'll subtract the congruent scores from the incongruent scores.

```
stroop_means$mean[2] - stroop_means$mean[1]
```

```
## [1] 0.07771698
```

As you can see, this value relates to the score for `conditionincongruent`, which shows how the incongruent condition differs from the baseline, congruent condition.

Aside from this, we have a whole host of other information that we might want to consider, including an adjusted R^2, which tells us about how much our explanatory predictors (e.g. condition) accounts for changes in the dependent variable. For now, this is all we need.

6.5.1.3 Assumptions

Simple general linear models make a few assumptions, of which:

- Linear relationship: your variables must be linearly related, otherwise a linear fit doesn't make sense
- Normality: Variables must be normally distributed
- Homoscedasticity (homogeneity of variance): your residuals are equal across the regression line
- No auto-correlation: your observations for your dependent variable must not impact one-another (e.g. eye-movements; if you get one success (i.e. a look at an object), the next eye-movement is likely also be a success (i.e. still looking) – this is bad!)

We've already looked at checking for normality and homoscedasticity, so we won't do this again here. We can also decide about auto-correlation prior to running our analyses, so if you understand your data we may not need to check this assumption. But we should probably check for a linear relationship. The best way to do this is to just eye-ball the data with a plot. This makes sense if your predictor is on a scale, but it's not really defined when we have groups, so we'll leave that for now.

6.5.2 Generalised Linear Model

What about cases in which we have a binomial response, e.g. if people got an answer correct or not? Here, we have to use a generalised linear model. This takes the same form as the general linear model, only these models use a link function, which allows the dependent variable to be a function of some error term other than the normal distribution.

We won't go into detail as to how the generalized linear model works, but just note that you can specify different link functions to describe your data, so for example if your data take a log distribution, you can specify a logit link.

We'll focus on a common case of fitting a logistic model to our data using a binomial distribution.

6.5.2.1 Preparing the Data

Below, we'll simulate some data looking at the effectiveness of a smoking intervention on smoking cessation rates. We have a simple outcome, 0 if people still smoke after the experiment (i.e. a failure), and 1 if they quit smoking (i.e. a success).

```r
set.seed(1000)
prob <- rep(c(0.3, 0.6), 50)
cond <- rep(c("control", "intervention"), 50)
subject <- seq(1:100)

smoking_dat <- tibble(
  subject = subject,
  cond = cond,
  outcome = rbinom(n = 100, size = 1, prob = prob)
)
```

Let's see what the rates of smoking look like:

```r
smoking_dat %>% group_by(cond) %>% summarise(mean = mean(outcome))
```

```
## # A tibble: 2 x 2
##   cond          mean
##   <chr>        <dbl>
## 1 control       0.24
## 2 intervention  0.54
```

Great! It looks like our intervention worked well. If only it was that easy in real life…

6.5.2.2 Running the Test

The generalised linear model is specified using the `glm()` function. Here, we can specify a few extra details about our model or accept the defaults. We'll overspecify to show these things work. Here, we've asked to fit the model using a binomial link function, using the `family = "binomial"` argument since our data come from a binomial decision and follow a binomial distribution.

```r
glm_ind <- glm(outcome ~ cond, data = smoking_dat, family = "binomial")
summary(glm_ind)
```

```
##
## Call:
## glm(formula = outcome ~ cond, family = "binomial", data = smoking_dat)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.2462  -0.7409  -0.7409   1.1101   1.6894
##
```

```
## Coefficients:
##                  Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -1.1527     0.3311  -3.481   0.0005 ***
## condintervention   1.3130     0.4361   3.011   0.0026 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 133.75  on 99  degrees of freedom
## Residual deviance: 124.10  on 98  degrees of freedom
## AIC: 128.1
##
## Number of Fisher Scoring iterations: 4
```

Again, we get a similar output to the linear model, but we've fit the data using a more appropriate model to account for the binary nature of the data. Notice that now we have z-values instead of *t*-values. Aside from that, all of your other interpretations remain the same.

6.5.2.3 Proportion Data

We can also fit models with proportion data for the dependent variable using the `glm()` function. To do so, you need the number of successes and the number of observations, specifying your model like so:

```
glm(successes/observations ~ cond, data = smoking_dat, family =
"binomial")
```

Or, if you just have the proportion and number of observations, you can fit it like so:

```
glm(proportion ~ cond, data = smoking_dat, family = "binomial",
weights = observations)
```

In this last case, you just specify to weight successes by the number of observations that you had for each data point. This weighting is important when you do more advanced analyses, like growth curve analysis with transformations on your data (e.g. empirical logit) as you can tell your model how much to "trust" each data point. But for our cases so far, this is all we need to know about the weights argument.

6.6 Exercises

If you don't have access to these, please download the repository from GitHub and open the **lesson_materials** folder. Open the relevant folder for this lesson.

If you work out of the file **06_simple_statistical_tests.Rmd** the code below will work to load the data set. If this fails then quit R and open it up from the .Rmd file above so your working directory is in the correct folder.

This data was saved as a .csv, so we need to use `read_csv()` to load the data into R.

```
library(tidyverse)
```

Next, we'll load all of the data sets.

```
binom_b_data <- read_csv("inputs/binom_between_data.csv")
binom_w_data <- read_csv("inputs/binom_within_data.csv")
bsubj_data <- read_csv("inputs/bsubj_data.csv")
wsubj_data <- read_csv("inputs/wsubj_data.csv")
corr_data <- read_csv("inputs/corr_data.csv")
```

6.6.1 Question 1

Using the `corr_data` data set, prepare to run a correlation for the height and weights of the participants. First, you have to check all of the assumptions are met for a parametric test. Do a visual inspection of the normality of the variables, and check that the two are linearly related.

6.6.2 Question 2

Using the information from Question 1, run a correlation for the height and weights of the participants. Adjust to a non-parametric test if necessary.

What is the correlation between the two variables? Can you work out the R^2? What does this tell us?

6.6.3 Question 3

Aggregate the scores of the `bsubj_data` data set by subject as an object called `bsubj_agg`. Output the mean scores aggregated by subject.

Then run a t-test between the two groups of A. Is there a significant difference between the two conditions?

6.6.4 Question 4

Check that a t-test was appropriate for question 3. We'll only check for **homogeneity of variance** here. What did the test show? Was a t-test appropriate?

6.6.5 Question 5

Aggregate the `wsubj_data` by subjects and save this as the object `wsubj_agg`. Then submit this aggregated data to an ANOVA, saving the output as the object `wsubj_aov`. What does the ANOVA show us?

6.6.6 Question 6

We should probably allow people to interpret our inferential tests with plots and descriptive statistics.

First, make a plot of the two conditions from Question 5, using your `wsubj_agg` data set. Then, output a table of means, standard deviations, and standard errors for the two groups. To calculate the standard error you need to take the standard deviation of the dependent variable divided by the square root of number of observations for the dependent variable (i.e. `sd(output)/sqrt(length(output))`).

6.6.7 Question 7

Fit a linear model to the `wsubj_agg` data and save this as `wsubj_lm`. How does this compare to the ANOVA from question 5? Compare the results of the parameter estimates from the linear model to that of your plot. How do these parameters compare?

6.6.8 Question 8

Fit a linear model to the Using the `binom_b_data`, and save this output as `binom_b_lm`. Look at the summary of the output. Is there a significant difference between the two groups?

6.6.9 Question 9

Fit a linear model to the Using the `binom_w_data`, and save this output as `binom_b_lm`. Look at the summary of the output. Is there a significant difference between the two groups? You will first have to convert the data to long format in order to use the formula syntax for your model. For this, set the key to "test_time", and the value to "success".

6.6.10 Question 10

Using the long-formatted `binom_w_data` from question 9, create a table containing the calculate means, standard deviations, standard errors, and 95% confidence intervals for the two test times.

In order to calcualte the upper and lower bounds of the 95% confidence interval, you will have to take the mean of the dependent variable - 1.96 times the **standard error** of the mean, and the mean of the dependent variable + 1.96 times the standard error of the mean respectively. Call these columns `lower_confint` and `upper_confint` respectively.

Next, plot a bar plot of this data with standard error bars.

Compare your hand-coded standard errors with those from ggplot. How do they compare? Do the bounds seem similar?

Chapter 7

Advanced Statistical Tests

In this session, we'll cover more advanced statistical tests, such as multifactorial experiments and experiments with factors of more than 2 levels.

Specifically, we'll cover:

- multilevel and multifactorial ANOVA
- multilevel and multiple regression
- factor coding (and why it matters in R)
- sum of squares (and why this matters in R)

As with the previous section, this section assumes some familiarity with basic statistics. It is beyond the score of this section to teach the basics behind all of these methods, but we will cover some background on how these methods work, and why you might choose to use them in different scenarios.

7.1 Getting Started

As always, we first need to load the `tidyverse` set of packages for this Chapter.

```
library(tidyverse)
```

7.2 Multilevel Analyses

Often, your experiments will have more than one level for a factor. However, how you code these factors determines the answers that you get from your analyses.

Take for example evaluating the efficacy of a host of drugs. We often want to compare performance of several drugs against a placebo. In this case, we might perform an analysis where we compare different levels of one factor (drug; with placebo, drug one, and drug two levels) against a reference level (drug level one; placebo). In this case, we would use **treatment** coding for our factors, as we only want to know if drug one and drug two perform differently from the placebo. That is, we are interested in the **simple effects** of the drug.

However, in other instances we might want to detect a main effect of a factor to see if there is an overall effect of condition, and you want to explore which conditions differ after the fact. In this case, we might want to use **deviation** or **sum** coding for our factors, and as such we want to test for the **main effect** of condition.

To explore main effects for the ANOVA and for the multilevel regression, we'll simulate some data. We'll use a similar (but more complex) example from the previous lesson.

We'll simulate data for a numerical stroop task, where subjects need to pick the highest value number from two numbers. Crucially, these numbers can be displayed such that the highest number is biggest in font size (congruent), smallest in font size (incongruent), or the two numbers have their fonts at an equal size (neutral). Our hypothesis is that participants will identify the larger number most quickly if the size and magnitude of the number of congruent, slower if they are neutral, and slowest if they are incongruent.

We'll generate 60 subjects, with the congruent reaction time being around 400ms, the incongruent reaction time being on average 60ms slower than this, and the neutral condition around 30ms slower than the congruent condition (and thus around 30ms faster than the neutral condition).

Finally, we'll gather the data together, assuming you'll often have your data in a long format.

```r
set.seed(1000)
stroop_dat <- tibble(
  subject = seq(1:60),
  congruent = rnorm(mean = 400, sd = 30, n = 60),
  incongruent = congruent + rnorm(mean = 60, sd = 10, n = 60),
  neutral = congruent + rnorm(mean = 30, sd = 10, n = 60)
  ) %>%
  gather(key = cond, value = reaction_time, 2:4) %>%
  mutate(
    cond = as.factor(cond),
    log_RT = log(reaction_time)
  )
```

To explore simple effects, we'll create some data that looks at the effects of a drug intervention on blood oxygen levels (outcome) for smokers. We have a control condition (placebo) where the oxygen level is on average 80mm Hg, “drug_one”, which on average should be no different to the placebo, and “drug_two”, which increases blood oxygen to around 100mm Hg. Here, we simply want to know if either of the drugs perform any better than the control condition. (Note, I have no idea if these are realistic values, but that’s not the point of this exercise!)

```r
# set parameters
set.seed(1000)
means <- rep(c(80, 80, 100), 100)
sds <- rep(c(5, 5, 7), 100)

# simulate data
cond <- rep(c("control", "drug_one", "drug_two"), 100)
subject <- seq(1:300)

smoking_dat <- tibble(
  subject = subject,
  cond = as.factor(cond),
  outcome = rnorm(n = 300, mean = means, sd = sds)
)
```

7.2.1 Preparation of Data

7.2.1.1 Factors

In the above examples, we’ve created data sets where our condition column is stored as a factor. Take a look at this:

```r
stroop_dat
```

```
## # A tibble: 180 x 4
##    subject cond      reaction_time log_RT
##      <int> <fct>             <dbl>  <dbl>
##  1       1 congruent          387.   5.96
##  2       2 congruent          364.   5.90
##  3       3 congruent          401.   5.99
##  4       4 congruent          419.   6.04
##  5       5 congruent          376.   5.93
##  6       6 congruent          388.   5.96
##  7       7 congruent          386.   5.96
##  8       8 congruent          422.   6.04
##  9       9 congruent          399.   5.99
## 10      10 congruent          359.   5.88
## # ... with 170 more rows
```

```
smoking_dat
```

```
## # A tibble: 300 x 3
##    subject cond     outcome
##      <int> <fct>      <dbl>
##  1       1 control     77.8
##  2       2 drug_one    74.0
##  3       3 drug_two   100.
##  4       4 control     83.2
##  5       5 drug_one    76.1
##  6       6 drug_two    97.3
##  7       7 control     77.6
##  8       8 drug_one    83.6
##  9       9 drug_two    99.9
## 10      10 control     73.1
## # ... with 290 more rows
```

This is because I used `mutate()` to change the column from a character data type, to a factor data type. The nice thing about storing your factors this way is that we can see the levels that make up our factor:

```
levels(stroop_dat$cond)
```

```
## [1] "congruent"   "incongruent" "neutral"
```

```
levels(smoking_dat$cond)
```

```
## [1] "control"  "drug_one" "drug_two"
```

If your data isn't stored as a factor, you can convert it to one using `as.factor()` or `factor()` on your column (e.g. `data$factor <- as.factor(data$factor)`).

7.2.1.2 Contrast Matrices

We can see how we specify our contrasts for our tests by using the `contrasts()` function any any factors.

Let's look back at the Stroop data when we only had two levels of condition, congruent and incongruent. Here, we'll subset the data and then use the `contrasts()` function to look at the contrast matrix. We'll also mutate the cond column to itself as a factor so we drop the level that is no longer present (i.e. neutral) ; this is just a little R trick!

```
# make data 2 levels and convert condition to a factor
stroop_subset <- stroop_dat %>%
  filter(cond %in% c("congruent", "incongruent")) %>%
  mutate(cond = factor(cond))

# see your contrasts
contrasts(stroop_subset$cond)
```

```
##             incongruent
## congruent             0
## incongruent           1
```

We can see from our contrast matrix that the congruent condition is assigned a value of 0, and the incongruent condition is assigned a value of 1. What this means is that our intercept in our linear model is the congruent condition, and we evaluate the effect of a shift in condition by 1 point (i.e. to the incongruent condition) against this reference level. Remember that in a linear model, the intercept is the point on the y-axis where x is 0.

Let's fit a linear model to see how this works. Notice that we fit the model and then (%>%) use the `summary()` function to see the output of the model:

```
lm(log_RT ~ cond, stroop_subset) %>% summary()
```

```
##
## Call:
## lm(formula = log_RT ~ cond, data = stroop_subset)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.15379 -0.04888  0.00581  0.03781  0.19539
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      5.978613   0.009196  650.14   <2e-16 ***
## condincongruent 0.145808   0.013005   11.21   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.07123 on 118 degrees of freedom
## Multiple R-squared:  0.5158, Adjusted R-squared:  0.5117
## F-statistic: 125.7 on 1 and 118 DF,  p-value: < 2.2e-16
```

We can see that our parameter estimate for the intercept is 5.979, and being in the congruent condition adds 0.145 to the intercept value of 5.979, which sums to 6.124. How does this equate to our mean scores?

```
stroop_subset %>%
  group_by(cond) %>%
  summarise(mean = mean(log_RT))
```

```
## # A tibble: 2 x 2
##   cond         mean
##   <fct>       <dbl>
## 1 congruent    5.98
## 2 incongruent  6.12
```

Great, our intercept corresponds to the mean of the congruent condition, and our effect of condition is the difference between the two means! That's because we used treatment coding, which is the default in R. This means that the intercept is alwasy the first alphabetic level of our conditions.

But what if we want to change the reference level of our intercept? This is important if we want to look at different contrasts when we have more than 1 level in a factor, or more than 2 factors in a model.

To change our intercept, we must change our contrast matrix. To change this we use the `contrasts()` function on our condition factor, to see what our contrasts are, and we can then define the contrasts we'd like using some inbuilt functions in R. Here, we'll change the coding to sum coding using `contr.sum`. (Others are available, such as `contr.helmert`, etc.)

```
contrasts(stroop_subset$cond) <- contr.sum
contrasts(stroop_subset$cond)
```

```
##             [,1]
## congruent      1
## incongruent   -1
```

Now, our contrast matrix sums to 0. Remember that our intercept is 0, so the intercept is now the mid-point between the 2 conditions, or the mean across both conditions. Let's refit our model and see this in action.

```
lm(log_RT ~ cond, stroop_subset) %>% summary()
```

```
##
## Call:
## lm(formula = log_RT ~ cond, data = stroop_subset)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.15379 -0.04888  0.00581  0.03781  0.19539
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)    
## (Intercept)  6.051517   0.006502  930.65   <2e-16 ***
## cond1       -0.072904   0.006502  -11.21   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.07123 on 118 degrees of freedom
## Multiple R-squared:  0.5158, Adjusted R-squared:  0.5117 
## F-statistic: 125.7 on 1 and 118 DF,  p-value: < 2.2e-16
```

Now our intercept is 6.052, and the effect of condition (now labelled `cond1`; the first (and only) contrast in our contrast matrix for condition) is -0.073. Crucially, since we only have 2 levels of condition, our interpretation of all of the statistics in this model remains the same, only our intercept paramter estimate is different to before. Further, the parameter estimate for condition (and the standard error of this estimate) are half as large as before, but this is just due to the scale on which our factor is evaluated; the t- and p-values remain the same.

How does the intercept relate to our means? Check it out!

```
stroop_subset %>% summarise(mean = mean(log_RT))
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1  6.05
```

The mean for both conditions is the intercept for our linear model.

7.2.2 Multilevel Regression

What happens if we have some more complex data? Let's look at the contrast matrix for the full Stroop data:

```
contrasts(stroop_dat$cond)
```

```
##             incongruent neutral
## congruent             0       0
## incongruent           1       0
## neutral               0       1
```

You can see that right now, the first row for the congruent condition sums to 0. Whereas the incongruent and neutral condition rows sum to 1. This means that the congruent condition will be the intercept in our model, and the incongruent and neutral conditions will be the other parameters in our model. Let's see this in action.

```
lm(log_RT ~ cond, data = stroop_dat) %>% summary
```

```
##
## Call:
## lm(formula = log_RT ~ cond, data = stroop_dat)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.180561 -0.052211  0.002842  0.042382  0.195386
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      5.978613   0.009308 642.314  < 2e-16 ***
## condincongruent 0.145808   0.013163  11.077  < 2e-16 ***
## condneutral      0.073206   0.013163   5.561 9.74e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0721 on 177 degrees of freedom
## Multiple R-squared:  0.4094, Adjusted R-squared:  0.4027
## F-statistic: 61.35 on 2 and 177 DF,  p-value: < 2.2e-16
```

Again, we'll check how these parameters correspond to our means for each group.

```
stroop_dat %>% group_by(cond) %>% summarise(mean = mean(log_RT))
```

```
## # A tibble: 3 x 2
##   cond         mean
##   <fct>       <dbl>
## 1 congruent    5.98
## 2 incongruent  6.12
## 3 neutral      6.05
```

Great, the intercept is indeed the mean for the congruent condition, and the incongruent condition (cond2) and neutral (cond3) are the difference between the intercept mean and that reference level's mean.

Instead, we could get sum coded effects as before.

```
contrasts(stroop_dat$cond) <- contr.sum
contrasts(stroop_dat$cond)
```

```
##             [,1] [,2]
## congruent      1    0
## incongruent    0    1
## neutral       -1   -1
```

Now the first 2 rows sum to 1, which together makes 2, and the final row sums to -2. Together, the intercept or 0 is the average across these contrasts. How does this affect our model?

```
lm(log_RT ~ cond, data = stroop_dat) %>% summary
```

```
##
## Call:
## lm(formula = log_RT ~ cond, data = stroop_dat)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.180561 -0.052211  0.002842  0.042382  0.195386
##
## Coefficients:
##              Estimate Std. Error  t value Pr(>|t|)
## (Intercept)  6.051618   0.005374 1126.106   <2e-16 ***
## cond1       -0.073005   0.007600   -9.606   <2e-16 ***
## cond2        0.072803   0.007600    9.580   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0721 on 177 degrees of freedom
## Multiple R-squared:  0.4094, Adjusted R-squared:  0.4027
## F-statistic: 61.35 on 2 and 177 DF,  p-value: < 2.2e-16
```

Now we can see that the intercept has changed, and now the congruent and incongruent condition are considered against the grand mean across all levels, so their terms have changed too.

```
stroop_dat %>% summarise(mean = mean(log_RT))
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1  6.05
```

Check the following values against the group means above: 6.0516 - 0.073 = 5.978613 and 6.0516 + 0.0728 = 6.124421, you can see that the group means for the congruent and incongruent condition equate to the intercept + the new paramter estimates.

Again, however, the main effect of our model remains unchanged regardless of the coding scheme used.

For now, we'll return to using treatment coding and we'll explore changing the order for the intercept. If we want to change the order, or determine exactly which level will be the intercept, we simply have to relevel our factor before defining our contrasts:

```
contrasts(stroop_dat$cond) <- contr.treatment
stroop_dat$cond <- factor(stroop_dat$cond, levels = c("neutral", "congruent", "incongru
```

Here, we simply asked R to remake our condition column from a factor of our condition column, but with specifically ordered levels.

Now when we specify our treatment contrasts, the reference level (i.e. 0) will be the first level that we specified above (neutral, in this instance).

```
contrasts(stroop_dat$cond)
```

```
##             congruent incongruent
## neutral             0           0
## congruent           1           0
## incongruent         0           1
```

We'll rerun the model to see how our intercept has changed.

```
lm(log_RT ~ cond, data = stroop_dat) %>% summary
```

```
##
## Call:
## lm(formula = log_RT ~ cond, data = stroop_dat)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.180561 -0.052211  0.002842  0.042382  0.195386
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      6.051819   0.009308 650.179  < 2e-16 ***
## condcongruent   -0.073206   0.013163  -5.561 9.74e-08 ***
## condincongruent  0.072602   0.013163   5.515 1.22e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0721 on 177 degrees of freedom
## Multiple R-squared:  0.4094, Adjusted R-squared:  0.4027
## F-statistic: 61.35 on 2 and 177 DF,  p-value: < 2.2e-16
```

This looks very similar to the grand mean, but notice the value of the parameter estimate beyond 3 decimal places. The paramter estimate is now 6.051819 which is the value of the mean for the neutral condition (while 6.051618 is the value of the grand mean across all conditions).

7.2.3 Multilevel ANOVA

Let's say we're interested in main effects and we want to see if there is a main effect of condition in the numerical stroop task. Nicely, the `aov()` function defaults to give us main effects in the form of a traditional ANOVA output.

Let's see how this works.

```
stroop_aov <- aov(log_RT ~ cond, data = stroop_dat)
summary(stroop_aov)
```

```
##                Df Sum Sq Mean Sq F value Pr(>F)
## cond            2 0.6378  0.3189   61.35 <2e-16 ***
## Residuals     177 0.9201  0.0052
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Now, our model corresponds to the model output that we got using the `lm()` function. Look at the bottom of the results from the linear model in the previous section, the F, df, and *p*-values are the same. This is because the `aov()` function is simply a wrapper for our linear model that gives us main effects! Under the hood, `aov()` is just `lm()`, only without the additional model coefficients from the linear model.

However, if we've already fitted a model using the `aov()` function, but we want to retrieve the model coefficients, we can use a different type of `summary()` function, `summary.lm()`. This makes R give us our model with a linear model output.

```
summary.lm(stroop_aov)
```

```
##
## Call:
## aov(formula = log_RT ~ cond, data = stroop_dat)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.180561 -0.052211  0.002842  0.042382  0.195386
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      6.051819   0.009308 650.179  < 2e-16 ***
## condcongruent   -0.073206   0.013163  -5.561 9.74e-08 ***
## condincongruent  0.072602   0.013163   5.515 1.22e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0721 on 177 degrees of freedom
## Multiple R-squared:  0.4094, Adjusted R-squared:  0.4027
## F-statistic: 61.35 on 2 and 177 DF,  p-value: < 2.2e-16
```

Again, the fit from this model gives coefficients which correspond to those that we determine using the `contrasts()` function.

This contrast coding will become more important as we fit more factors in our models.

7.2.4 Multiple Contrasts

Let's say we have the smoking data set, and we're interested in whether two drugs differ from a placebo. We can fit the model with a treatment coded contrast matrix so that the intercept is the placebo (control group), and the other conditions are compared against this reference value. We'll fit this with a linear model, seeing as we care about the model coefficients.

```
smoking_lm <- lm(outcome ~ cond, data = smoking_dat)
summary(smoking_lm)
```

```
##
## Call:
## lm(formula = outcome ~ cond, data = smoking_dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.7906  -3.1192   0.0561   3.1747  18.4927
##
```

```
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)    79.6866     0.5508 144.670   <2e-16 ***
## conddrug_one    0.7748     0.7790   0.995    0.321
## conddrug_two   20.5112     0.7790  26.331   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.508 on 297 degrees of freedom
## Multiple R-squared:   0.75,  Adjusted R-squared:  0.7483
## F-statistic: 445.4 on 2 and 297 DF,  p-value: < 2.2e-16
```

As you can see, the two drug conditions are compared aginst the intercept for the placebo condition. However, what if we want to consider the effectiveness of the two drugs against one-another?

We have two options:

1. Relevel your factor so that you can compare drug_one to drug_two (i.e. set drug_one as the intercept)
2. Calculate the differences in the parameter estimates for the two rows in the table.

The first option is easiest if we only have a few levels in a factor. The second is handy if we have a lot of levels, but requires us to specify a complex contrast matrix. For the first option, we simply need to do something like this:

```
smoking_dat$cond <- factor(smoking_dat$cond, levels = c("drug_one", "drug_two", "control"))
smoking_lm2 <- lm(outcome ~ cond, data = smoking_dat)
summary(smoking_lm2)
```

```
##
## Call:
## lm(formula = outcome ~ cond, data = smoking_dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.7906  -3.1192   0.0561   3.1747  18.4927
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)    80.4614     0.5508 146.077   <2e-16 ***
## conddrug_two   19.7364     0.7790  25.337   <2e-16 ***
## condcontrol    -0.7748     0.7790  -0.995    0.321
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.508 on 297 degrees of freedom
## Multiple R-squared:   0.75,  Adjusted R-squared:  0.7483
## F-statistic: 445.4 on 2 and 297 DF,  p-value: < 2.2e-16
```

Now our intercept is drug_one, the parameter estimate for conddrug_two is the difference between drug_one and drug_two, and condcontrol corresponds to the parameter estimates for the difference between drug_one and the control condition. Notice that condcontrol here equates to conddrug_one in the original model.

For more complex cases (i.e. several levels) there are helper packages such as `multcomp` which allows you to specify a contrast matrix and conduct multiple comparisons on your model, but we won't get into that here.

7.3 Multifactorial Analyses

Now we'll look at cases where we have more than one factor. Here, we'll use one of R's inbuilt data sets, ToothGrowth, which looks at the effect of vitamin C on tooth growth in guinea pigs. This data set has 3 columns:

- len: tooth length
- supp: supplement type (vitamin C supplement or orange juice)
- dose: strength of dose

Most importantly, supplement is a factor with 2 levels, and dose is a numeric variable with 3 levels, but the same principles apply to more complex designs.

Load the data, and save it as a tibble with the columns saved as factors:

```
# load the data
data("ToothGrowth")

# convert to tibble and make dose a factor
tooth <- ToothGrowth %>%
  mutate(dose = factor(dose)) %>%
  as.tibble()

# see the output
tooth
```

```
## # A tibble: 60 x 3
##       len supp  dose
##    <dbl> <fct> <fct>
##  1   4.2 VC    0.5
##  2  11.5 VC    0.5
##  3   7.3 VC    0.5
##  4   5.8 VC    0.5
##  5   6.4 VC    0.5
##  6  10   VC    0.5
##  7  11.2 VC    0.5
##  8  11.2 VC    0.5
##  9   5.2 VC    0.5
## 10   7   VC    0.5
## # ... with 50 more rows
```

7.3.1 Multifactorial ANOVA

We'll conduct an ANOVA to check for a main effect of each factor, and the interaction between them. You already know how to check for main effects, and we can simply check for more main effects by adding them to the formula like so:

`dependent variable ~ factor_one + factor_two`

To check for an interaction, we specify it like so:

`dependent variable ~ factor_one + factor_two + factor_one : factor_two`

Or, to save on typing, like so (these two are equivalent):

`dependent variable ~ factor_one * factor_two`

First, we'll check how our factors are instantiated (i.e. what are the contrasts)?

```
contrasts(tooth$supp)
```

```
##    VC
## OJ  0
## VC  1
```

```
contrasts(tooth$dose)
```

```
##     1 2
## 0.5 0 0
## 1   1 0
## 2   0 1
```

Here, we're interested in the main effects of both factors, and whether there's an interaction between them. Since we aren't going to get the parameter estimates from this model using the `summary.lm()` function, we don't need to specify the contrast matrix required for the ANOVA here.

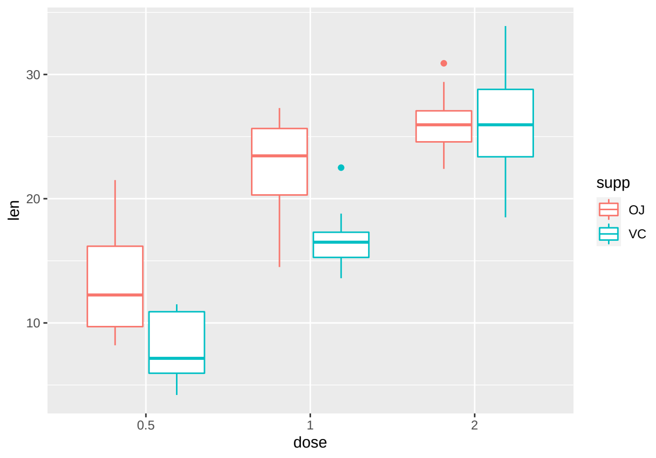
We'll save this as **tooth_aov** as we want to do some follow up tests later on.

```
tooth_aov <- aov(len ~ supp * dose, data = tooth)
summary(tooth_aov)
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)
## supp         1  205.4   205.4  15.572 0.000231 ***
## dose         2 2426.4  1213.2  92.000  < 2e-16 ***
## supp:dose    2  108.3    54.2   4.107 0.021860 *
## Residuals   54  712.1    13.2
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As you can see, we have a significant main effect of both factors, and a significant interaction. As such, we need to do some follow up tests to see where our effect lies in the interaction. A quick check with a plot always helps:

```
ggplot(data = tooth, mapping = aes(x = dose, y = len, colour = supp)) +
  geom_boxplot()
```



It looks like there's a main effect of dose overall, but also that supplement type probably only matters for lower doses. Let's check these using some pairwise tests. As such, perform pairwise comparisons to see where our differences lie.

7.3.1.1 Multiple Comparisons

One option is to perform multiple comparisons using Tukey Honest Significant Difference. We do this by applying the `TukeyHSD()` function to our fitted model **tooth_aov**. Here, we also have to specify the confidence level, which is by default 95%.

```
TukeyHSD(tooth_aov, conf.level = 0.95)
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = len ~ supp * dose, data = tooth)
##
## $supp
##       diff       lwr       upr     p adj
## VC-OJ -3.7 -5.579828 -1.820172 0.0002312
##
## $dose
##         diff       lwr       upr   p adj
## 1-0.5  9.130  6.362488 11.897512 0.0e+00
## 2-0.5 15.495 12.727488 18.262512 0.0e+00
## 2-1    6.365  3.597488  9.132512 2.7e-06
##
## $`supp:dose`
##                diff        lwr        upr     p adj
## VC:0.5-OJ:0.5 -5.25 -10.048124 -0.4518762 0.0242521
## OJ:1-OJ:0.5    9.47   4.671876 14.2681238 0.0000046
## VC:1-OJ:0.5    3.54  -1.258124  8.3381238 0.2640208
## OJ:2-OJ:0.5   12.83   8.031876 17.6281238 0.0000000
## VC:2-OJ:0.5   12.91   8.111876 17.7081238 0.0000000
## OJ:1-VC:0.5   14.72   9.921876 19.5181238 0.0000000
## VC:1-VC:0.5    8.79   3.991876 13.5881238 0.0000210
## OJ:2-VC:0.5   18.08  13.281876 22.8781238 0.0000000
## VC:2-VC:0.5   18.16  13.361876 22.9581238 0.0000000
## VC:1-OJ:1     -5.93 -10.728124 -1.1318762 0.0073930
## OJ:2-OJ:1      3.36  -1.438124  8.1581238 0.3187361
## VC:2-OJ:1      3.44  -1.358124  8.2381238 0.2936430
## OJ:2-VC:1      9.29   4.491876 14.0881238 0.0000069
## VC:2-VC:1      9.37   4.571876 14.1681238 0.0000058
## VC:2-OJ:2      0.08  -4.718124  4.8781238 1.0000000
```

We get a lot of output here corresponding to every possible comparison we could ever want. We can remedy this by saving the output of TukeyHSD to an object, and selecting only the relevant information, **supp:dose**. Notice that we put this name in backticks so we can use the special character `:` without R interpreting it as something else.

```
tukey_aov <- TukeyHSD(tooth_aov, conf.level = 0.95)
tukey_aov$`supp:dose`
```

```
##                  diff        lwr        upr        p adj
## VC:0.5-OJ:0.5 -5.25 -10.048124 -0.4518762 2.425209e-02
## OJ:1-OJ:0.5    9.47   4.671876 14.2681238 4.612304e-06
## VC:1-OJ:0.5    3.54  -1.258124  8.3381238 2.640208e-01
## OJ:2-OJ:0.5   12.83   8.031876 17.6281238 2.125153e-09
## VC:2-OJ:0.5   12.91   8.111876 17.7081238 1.769939e-09
## OJ:1-VC:0.5   14.72   9.921876 19.5181238 2.985978e-11
## VC:1-VC:0.5    8.79   3.991876 13.5881238 2.100948e-05
## OJ:2-VC:0.5   18.08  13.281876 22.8781238 4.855005e-13
## VC:2-VC:0.5   18.16  13.361876 22.9581238 4.821699e-13
## VC:1-OJ:1     -5.93 -10.728124 -1.1318762 7.393032e-03
## OJ:2-OJ:1      3.36  -1.438124  8.1581238 3.187361e-01
## VC:2-OJ:1      3.44  -1.358124  8.2381238 2.936430e-01
## OJ:2-VC:1      9.29   4.491876 14.0881238 6.908163e-06
## VC:2-VC:1      9.37   4.571876 14.1681238 5.774013e-06
## VC:2-OJ:2      0.08  -4.718124  4.8781238 1.000000e+00
```

7.3.2 Multiple Regression

7.3.2.1 Three-Level Factors

If you want to see whether there's a main effect or interaction for 3 level factors, the easiest way is to simply run an ANOVA. With the `lm()` function, comparisons for the contribution of each factor in our model starts to become more difficult when we have factors with more than 2 levels. That's because we have to construct some numeric variables for these factors, and we need to perform model comparisons on a full model (with interactions) and reduced models (without) to see which model best fits our data.

To do this, we simply have to centre our two factors, with these stored as numeric variables.

```
# code new centered variables
tooth$supp_dev <- (tooth$supp == "VC") - mean(tooth$supp == "VC")
tooth$dose_dev_one <- (tooth$dose == "0.5") - mean(tooth$dose == "0.5")
tooth$dose_dev_two <- (tooth$dose == "1") - mean(tooth$dose == "1")

# inspect changes
head(tooth)
```

```
## # A tibble: 6 x 6
##     len supp  dose  supp_dev dose_dev_one dose_dev_two
##   <dbl> <fct> <fct>    <dbl>        <dbl>        <dbl>
## 1   4.2 VC    0.5        0.5        0.667       -0.333
## 2  11.5 VC    0.5        0.5        0.667       -0.333
## 3   7.3 VC    0.5        0.5        0.667       -0.333
## 4   5.8 VC    0.5        0.5        0.667       -0.333
## 5   6.4 VC    0.5        0.5        0.667       -0.333
## 6  10   VC    0.5        0.5        0.667       -0.333
```

This centering works similarly to the contrast matrices we used before. In fact, it is often the better option if we have unbalanced data sets, as it accounts for the mismatch in the number of observations in each level of a factor. The centering used in this case is called **deviation coding** and is just like sum coding, only the parameter estimates and standard errors are half as large (but all other interpretations, e.g. *t*- and *p*-values remain the same).

Now, when we fit a model with **supp_dev**, the intercept will be the mean of the two conditions.

When it comes to fitting a 3 level factor, we need to do the same thing for only two of the levels. When we then add this to a model and test for an interaction, we test for the interaction for **supp_dev** across both columns for our deviation coded doses.

```
tooth_lm_full <- lm(len ~ supp_dev * (dose_dev_one + dose_dev_two), data = tooth)
```

Next, we construct a reduced model, and use the `anova()` function to compare the two models against one another.

```
tooth_lm_reduced <- lm(len ~ supp_dev + (dose_dev_one + dose_dev_two), data = tooth)
anova(tooth_lm_full, tooth_lm_reduced)
```

```
## Analysis of Variance Table
##
## Model 1: len ~ supp_dev * (dose_dev_one + dose_dev_two)
## Model 2: len ~ supp_dev + (dose_dev_one + dose_dev_two)
##   Res.Df    RSS Df Sum of Sq     F  Pr(>F)
## 1     54 712.11
## 2     56 820.43 -2   -108.32 4.107 0.02186 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This checks for how much variance is explainedby each model. As you can see, the interaction term in our model makes a significant contribution to the variance explained, so this indicates that there's an interaction in our model, and as such we should explore this.

A quick and easy way to do so is to run a number of *t*-tests or linear models on subsets of the data, adjusting for the *p*-values where necessary.

One way to achieve this for multifactorial designs is to create a new column which is a combination of the two factors.

```
# create combined factor column
tooth$interact <- interaction(tooth$supp, tooth$dose)

# check levels of our new factor
levels(tooth$interact)
```

```
## [1] "OJ.0.5" "VC.0.5" "OJ.1"   "VC.1"   "OJ.2"   "VC.2"
```

Then we simply use the `pairwise.t.test()` function, and supply it our dependent variable and our grouping factor. This method defaults to Holm's sequential bonerroni, which is like bonferroni adjustment of *p*-values, but rather than multiplying them all by the total number of comparisons, we instead multiply them incrementally, so the largest *p*-value is multiplied by 1, the next smallest by 2, the next smallest by 3, etc.

```
pairwise.t.test(tooth$len, tooth$interact)
```

```
##
##  Pairwise comparisons using t tests with pooled SD
##
## data:  tooth$len and tooth$interact
##
##        OJ.0.5  VC.0.5  OJ.1   VC.1    OJ.2
## VC.0.5 0.0105  -       -      -       -
## OJ.1   3.2e-06 2.6e-11 -      -       -
## VC.1   0.1346  1.0e-05 0.0035 -       -
## OJ.2   1.6e-09 1.9e-14 0.1346 3.8e-06 -
## VC.2   1.4e-09 1.7e-14 0.1346 3.6e-06 0.9609
##
## P value adjustment method: holm
```

There are other ways to automate *t*-tests in R. For example, you could create a function to subset your data, apply a test, and adjust the *p*-values manually, but this is not necessary here.

7.3.2.2 Two-Level Factors

Things are a lot easier when we have factors with only 2 levels. Let's assume our data had only 2 factors with 2 levels each. How would we test for main effects and interactions with a linear model?

To explore this question, we'll subset our data to only 2 levels for dose, and we'll keep only the original columns. (We use `dplyr::select` to be specific that we want select from the `dplyr` package, not another loaded package.). Remember, we want to recreate dose as a factor again so it doesn't remember our old level.

```
tooth_sub <- tooth %>%
  dplyr::select(1 : 3) %>%
  filter(dose %in% c(0.5, 1)) %>%
  mutate(dose = factor(dose))
```

Now, we should check our contrasts as usual.

```
contrasts(tooth_sub$supp)
```

```
##    VC
## OJ  0
## VC  1
```

```
contrasts(tooth_sub$dose)
```

```
##     1
## 0.5 0
## 1   1
```

We should make these sum coded contrasts so we can then get main effects from our model.

```
contrasts(tooth_sub$supp) <- contr.sum
contrasts(tooth_sub$dose) <- contr.sum
```

Then we can fit the model again.

```
tooth_sub_lm <- lm(len ~ supp * dose, data = tooth_sub)
summary(tooth_sub_lm)
```

```
##
## Call:
```

```
## lm(formula = len ~ supp * dose, data = tooth_sub)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -8.20  -2.72  -0.27   2.65   8.27
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  15.1700     0.5537  27.397  < 2e-16 ***
## supp1         2.7950     0.5537   5.048 1.30e-05 ***
## dose1        -4.5650     0.5537  -8.244 8.25e-10 ***
## supp1:dose1  -0.1700     0.5537  -0.307    0.761
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.502 on 36 degrees of freedom
## Multiple R-squared:  0.7221, Adjusted R-squared:  0.6989
## F-statistic: 31.18 on 3 and 36 DF,  p-value: 4.098e-10
```

The only thing to note here is that we've used sum coding other than deviation coding with the 3-level case. Our interpretation of effects stays the same here, only our paramter estimates and standard errors will be twice as large with sum coding when compared to deviation coding.

If your factors are not sum/deviation coded, then you won't observe main effects. So we can test for the effect of dose at the first level of supplement as follows:

```
contrasts(tooth_sub$supp) <- contr.treatment
contrasts(tooth_sub$supp)
```

```
##    2
## OJ 0
## VC 1
```

OJ will now be our reference value for the intercept. Additionally, the effect of dose will be compared at this intercept value, so we now have simple effects of dose, and main effects of supplement. The interpretation for our interaction terms remains the same regardless of our coding approach.

```
tooth_sub_lm2 <- lm(len ~ supp * dose, data = tooth_sub)
summary(tooth_sub_lm2)
```

```
##
## Call:
```

```
## lm(formula = len ~ supp * dose, data = tooth_sub)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -8.20  -2.72  -0.27   2.65   8.27
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  17.9650     0.7831  22.942  < 2e-16 ***
## supp2        -5.5900     1.1074  -5.048 1.30e-05 ***
## dose1        -4.7350     0.7831  -6.047 6.02e-07 ***
## supp2:dose1   0.3400     1.1074   0.307    0.761
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.502 on 36 degrees of freedom
## Multiple R-squared:  0.7221, Adjusted R-squared:  0.6989
## F-statistic: 31.18 on 3 and 36 DF,  p-value: 4.098e-10
```

7.4 Mixed Analyses

Mixed factorial analyses are just a special form of factorial analyses. Here, one factor is between subjects and another factor is within subjects. Here, we'll create some new data based on our smoking data that reflects a mixed design.

```r
# set parameters
set.seed(1000)
means <- rep(c(80, 85), 100)
sds <- rep(c(5, 5, 7), 100)
cond <- rep(c("control", "drug_one"), 100)
subject <- seq(1:200)

# simulate data
smoking_dat_extra <- tibble(
  subject = subject,
  cond = as.factor(cond),
  time_one = rnorm(n = 200, mean = means, sd = sds),
  time_two = time_one + rnorm(n = 200, mean = 5, sd = sds)
  ) %>%
  gather(key = test_time,
         value = recovery,
         3:4
         ) %>%
  mutate(test_time = as.factor(test_time)) %>%
  arrange(subject)

# view the data
smoking_dat_extra
```

```
## # A tibble: 400 x 4
##    subject cond     test_time recovery
##      <int> <fct>    <fct>        <dbl>
##  1       1 control  time_one      77.8
##  2       1 control  time_two      87.4
##  3       2 drug_one time_one      79.0
##  4       2 drug_one time_two      87.6
##  5       3 control  time_one      80.3
##  6       3 control  time_two      73.5
##  7       4 drug_one time_one      88.2
##  8       4 drug_one time_two      92.0
##  9       5 control  time_one      76.1
## 10       5 control  time_two      77.8
## # ... with 390 more rows
```

Now our smoking data is a little more complex, and a simpler in parts. For one, we now only have two drugs, the placebo (control) or drug_one. However now we've tested people at two time points to see if time plays a factor in their recovery.

7.4.1 Mixed Linear Models

First, we'll make sure our factors are sum coded so that we can obtain main effects of both factors.

```
contrasts(smoking_dat_extra$cond) <- contr.sum
contrasts(smoking_dat_extra$test_time) <- contr.sum
```

Then, we'll refit the model as before.

```
mixed_lm <- lm(recovery ~ cond * test_time,
                 data = smoking_dat_extra)
summary(mixed_lm)
```

```
##
## Call:
## lm(formula = recovery ~ cond * test_time, data = smoking_dat_extra)
```

```
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -23.7613  -3.8794   0.4193   4.3061  20.2756
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)       85.3816     0.3370 253.385  < 2e-16 ***
## cond1             -2.6232     0.3370  -7.785 6.15e-14 ***
## test_time1        -2.5064     0.3370  -7.438 6.40e-13 ***
## cond1:test_time1   0.0612     0.3370   0.182    0.856
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.739 on 396 degrees of freedom
## Multiple R-squared:  0.2265, Adjusted R-squared:  0.2206
## F-statistic: 38.65 on 3 and 396 DF,  p-value: < 2.2e-16
```

Now our intercept corresponds to the grand mean, and our effects of condition and test time are main effects of these two factors. The interaction again tells us that we need to explore the data. For this, we'd just use the same methods from above.

However, this model doesn't account for the dependencies between subjects and their scores. In order to do this, we should use the 'aov()' function, which is simply a wrapper for the linear model, but allows for accounting for between-subjects factors. Alternatively, we can improve on our linear models by making them account for subjects-based effects, and we'll cover this in later sessions on **mixed effects modelling**.

7.4.2 Mixed ANOVA

With a mixed ANOVA, we are simply just including an error term for our within-subjects factor. We do this in the same way as for within-subjects ANOVAs, only leaving the error term off the between-subjects factor.

In this case, subjects take part in both test time conditions, so we need to nest this within our data.

```
mixed_aov <- aov(recovery ~ cond * test_time +
                   Error(subject/test_time),
                 data = smoking_dat_extra)
summary(mixed_aov)
```

```
##
```

```
## Error: subject
##      Df Sum Sq Mean Sq
## cond  1  261.2   261.2
##
## Error: subject:test_time
##           Df Sum Sq Mean Sq
## test_time  1   2077    2077
##
## Error: Within
##                 Df Sum Sq Mean Sq F value   Pr(>F)
## cond             1   2738  2738.1  60.877 5.51e-14 ***
## test_time        1    454   453.9  10.091  0.00161 **
## cond:test_time   1      1     1.4   0.031  0.85956
## Residuals      394  17721    45.0
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Now we have our main effects of condition, test time, and the interaction between them.

7.5 A Note on Sum of Squares

In some cases, you might notice that the output of your ANOVA models does not correspond to the output of that from SPSS or other statistical packages. That's because R defaults to type-I sum of squares, and SPSS defaults to type-III sum of squares.

Put simply, when you have an unbalanced design, such as when you have an unequal number of participants in each group, the way in which the sum of squares is calculated will affect the output and interpretation of your ANOVA.

With type-I sum of squares, which is R's default, the order in which you specify factors in your model will determine how much variance is explained by each factor. With type-I sum of squares, the terms are considered in the model in a sequential order. Take a look at this in action below. We'll use the **tooth_sub** data from before, but we'll remove the first observation.

```
tooth_sub_unbalanced <- tooth_sub[2:40,]
```

```
supp_first <- aov(len ~ supp * dose, data = tooth_sub_unbalanced)
summary(supp_first)
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)
## supp         1  259.4   259.4  21.331 5.05e-05 ***
## dose         1  777.3   777.3  63.920 2.09e-09 ***
## supp:dose    1    2.9     2.9   0.242    0.626    
## Residuals   35  425.6    12.2                     
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
dose_first <- aov(len ~ dose * supp, data = tooth_sub_unbalanced)
summary(dose_first)
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)    
## dose         1  753.3   753.3  61.948 2.98e-09 ***
## supp         1  283.4   283.4  23.304 2.70e-05 ***
## dose:supp    1    2.9     2.9   0.242    0.626    
## Residuals   35  425.6    12.2                     
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Here, you can see that the sum of squares, and consequently the F-values (and hence p-values) differ depending on the order in which we entered our factors in the model. What happens is that the first term in the model is evaluated first, and any variance explained that is shared by dose and supp is instead given to the first term in the model. Therefore, we shouldn't really use type-I sum of squares unless our two factors are entirely independent of one-another.

Alternatively, we have type-II sum of squares, which evaluates the main effects of your factors while ignoring the variance explained by interaction terms. Crucially, if you do have an interaction, type-II sum of squares will not accurately reflect the variance explained by the interaction as that variance is given over to the main effect terms. Crucially, the coding scheme used for type-II sum of squares does not matter.

In Psychology, we typically use type-III sum of squares, where main effects are computed while taking the interaction into account. This method is recommended for unbalanced designs, although there is some debate on this. However, type-III sum of squares only work with orthogonal contrast matrices (i.e. sum/helmert coding), so make sure you pay attention to how you've coded your factors prior to fitting a model.

The way to refit our model with type-III sum of squares is to use the `Anova()` (notice the capital A) function from the **car** package, specifying the type for your sum of squares. This function takes the object of your model fitted by `aov()` and the type for the sum of squares you would like to compute. Note that this function defaults to type-II sum of squares, so we need to be specific in our function call.

First, install this package and load it, then run the function on the two model fits from above.

```r
# install.packages("car") # install once per machine
library(car) # load each time you open R
```

```
## Loading required package: carData

##
## Attaching package: 'car'

## The following object is masked from 'package:dplyr':
##
##     recode

## The following object is masked from 'package:purrr':
##
##     some
```

```r
Anova(supp_first, type = "III")
```

```
## Anova Table (Type III tests)
##
## Response: len
##             Sum Sq Df F value    Pr(>F)
## (Intercept) 6454.8  1 530.797 < 2.2e-16 ***
## supp         281.6  1  23.158 2.824e-05 ***
## dose         448.4  1  36.873 6.201e-07 ***
## supp:dose      2.9  1   0.242    0.6258
## Residuals    425.6 35
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
Anova(dose_first, type = "III")
```

```
## Anova Table (Type III tests)
##
## Response: len
##             Sum Sq Df F value    Pr(>F)
## (Intercept) 6454.8  1 530.797 < 2.2e-16 ***
## dose         448.4  1  36.873 6.201e-07 ***
## supp         281.6  1  23.158 2.824e-05 ***
## dose:supp      2.9  1   0.242    0.6258
## Residuals    425.6 35
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As you can see, the results of both analyses are equivalent for the unbalanced data set using this function.

As a final note, in all cases we should check that the tests we run meet the assumptions of said test. For factorial ANOVAs, this means checking the previous assumptions for a one-way ANOVA, while also checking for a lack of multicollinearity between the factors in the design, which we won't cover here.

7.6 Exercises

For these exercises, we will look at the core concepts from this lesson. Here, we'll use a simulated data set that's saved as a .csv in the inputs folder.

If you don't have access to these, please download the repository from GitHub and open the **lesson_materials** folder. Open the relevant folder for this lesson. If you work out of the file **07_advanced_statistical_tests.Rmd** the code below will work to load the data set. If this fails then quit R and open it up from the .Rmd file above so your working directory is in the correct folder.

```
library(tidyverse)
# install.packages("car") # install if necessary (once per machine)
library(car)
library(ggpirate)
```

Next, we'll load the data set for this exercise.

```
factorial_data <- read_csv("inputs/factorial_data.csv")
```

7.6.1 Question 1

Aggregate the data by subject such that you have 1 score per subject for the mean of the column Y across both A and B columns. Call the Y column mean_Y. Assign this to the object **data_agg**.

7.6.2 Question 2

Generate summary statistics for the data set and a pirate plot of the data set. We want means, SDs, and ns for the table of summary statistics.

7.6.3 Question 3

Define your contrasts such that you have main effects and interactions from any model fits. You may need to check your data types for this to work properly. *Hint*: use `mutate()` after the `ungroup()` function

7.6.4 Question 4

Fit the data set checking for all main effects and interactions.

7.6.5 Question 5

Refit your model from Question 5 using type-III sum of squares. Did that make any difference to your sum of squares? Why or why not?

7.6.6 Question 6

Explore the parameter estimates from your ANOVA model, and briefly report your findings in text (1 sentence max).

Chapter 8

Simulation and Calculating Power

In this session we'll cover how to simualte data to improve our statistical inferences. We'll use simulation to perform power analyses, and to understand more about how *p*-values work. Finally, we'll look at how you can calculate power easily and quickly in several pre-existing packages and functions in R.

Specifically, we'll cover:

- simulating data in R
- simulation-based power analyses
- packages for power analysis

8.1 Getting Started

As always, we first need to load the `tidyverse` set of package to perform the data manipulation for this chapter.

```r
# load packages
library(tidyverse)
```

8.2 Simulating Data

First off, we'll explore different ways to simulate data using some of the basic functions in R. We'll look at randomly sampling data that adheres to the following distributions:

- Uniform, using `runif()` (read: **r**andom **unif**orm)
- Normal, using `rnorm()` (read: **r**andom **norm**al)
- Binomial, using `rbinom()` (read: **r**andom **binom**ial)

Additionally, we'll look at randomly sampling data with and without replacement, and how we can determine bias in our sampling procedure by defining the probability with which items should be sampled.

We'll look at sampling data following these distributions as this provides us with a better understanding of our data, and equips us with the means to evaluate our research plans prior to the expensive (in terms of time and money) procedure of testing. Put simply, simulating data is useful because it allows us to do things like calculate power for simple and complex experimental designs, and to observe when participants are performing to change in tasks.

8.2.1 The Uniform Distribution

With the uniform distribution, we sample numbers such that all values are equally likely to be drawn from our range. Try this out a few times:

```r
runif(n = 5, min = 1, max = 10)
```

```
## [1] 3.300586 2.342055 4.873663 9.191708 5.319560
```

In all likelihood, you should get different numbers to those above, but your numbers will all be between 1 and 10 (excluding these two values). You can increase the amount of numbers sampled by changing `n` to anything else. For example, try this:

```r
runif(n = 10, min = 1, max = 10)
```

```
##  [1] 8.011183 5.748096 1.380451 7.302121 5.467758 6.987622 2.161111 5.345088
##  [9] 5.670862 4.214587
```

Now we have 10 numbers between 1 and 10.

Finally, we can change the minimum and maximum values, or the range of values we want to sample from. Let's sample 10 numbers from between 0 and 1.

```r
runif(n = 10, min = 0, max = 1)
```

```
##  [1] 0.90863606 0.60394266 0.78368223 0.47517589 0.03186038 0.51023136
##  [7] 0.51073424 0.58464829 0.76034355 0.08140943
```

What happens if we calculate the mean from 10 samples from a uniform distribution? What about 100, 1000, or even 10000?

```
runif(n = 10, min = 0, max = 1) %>% mean()
```

```
## [1] 0.4962173
```

```
runif(n = 100, min = 0, max = 1) %>% mean()
```

```
## [1] 0.4754119
```

```
runif(n = 1000, min = 0, max = 1) %>% mean()
```

```
## [1] 0.4878579
```

```
runif(n = 10000, min = 0, max = 1) %>% mean()
```

```
## [1] 0.5000408
```

As you can see, as we draw more samples, the mean of these samples better represents the true distribution of our sampling procedure. Think about how this could impact your studies. With small sample sizes, we're likely to get skewed data that tends towards the extremes.

8.2.2 The Normal Distribution

Alternatively, we can sample continuous data from the normal distribution. We might use this to generate data on a continuous scale with a normal distribution in the population, such as the height and weight of a participant, or even their IQ.

To do this, we use the `rnorm()` function. You may have noticed me doing this when creating data for the previous sessions, and that's with good reason; we often fit our data with models that assume a normal distribution of our variables, and so creating data with this assumed distribution often meets the assumptions of the tests we'd like to run.

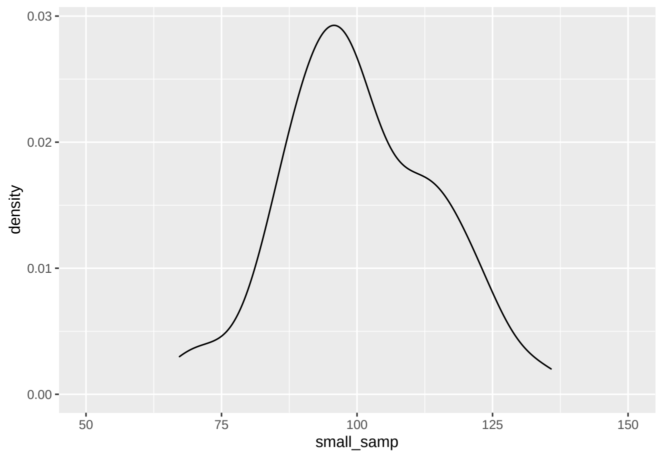
This sampling function provides us with a good range of values if we know (or can guess at) the mean and standard deviation of a sample from a population. Let's see how this works. Below, we'll sample some data from a population where we know the mean and standard deviation; the IQ test. Here, the mean is 100 and the standard deviation is 15. Let's go ahead and sample 10 people from this population.

```
rnorm(n = 10, mean = 100, sd = 15)
```

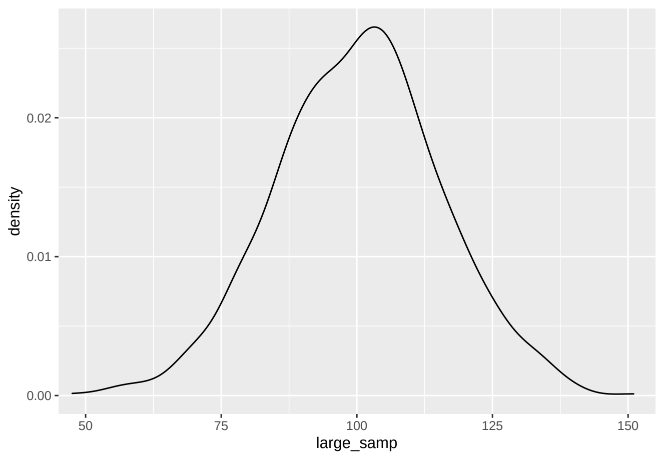
```
##  [1]  82.16277  99.75992 128.28859  76.65559  73.06495  97.49910  96.44725
##  [8] 102.10636  91.52518 125.11196
```

OK, we have a good range of values for our IQ test here, but as before, with a small sample size our distribution will be skewed. Let's observe how this plays out with a density plot. (Here, I'm being lazy and not defining `mapping = aes()`, as ggplot understands that what we put in `aes()` will be mapped to the data/plot.)

```
set.seed(5)
small_samp <- rnorm(n = 100, mean = 100, sd = 15)
large_samp <- rnorm(n = 1000, mean = 100, sd = 15)

ggplot() + geom_density(aes(small_samp)) + coord_cartesian(x = c(50, 150))
```

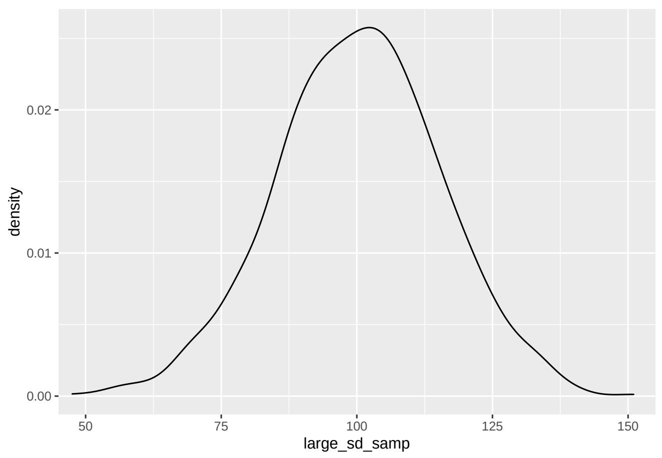


```
ggplot() + geom_density(aes(large_samp)) + coord_cartesian(x = c(50, 150))
```

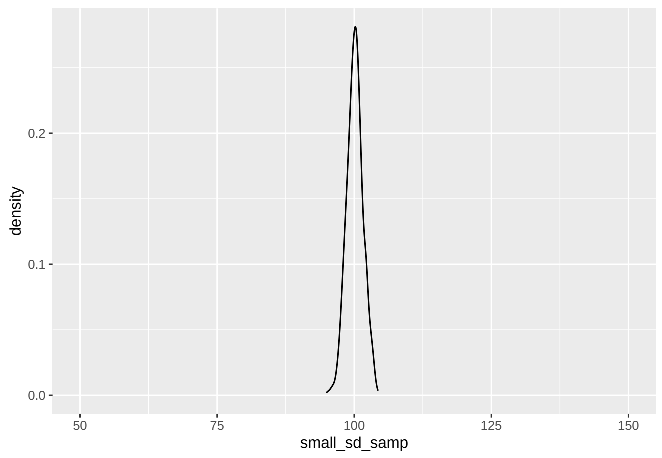


As before, the data seem to fit the population distribution better when we have larger sample sizes. Let's change the standard deviation to see how variance in our samples influences the distribution of effects. Here, we'll just focus on the larger sample size:

```
set.seed(5)
large_sd_samp <- rnorm(n = 1000, mean = 100, sd = 15)
small_sd_samp <- rnorm(n = 1000, mean = 100, sd = 1.5)

ggplot() + geom_density(aes(large_sd_samp)) + coord_cartesian(x = c(50, 150))
```



```
ggplot() + geom_density(aes(small_sd_samp)) + coord_cartesian(x = c(50, 150))
```



As you can see, as we sample from a population with a smaller standard deviation, our samples will be closer to the mean score. This has important implications for our statistical tests; the larger the standard deviation within our samples, the more samples we'll need to detect an effect.

8.2.3 The Binomial Distribution

Next, we'll look at sampling from a binomial distribution. We might use this to generate data on a binomial scale, such as the probability of a success/failure in a task (e.g. did they get the question right or not?). We sample from a binomial distribution using the `rbinom()` function.

This function takes 3 arguments: `n`, the number of observations we'd like to see; `size`, the number of trials within an observation; and `prob` the probability of a success (1) (vs. failure, 0) on each trial.

Below, we'll get 10 observations, of 1 trial each, with a success rate of 50% on each trial.

```
set.seed(1000)
rbinom(n = 10, size = 1, prob = 0.5)
```

```
##  [1] 0 1 0 1 1 0 1 1 0 0
```

As you can see, our sampling produced 5 successes (1) and 5 failures (0). What if we want to make each observation the number of successes in 20 trials. Let's change the size argument to 20.

```r
set.seed(1000)
rbinom(n = 10, size = 20, prob = 0.5)
```

```
##  [1]  9 12  7 11 10  7 11 10  8  9
```

Now each trial roughly follows a 50% success rate as before, but each observation is out of 20 trials. We can convert these to proportions by simply dividing the whole thing by the same value as our size argument.

```r
set.seed(1000)
rbinom(n = 10, size = 20, prob = 0.5) / 20
```

```
##  [1] 0.45 0.60 0.35 0.55 0.50 0.35 0.55 0.50 0.40 0.45
```

Did you notice the repetition of 20 in our code? We'll save some repetition for the value of 20 by defining the number of trials in a separate object, and passing the name of this object to our argument. This should get you used to using objects as holders for values as a precursor to writing your own functions!

```r
# define number of trials
n_trials <- 20

set.seed(1000)
rbinom(n = 10, size = n_trials, prob = 0.5) / n_trials
```

```
##  [1] 0.45 0.60 0.35 0.55 0.50 0.35 0.55 0.50 0.40 0.45
```

Also, I think our code is a bit more readable now that we know what the 20 represents!

Can you guess what the mean of a 1000 samples might look like? What about the distribution of these? Try calculating the mean of 1000 samples of 20 trials with a probability of 0.5 and plot this out using a histrogram.

8.2.4 Flexible Sampling

Finally, we can also randomly sample discrete variables with pre-defined probabilities. Nicely, we can use the same function to do this for strings of character or for integers. To do this, we use the `sample()` function. This is useful for creating dummy data sets where we might want to randomly sample names for participants, or outcomes with a bias on our draws.

We can simply define what to sample from with the `x` argument. If this is an integer, R will sample from any positive integer up to and including this limit. As before we can ask for any number of samples with the `size` argument.

Finally, we can specify if we want to replace these numbers with the `replace` argument. If we set this to true, R puts any drawn numbers back in the bag before picking them out again. If we set this to FALSE, R will not put the numbers back in the bag before drawing again, so we can't redraw the same number.

```
set.seed(1000)
sample(x = 10, size = 10, replace = TRUE)
```

```
##  [1] 4 6 3 8 3 2 6 6 6 1
```

Did you notice how we got 3 twice? Let's change the `replace` argument to `FALSE` to see what happens.

```
set.seed(1000)
sample(x = 10, size = 10, replace = FALSE)
```

```
##  [1]  4  6  3  5  8  7  2 10  9  1
```

Now we only get unique numbers. If we ask to draw more numbers than we have unique values, this will fail.

```
set.seed(1000)
sample(x = 10, size = 11, replace = FALSE)
```

We can also sample from characters, such as when we might want to create some data for the names of participants.

```
set.seed(1000)
names <- c("Glenn", "Nik", "Vera", "Neil")
sample(x = names, size = 2, replace = TRUE)
```

```
## [1] "Neil" "Neil"
```

We can also set some pre-defined probabilities for the things we'll sample from. Let's say the probability of selecting "Glenn" is most likely in 10 draws.

```
set.seed(1000)
names <- c("Glenn", "Nik", "Vera", "Neil")
sample(x = names, size = 10, replace = TRUE, prob = c(0.7, 0.1, 0.1, 0.1))
```

```
##  [1] "Glenn" "Vera"  "Glenn" "Glenn" "Glenn" "Glenn" "Vera"  "Glenn" "Glenn"
## [10] "Glenn"
```

As you can see, Glenn gets drawn most often because this has the highest probability to be drawn (i.e. the first thing in `x` is related to the first probability in `prob`).

This sampling procedure is very flexible for simulating data if we have some idea about the probabilities for things to be sampled. Let's look at an example where sampling like this is useful.

8.3 Sampling for Inference

Imagine we're studying a person who claims to be a psychic. We decide to test their claim by getting them to take part in an experiment. In this experiment, we have 10 cards each with 1 of 5 symbols on the back. The psychic says that they can guess, above chance, which symbol is on the back of the card.

Let's simulate their data for 10 attempts.

```
# sample the true values of the cards
set.seed(13)
psychic_outcome <- rbinom(n = 1, size = 10, prob = 0.2)

# see the true values
psychic_outcome
```

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

The psychic guessed 3 cards correctly out of 10. However, we know that on each trial, the probability of a success is only 20% (i.e. 1 in 5). Was the psychic telling the truth, or should we have reason to doubt their claim?

```
binom.test(x = psychic_outcome, n = 10, p = 0.2)
```

```
##
##  Exact binomial test
##
```

```
## data:  psychic_outcome and 10
## number of successes = 3, number of trials = 10, p-value = 0.4296
## alternative hypothesis: true probability of success is not equal to 0.2
## 95 percent confidence interval:
##  0.06673951 0.65245285
## sample estimates:
## probability of success
##                    0.3
```

So, it seems that although they performed above chance, the sample size is too small to consider this difference between the observed and expected probability of success to be meaningful.

What if we change it so the psychic got it right 30 times in 100 chances? We'll simply multiply the number of successes and observations by 10 and test for a significant difference between the observed and expected probability.

```
binom.test(x = psychic_outcome * 10, n = 10 * 10, p = 0.2)
```

```
##
##  Exact binomial test
##
## data:  psychic_outcome * 10 and 10 * 10
## number of successes = 30, number of trials = 100, p-value = 0.01695
## alternative hypothesis: true probability of success is not equal to 0.2
## 95 percent confidence interval:
##  0.2124064 0.3998147
## sample estimates:
## probability of success
##                    0.3
```

Now we have a significant effect showing indicating that we should reject the null hypothesis. But do any of you really believe that the psychic is truly psychic? Remember that we used the binomial sampling procedure to come up with these scores in the first place, so we know that the true probability of a success on each trial is 0.2, or 20%, which is chance.

Let's change our sampling procedure into a function so we can better explore whether or not we should trust the outcome of this one study. Before we do so, we'll see how functions work, and why they are useful in our work.

8.3.1 Understanding p-values and Type-I Errors

User defined functions in R take the form of:

```r
function_name <- function(parameters_to_vary) {
  things_for_the_function_to_do
}
```

We first have to specify a name for our function. Often, it's best to use verb-like naming here, as your functions are things you want to do to something (e.g. calculate descriptive statitics etc.). You then assign to this name how the function works. Within the function call in parentheses, you can define different parameters of your function that you'd like to vary. This simply makes your function flexible to doing things to data with different inputs. Finally, within the body of the function, `{here is the body}`, we define things to do to your parameters that are passed to the function. The last thing that you type in your function is what your function returns to you when you run it (i.e. what you will see when you run the function).

Let's transform our sampling and testing procedure into a function. I've commented this code so you understand how it works.

```r
# define function name and parameters to vary
simulate_psychic_test <- function(n_trials, prob_success) {

  # sample binomial data (successs or not) for varying trials and probabilities
  psychic_outcome <- rbinom(n = 1, size = n_trials, prob = prob_success)

  # run a binomial test on the sampled data against your proability
  test <- binom.test(x = psychic_outcome, n = n_trials, p = prob_success)

  # retrieve the p-value, this is what is returned in our function
  test$p.value
}
```

Congratulations, you've just made your first user-defined function in R! Here, we've defined a function that will take a number of trials and a probability of success on each trial. It will then sample some data under those parameters, before testing it against the probability of success. Finally, it will extract the p-value from that test and return it to you. Let's see this in action.

Run and rerun the code below several times to see what happens. What do the p-values look like? Remember, a value below 0.05 is typical in psychology for the cut-off point where we'll reject the null hypothesis of no difference between the observed and predicted probabilities of success.

```r
simulate_psychic_test(n_trials = 100, prob_success = 0.2)
```

```
## [1] 0.5322562
```

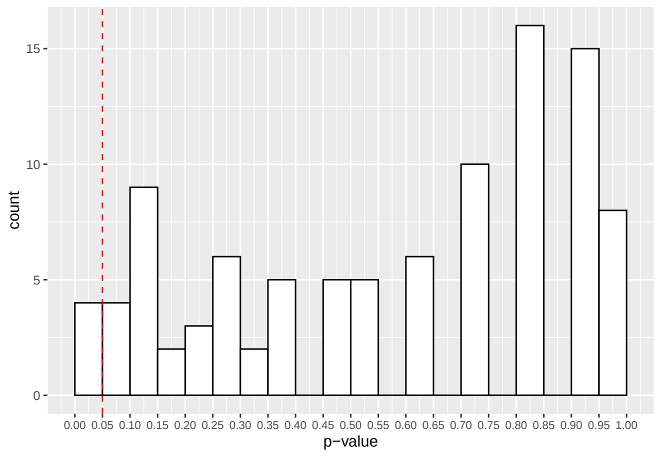
Ok, so you've seen how these p-values shift even when we know there's no true effect. Let's simulate the experiment 100 times to see what happens to the distribution of the p-values.

Nicely, R has a `replicate()` function, which replicates whatever we ask it to do by a pre-defined number. This replicate function takes `n` as an argument, which is the number of replication to carry out, plus whatever you want to replicate.

Can you predict what will happen?

```
simulations <- replicate(n = 100,
                         simulate_psychic_test(n_trials = 100,
                                               prob_success = 0.2
                                               )
                         )
```

```
ggplot() +
  geom_histogram(mapping = aes(simulations),
                 binwidth = 0.05,
                 boundary = 0,
                 fill = "white",
                 colour = "black") +
  scale_x_continuous(limits = c(0, 1),
                     breaks = seq(from = 0, to = 1, by = 0.05)
                     ) +
  geom_vline(xintercept = 0.05, linetype = 2, colour = "red") +
  labs(x = "p-value") +
  theme(axis.text.x = element_text(size=8))
```

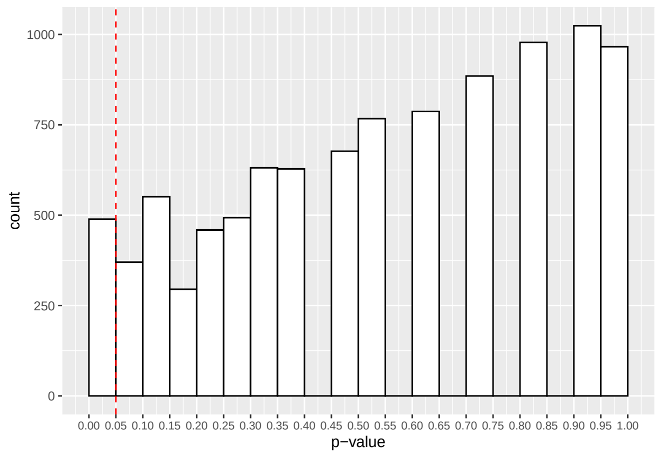


The red cut-off line shows the number of values that fall below our 0.05 cut-off point for our p-values. That looks like roughly 3 in 100 samples, or 3% of our simulated studies.

Try increasing the number of replications for the experiment to see how the p-values change. How does it look with 10,000 simulated studies?

```r
simulations <- replicate(n = 10000,
                         simulate_psychic_test(n_trials = 100,
                                               prob_success = 0.2
                                               )
                         )
```

```r
ggplot() +
  geom_histogram(aes(simulations),
                 binwidth = 0.05,
                 boundary = 0,
                 fill = "white",
                 colour = "black") +
  scale_x_continuous(limits = c(0, 1),
                     breaks = seq(from = 0, to = 1, by = 0.05)
                     ) +
  geom_vline(xintercept = 0.05, linetype = 2, colour = "red") +
  labs(x = "p-value") +
  theme(axis.text.x = element_text(size=8))
```



Here we have roughly 500 studies showing p-values below 0.05. Thats 5% of the 10,000 studies. Is that surprising? Even when there's no effect in the population, with repeated sampling we're bound to find some cases where we have p-values below 0.05. That's why replications are so important for our research!

This false-positive rate is the type-I error rate of our study, and is determined by our p-value. This determines the amount of time we will incorrectly reject the null hypothesis in the long run (i.e. if we run several studies over time).

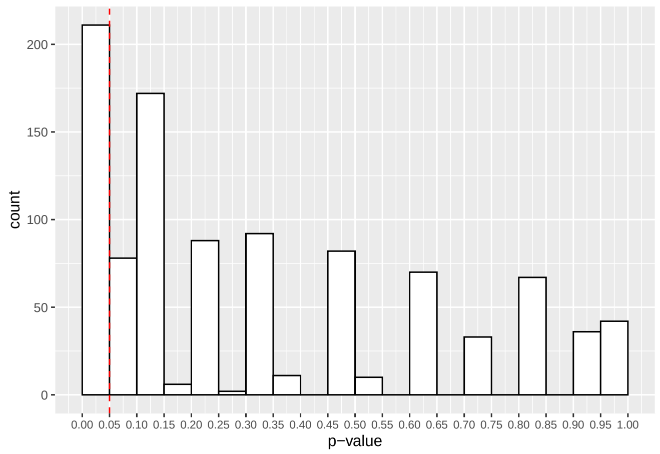
8.3.2 Understanding Power and Type-II Errors

What if we change our function so that we can give the psychic a different probability to the true probability of a success?

```r
# define function name and parameters to vary
simulate_psychic_test <- function(n_trials, prob_success, psychic_success) {

  # sample binomial data (successs or not) for varying trials and probabilities
  psychic_outcome <- rbinom(n = 1, size = n_trials, prob = psychic_success)

  # run a binomial test on the sampled data against your proability
  test <- binom.test(x = psychic_outcome, n = n_trials, p = prob_success)

  # retrieve the p-value, this is what is returned in our function
  test$p.value
}
```

```r
true_effect_simulations <- replicate(n = 1000,
                         simulate_psychic_test(n_trials = 100,
                                               prob_success = 0.2,
                                               psychic_success = 0.25
                                               )
                         )
```

```r
ggplot() +
  geom_histogram(aes(true_effect_simulations),
                 binwidth = 0.05,
                 boundary = 0,
                 fill = "white",
                 colour = "black") +
  scale_x_continuous(limits = c(0, 1),
                     breaks = seq(from = 0, to = 1, by = 0.05)
                     ) +
  geom_vline(xintercept = 0.05, linetype = 2, colour = "red") +
  labs(x = "p-value") +
  theme(axis.text.x = element_text(size=8))
```



Now it looks like we have a much larger proportion of studies that give us *p*-values below 0.05. Actually, it's 211, or 21.1%.

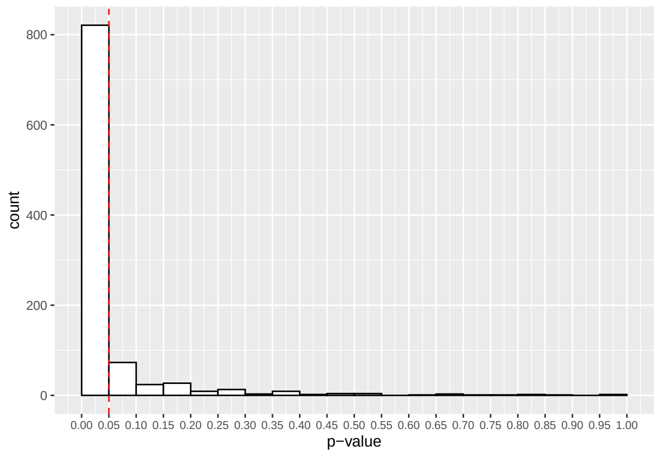
What does this tell us? Actually, this is the power of our study, or the probability for us to detect a true effect when it is present in the population. As such, 1-power is the type-II error rate, or the probability to reject the alternative hypothesis when a true effect is present in the population.

At only 21.1%, this is inadequate for our purposes. We generally aim for at least 80% in psychological research. Here we have a much larger chance of committing a type-II error, or incorrectly rejecting the alternative hypothesis when there is a true effect in the population.

We can use this simulating method to help determine the parameters of our study. Let's look into increasing the sample size in our study, or the number of observations we make during the experiment.

```
true_effect_simulations <- replicate(n = 1000,
                         simulate_psychic_test(n_trials = 600,
                                               prob_success = 0.2,
                                               psychic_success = 0.25
                                               )
                         )
```

```
ggplot() +
  geom_histogram(aes(true_effect_simulations),
                 binwidth = 0.05,
                 boundary = 0,
                 fill = "white",
                 colour = "black") +
  scale_x_continuous(limits = c(0, 1),
                     breaks = seq(from = 0, to = 1, by = 0.05)
                     ) +
  geom_vline(xintercept = 0.05, linetype = 2, colour = "red") +
  labs(x = "p-value") +
  theme(axis.text.x = element_text(size=8))
```



That looks better! Now, in most experiments, we'll get a *p*-value below 0.05. How many studies detected this true effect? Our power for this study is:

```
mean(true_effect_simulations < 0.05)
```

```
## [1] 0.821
```

Great, at 600 trials we now have an 82.1% chance to detect a true effect if present. It looks like we'll have one tired psychic by the end of this experiment!

As long as you can construct data that fits the form of your planned experiment you can use simulation to calculate the predicted power under the parameters that you set. As long as we have *t*- or *p*-values to capture, we can simulate data, fit our model, and count the number of *p*-values below our threshold (or *t*-values above our threshold) where we reject the null hypothesis in order to come up with the power of our study. Typically, around 1000 samples is enough for this.

8.3.3 Flexible Power Analyses with Simulation

8.3.3.1 Simulation for Simple Designs

Now we understand how we can use simulation to create data sets and observe the probability that we'll detect a true effect with a simple design, we can build on our knowledge to look at more complex designs.

Let's return to the IQ study data. Remember, that the IQ test has a mean of 100 and standard deviation of 15. Let's run a power analysis to detect whether a predicted sample of people differ from this average range.

```r
calculate_IQ_power <- function(n, sample_mean, sample_sd, pop_mean, pop_sd) {
  sample_data <- rnorm(n = n, mean = sample_mean, sd = sample_sd)
  test <- t.test(sample_data, mu = pop_mean, sd = pop_sd)
  test$p.value
}
```

Now let's run our function, replicated 1000 times. We'll sample assuming our population and sample have the same standard deviation, but the sample mean is 5 points higher than the general population. Finally, we'll sample 60 people.

We'll then calculate the mean number of *p*-values below our 0.05 cut off point to see our power.

```r
p_vals <- replicate(1000, calculate_IQ_power(60, 105, 15, 100, 15))
mean(p_vals < 0.05)
```

```
## [1] 0.735
```

Great, 73.5% power is not bad at all. Maybe we should bump up the sample size a little to get to 80%.

```r
p_vals <- replicate(1000, calculate_IQ_power(75, 105, 15, 100, 15))
mean(p_vals < 0.05)
```

```
## [1] 0.835
```

It looks like around 75 participants does a good job for us. What if the standard deviation for our sample was smaller? i.e. what if the samples tend to be more similar in their mean scores?

```r
p_vals <- replicate(1000, calculate_IQ_power(20, 105, 7.5, 100, 15))
mean(p_vals < 0.05)
```

```
## [1] 0.82
```

Now it looks like we only need around 20 participants to achieve the same power.

What if the mean score is half as large as before?

```
p_vals <- replicate(1000, calculate_IQ_power(70, 102.5, 7.5, 100, 15))
mean(p_vals < 0.05)
```

```
## [1] 0.793
```

Now we need more participants, even though the standard deviation is as small as before.

As you can see, sample size, effect sizes, and variance all affect the ability to detect a true effect.

8.3.3.2 Simulation for More Complex Designs

So far, we've looked at simulation for cases where we have a baseline condition to compare against. However, we often want to compare to or more groups against one another to determine if they differ on some outcome variable. For a *t*-test, this is as simple as sampling two sets of data using `rnorm()` and comparing the two against one another. However, we'll look at a more complex 2 by 2 between-subjects design, which can be adapted to most scenarios.

Here, participants take part in 1 combination of 2 factors (A and B) each with 2 levels. To get started we first have to create a tibble to hold our simulated data.

```
# how many subjects?
n <- 60

# create your design
design <- expand.grid(A = c(-.5, .5), B = c(-.5, .5))

# create table of data
data <- tibble(subject = 1: n,
               A = rep(design$A, n/4),
               B = rep(design$B, n/4)
               )

# look at the data
data
```

```
## # A tibble: 60 x 3
##    subject     A     B
##      <int> <dbl> <dbl>
##  1       1  -0.5  -0.5
##  2       2   0.5  -0.5
##  3       3  -0.5   0.5
##  4       4   0.5   0.5
##  5       5  -0.5  -0.5
##  6       6   0.5  -0.5
##  7       7  -0.5   0.5
##  8       8   0.5   0.5
##  9       9  -0.5  -0.5
## 10      10   0.5  -0.5
## # ... with 50 more rows
```

Notice that we defined our two condition columns with the values of -.5 and .5. Think back to how this is modelled in a linear model in R. What might the intercept term be? It will be the mean across both conditions.

Next, we need to set up the coefficients and error term for our model. We want the intercept to be 400, a shift in A to be the intercept + 30, a shift in B to be the intercept - 50, and the interaction term to be the intercept - 40. Finally, our error term has a mean of 0 and standard deviation of 40.

We then create our data using the formula for the linear model, and add this to our data.

```
# set up coefficients
alpha = 400
beta1 = 30
beta2 = -50
beta3 = -40
err <- rnorm(n, 0, sd = 40)

# create data from the linear model
data$y <- alpha + (beta1*data$A) + (beta2*data$B) + (beta3*data$A*data$B) + err

# see the data
data
```

```
## # A tibble: 60 x 4
##    subject     A     B     y
##      <int> <dbl> <dbl> <dbl>
##  1       1  -0.5  -0.5  421.
##  2       2   0.5  -0.5  472.
##  3       3  -0.5   0.5  408.
##  4       4   0.5   0.5  443.
##  5       5  -0.5  -0.5  423.
##  6       6   0.5  -0.5  479.
##  7       7  -0.5   0.5  405.
##  8       8   0.5   0.5  386.
##  9       9  -0.5  -0.5  498.
## 10      10   0.5  -0.5  428.
## # ... with 50 more rows
```

Let's see how this data would look when modelled.

```
fit <- lm(y ~ A * B, data = data)
summary(fit)
```

```
##
## Call:
## lm(formula = y ~ A * B, data = data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -108.540  -30.991    8.045   24.951   89.673
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  401.193      5.694  70.456  < 2e-16 ***
## A             17.397     11.388   1.528   0.1322
## B            -55.695     11.388  -4.890 8.85e-06 ***
## A:B          -47.358     22.777  -2.079   0.0422 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 44.11 on 56 degrees of freedom
## Multiple R-squared:  0.3532, Adjusted R-squared:  0.3185
## F-statistic: 10.19 on 3 and 56 DF,  p-value: 1.868e-05
```

Finally, we need to fit a model to our simulated data and extract the *p*-value. We need to do this fitting and extracting process several times before we can calculate our power to detect an effect. However, we need to know which effect we're interested in detecting. Is it just an effect of A, B, or the interaction? That choice is up to you, but for now we'll simply look at the probability to detect an effect for the interaction.

Thankfully, we can save on writing long functions and/or loops when we have some data like this by fitting our model once, and then using the `simulate()` function in R.

In this function, we need to pass our original data and model fit. We then create new data from our original data, but with a simulated response variable (`sim_y`) made from the `simulate()` function which is applied to our original model. We then refit the model with our new data and response variable, and we use a helper function `broom::tidy()` which puts model outputs in a nice table so we can easily select the *p*-value for the 4th row (i.e. the interaction). Try running `str()` on both **fit** and **tidy_table** to see what I mean.

```
simulate_model_fit <- function(data, model_fit) {
  # simulate data and add to tibble
  new_data <- mutate(data, sim_y = simulate(fit)$sim_1)

  # refit model
  refit_model <- lm(sim_y ~ A * B, data = new_data)

  # extract p-value for interaction
  broom::tidy(refit_model)$p.value[4]
}
```

Finally, we can use `replicate()` to do this process however many times we'd like. We then calculate the power by seeing how often we find *p*-values below 0.05 for the interaction term.

```
complex_power <- replicate(1000, simulate_model_fit(data, fit))
mean(complex_power < .05)
```

```
## [1] 0.529
```

Nice, we have very high power for this study!

Feel free to reuse and adapt the script above to your own needs. Try looking at the impact of inputting different parameter estimates and designs.

8.4 Performing Power Analyses from Packages

The first option of calculating power through simulation is very flexible, and I think it helps you to think more deeply about your data when compared to running canned power analyses. Also, there aren't many ways to easily calculate power with mixed effects models, and so simulation is our best bet at the moment. However, there are options out there if you want to calculate power for most designs without the need to do simulation.

R has an inbuilt power analysis function for *t*-tests called `power.t.test()`, and for ANOVAs called `power.anova.test()`.

Let's see how the ANOVA function works, and you can work back to the *t*-test function if necessary. In this function, we just have to define the number of groups we have, the number of observations in each group (typically participants in psychology), and the variance for any between-subjects and within-subjects variables.

Making the below assumptions, it looks like our study is overpowered.

```
power.anova.test(groups = 4, n = 20, between.var = 1, within.var = 3)
```

```
##
##      Balanced one-way analysis of variance power calculation
##
##          groups = 4
##               n = 20
##     between.var = 1
##      within.var = 3
##       sig.level = 0.05
##           power = 0.9679022
##
## NOTE: n is number in each group
```

Alternatively, we can predefine our desired power (typically 80% in psychology), and this will work out the number of participants needed per group.

```
power.anova.test(groups = 4, between.var = 1, within.var = 3, power = .80)
```

```
##
##      Balanced one-way analysis of variance power calculation
##
##          groups = 4
##               n = 11.92613
##     between.var = 1
##      within.var = 3
##       sig.level = 0.05
##           power = 0.8
##
## NOTE: n is number in each group
```

Great, it looks like we only need 12 people in each group to achieve a power of 80%.

There are other options which allow you to specify means or effect sizes to determine power, and as such these may be more user friendly. One option is the `pwr` package, which contains a number of functions for calcualting power with different designs. Have a look at this in your own time to see if it will be useful to you.

8.5 Exercises

8.5.1 Introduction and Setup

For these exercises, we will look at the core concepts from this lesson. Here, you'll make your own data using simulation-based techniques, and you'll use those same techniques to calculate power for your design.

8.5.2 Question 1

Create some data for 60 participants (split into two groups), A and B. Save this as a tibble called **sim_data**.

Make a column containing the subject ID (called **subject_id**), a column for condition (called **condition**), and a column for the outcome (called **outcome**).

You will have to use the `seq()` function to make sequential numbers for the subject ID, the `rep()` function with the `c()` function to make the conditions, and the `rnorm()` function with the `c()` function to make the scores.

For the outcome column, we want a mean of 180, and standard deviation of 10 for the first group, and a mean of 190 and standard deviation of 10 for the second group.

8.5.3 Question 2

Generate summary statistics for the data set. We want means, SDs, and ns for the table of summary statistics. How do these means and standard deviations match the sampling procedure? Are you surprised at all by these values?

8.5.4 Question 3

Conduct an independent samples *t*-test looking at differences in the scores across the two groups.

8.5.5 Question 4

Turn the sampling procedure into a function called `sim_data`.

The arguments the function can take are:

- n_one: the number of observations for group 1
- n_two: the number of observations for group 2

- mean_one: the mean score in the population for group 1
- mean_two: the mean score in the population for group 2
- sd_one: the standard deviation in the population for group 1
- sd_two: the standard deviation in the population for group 2

Remember that these arguments come within the paretheses after the blue `function` part.

Hint: Just copy and paste your code for 1, and replace the things you want to vary with the name of a variable that will define your function arguments.

Test that your function works by passing it some parameters. Pass it an n of 80 for both groups, a mean of 200 and SD of 10 for group 1, and a mean of 195 and SD for 10 for group 2.

Assign the outcome of your function to an object to save the outcome. Call this object **new_data**.

8.5.6 Question 5

Make a new function called `sim_test`, which uses your simulation function from above to make some data before it runs an independent samples *t*-test on the data. Finally, this function should output the *p*-value from the *t*-test. Note that this function should take as arguments the same arguments from the function in question 4. Can you figure out why this is the case?

Test that your function works by passing it the same arguments as in question 4, and assign this to the object `new_data_p`.

8.5.7 Question 6

Calculate the power for your study based on the parameters from question 4 and 5 (ns of 80; mean of 200 and 195, SDs of 10). Do so by running your function 1000 times and capturing the p-values. Assign this to the object **power_data**.

Top Tip: If you need to test your code (which you should), make sure you use a low number like 10 for your replications before running it 1000 times to answer this question. This will save time and avoid flooding your console etc. if you make a mistake.

8.5.8 Question 7

Calculate the power of your study from the *p*-values in question 6. What is the power for your study? (Your values will differ from mine as we aren't using `set.seed()`, but we're instead using proper random sampling.)

8.5.9 Question 8

Make a plot of the p-values from your simulated data. Make this with a binwidth of 0.05, a boundary of 0 and with white bars with a black border.

Also, make sure that the breaks happen at every 0.05 tick along the x-axis.

What percentage of your data lie below the 0.05 line? What does this correspond to?

8.5.10 Question 9

Remake your data from question 6, only with means of 200 in each group. Call this **power_data_null**.

Then calculate the power as in question 7, and plot your data as in question 8.

Note: The sampling procedure may take a while to run.

What percentage of the data fall below the 0.05 line on the plot? What does this correspond to? What percentage falls within each bin above the 0.05 line? Are you surprised at the distribution of the p-values?

Chapter 9

Mixed Effects Models

In this session we'll cover Linear/Hierarchical Mixed Effects Modelling. We'll cover why you should use mixed effects modelling for your own analyses, how these models work, and how to define your models properly in R. Specifically, we'll cover:

- Fixed and Random Effects
- Random Intercepts and Slopes
- Nested and Crossed Random Effects
- Partial-Pooling of Data
- Calculating *p*-values
- Generalised Mixed Effects Models

One nice introduction to mixed effects models is provided by Bodo Winter in two parts: part one and part two and you might want to check this out as further reading from this chapter.

9.1 Getting Started

As always, we first need to load the `tidyverse` set of package. However, for this chapter we also need the `lme4` package. This package allows us to run mixed effects models in R using the `lmer` and `glmer` commands for linear mixed effects models and generalised linear mixed effects models respectively. These models are similar to linear models and generalised lienar models in that the first can take continuous, unbounded data, and the second takes bounded, discrete data.

```r
# load packages
library(tidyverse)
library(lme4)
```

Additionally, we'll load the data set for this study. This is the inbuilt `sleepstudy` data set from `lme4`, but I've simulated 2 additional participants before removing a lot of their observations. This is to show you just how powerful mixed effects models are when we have missing data.

9.2 Why Mixed Effects Models

In all of the previous analyses we've used so far, our models have assumed that we have independence between cases in our data. However, this assumption is often violated, which can cause some serious problems when it comes to interpreting our tests.

Let's assume we're interested in language learning in children from monolinguial and bilingual backgrounds. We may decide to test how well children learn an artificial language by going to several schools and testing children in the same year group from different classes.

If we fit a traditional linear model to our data, we implicitly make the assumption that learning should be the same regardless of the class or school the child attends. Yet we know this is unlikely to be true; schools reflect the demographics of an area and the school ethos, and individual teachers may be better or worse at teaching linguistics to children. This means that children within the same school are likely to be more similar to one another than to children from different schools. Moreover, children within the class are likely to be more similar to one another than to children in different classes. In this way, our data should be nested such that children should be defined as part of a specific class in a specific school, and our models should reflect this ordering.

With mixed effects models, we can directly model this dependency in our data. Moreover, mixed effects models make fewer strict assumptions to other tests (Field et al. (2012)), such as:

- Homogeneity of regression slopes: Mixed effects models can directly model variability in slopes, so we needn't make any assumption that slopes are similar across conditions (which is often untrue).
- Assumption of independence: with independent-samples tests we often assume that data are unrelated to one-another. But again, this is often not the case; take learning over time. The same person provides multiple measurements at multiple time points. Mixed effects models can handle this well.
- Complete data: Often, with traditional analyses you just have to throw away entire sources of data (e.g. participants) if one case (e.g. one trial) is missing. This isn't the case with mixed effects models, where we can estimate missing data based on what we have available to us.

For these reasons, by default you should probably pick mixed effects models to model your data.

9.3 How Do Mixed Effects Models Work?

9.3.1 Fixed and Random Effects

So far, we've only talked about our parameter estimates from a linear model in terms of factors and levels. However, you can also consider the factors in your study in terms of whether they are fixed or random effects. Typically, a **fixed effect** will contain all possible levels of a factor in the experiment, while a **random effect** will be a random sample of possible levels of a factor. In effect, we can only generalise our fixed effects results to the levels within an experiment, but we can generalise from our random effects to levels beyond those that took part in our study.

Returning to our classroom example, if we tested languge learning in one of two conditions in students from different schools and and classes, our fixed effect would be the condition for the study. Hence, the random effects would be everything else we'd like to generalise to outside of our study. We probably want to generalise our results to all classes and all schools, so our random effects would be students within classes within schools.

In traditional models, you can say that all of the effects of interest, or the factors in the model, are defined as fixed effects. However, with mixed effects models, we can have both fixed and random effects in our model (hence **mixed effects** models).

To fit a mixed effects model, we simply define our fixed effects like we normally do for a linear model. However, there are different ways that we can define our random effects. Your choice of definition incorporates your belief of how the random effects work. Let's explore how the way in which we define our random effects influences our model fit.

9.3.1.1 Fixed Intercept and Slope

To look at how mixed effects models work, we'll use some toy data based on the `sleepstudy` data set from **lme4**. I made this quickly in another script, and you can load it using the code below.

This just makes three groups within our data, and we can look at this like conducting our study in three phases, with three different groups of people. As with the `sleepstudy` data set, this data set looks at the effect of days without sleep on reaction times.

```
sleep_groups <- read_csv("inputs/sleep_study_with_sim_groups.csv")
```

```
## Rows: 540 Columns: 4
## -- Column specification ------------------------------------
## Delimiter: ","
## chr (1): Group
## dbl (3): Reaction, Days, Subject
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

If we were to fit this data to a traditional linear model, and we ignored the fact that we have a random effect in our study (Group), then our linear model coefficients might look like this:

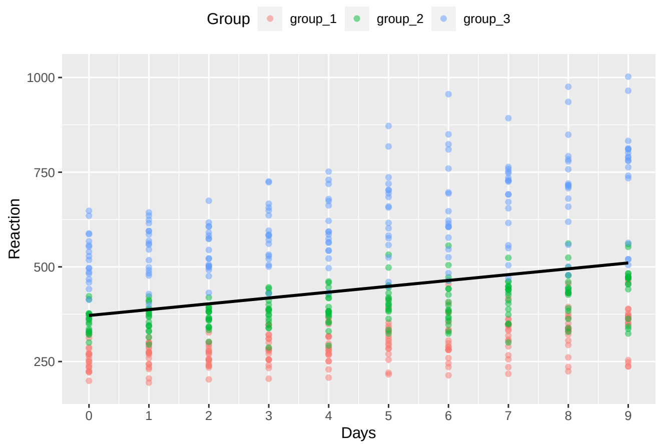
```
lm(Reaction ~ Days, data = sleep_groups)
```

```
##
## Call:
## lm(formula = Reaction ~ Days, data = sleep_groups)
##
## Coefficients:
## (Intercept)         Days
##       371.7         15.4
```

That is, we would get an intercept of approximately 351.5 and a slope of approximately 10.3 which is used to define the relationship for days on reaction times across all participants. This type of model looks against the raw data for the study looks like this:

```
ggplot(data = sleep_groups, mapping = aes(x = Days, y = Reaction)) +
  geom_point(na.rm = T, aes(col = Group), alpha = 0.5) +
  geom_smooth(method = "lm", na.rm = T, col = "black", se = F) +
  scale_y_continuous(limits = c(180, 1020)) +
  scale_x_continuous(breaks = seq(1:10) - 1) +
  theme(legend.position = "top")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



But, you can see from the colour of the points that we have 3 distinct groups of participants in our model. It's clear that the three groups have different intercepts, and group 3 has a much steeper slope than the other two groups. Perhaps, then, we should model this difference across groups, even if it's not the main factor of interest in our study.

9.3.1.2 Random Intercepts

We define a mixed effects model in a similar way to a traditional linear model. Here, the only difference is we run the model using the `lmer()` function from **lme4**, rather than the `lm()` function from base R, and we specify our random effects as well.

```
# fit random intercepts model
intercepts_model <- lmer(Reaction ~ Days + (1 | Group), data = sleep_groups)
```

As with regular linear models, we define our response variable as the column name to the left of the `~` (read: tilde). We then define our fixed effects to the right of the `~`.

After this, we define our random effects in parentheses. Here, we have specified that we want **random intercept (1)** for each group; `(1 | Group)`. Notice that we have a `|` (read: pipe) between the 1 and group. This specifies what you want to calculate, by which random factor - here it is random intercepts (1) by the Group random factor.

Finally, as always we specify our data set, which should be in long format.

Let's see how defining random intercepts affects the coefficients for the groups in our model. We can extract model coefficients using the `coef()` function. I specifically got the Group coefficients by asking for this using the dollar notation after the function call. To make this easier on the eyes I've renamed the columns, and created an identifier for the intercepts and slopes (i.e. the Group column).

```
# see group coefficients
model_coefs <- coef(intercepts_model)$Group %>%
  rename(Intercept = `(Intercept)`, Slope = Days) %>%
  rownames_to_column("Group")

# see coefficients
model_coefs
```

```
##     Group Intercept    Slope
## 1 group_1  229.3394 15.40385
## 2 group_2  329.0908 15.40385
## 3 group_3  556.6432 15.40385
```

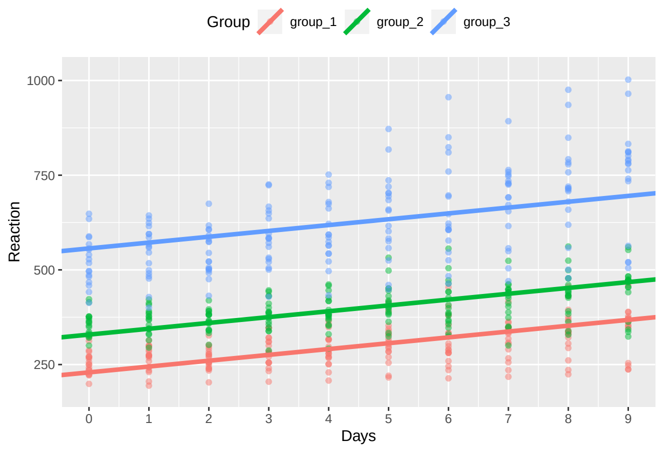
How would this model look against our raw data?

First, we'll join these coefficients to our original data so we can plot the individual lines.

```
sleep_groups_rani <- left_join(sleep_groups, model_coefs, by = "Group")
```

Then we'll plot our original data with our new, random intercepts model. We'll save this under the name `model_coef_plot` because we can use a simple ggplot function to update models without repeating code.

```
model_coef_plot <- ggplot(data = sleep_groups_rani,
       mapping = aes(x = Days,
                     y = Reaction,
                     colour = Group)
       ) +
  geom_point(na.rm = T, alpha = 0.5) +
  geom_abline(aes(intercept = Intercept,
                  slope = Slope,
                  colour = Group
                  ),
              size = 1.5
              ) +
  scale_y_continuous(limits = c(180, 1020)) +
  scale_x_continuous(breaks = seq(1:10) - 1) +
  theme(legend.position = "top")

# see the plot
model_coef_plot
```



Now you can see that we fit a model where the intercept (the y value at x = 0) differs across the 3 groups, but the slope of the lines is the same.

You might pick a random effects structure with random intercepts for data like this if you expect the three groups to differ from one another at the start of testing, but the effect of days on reaction times is the same across groups.

9.3.1.3 Random Slopes

Alternatively, we can specify a random effects structure with random slopes but fixed intercepts. We might pick this form if we expect all groups to start off at around the same score, but that the effects of days on reaction times differs across groups.

To specify this random effects structure, we change the 1 to a 0 in our random effects structure (specifying no or 0 random intercepts) and we put Days to the left of the |, meaning that we want to calculate random slopes for the effect of Days for each Group.

```
# fit random slopes model
model <- lmer(Reaction ~ Days + (0 + Days | Group), data = sleep_groups)

# see group coefficients
model_coefs <- coef(model)$Group %>%
  rename(Intercept = `(Intercept)`, Slope = Days) %>%
  rownames_to_column("Group")

# see coefficients
model_coefs
```

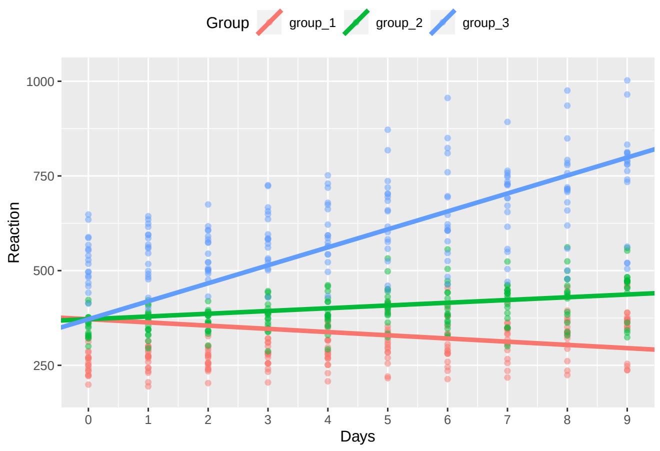
```
##     Group Intercept     Slope
## 1 group_1  371.6912 -8.479087
## 2 group_2  371.6912  7.249332
## 3 group_3  371.6912 47.441298
```

Next, we'll join these coefficients to our original data.

```
sleep_groups_rans <- left_join(sleep_groups, model_coefs, by = "Group")
```

Then we'll plot our original data with our new, random intercepts model. Here, we'll use a new, cheat function from ggplot, %+% (read: add components). This takes a fitted plot from ggplot, and replaces the data from that plot with whatever comes to the right of the function.

```
model_coef_plot %+% sleep_groups_rans
```



In the above plot, we can now see that our data has the same intercept across groups, but different slopes. It seems that the effect of days without sleep is greatest for group 3.

9.3.1.4 Random Intercepts and Slopes

Finally, we can specify that we want random intercepts and slopes for the group. Here, we just change the 0 back to a 1 in the random effects call.

```
# fit random intercepts model
model <- lmer(Reaction ~ Days + (1 + Days | Group), data = sleep_groups)
```

```
## Warning in checkConv(attr(opt, "derivs"), opt$par, ctrl = control$checkConv, :
## Model failed to converge with max|grad| = 0.00268565 (tol = 0.002, component 1)
```

Let's see how these coefficients look.

```
# see group coefficients
model_coefs <- coef(model)$Group %>%
  rename(Intercept = `(Intercept)`, Slope = Days) %>%
  rownames_to_column("Group")

# see coefficients
model_coefs
```

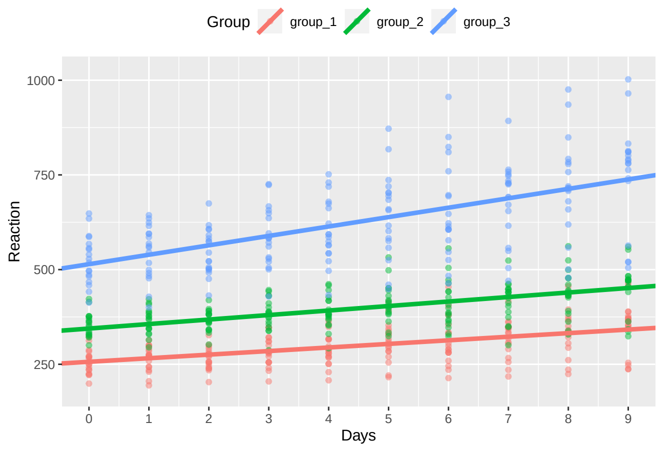
```
##     Group Intercept     Slope
## 1 group_1  256.4951  9.457048
## 2 group_2  344.2963 11.894270
## 3 group_3  514.2821 24.860226
```

Next, we'll join these coefficients to our original data.

```
sleep_groups_ranis <- left_join(sleep_groups, model_coefs, by = "Group")
```

Then we'll plot our original data with our new, random intercepts and random slopes model.

```
model_coef_plot %+% sleep_groups_ranis
```



As you can see, our model now fits random intercepts and slopes for each group of our participants.

9.3.2 Specifying your Random Effects Structure

There's some debate on how to specify your random effects, but one good rule of thumb is to keep it maximal. From several simulations, Dale Barr and colleagues have found that models that use the maximal random effects structure justified by the design of the study tend to have a good balance between power and type-I error rates (specifically keeping error below the nominal α).

What does a maximal random effect structure look like? Typically, it will fit random intercepts, slopes, and the correlation between the two for any main effects and interactions in the model.

Not all researchers agree that this is always the best choice. Indeed, some argue that you should evaluate the goodness of fit for models with and without more complex terms in the model (such as random slopes and interactions) in order to get the best trade off between power and type-I error rates, especially at smaller sample sizes, and that maximal models inflate type-II error rates. The choice is up to you, but be informed about the choice you make.

9.3.2.1 Crossed and Nested Random Effects

To understand how to define your random effects, we'll look at some more simulated data. For illustration purposes, we won't fit these models (we'd really need more data for that), but just pay attention to the structure of the models.

9.3.2.1.1 Crossed Random Effects The random effects structure that you take should reflect the reality of the data. If you have a design where observations can be assigned to more than one random effect simultaneously, then these random effects are said to be crossed.

This is often the case when you have a repeated measures design where participants provide multiple observations through responding to several items. In this instance, each observation comes from an individual subject on an individual item at the same time, making each observation a unique combination of these random factors. Hence, the random effects of subjects and items are said to be crossed.

For this example, our data could look like the following:

```r
crossed_data <- tibble(
  Subject = rep(1:2, 5),
  Item = rep(1:5, each = 2),
  Condition = c(rep(c("A", "B"), 2), rep(c("B", "A"), 3)),
  Response = rnorm(n = 10, mean = 100, sd = 10)
)
crossed_data
```

```
## # A tibble: 10 x 4
##    Subject  Item Condition Response
##      <int> <int> <chr>        <dbl>
##  1       1     1 A             92.8
##  2       2     1 B             85.0
##  3       1     2 A            108.
##  4       2     2 B            109.
##  5       1     3 B            100.
##  6       2     3 A            106.
##  7       1     4 B             92.8
##  8       2     4 A            111.
##  9       1     5 B            118.
## 10       2     5 A            105.
```

Here, both participants take part in both conditions (A or B), hence this is a within-subjects design (but note that this dummy set is not balanced; real designs should be where possible). The condition (A or B) is therefore within individual subjects and individual items, but items are not only associated with one subject. Hence, our random effects are crossed, and should look like this:

```
lmer(Response ~ Condition + (1 | Subject) + (1 | Item), data = crossed_data)
```

9.3.2.1.2 Nested Random Effects Sometimes we get cases when our data are not crossed. For example, if you're testing participants from separate classes in separate schools, then your data can be said to be nested. The data are nested because observations come from students, which are a part of a specific class, and a specific school. In this instance, you should construct your data to reflect this nesting, ending up with something like this:

```
nested_data <- tibble(
  Student = seq(1:10),
  Class = rep(seq(1:5), 2),
  School = c(rep(1, 5), rep(2, 5)),
  Intervention = rep(c("yes", "no"), 5),
  Outcome = rnorm(n = 10, mean = 200, sd = 20)
)

nested_data
```

```
## # A tibble: 10 x 5
##    Student Class School Intervention Outcome
##      <int> <int>  <dbl> <chr>          <dbl>
##  1       1     1      1 yes             195.
##  2       2     2      1 no              191.
##  3       3     3      1 yes              184.
##  4       4     4      1 no               169.
##  5       5     5      1 yes              249.
##  6       6     1      2 no               232.
##  7       7     2      2 yes              224.
##  8       8     3      2 no               199.
##  9       9     4      2 yes              170.
## 10      10     5      2 no               171.
```

As you can see, your data are structured so that you have individuals with an identifier (column) for the class that they're in and for the school that they're in. We then have a column indicating whether or not students received an intervention, and their outcome on some test.

As a result, class 1 in school 1 is different to class 1 in school 2. How do you tell R to treat the data such that classes are nested within schools? Or to account for the school and class effects for each participant? We use a formula like this:

```
lmer(Outcome ~ Intervention + (1 | School/Class/Student), data = nested_data)
```

Here, our random effects structure defines random intercepts for students, adjusting for similarities for students within the same class and school.

It's sometimes hard to figure out if you have nested or crossed (i.e. not nested) data, and you can help your thinking (and Rs functioning) by simply using unique class identifiers for your school and class combinations. We can make this by simply pasting the identifiers for the School and Class together.

```
nested_data$Class_ID <- paste(nested_data$School, nested_data$Class, sep = "_")
nested_data
```

```
## # A tibble: 10 x 6
##    Student Class School Intervention Outcome Class_ID
##      <int> <int>  <dbl> <chr>          <dbl> <chr>
##  1       1     1      1 yes             195. 1_1
##  2       2     2      1 no              191. 1_2
##  3       3     3      1 yes             184. 1_3
##  4       4     4      1 no              169. 1_4
##  5       5     5      1 yes             249. 1_5
##  6       6     1      2 no              232. 2_1
##  7       7     2      2 yes             224. 2_2
##  8       8     3      2 no              199. 2_3
##  9       9     4      2 yes             170. 2_4
## 10      10     5      2 no              171. 2_5
```

Now your results will be exactly the same if you use either of these two formulas:

```
lmer(Outcome ~ Intervention + (1 | School/Class_ID), data = nested_data)
lmer(Outcome ~ Intervention + (1 | School) + (1 | Class_ID), data = nested_data)
```

If you don't have unique identifiers, using this second structure will give you incorrect results as R will think that every class was in every school, which is not the case.

These random effects structures take the simplest form where we have random intercepts only. In most cases, it's best to fit a more complex model depending upon the data you have.

9.3.2.2 Exploring Different Random Effects Structres

Let's say we have a within subjects design, where participants see half of the items in our study in one condition, and half in another condition. Let's take the `crossed_data` example from before.

```
crossed_data
```

```
## # A tibble: 10 x 4
##    Subject  Item Condition Response
##      <int> <int> <chr>        <dbl>
##  1       1     1 A             92.8
##  2       2     1 B             85.0
##  3       1     2 A            108.
##  4       2     2 B            109.
##  5       1     3 B            100.
##  6       2     3 A            106.
##  7       1     4 B             92.8
##  8       2     4 A            111.
##  9       1     5 B            118.
## 10       2     5 A            105.
```

Here, we could fit random intercepts and slopes for subjects and items by adding condition (our slope term, or change in response for changes in 1 unit of condition) to the left of each term (along with the random intercept).

```
lmer(Outcome ~ Condition + (1 + Condition | Subject) + (1 + Condition | Item), data = data)
```

What if participants take part in several blocks of this study? We might expect that their performance will improve from block to block. Here, we'd just have to add a fixed effect of study block and random effects on any factors that are affected by the blocking. If you repeat your items across blocks with the same subjects, you'd have random effects of block on subjects and items (along with condition). If you have new items in each block, you'd only have random slopes of block for subjects. These random effects structures look like this:

```r
# same items within each block
lmer(
  Outcome ~ Condition + Block +
    (1 + Condition + Block | Subject) +
    (1 + Condition + Block | Item),
  data = data
  )

# new items within each block
lmer(
  Outcome ~ Condition + Block +
    (1 + Condition + Block | Subject) +
    (1 + Condition | Item),
  data = data
  )
```

As you might have guessed, for any between subjects factors, we cannot have a random slope of condition for that factor for each subject as subjects only see one condition. Here, our model might look like this:

```r
lmer(
  Outcome ~ Condition +
    (1 | Subject) +
    (1 + Condition | Item),
  data = data
  )
```

Where we can have random slopes for condition by item (as all items presumably see all conditions), but only random intercepts for subjects as they only see one condition.

Finally, as with all linear models, if we want an interaction between factors, we simply use an asterisk between factors to define our interactions, or a colon if we only want the interaction term and not main effects. This could take the form below if we want random intercepts and slopes for both factors on subjects and items, with main effects and interactions on our fixed factors.

```r
lmer(
  Outcome ~ factor_A * factor_B +
    (1 + factor_A * factor_B | Subject) +
    (1 + factor_A * factor_B | Item),
  data = data
  )
```

If you'd like to find out more about different structures, Kristoffer Magnusson has a great guide on this.

9.3.3 Partial-Pooling of Data

Here, we'll follow the first part of TJ Mahr's excellent write up of partial-pooling, and why this is so important for mixed effects models. Additionally, we'll see how partial-pooling allows mixed effects models to account for missing data.

We'll use the same data set as in TJ's article, the `sleepstudy` data set from the `lme4` package. I've already made some changes to this data set by convering it to a tibble and adding a few participants with missing data at the end by simulating new values based on the old participants, and simply deleting a few cells. Load the .csv file from the lesson materials folder to take a look at the data.

```
sleep_study <- read_csv("inputs/sleep_study_with_sim.csv")
```

```
## Rows: 210 Columns: 3
## -- Column specification ------------------------------------
## Delimiter: ","
## dbl (3): Subject, Days, Reaction
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

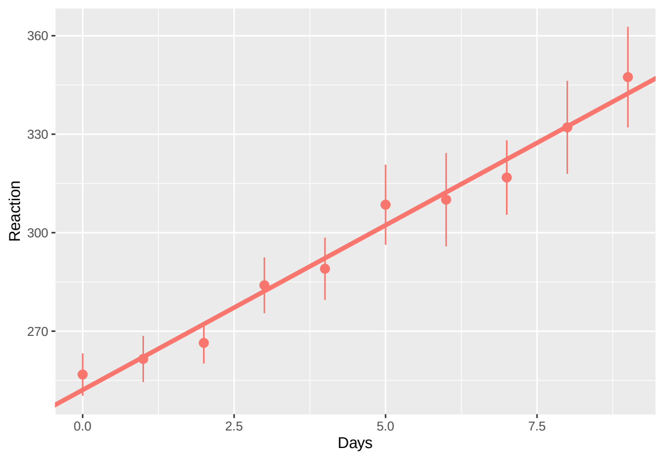
First off, we'll fit a linear model to the data using complete pooling. This means that all of our data from each participant is put together to come up with one mean intercept and slope which is the same across all participants.

```
# fit model
complete_pooling <- lm(Reaction ~ Days, data = sleep_study)

# tidy up and print model coefficients
complete_pooling_coefs <- coef(complete_pooling)
complete_pooling_coefs
```

```
## (Intercept)        Days
##   252.09397    10.03451
```

Next, we'll fit this model against our mean scores (with standard errors), observing how an increase in the days without sleep decreases reaction time.

```
ggplot(data = sleep_study, mapping = aes(x = Days, y = Reaction)) +
  geom_abline(aes(intercept = complete_pooling_coefs[1],
                  slope = complete_pooling_coefs[2]
                  ),
              colour = "#F8766D",
              size = 1.5
              ) +
  stat_summary(fun.data = "mean_se",
               geom = "pointrange",
               na.rm = T,
               colour = "#F8766D")
```

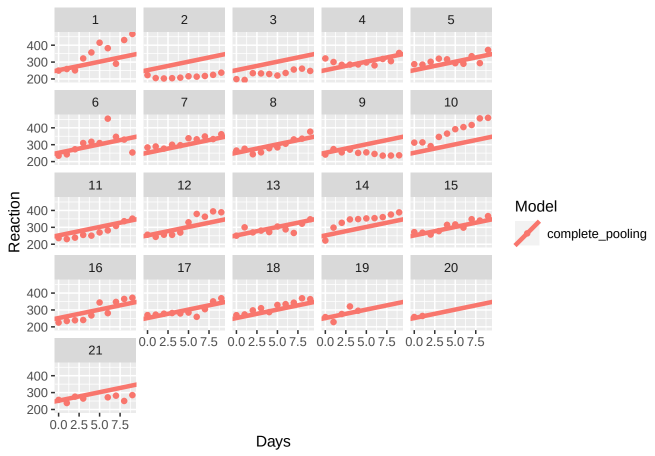


You can see that this line fits the mean scores pretty well. But how does the data look at the individual level? Again, we'll save this plot as an object so we can update it later on. We'll also join these coefficients with our data so that we can easily update our plot when we explore different pooling choices.

```
complete <- tibble(
  Subject = seq(1:21),
  Intercept = complete_pooling_coefs[[1]],
  Slope = complete_pooling_coefs[[2]],
  Model = "complete_pooling"
  )

model_coefs <- left_join(sleep_study, complete, by = "Subject")
```

Let's see how the data look with no pooling.

```
pooling_plot <- ggplot(data = model_coefs,
       mapping = aes(x = Days,
                     y = Reaction,
                     colour = Model)
       ) +
  geom_abline(aes(intercept = Intercept,
                  slope = Slope,
                  colour = Model),
              size = 1.5
              ) +
  geom_point(na.rm = T) +
  facet_wrap(~Subject)

# see the plot
pooling_plot
```



Wow, so it looks like the model fits participant 18 very well. But what about participants 1, 2, and 14? Maybe a different model would fit their data better. It doesn't do a good job of explaining their responses, that's for sure!

One alternative to this complete pooling method is the opposite: no pooling. Here, we instead fit an individual intercepts and slopes to each individual in the data. We can do this by using the `lmer()` function from **lme4**. Here, our formula takes the form where reaction times are a function of the days (of deprived sleep) within each participant.

```
# fit the model
no_pooling <- lmer(Reaction ~ Days | Subject, data = sleep_study)

# extract and view model coefficients
```

```
no_pooling_coefs <- coef(no_pooling)$Subject %>%
  rename(Intercept = `(Intercept)`, Slope = Days)

head(no_pooling_coefs)
```

```
##         Slope Intercept
## 1 20.60380801  249.4076
## 2 -0.02559952  219.9904
## 3  3.66038038  219.0879
## 4  4.24609733  281.1159
## 5  6.27936095  278.4033
## 6  9.56692414  263.1531
```

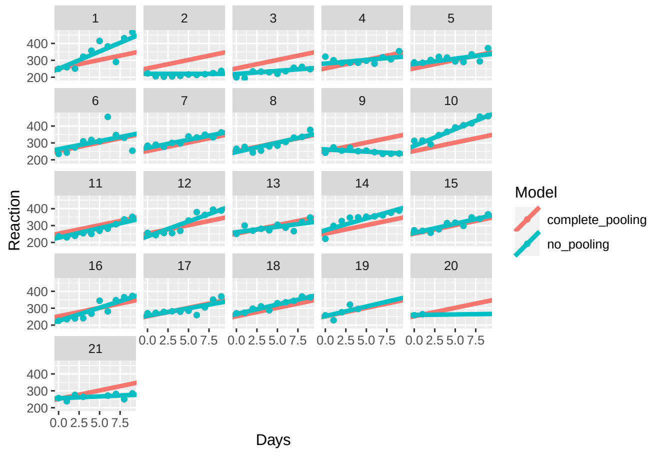
We can see for the first 6 participants their intercepts and slopes (Days) all differ from one another. How does this look in our plot? First, we'll add the two coefficients together for easy plotting.

```
none <- tibble(
  Subject = seq(1:21),
  Intercept = no_pooling_coefs$Intercept,
  Slope = no_pooling_coefs$Slope,
  Model = "no_pooling"
)
complete_none <- bind_rows(complete, none)
model_coefs <- left_join(sleep_study, complete_none, by = "Subject")
```

Now we've got our data in the correct format for plotting, we can have a look at how our models fit our data. Which do you think best matches the individual? Is this model any good when we have a lot of missing data?

Take a look at participant 20. Do you think the no pooling model is any better than the complete pooling model? What about participant 2?

```
pooling_plot %+% model_coefs
```



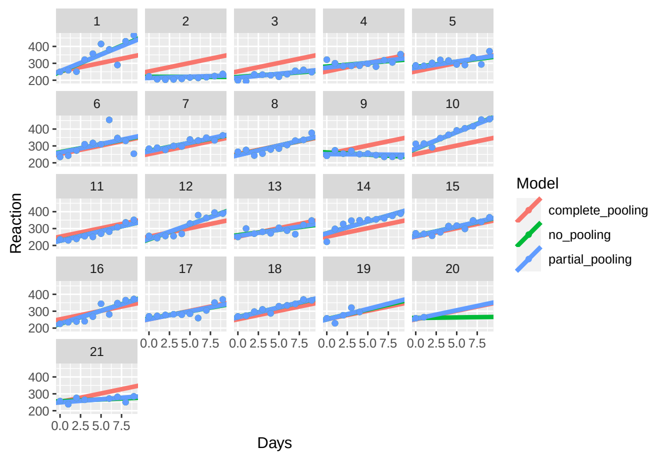
Finally, we get to where **lme4** shines; partial pooling of results. Here, we'll fit a mixed effects model where we have fixed effects of Days, which are assumed to be

```
partial_pooling <- lmer(Reaction ~ Days + (Days | Subject), data = sleep_study)
```

```
# extract model coefficients
partial_pooling_coefs <- coef(partial_pooling)$Subject

# make a tibble for partial pooling
partial <- tibble(
  Subject = seq(1:21),
  Intercept = partial_pooling_coefs$`(Intercept)`,
  Slope = partial_pooling_coefs$Days,
  Model = "partial_pooling"
)

# clean up and combine with other models
partial <- partial %>%
  left_join(sleep_study, by = "Subject")

all_pools <- bind_rows(model_coefs, partial)
```

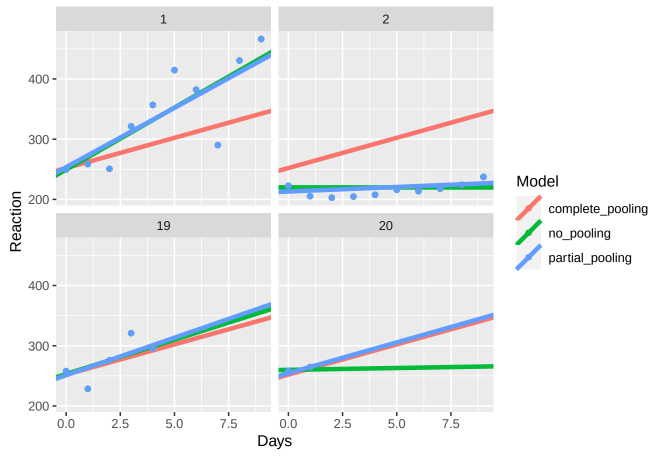
Next, we can plot the different models against one another to see what happens.

```
pooling_plot %+% all_pools
```



As with TJ's article, we'll also zoom in to some of these participants too see what happens.

```r
subset_pools <- all_pools %>% filter(Subject %in% c(1, 2, 19, 20))
pooling_plot %+% subset_pools
```



Here, you can see that in the first two cases (1 and 2), the coefficients of the partial pooling model closely matches that of the no pooling model, showing that this model closely captures the individual differences within each participant.

Look at participant 20. You'll notice that the partial and complete pooling lines are more similar to one another. That's because while the partial pooling model fits individual intercepts and slopes for each participant, it is pulled towards the mean, i.e. the overall intercept and slope across all participants. That's called shrinkage, where extreme values are pulled towards the average. This means that in cases where we have missing data, our model takes the best bet based on the grand mean, rather than trying to guess an individual intercept and slope based on that one person's data.

If you'd like to learn more about how this works, I really recommend consulting the source of this work, the article by TJ Mahr on partial pooling.

9.4 Interpreting Mixed Effects Model Output

Now that we've seen how mixed effects models work, we'll look at how to interpret the output of a mixed effects model. For this part, we'll use the `lexdec` dataset from the `languageR` library. Load this by loading the library and running the code below.

```
library(languageR)
lex_dec <- as.tibble(lexdec) %>%
  select(Subject, Trial, Word, NativeLanguage, RT)
```

In this data set, we have log transformed reaction times for English words rated by native and non-native speakers of English. Let's assume that we are interested in the influence of native language on reaction times. First, we want to set the intercept to the mean across both groups. Since the data isn't perfectly balanced, we should avoid the base `contrasts()` method from R, and instead centre our factors, so that one level is around -.5 and the other is .5. If your values aren't exactly -.5/.5, then your data set is unbalanced, so your should use this method.

```
# centre the factor
lex_dec$lang_c <- (lex_dec$NativeLanguage == "English") - mean(lex_dec$NativeLanguage == "English

# see the result
head(lex_dec)
```

```
## # A tibble: 6 x 6
##   Subject Trial Word       NativeLanguage    RT lang_c
##   <fct>   <int> <fct>      <fct>          <dbl>  <dbl>
## 1 A1         23 owl        English         6.34  0.429
## 2 A1         27 mole       English         6.31  0.429
## 3 A1         29 cherry     English         6.35  0.429
## 4 A1         30 pear       English         6.19  0.429
## 5 A1         32 dog        English         6.03  0.429
## 6 A1         33 blackberry English         6.18  0.429
```

Then we'll fit our model. We'll start out with a relatively simple model where we have crossed random effects with random intercepts by subjects (Subject) and items (Word). We'll then get a summary of our model to see what the `lmer` output looks like. We'll fit all models with ML since we're going to compare them to one another.

```
lexdec_mod <- lmer(RT ~ lang_c + (1 | Subject) + (1 | Word), data = lex_dec, REML = F)
summary(lexdec_mod)
```

```
## Linear mixed model fit by maximum likelihood  ['lmerMod']
## Formula: RT ~ lang_c + (1 | Subject) + (1 | Word)
##    Data: lex_dec
##
##      AIC      BIC   logLik deviance df.resid
##   -900.1   -873.0    455.1   -910.1     1654
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -2.3600 -0.6142 -0.1196  0.4614  5.9658
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  Word     (Intercept) 0.005896 0.07678
##  Subject  (Intercept) 0.016741 0.12939
##  Residual             0.029842 0.17275
## Number of obs: 1659, groups:  Word, 79; Subject, 21
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)  6.38509    0.02983 214.052
## lang_c      -0.15582    0.05769  -2.701
##
## Correlation of Fixed Effects:
##        (Intr)
## lang_c 0.000
```

There's a few things to unpack here. First, our model was fitted with **restricted maximum likelihood (REML)** rather than **maximum likelihood (ML)**. These are just different ways to estimate the parameters of the model. Andy Field (Field et al. (2012)) states that ML is typically better at estimating parameters for fixed regression parameters, and REML for random variances. The choice normally only makes a minor difference to the outcomes. However, we have to use ML if we want to compare model fits against one another.

Of most intereest to us are the random effects and fixed effects of the analysis. The random effects tells us how much variance in our data is captured by our random effects. Here, we can see that most of the variance is captured by the residuals (i.e. unexplained variance), but including random intercepts for subjects and items goes some way to explaining the differences in scores. Here, means are always assumed to be 0, so we only get the variance and standard deviation for these terms.

Next, we have the fixed effects. These roughly correspond to the parameter estimates from regular linear models that we've looked at. As we can see, we have a *t*-value of 2.57 for the effect of native language. The negative sign indicates that response times are slower for non-native speakers compared to the grand mean.

Finally, we get a correlation of our fixed effects, this is not the degree to which our fixed effects are related to one another, but is used for constructing confidence ellipses. We typically don't use this in reporting our statistics.

9.4.1 Calculating p-values for Parameter Estimates

You may have noticed that `lme4` doesn't provide you with *p*-values for your parameter estimates. That's because there's some discussion on whether or not we can accurately do so with mixed effects models, and also because the creators of lme4 would rather you use parameter estimates and their standard errors to infer the strenth of evidence for an effect.

We can easily calculate these using Kenward-Roger or Satterthwaite approximations, which are provided by the **lmerTest** and **afex** packages in R.

However, to avoid loading more packages, we can use the normal approximation to calculate *p*-values. The reasoning here is that with higher degrees of freedom, the t distribution matches the z distribution. Thus, we can treat the t-value as a z-value, essentially assuming infinite degrees of freedom.

Here, we look up the *p*-value that matches the absolute (non-signed) *t*-value for our parameters in the normal distribution. Then we subtract that value from 1 to get the probability of a *t*-value exceeding the one we have, and multiply that by 2 to get a two-tailed *p*-value.

We'll first ensure we have the `broom.mixed` function loaded to tidy up our model results from `lme4`. This can be installed like all packages from CRAN using:

```r
install.packages("broom.mixed") # install broom.mixed
library(broom.mixed) # load broom.mixed
```

Then we can tidy up our `lme4` output and calculate *p*-values using the normal approximation as follows:

```r
lexdec_mod %>%
  tidy("fixed") %>%
  mutate(p_value = 2*(1 - pnorm(abs(statistic))))
```

```
## # A tibble: 2 x 6
##   effect term        estimate std.error statistic p_value
##   <chr>  <chr>          <dbl>     <dbl>     <dbl>   <dbl>
## 1 fixed  (Intercept)    6.39     0.0298    214.   0
## 2 fixed  lang_c        -0.156    0.0577     -2.70 0.00692
```

Just be aware that this approach somewhat inflates type-I error rates (but only slightly so with large samples).

Another alternative we can use to assess whether we have a main effect of a factor with multiple levels is to use a model comparisons approach, which we'll look at next.

9.4.2 Model Selection

With the model comparisons approach, we simply fit our full model, and a model that is exactly the same as the full model, only without the fixed effect of interest. Here, because we want to check for a main effect of language, we simply fit a model with only an intercept term (and without an effect of language). Notice that the random effects stay the same.

```
lexdec_mod_reduced <- lmer(RT ~ 1 + (1 | Subject) + (1 | Word), data = lex_dec, REML =
```

Then, we use the `anova()` function to check whether inclusion of the language fixed factor significantly improves model fit. Notice also that if we originally fit our models with REML, R will refit them with ML instead.

```
anova(lexdec_mod, lexdec_mod_reduced)
```

```
## Data: lex_dec
## Models:
## lexdec_mod_reduced: RT ~ 1 + (1 | Subject) + (1 | Word)
## lexdec_mod: RT ~ lang_c + (1 | Subject) + (1 | Word)
##                    npar     AIC     BIC logLik deviance  Chisq Df Pr(>Chisq)
## lexdec_mod_reduced    4 -895.86 -874.20 451.93  -903.86
## lexdec_mod            5 -900.12 -873.05 455.06  -910.12 6.2605  1    0.01235 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Here, we're interested in the change in log-likelihood when we add terms to our model. Here, the increase in degrees of freedom should be one for each model you compare. We can see that adding the fixed effect of language significantly improved model fit, $\chi^2(1) = 6.261, p = 0.012$. This means that we have a significant main effect of language.

We can similarly use this method to assess whether or not we should include random effects in our model, however whether or not you should select your random effects on the fly is again up for debate.

```
lexdec_slope <- lmer(RT ~ lang_c + (1 | Subject) + (lang_c || Word), data = lex_dec, REML = F)
lexdec_slope_cov <- lmer(RT ~ lang_c + (1 | Subject) + (lang_c | Word), data = lex_dec, REML = F)
anova(lexdec_mod, lexdec_slope, lexdec_slope_cov)
```

```
## Data: lex_dec
## Models:
## lexdec_mod: RT ~ lang_c + (1 | Subject) + (1 | Word)
## lexdec_slope: RT ~ lang_c + (1 | Subject) + ((1 | Word) + (0 + lang_c | Word))
## lexdec_slope_cov: RT ~ lang_c + (1 | Subject) + (lang_c | Word)
##                  npar     AIC     BIC logLik deviance   Chisq Df Pr(>Chisq)
## lexdec_mod          5 -900.12 -873.05 455.06  -910.12
## lexdec_slope        6 -903.05 -870.57 457.53  -915.05  4.9348  1    0.02632 *
## lexdec_slope_cov    7 -921.40 -883.51 467.70  -935.40 20.3514  1  6.445e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

To understand what happened here, we need to look at our random effects. Look at the summary of the model with only random intercepts by subject and item. You can see that the random effects here are calculated for the intercept terms and for the residuals.

```
summary(lexdec_mod)
```

```
## Linear mixed model fit by maximum likelihood  ['lmerMod']
## Formula: RT ~ lang_c + (1 | Subject) + (1 | Word)
##    Data: lex_dec
##
##      AIC      BIC   logLik deviance df.resid
##   -900.1   -873.0    455.1   -910.1     1654
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -2.3600 -0.6142 -0.1196  0.4614  5.9658
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  Word     (Intercept) 0.005896 0.07678
##  Subject  (Intercept) 0.016741 0.12939
##  Residual             0.029842 0.17275
## Number of obs: 1659, groups:  Word, 79; Subject, 21
```

```
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)  6.38509    0.02983 214.052
## lang_c      -0.15582    0.05769  -2.701
##
## Correlation of Fixed Effects:
##        (Intr)
## lang_c 0.000
```

Look at the next model, with random slopes added. Here, we use the double pipe (||) in our random effects to specify that we want random intercepts and slopes, but we don't want to calculate the intercept-slope covariance. (But, note that this only works for numeric, continuous predictors; centering again helps here.) This can be seen in our random effects where we have `Word lang_c` which shows the slope for language by item, `Word.1 (Intercept)` which is the intercept for items, and `Subject (Intercept)` which shows the intercept for the subjects.

```
summary(lexdec_slope)
```

```
## Linear mixed model fit by maximum likelihood  ['lmerMod']
## Formula: RT ~ lang_c + (1 | Subject) + ((1 | Word) + (0 + lang_c | Word))
##    Data: lex_dec
##
##      AIC      BIC   logLik deviance df.resid
##   -903.1   -870.6    457.5   -915.1     1653
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -2.5314 -0.6105 -0.1222  0.4714  6.0190
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  Word     lang_c      0.002351 0.04849
##  Word.1   (Intercept) 0.005924 0.07697
##  Subject  (Intercept) 0.016757 0.12945
##  Residual             0.029237 0.17099
## Number of obs: 1659, groups:  Word, 79; Subject, 21
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)  6.38509    0.02984 213.959
## lang_c      -0.15582    0.05797  -2.688
##
```

```
## Correlation of Fixed Effects:
##        (Intr)
## lang_c 0.000
```

Finally, if we look at the more complex model, we can see that we have intercepts and slopes for the items, and we've calculated the covariance between these terms (in the `Corr` column.

```
summary(lexdec_slope_cov)
```

```
## Linear mixed model fit by maximum likelihood  ['lmerMod']
## Formula: RT ~ lang_c + (1 | Subject) + (lang_c | Word)
##    Data: lex_dec
##
##      AIC      BIC   logLik deviance df.resid
##   -921.4   -883.5    467.7   -935.4     1652
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -2.6132 -0.6175 -0.1112  0.4677  6.1894
##
## Random effects:
##  Groups   Name        Variance Std.Dev. Corr
##  Word     (Intercept) 0.005924 0.07697
##           lang_c      0.002349 0.04846  -0.98
##  Subject  (Intercept) 0.016757 0.12945
##  Residual             0.029237 0.17099
## Number of obs: 1659, groups:  Word, 79; Subject, 21
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)  6.38509    0.02984 213.961
## lang_c      -0.15582    0.05797  -2.688
##
## Correlation of Fixed Effects:
##        (Intr)
## lang_c -0.027
```

It seems that this model better explains our data than the other models, so we should probably use this one. Still, the choice of random effects structure can be as much driven by your assumptions and knowledge of the source of the data as from these data driven techniques.

9.4.3 Failure to Converge: What Should I Do?

Often, when you attempt to fit the maximal random effects structure you will get an error saying that your model failed to converge. This means that you don't have enough information in your data (or your data is so noisy) that you cannot properly identify the full variance covariance matrix for the model. This happens a lot if you have variances that are equal or nearly equal to 0, or correlations in terms that are equal to -1 or 1.

One way to avoid problems with converging is to centre your factors (as we have done above) and use numeric predictors in your model. You can also try to use a different optimiser by specifying the options in `lme4`. You can specify your optimiser like so. Check out the summaries of these models to see how they differ. Basically, these optimisers are just different methods for attempting to find the best fitting model for your data.

```
nelder_mod <- lmer(
  RT ~ lang_c + (1 | Subject) + (lang_c | Word),
  data = lex_dec, REML = F,
  control = lmerControl(optimizer = "Nelder_Mead")
  )
boby_mod <- lmer(
  RT ~ lang_c + (1 | Subject) + (lang_c | Word),
  data = lex_dec, REML = F,
  control = lmerControl(optimizer = "bobyqa")
     )
```

Also, if you have complex models, with intercepts and slopes calculated for subjects and items for something that is also a fixed effect, your model can effectively run out of the degrees of freedom to estimate the correlations between your intercepts and slopes. One work around here is to just simplify your model. There are a number of suggestions for how you should do this.

In psychological research, one suggested procedure is as follows:

For a model specified as `lmer(outcome ~ A * B + (1 + A * B | subjects) + (1 + A * B | items), data)`:

- First remove the correlations between random effects, i.e. `(1 + A + B | subjects) + (1 + A + B | items)`
- If that doesn't converge, remove the by-items slopes for one factor (which explains the least variance) first. `(1 + A + B | subjects) + (1 + A | items)`
- If that doesn't converge, remove the reamining by-items slopes `(1 + A + B | subjects) + (1 | items)`
- Remove slopes on subjects until it eventually converges

The reasoning here is that in our experiments we attempt to control as many confounds in our items as possible, such that variance within items should be smaller than variance within subjects.

If your model doesn't converge with random intercepts by both terms, you should probably explore your data for potential problems.

9.5 Test Assumptions

While mixed effects models get around many of the stricter assumptions from linear models, these models still make some assumptions. Some assumptions you need to worry about include:

- Linearity: Unsurprisingly, linear models require linearly related data. You can check for this with a graph as in previous sections. Alternatively, you can fit a non-linear function to your model.
- Independence: If your data are not independent (e.g. with multiple responses per participant) then your model must specify this is the case through your random effects. Failing to do so will lead to violations of this assumption, and incorrectly fitted models.
- Normal Distribution of Random Coefficients: random coefficients are assumed to be normally distributed around the model. For random intercepts models this means that your intercepts are supposed to be normally distributed around the overall model (Field et al. (2012)).
- Multicollinearity: This is where one predictor in your model can be can be reasonably accurately linearly predicted from the other predictors in your model. You can check for this by using the `vif()` function from the `usdm` library. Centering your predictors can help with problems of multicollinearity.
- Homoscedasticity: The error term in your model should be the same across all of your variables. You could use a Levene's test or a plot of residuals vs. fitted values, as with regular linear models.
- Outliers: Large outliers will skew your data. You can check for these using any traditional means, e.g. eye-balling your data via graphs or excluding based on some reasonable criteria (e.g. + 3SD above the mean). Just make sure you have a principled decision for excluding data.

9.6 Generalised Mixed Effects Models

Generalised mixed effects models are performed in a similar way to mixed effects models, but are applied to cases when your outcome or dependent variable is not normally distributed, e.g. logistic regression for binary response data. These models use the same logic as generalised linear models, and the interpretation is very similar to mixed effects models. However here the *p*-values are calculated for the fixed effects coefficients straight out of the model!

We implement these models for binary response data like so:

```
glmer(
  DV_binom ~ # binomial dependent variable
    A * B + # fixed effects
    (A * B | subject) + (A * B | item), # random effects
  family = binomial, # family: type of distribution
  data = data,
  glmerControl(optimizer = "bobyqa") # options; notice glmerControl (not lmer)
  )
```

For proportions, we simply set the weights of the model to the number of success and provide the proportions as our dependent variable.

```
glmer(
  DV_prop ~ # dependent variable as a proportion
    A * B + # fixed effects
    (A * B | subject) + (A * B | item), # random effects
  family = binomial, # family: type of distribution
  weights = N_observations, # number of observations making up the proportion
  data = data,
  glmerControl(optimizer = "bobyqa")# options; notice glmerControl (not lmer)
  )
```

Alternatively, we can calculate these within the model if we have the number of successes and the number of observations as columns in our model.

```
glmer(
  DV_successes/N_observations ~ # calculate proportion
    A * B + # fixed effects
    (A * B | subject) + (A * B | item), # random effects
  family = binomial, # family: type of distribution
  weights = N_observations, # number of observations making up the proportion
  data = data,
  glmerControl(optimizer = "bobyqa") # options; notice glmerControl (not lmer)
  )
```

Here, our parameter estimates are interpreted as log likelihoods.

9.7 A Note on Power, Effect Sizes, and Pairwise Comparisons

9.7.1 Power

Power analysis is rather complex for mixed effects models. Here, simulation is key, and there are some packages out there to help you perform your own power analyses with little coding, such as SIMR (`install.packages("simr")`). You can find a paper on how to use SIMR here.

9.7.2 Effect Sizes

Effect sizes for individual parameter estimates and/or the entire model fit are often not reported with mixed effects models due to the practical and/or technical difficulties posed by calculating effect sizes. However, there are some options out there, such as that offered by the `r2beta()` function from the **r2glmm package**, or the `r.squaredGLMM()` function from the **MuMIn package**. I'll leave it up to you to explore these options, but we might return to this in the final lesson.

9.7.3 Pairwise Comparisons

Pairwise comparisons can be conducted in the same way as for linear models. You can either subset your data to the a single level of one factor to explore the simple effects for another factor, adjusting your *p*-values manually in the process, or you can use the `glht()` function from the **multcomp** package.

9.8 Exercises

For these exercises, we will look at the core concepts from this lesson. Here, we'll use a simulated data set that's saved as a .csv in the inputs folder.

If you don't have access to these, please download the repository from GitHub and open the **lesson_materials** folder. Open the relevant folder for this lesson. If you work out of the file **09_mixed_effects_models.Rmd** the code below will work to load the data set. If this fails then quit R and open it up from the .Rmd file above so your working directory is in the correct folder.

```
library(tidyverse)
library(lme4)
```

```r
dat <- read_csv("inputs/factorial_data.csv")
```

9.8.1 Question 1

Summarise the data, creating means, standard deviations, and ns for each group in the data set.

9.8.2 Question 2

Centre the factors A and B in preparation for fitting the model. Define the centred variables for A and B as **A_c** and **B_c** respectively. Return the data set to see what you did to the data frame.

9.8.3 Question 3

a. Find maximal random effects structure for the model and call this **maximal_model**. Return the a summary of the model output when you're done. Assume that the data from which this is taken is not nested. This model may take a while to fit, so be patient!
b. Where is most of the variance explained? What does this tell us about the model (i.e. is it overfitting the data)?
c. Are there any perfect correlations between the random effects? What does this tell us about our random effects structure?

9.8.4 Question 4

Explore the whether inclusion of the interaction significantly improves model fit over a model with just main effects. Be sure to include all random effects in all models.

9.8.5 Question 5

Calculate p-values for the parameters in your maximal model (including interactions) using the normal approximation. Round each number to 2 decimal places.

9.8.6 Question 6

Explore the interaction by subsetting the data to each level of factor A and fitting a model containing factor B. Then do the same for the remaining factor. Where does the interaction lie?

First, subset the data as described above, and assign this to four objects called A1_dat, A2_dat, B1_dat, and B2_dat respectively.

Then, fit four models looking at the main effect of one factor within each subset of data.

Next, extract a tidied version of the coefficients from each model and add a column identifying which comparison these models are from. Assign these to A1_comp, A2_comp, B1_comp and B2_comp respectively, showing that these objects contain the comparisons for each level.

Finally, bind the rows together from your comparisons, and create a *p*-value for your data using the normal approximation. Be sure to bonferroni correct your data. After the correction, make sure that you reset any *p*-values above 1 to 1 (as *p*-values above 1 don't make any sense).

9.8.7 Question 7

Make a plot showing the interaction between A and B.

Chapter 10

Reporting Reproducible Research

In this session, we'll cover how you can use Rmarkdown for reproducible documentation of analyses. In doing so, we'll cover some aspects that improve the quality of your research, such as reporting effect sizes and the spread/uncertainty in your data (e.g. 95% confidence intervals) and drawing inferences from null results. Furthermore, we'll cover some aspects of good research practice which go hand in hand with reproducible research, such as data and code archiving and pre-registration.

Specifically, we'll cover:

- Calculating effect sizes from the *r* and *d* families
- Evaluating evidence for the null hypothesis in the Neyman-Pearson framework using the two one-sides tests (TOST) procedure
- Calculating the 95% confidence interval for parameter estimates
- Simulation for understanding research practices
- Data and code archiving, and pre-registration on the Open Science Framework

This session is heavily informed and inspired by the excellent Improving Your Statistical Inferences course by Daniel Lakens. If you get the chance, please enrol on this course. You won't regret it!

10.1 Getting Started

As always, we first need to load the `tidyverse` set of package. However, for this chapter we also need the **sjstats**, **effsize**, and **TOSTER** packages. The first two of these new pacakges allow us to calcualte effect sizes, and the final package allows us to conduct the TOST procedure for evaluating evidence for the null hypothesis. Everything else for creating reproducible documents in R comes loaded with RStudio.

```
# load packages
library(tidyverse)
library(sjstats)
library(effsize)
library(TOSTER)
```

10.2 Creating Reproducible Documents

10.2.1 Creating a Reproducible Report

When doing your research, you'll often have to write up a number of reports for your studies, be this papers, presentations, or blog posts. One advantage of using R for your analyses is that every time you make a small change to your analysis, for example testing more participants, you have all of the code there to reproduce your analyses using the exact same method as before. However, even with R, you don't want to recreate your results and translate your outputs every time you change to a new format. Imagine the pain involved in changing all of your table outputs and plots by hand every time you make a minor change? On top of this, you're very likely to introduce some mistakes when transcribing your results. To remedy this, we can use R to produce reproducible reports.

The advantage of this over simple scripts to conduct your analyses and a separate document used to write up these analyses is that every time you change something in regards to your experiment, you can automatically update everything in your report, reducing the chance of human error. Furthermore, by using reproducible reports, others can read your work and see how you calculated your statistics, which increases the transparency of your work. Why is this a good thing? Aside from catching mistakes in your code, you're leaving a record for other researchers to follow in their own work. That complex analysis you did can then be reproduced by others, helping to push science forward!

We've already used one method for producing reproducible reports in R; the R Notebook. This updates every time you save your document, without rerunning your code. However, we can also produce a similar R Markdown report which requires you to compile your code before it produces output. Here, you include text and code chunks like in your R Notebooks, but you use the Knit button create an output which translates your code into an HTML, PDF, or Word document. I've shown you how to do this below.

Creating R Markdown Document

Once you've elected to create an R Markdown document, you can choose your output type. For documents, we have the options discussed above. But you can also create presentation slides in R. For the slides in this course, I used the ioslides option. This is nice if you want to include code or the outputs of code directly in your presentations; no more dragging and resizing images by hand!

R Markdown Document Options

For this example, I'll create an HTML document from our code, but all of the same principles apply with different outputs.

First off, you need to pick a name for your document. Here, I've set it to `test_html_document`. This will be the title of your document once you've saved and rendered it.

Defining Your Title and Author in the R Markdown Options

Just save your document somewhere, preferably in its own folder, and you're good to go. Once you've done that, you can change your code and text. Any time you want to render the HTML (or Word, or PDF file) just click the Knit button, and you'll see a preview of your output once it is done making the document. You can use this to work on the fly, or just look at the output itself (i.e. double clicking the HTML file in your folder) to see how it looks on completion. R turns script like this below into a nice output!

The Markdown Environment

Remember that anything in the code chunks (inserted with Insert, R) from the editor bar in RStudio will be evaluated and you'll see the output of this code when rendered. You can also include in line code by using one backtick followed by R and then your code and closing this call with another backtick. This is nice for if you want to specify the mean and standard deviation for a group in text.

Formatting text in Markdown is quite easy, and you can find an R Markdown Cheat Sheet here. This covers all of the basics of getting started, and gives you examples of how to do certain features. This does such a good job that I won't cover it here.

10.2.2 Folder Structure for Reproducible Analyses

Now, we can store all of our analysis and write up in one folder. One way to structue your work is by having everything for one experiment in one folder, called something like "Experiment 1". Within this folder, you can have several sub-directories for the different things to do with your experiment. This can have as many sub-directories as you like, such as Ethics, Grant Applications, etc., but it's often good practice to have a folder for your Data, Analysis, and Output. The Data folder should just contain any raw data for your experiment. The Analysis folder can contain any R scripts used to produce your outputs, such as a data cleaning script, which makes your raw data tidy and your code for your R Markdown document. Finally, your Output folder will be where your HTML/Word/PDF documents will get saved when knitted.

Remember using relative file paths? This will be handy for getting R to read data from different folders within the Experiment 1 folder. For example, if we keep our R Markdown file in the Analysis sub folder, we can read data from the data folder by using two dots before the folder and file name. This tells R to look up from our current directory (the Analysis folder) by one folder. So now R is in the Experiment 1 folder. We then tell it to go to the Data folder `/Data/` and read the raw data from the .csv file called `raw_data.csv`.

```
data <- read_csv("../Data/raw_data.csv")
```

Similarly, we can then keep a normal R script that will render our document for us (without using the Knit button) called something like `render_document.R` in our Analysis folder. In this, the only code we need is as follows:

```
rmarkdown::render('test_html_document.Rmd',
                  output_file = '../Output/test_html_document.html'
                  )
```

When we run this code, R will execute everything in the R Markdown file, and save the rendered output in our Output folder. This is less convenient than using the Knit button, which is easy if you want your RMarkdown file and HTML output in the same folder. But if we want a nicely structured file system, using the above method is best.

You can find an example of how this works in the `reproducible_example` folder on in the `lesson_materials` folder for lesson 10.

Here, you'll notice I use Kable for printing tables, e.g.

```
# make example data
tooth_dat <- ToothGrowth %>% group_by(supp) %>% summarise(mean_len = mean(len))

# print table
library(knitr)
kable(tooth_dat)
```

supp	mean_len
OJ	20.66333
VC	16.96333

This just turns data frames into a nice table output like you'd get in Word. The benefit over just printing data frames is that you get everything in a normal table output, so your data is ordered correctly. You can specify a number of options for Kable, which I won't cover here. But just know that you can change how anything looks to your liking.

10.2.3 Using R Markdown Templates

We can use different R Markdown templates to save us from specifying too many options. One popular template is the 6th Edition APA article template from the **papaja** package. You can install this from GitHub using the following code. If the install doesn't work check you've installed devtools first and install if needed by uncommenting the first line.

```
# install.packages("devtools")
devtools::install_github("crsh/papaja")
```

The GitHub page for pajapa has lots of nice documentation on how to use the different helper functions from this package. This makes for easy and pretty reporting in your document.

You may already have your workflow geared towards word, where you can easily use reference managers like Mendeley to input and report all of your references. One way to get the most out of this is to export your references as a .bib file, and include this in the same folder as your R Markdown document scripts. Then you just include the code `bibliography: bibliography.bib` under the output command in the header of your R Markdown file, and just use the shorthand to refer to your references in the R Markdown text, e.g. `@reference_date` and R will sort the references for you. You can find help on using bibliographies in RMarkdown at the RStudio website.

If all of this sounds a little overwhelming, you could just use RMarkdown to make your Results section, and copy those sections across to your working Word file on completion, but you lose a lot of the benefits of automation by doing so.

10.3 Open Science

Reproducbile reports are just one step towards Open Science. Why should we adopt Open Science practices? Briefly, we improve the quality of our work, and create a more collaborative atmosphere in science. Here are just some ways to make your research more open. You should try to adopt:

- **Reproducible reports**: all code for creating analyses and visualisations is included along with (and goes into making) the document, so people can see what you did. RMarkdown is useful for this.
- **Pre-prints**: post your articles on a pre-print repository. This means that people can read and give you feedback on your work prior to publication. Many journal articles are fine with you having a pre-print out there, so that's not much of a concern anymore. One example is the PsyArχiv for articles in psychology.

- **Pre-registration and registered reports**: post a document considering your hypotheses, study design (including all measured and manipulated variables), and a plan for analysis. This should include details such as a plan for how to exclude outliers, a justification for your α-level, the number and type of tests conducted, any correction to p-values if necessary, and a justification of your sample size. This sample size justification can be based on power analyses or even things like time/money constraints, but be clear how you defined your sample size. Pre-registrations allow us to distinguish confirmaty from exploratory analyses, and they formalise your type-I error rate, ensuring that researcher degrees of freedom (e.g. trying different tests, adding more participants, etc.) are considered so you don't inflate your type-I error rate. You can pre-register your studies using a quick and easy form from As Predicted, or alternatively you can host your study data, materials, and code on the OSF and include your pre-registration there.
- **Open access publishing**: when possible, publish in Open Access journals. This means that everyone, including the general public, can access your research. If their tax money goes to your science, surely they should have access to the results. There are many Open Access journals, so I'll leave it up to you to decide where to publish!
- **Open materials, data, and code**: sharing your materials, data, and code allows for other researchers to build on your work. Furthermore, knowing that it's out there on the web will make you spend more time checking everything is in order. One bonus is that if your hard drive burns down and you lose a local copy of your work, it's hosted on a server somewhere, so you'll always have access to it! Some repositories that are useful for this are the OSF and GitHub.

One thing to bear in mind when sharing your materials, data, and code associated with a project is that you should make everything easy to understand for someone who doesn't know your study. That's why it's useful to include a README file along with your study data explaining (briefly) what was done, what each column and observation represents in your data, and how your code works. As such, commenting your code is a must. Just aim to explain why you're doing something (not always what you're doing) as that is often obvious from the code itself. Using a structured system for your folders and files makes it easier for people to know hwere to look (e.g. I want the raw data, so I'll go to the Data folder and grab the raw_data.csv file!)

All of this goes hand in hand with coding in R. We'll briefly consider some of the problems associated with flexible analyses masquerading as confirmatory hypothesis testing later in this session in an attempt to hammer home the importance of pre-registration. Note that pre-registration doesn't preclude flexible analyses or changes to your analysis plan; you just have to state what was changed and why. It's purely a system of accountability, not restriction. One benefit is that you can use one-sided tests if you make a prediction and pre-register it! For me, pre-registration gets me to think deeply about a study prior to running it, often catching mistakes prior to running the study. This saves time and money re-reunning participants with the fixed study procedure. It also means that I have a record of what I have to do or what I did, so writing up reports is so much easier!

10.4 What to Report

Any report you submit should include a number of components that we've already covered in previous sessions. You should include graphs to display your data, along with descriptive (i.e. means, standard deviations, n, etc.) and inferential statistics (where test assumptions have been checked). We've covered all of this in previous sessions, but we haven't spent much time on the other aspects. As such, we'll look into what else you should report.

Ideally, you should also try to include some measure of uncertainty around the parameter estimates in any models that you report, such as confidence intervals. Furthermore, you should try to include some measure of effect size, so we can really interpret the magnitude of an effect when significant. Finally, if have null results it's often interesting to explore why this is. We can evaluate evidence in support of the null (no effect; contrasting with an insensitive study due to small sample size etc) under different frameworks, but we'll cover one method in the Neyman-Pearson null hypothesis significance testing approach later in this lesson; the TOST procedure. We'll explore ways to do these last 3 later in this session.

10.5 Effect Sizes

When we encountered correlations, we covered effect sizes briefly as the base correlation function in R, `cor.test()`, by default reports a measure of standardised effect sizes. Unfortunately, not all R functions for running hypothesis tests report a measure of standardised effect size. Indeed, there is a lack of consistency of what is reported; e.g. *t*-tests but not ANOVAs report 95% confidence intervals around the parameter estimate(s). As such, we often have to do things like calculate standardised effect sizes using other means. Luckily, there are a number of packages that can do this for us.

Although there are other packages out there for producing standardised effect sizes, we'll use the **sjstats** package for calculating effect size estimates for ANOVA and linear regression models and the **effsize** package for calculating effect sizes from *t*-tests.

But why should you report standardised effect sizes anyway? Well, as sample size increases, we're more likely to find a significant effect between two groups even if this effect is really small. Effect sizes tell us how much we should care about the effect. But still, you can get similar information by looking at raw effect sizes (e.g. reaction times between group 1 and 2), so why report standarised effect sizes at all? Standised effect sizes allow us to compare effects across different studies (which often have different designs). As a result of this, meta-analysis is reliant on the reporting of standrdised effect sizes, so you'll be doing science a favour by reporting standardised effect sizes and not forcing other researchers to recompute these values when it comes to including your study in a meta-analysis. Finally, many packages and programs that allow you to analytically calculate the planned power for a study rely on you using standarised effect sizes. So, at the very least, you'll be doing yourself a favour in the future if you plan a similar study to one you've ran previously and you want to calculate your planned power prior to running the new study.

10.5.1 Cohen's d and Pearson's r

We have 2 families of effect sizes that we can choose to use in reporting our analyses; the *d* family and the *r* family. The differences between these families (and within the families) is briefly summarised below:

- The ***d* family** reports standised effect sizes in terms of **standardised mean differences**. These go from 0 (no effect) to ∞ (infinity). The most simple case is Cohen's *d* which is calculated by taking the differences between scores (e.g. across groups) and dividing them by the pooled standard deviation (i.e. ignoring grouping). This effect size family has different versions, such as:
  - d_{z} (i.e. the difference between x and y, z) for within-subjects designs;
  - d_{av} for a within-subjects design corrected for correlations between measures;
  - or Hedge's *g* for an unbiased version of Cohen's *d* which corrects for the fact that we often do not know the population standard deviaiton.

This final case typically provides a better estimate of effect size (population-based measures often overestimate effect sizes), especially for small sample sizes. Due to this independence of the study design, some researchers argue that Cohen's *d* does a better job than many *d*-like effect sizes as it is insensitive to your experimental design. Thus, it can be generalised more easily to different experimental designs that don't match your study.

- The ***r* family** reports standardised effect sizes in terms of the **strength of association between variables**. This is calculated in terms of the difference between our observations and the group mean, squared in order to stop negative scores cancelling out positive scores, and then summed up. These terms go from 0 (no association) to 1 (total association). Crucially, these can be reported in squared terms where we look at the proportion of total variance explained by an effect (e.g. based on r^2). This effect size family, has different versions, such as:

- η^2 (eta squared) for more than 2 sets of observations;
- ω^2 (omega squared) which corrects for the fact that we don't know the population variance, only the sample variance; (This is good when sample size is small or when we have many levels in a factor.)
- ϵ^2 (epsilon squared), which is possibly the least biased of all alternatives.

Crucially, these effect size estimates, such as η^2, sum to 100%, so we can use partial forms of these explain variance by only 1 factor. This is useful in experimental designs, and allows us to compare effect sizes between studies.

Selection of which effect size to use should come from an informed position. As such, I recommend that you read this great article by Daniel Lakens which explains the differences between effect size measures above in relatively simple terms.

How should you select between reporting d or r? When sample sizes are very different across groups, d is typically less biased than r (McGrath & Meyer, 2006), but many find r easier to interpret.

In using standardised effect sizes to calculate power you should generally avoid benchmarks, such as small, medium, and large effects, and instead focus on having an informed idea from the wider literature. However, be wary of publication bias which may overstate effect sizes. One alternative here is to specify your smallest effect size of interest in your study, and design your study around detecting that.

10.5.2 Calculating Effect Sizes

10.5.2.1 Cohen's d and Hedge's g

Look at the reults of an independent-sample *t*-test.

```r
set.seed(5)
ind_t_data <- tibble(
  dv = c(rnorm(n = 10000, mean = 100, sd = 10),
         rnorm(n = 10000, mean = 99, sd = 10)),
  id = c(rep("a", 10000), rep("b", 10000))
  )

t.test(dv ~ id, data = ind_t_data)
```

```
##
##  Welch Two Sample t-test
##
## data:  dv by id
## t = 7.8329, df = 19998, p-value = 5.006e-15
## alternative hypothesis: true difference in means between group a and group b is not
## 95 percent confidence interval:
##  0.8395062 1.3998889
## sample estimates:
## mean in group a mean in group b
##        100.0181         98.8984
```

It's significant, but should we really care about this difference between these groups? One way to check this would be to look at the unstandardised effect sizes.

```
ind_t_data %>% group_by(id) %>% summarise(mean_dv = mean(dv))
```

```
## # A tibble: 2 x 2
##   id    mean_dv
##   <chr>   <dbl>
## 1 a       100.
## 2 b        98.9
```

Well, a mean difference of around 1.2 doesn't seem like much to me. But, of course this all needs to be put in the perspective of the study you're running. We can calculate standardised effect size for this study using the `cohen.d()` function from the **effsize** package. Here, we'll make sure that we get effect sizes reported without the correction for paired data (as these samples are independent), and we'll make sure that we get Hedge's g and not Cohen's d.

```
cohen.d(dv ~ id, paired = F, hedges.correction = T, data = ind_t_data)
```

```
##
## Hedges's g
##
## g estimate: 0.1107692 (negligible)
## 95 percent confidence interval:
##      lower      upper
## 0.08302926 0.13850919
```

Try changing the parameters in the arguments in the `cohen.d()` function above. How do the results change if we get Cohen's d? Nicely, we get a 95% confidence interval around our estimates by default.

10.5.2.2 Eta Squared and Omega Squared

For calculating effect sizes from ANOVA designs, we'll use the **sjstats** package. Here, we'll see how we can simply calculate different alternatives from the *r* family of effect sizes using this package. For this example, we'll use the **ToothGrowth** data set that we used in Chapter 7. Remember, this looks at the effect of different supplements, and doses of those supplements, on tooth growth in guinea pigs.

```
# load the data
data("ToothGrowth")

# convert to tibble and make dose a factor
tooth <- ToothGrowth %>%
  mutate(dose = factor(dose)) %>%
  as.tibble()

# see the output
tooth
```

```
## # A tibble: 60 x 3
##      len supp  dose
##    <dbl> <fct> <fct>
##  1   4.2 VC    0.5
##  2  11.5 VC    0.5
##  3   7.3 VC    0.5
##  4   5.8 VC    0.5
##  5   6.4 VC    0.5
##  6  10   VC    0.5
##  7  11.2 VC    0.5
##  8  11.2 VC    0.5
##  9   5.2 VC    0.5
## 10   7   VC    0.5
## # ... with 50 more rows
```

Next, we'll fit a linear model to the data.

```
tooth_aov <- aov(len ~ supp * dose, data = tooth)
summary(tooth_aov)
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)
## supp         1  205.4   205.4  15.572 0.000231 ***
## dose         2 2426.4  1213.2  92.000  < 2e-16 ***
## supp:dose    2  108.3    54.2   4.107 0.021860 *
## Residuals   54  712.1    13.2
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

OK, so we have a significant main effect of both factors, and a significant interaction. As such, we probably want the effect sizes for this. We can do this using the `eta_sq()` function for η^2 and `omega_sq()` for ω^2. For both functions, we can set the `partial` argument to true (`TRUE`/`T`) or false (`FALSE`/`F`) to get the partial forms of these effect sizes.

10.5.2.2.1 Eta Squared

First, we'll calculate η^2.

```
eta_sq(tooth_aov, partial = F)
```

```
## Warning: 'eta_sq' is deprecated.
## Use 'effectsize::eta_squared()' instead.
## See help("Deprecated")
```

```
## # Effect Size for ANOVA (Type I)
##
## Parameter | Eta2 (partial) |       95% CI
## -----------------------------------------
## supp      |           0.22 | [0.08, 1.00]
## dose      |           0.77 | [0.68, 1.00]
## supp:dose |           0.13 | [0.01, 1.00]
##
## - One-sided CIs: upper bound fixed at [1.00].
```

10.5.2.2.2 Omega Squared

Then we'll calculate ω^2.

```
omega_sq(tooth_aov, partial = F)
```

```
## Warning: 'omega_sq' is deprecated.
## Use 'effectsize::omega_sqared()' instead.
## See help("Deprecated")
```

```
## # Effect Size for ANOVA (Type I)
##
## Parameter | Omega2 (partial) |       95% CI
## -------------------------------------------
## supp      |             0.20 | [0.06, 1.00]
## dose      |             0.75 | [0.65, 1.00]
## supp:dose |             0.09 | [0.00, 1.00]
##
## - One-sided CIs: upper bound fixed at [1.00].
```

Try changing the partial argument in both of these functions to see how this works.

Unfortunately, I'm not aware of any package at the moment that collects absolutely every effect size metric that you could possibly compute in R. However, as you can see it's relatively easy to calculate effect sizes with many packages in R as long as you know what you want to calcualte.

10.6 Confidence Intervals

Any report should contain uncertainty about the parameter estimates. This can be easily calculated using the `confint()` function on any model object. There are different methods for calculating confidence intervals in this function. The default is profile based confidence intervals, but you can specify to calculate these via parameteric bootstrapping, which, assuming your model is correct (i.e. you're not violating assumptions), tends to be preferred. We can try this with our fitted model, `tooth_aov`. How these are calculated tends to be more a concern for mixed effects models, but we can see how we get confidence intervals around our parameter estimates from our ANOVA model.

```
confint(tooth_aov, method = "boot")
```

```
##                    2.5 %    97.5 %
## (Intercept)  10.9276907 15.532309
## suppVC       -8.5059571 -1.994043
## dose1         6.2140429 12.725957
## dose2         9.5740429 16.085957
## suppVC:dose1 -5.2846186  3.924619
## suppVC:dose2  0.7253814  9.934619
```

It's really that easy!

10.7 Evaluating Evidence for the Null Hypothesis

Often in psychology we end up with a non-significant results from our statistical tests. However, it's very difficult to interpret what this means. You may be tempted to conclude that a non-significant result means that there is no effect in the population, but this is incorrect. In fact, it's likely that nothing in the real world has an effect size of exactly 0. As such, we need a way to determine whether or not an effect is small enough that we don't care about it. One way to do this in the frequentist framework is with equivalence tests.

One method for testing for equivalence is the two one-sided test (TOST) procedure (Schuirmann, 1987) whereby we specify the upper and lower bounds within which we deem effects to be statistcally equivalent. For example, let's say we test two groups of people on an IQ test, and group 1 scores 100, while group 2 scores 102. This is a difference of 2 IQ points between groups. If our bounds for equivalence are difference scores of -5 and +5, we can say that any difference score within these bounds is statistically equivalent, and as such we can conclude the absence of an effect in the population. Since our score of 2 is between these bounds, we state that the two groups are equivalent.

With the TOST procedure, we effectively test for an effect more extreme than our bounds using one-sided *t*-tests. As such, the null hypothesis of the TOST procedure is the presence of a true effect (i.e. the scores don't differ from our upper and lower bounds; they aren't within our equivalence bound) and the alternative hypothesis is that our effects fall within the lower and upper bounds (i.e. the scores differ from the lower and upper bounds; they are both smaller than the upper, and larger than the smaller bounds). This takes some getting used to, but just remember that **significant effects reveal equivalence**.

10.7.1 Independent Samples TOST

Here, we'll take the data from our independent samples *t*-test from before. Remember, the data revealed that the difference between the groups was around 1.2 points on some test, as shown below.

```r
ind_t_data %>% group_by(id) %>% summarise(mean_dv = mean(dv))
```

```
## # A tibble: 2 x 2
##   id    mean_dv
##   <chr>   <dbl>
## 1 a       100.
## 2 b        98.9
```

10.7.1.1 TOST for Raw Scores

Here, we'll test for equivalence based on raw scores and effect sizes in our samples. For the first case we'll use the `TOSTtwo.raw()` function to calculate equivalence for an independent-samples *t*-test based on raw scores. First, we need to calculate some summary statistics from our groups.

```
# make a summary
ind_summary <- ind_t_data %>%
  group_by(id) %>%
  summarise(mean = mean(dv),
            sd = sd(dv),
            n = n()
            )
ind_summary # return the summary
```

```
## # A tibble: 2 x 4
##   id     mean    sd     n
##   <chr> <dbl> <dbl> <int>
## 1 a     100.   10.1 10000
## 2 b      98.9  10.1 10000
```

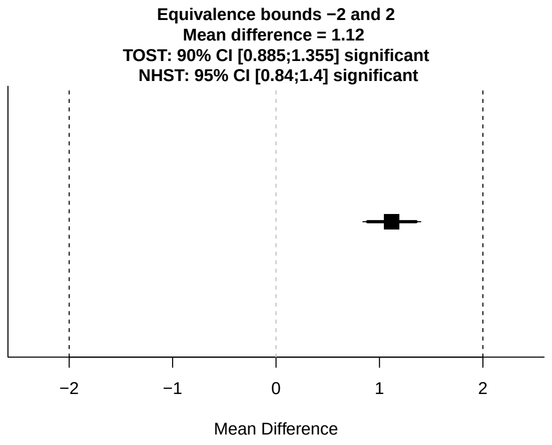
Then, we'll pass these results on to the function. Here, we'll use a new function called `with()` which makes R evaluate our column names in the scope of the summary data frame (or more generally, in whatever data you pass to it). This just stops us from having to type `ind_summary$mean[1]`, `ind_summary$mean[2]`, etc. Within this, we use the TOST function and we have to define the means, standard deviations, and *n*s for each group. Group A here is always the first row in each column, so we use `mean[1]` to refer to their mean, etc. Aside from their details from the study, we need to determine our upper and lower bounds. These are -2 and 2 here, saying we don't care about differences of only 2 between groups. Finally, we set our alpha level to determine out type-I error rate, and we specify if we assume equal variances.

```
with(ind_summary,
     TOSTtwo.raw(m1 = mean[1],
                 m2 = mean[2],
                 sd1 = sd[1],
                 sd2 = sd[2],
                 n1 = n[1],
                 n2 = n[2],
                 low_eqbound = -2,
                 high_eqbound = 2,
                 alpha = 0.05,
                 var.equal = TRUE
                 )
     )
```

```
## Note: this function is defunct. Please use tsum_TOST instead
```

```
## TOST results:
```

```
## t-value lower bound: 21.82   p-value lower bound: 0.000000000000000000000000000000
## t-value upper bound: -6.16   p-value upper bound: 0.0000000004
## degrees of freedom : 19998
##
## Equivalence bounds (raw scores):
## low eqbound: -2
## high eqbound: 2
##
## TOST confidence interval:
## lower bound 90% CI: 0.885
## upper bound 90% CI:  1.355
##
## NHST confidence interval:
## lower bound 95% CI: 0.84
## upper bound 95% CI:  1.4
##
## Equivalence Test Result:
## The equivalence test was significant, t(19998) = -6.158, p = 0.000000000375, given

##


##
## Null Hypothesis Test Result:
## The null hypothesis test was significant, t(19998) = 7.833, p = 0.00000000000000501

##


## NHST: reject null significance hypothesis that the effect is equal to 0
## TOST: reject null equivalence hypothesis
```



Here, we get an output telling us the result of our test in regards to a normal *t*-test (NHST), and under the TOST procedure. We also get reminder of our equivalence bounds, and we get some confidence intervals around our scores. In this case, the normal *t*-test (NHST) was significant, as was the TOST in both directions. This means that although we found a statistically significant difference between the groups, they are statistically equivalent with our criteria. As you can see, differences between groups can be statistically significant, but the effect is so small that we don't care.

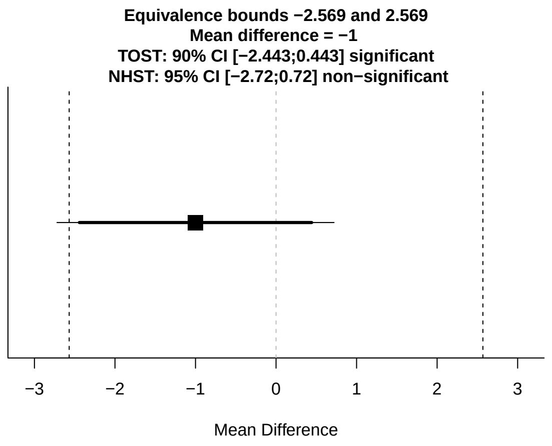
10.7.1.2 TOST for Effect Sizes

Next, we'll look at a case when we already know the means and standard deviations, but we define our bounds in terms of effect sizes. For this, we only need the `TOSTtwo()` function. The main difference here is that we specify our bounds in terms of effect sizes (in this case, Cohen's *d*), but everything else remains the same. We'll use a simpler case where we just input some values for the means, standard deviations, and *n*s.

```
TOSTtwo(
  m1 = 100,
  m2 = 101,
  sd1 = 9,
  sd2 = 8.1,
  n1 = 192,
  n2 = 191,
  low_eqbound_d = -0.3,
  high_eqbound_d = 0.3,
  alpha = 0.05,
  var.equal = FALSE
  )
```

```
## Note: this function is defunct. Please use tsum_TOST instead
```

```
## TOST results:
## t-value lower bound: 1.79    p-value lower bound: 0.037
## t-value upper bound: -4.08   p-value upper bound: 0.00003
## degrees of freedom : 377.25
##
## Equivalence bounds (Cohen's d):
## low eqbound: -0.3
## high eqbound: 0.3
##
## Equivalence bounds (raw scores):
## low eqbound: -2.5686
## high eqbound: 2.5686
##
## TOST confidence interval:
## lower bound 90% CI: -2.443
## upper bound 90% CI:  0.443
##
## NHST confidence interval:
## lower bound 95% CI: -2.72
## upper bound 95% CI:  0.72
##
## Equivalence Test Result:
## The equivalence test was significant, t(377.25) = 1.793, p = 0.0369, given equivaler

##


##
## Null Hypothesis Test Result:
## The null hypothesis test was non-significant, t(377.25) = -1.143, p = 0.254, given a

##


## NHST: don't reject null significance hypothesis that the effect is equal to 0
## TOST: reject null equivalence hypothesis
```



Here, we can see that the Welch's independent samples *t*-test was non-significant, but the TOST was significant. This means that we didn't find a significant difference between groups and the groups are statistically equivalent.

This pacakge covers TOST for meta-analysis, one- independent- and paired-samples *t*-tests. Check out these functions by typing `??TOST` in the console.

There's a lot more to understanding this procedure than I can cover here, but if you're interested in equivalence tests, you should check out this primer by the package author, Daniel Lakens.

Next, we'll cover how we can use R to understand just why we need other open science practices along with the statistics that we calculate and report in R.

10.8 Understanding Problematic Design Choices

Finally, we'll touch briefly on why you should adopt Open Science practices now that you know how to create reproducible reports. We'll use some simulations to explore cases where you may inadvertantly inflate your error rate through flexible analysis plans. Namely, we'll look at optional stopping; choosing when you should stop recruiting participants on the fly based on your test statistics (and how to avoid inflating your error rate), and why statistics can't save you from poor hypotheses.

10.8.1 Optional Stopping (Without Correction)

The real strength in null hypothesis significance testing is that it enables us to make statements about how how often we'll be wrong in the long run (i.e. to make a type-I error) if we make our inferences based on a certain threshold *p*-value. However, this is only the case if you stick to your research plan from the outset.

Let's look at a case where we can end up fooling ourselves that we have an effect when there isn't a true effect in the population if we change our desgn on the fly. This example is inspired by the section on optional stopping in the Improving your Statistical Inferences course by Daniel Lakens.

Here, we'll test our hypothesis that two groups have different IQs (which isn't the case), but if after the first 60 participants we have a non-significant result we'll recruit another 20 and see how this affects our *p*-value distribution.

First, we'll make some functions to (1) sample data and put them in a tibble with a group identifier, (2) run a *t*-test on those groups and capture the *p*-value, and (3) a biased test where we run a *t*-test on our groups, but if it's non-significant, we run additional participants and rerun the *t*-test with the larger sample size.

```r
# sample data and make a tibble with DV and ID based on input group sample size
# this is fixed at means of 100 and sd of 15 for each group
sample_groups <- function(n_group){
  group_one <- rnorm(n = n_group, mean = 100, sd = 15)
  group_two <- rnorm(n = n_group, mean = 100, sd = 15)
  data <- tibble(dv = c(group_one, group_two),
                 id = c(rep("group_one", n_group),
                        rep("group_two", n_group)
                        )
  )
}

# run a t-test and extract the p-value for a tibble
# columns must be named dv and id for dependent variables and group ID
test_groups <- function(tibble_data) {
  test <- t.test(dv ~ id, data = tibble_data, paired = FALSE)
  test$p.value
}

# optional stopping without correction:
# sample data and extract p-values from a t-test
# if non-significant, sample additional people and retest
bias_test <- function(sample_size, additional_n) {
  original_data <- sample_groups(sample_size)
  p_val <- test_groups(original_data)
  ifelse(p_val >= .05,
         p_val <- test_groups(
           rbind(original_data, sample_groups(additional_n))
           ),
         p_val
         )
}
```

Now that we have our functions ready, we can caputre all *p*-values assuming we run the study 1000 times with the idea of testing and getting more participants before re-testing if non-significant.

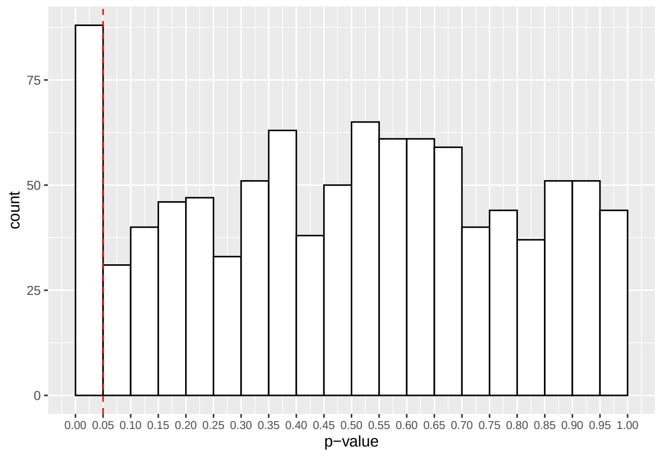
```r
set.seed(100) # uncomment if you want different values each time
p_vals <- replicate(n = 1000, bias_test(30, 10))

# calculate the proportion of p-values below .05
mean(p_vals < 0.05)
```

```
## [1] 0.088
```

What we see is that 8.8% of our *p*-values are signficant below the .05 α-level. Really though, this should be 5% if the null hypothesis is true (which it is). This means that we're inflating our type-I error rate by being so flexible in our testing and analysis procedure. It's easy to fool yourself this way!

Let's have a look at a plot of our *p*-values. You can see that we have many more *p*-values below the .05 threshold. This is despite the fact that *p*-values should be evenly distributed under the null hypothesis.

```r
p_plot <- ggplot() +
  geom_histogram(aes(p_vals),
                 binwidth = 0.05,
                 boundary = 0,
                 fill = "white",
                 colour = "black") +
  scale_x_continuous(limits = c(0, 1),
                     breaks = seq(from = 0, to = 1, by = 0.05)
                     ) +
  geom_vline(xintercept = 0.05, linetype = 2, colour = "red") +
  labs(x = "p-value") +
  theme(axis.text.x = element_text(size=8))
p_plot
```



One way to remedy this situation is through optional stopping with a *p*-value correction. If you specify how many times you will look at your data and run a test before deciding whether or not to test more participants, you can correct your *p*-value to account for the infaltion in type-I error rate. One simple way is with the Bonferroni correction. Take your *p*-value, and multiply it by how many times you tested the data. There are less conservative procedures out there, but this is one simple method.

10.8.2 Bad Hypotheses

If you have an outlandish hypothesis, where your hypothesis is unlikely to be true, you have a very low chance that a positive result indicates your hypothesis is true. We'll explore an example from the excellent Statistical Rethinking by Richard McElreath to test the truth of this statement. (Honestly, this book has changed how I understand statistics!)

To test this idea this, we can rely on Bayes' theorem. I won't show you the ins and outs of this theorem, but just know that the function that allows you to change the power of your study, your alpha level, and the probability that your hypothesis is true. The default values make sure that you have a design with 90% power and a 5% α-level, assuming that your hypothesis only has a 1% chance of being true.

```
test_prob_hyp_true <- function(power = 0.90, alpha = 0.05, hyp_chance = 0.01){
  # calculate probabilities
  prob_false <- 1 - hyp_chance
  prob_positive <- power * hyp_chance + alpha * prob_false
  prob_positive_is_true <- power * hyp_chance / prob_positive

  # output probability of a positive result being true
  print(paste("There is a",
              round(prob_positive_is_true*100, 2),
              "% chance of a positive result being true.")
  )
}
```

We'll test the probability of a positive result being true under these default assumptions.

```
test_prob_hyp_true()
```

```
## [1] "There is a 15.38 % chance of a positive result being true."
```

But what if we change the probability of the hypothesis being true to even 25%?

```
test_prob_hyp_true(hyp_chance = 0.25)
```

```
## [1] "There is a 85.71 % chance of a positive result being true."
```

Even with only a $\frac{1}{4}$ chance of our hypothesis being true, we can now more confidently accept a positive result as being positive. The take home message here is that even if you design and test your studies to your best abilities, with very high power, positive findings based on bad hypotheses are likely to be misleading; good statistics can't save you from bad hypotheses.

10.9 The Final Lesson

Congratulations, you've made it! Now watch this video and follow Jeff's advice.

If you ever get stuck with anything, these two book covers always help me.

<img src="..img/Cat.jpg"; style="max-width:350px"; align="left"> <img src="..img/Frog.jpg"; style="max-width:350px"; align="right>

10.10 Exercises

For these exercises, we will look at the core concepts from this lesson. As always, load the libraries necessary for this lesson.

```r
library(tidyverse)
library(effsize)
library(TOSTER)
```

For this exercise, we'll look at doing a full analysis using a data set and report from the excellent Open Stats Lab. For simplicity, we'll load this data directly from a csv file stored on GitHub. (To get a link for this, you must view the raw file on GitHub online.)

Schroeder and Epley(2015) explored whether you'd be more likely to get a new job after delivering a speech describing your skills, or if you wrote a speech and had a potential employer read it out.

Prediction: A person's speech (i.e., vocal tone, cadence, and pitch) communicates information about intelligence better than written words. Participants: 39 professional recruiters in one of two conditions. Conditions: Listen to audio recording of speech, or read speech aloud. Dependent Variable: Intellect aggregated across intelligence, competence, and thoughtfulness.

Also rated overall impression (composite of positive and negative), and how likely they would be to hire them (0 = not at all, 10 = extremely likely).

```r
# read in the data
raw_data <- read_csv(
  "https://raw.githubusercontent.com/gpwilliams/r4psych/master/lesson_materials/10_reporting_repr
  )
```

10.10.1 Question 1

Subset the data to the columns relevant to the research question, and make the names all lowercase for consistency. Call this, `data_subset`.

10.10.2 Question 2

Summarise the data, creating means, standard deviations, and ns for each group in the data set. Call this `data_summary`. Put the summary in a table using the `kable()` command.

10.10.3 Question 3

Create a plot to display the differences across the groups.

10.10.4 Question 4

Test the assumptions of an independent-samples *t*-test.

10.10.5 Question 5

Assuming the test assumptions are met, fit a model to your data, save this as `t_test_output`. For this case, we will just look at the effect of Intellect. Output the test on completion. (Note: do not use `summary()` here, instead just type the name of the object.)

10.10.6 Question 6

Calculate effect sizes for the *t*-test. The choice of effect size is up to you.

10.10.7 Question 7

Perform a TOST on your data evaluating the equivalence against a lower bound of -0.5 and upper bound of 0.5. What does this tell us? (Hint: Use the `with()` function on the data_summary object to access the columns for that data source.)

10.10.8 Question 8

A. Fit a linear model to the data. Save this as `lm_test_output`. How does this compare with the *t*-test? B. Output 95% confidence intervals for the linear model. Why do they differ to that of the *t*-test?

Bibliography

Field, A., Miles, J., and Field, Z. (2012). *Discovering statistics using R*. Sage publications.

Grolemund, G. and Wickham, H. (2011). Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3):1–25.

R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1.

Wickham, H., Francois, R., Henry, L., and Müller, K. (2017a). *dplyr: A Grammar of Data Manipulation*. R package version 0.7.4.

Wickham, H. and Henry, L. (2018). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.8.0.

Wickham, H., Hester, J., and Francois, R. (2017b). *readr: Read Rectangular Text Data*. R package version 1.1.1.