Introduction to R

Ryan Donovan

2024-03-12

Contents

Chapter 1

Introduction

This series of workshops describes how to use R to import, clean, and process psychological data. All materials, data, and information in these workshops are used for educational purposes only. This document should only be shared within the University of Galway's School of Psychology and is not intended for widespread dissemination. The workshop's e-book is very much in its draft stages and will be updated and refined in the future. Several materials are adapted from various online resources on teaching R.

1.1   Who is this resource for?

These workshops are designed to help people who come from a psychology or social science background learn the necessary programming skills to use R effectively in their research. These workshops are intended for individuals with no programming experience whatsoever, teaching the necessary programming skills and ideas required to conduct statistical techniques in psychology (e.g., Power Analyses, Correlation, ANOVA, Regression, Mediation, Moderation).

These workshops are **not** for people interested in learning about statistical theory or the who, what, where's of any of the aforementioned statistical techniques. I want these workshops to focus entirely on how to perform statistical analyses in R; I assume you know the rest or know how to access that information.

1.2   Should I learn R?

There are many reasons to learn R.

Psychological research is increasingly moving towards open-science practices. One of the key principles of open-science is that all aspects of data handling

- including data wrangling, pre-processing, processing, and output generation - are openly accessible. This is not only an abstract want or desire; several top-tier journals require that you submit R scripts along with any manuscripts. If you don't know how to use R (or at least no one in your lab does), then this may put you at a disadvantage.

R enables you to import, clean, analyse, and publish manuscripts from R itself. You do not have to switch between SPSS, Excel, and Word or any other software. You can conduct your statistical analysis directly in R and have that "uploaded" directly to your manuscript. In the long run, this will save you so much time and energy.

R is capable of more than statistical analysis. You can create websites, documents, and books in R. This e-book was developed in R! While these initial workshops will not be discussing how to do this (although it is something that I would like to do in the future), I wanted to mention it as an example of how powerful R can be.

1.3   What will I learn to do in R?

The following workshops will teach you how to conduct statistical analysis in R.

R is a statistical programming language that enables you to wrangle, process, and analyse data. By the end of these workshops, you should be able to import a data file into R, do some processing and cleaning, compute descriptive and inferential statistics, generate nice visualisations, and output your results.

The learning objectives of this course are:

- Learn how to import and create datasets in R.

- Learn and apply basic programming concepts such as data types, functions, and loops.

- Learn key techniques for data cleaning in R to enable statistical analysis.

- Learn how to create APA-standard graphs in R.

- Learn how to deal with errors or bugs with R code.

- Learn how to export data.

1.4   What will I not learn to do in R?

This is not an exhaustive introduction to R. Similar to human languages, programming languages like R are vast and will take years to master. After this

course, you will still be considered a "newbie" in R. But the material covered here will at least provide you a solid foundation in R, enabling you to go ahead and pick up further skills if required as you go on.

This course will teach you data cleaning and wrangling skills that will enable you to wrangle and clean a lot of data collected on Gorilla or Qualtrics. But you will not be able to easily handle all data cleaning problems you are likely to find out in the "wild" world of messy data. Such datasets can be uniquely messy, and even experienced R programmers will need to bash their head against the wall a few times to figure out a way to clean that dataset entirely in R. If you have a particularly messy dataset, you might still need to use other programmes (e.g., Excel) to clean it up first before importing it to R.

Similarly, do not expect to be fluent in the concepts you learn here after these workshops. It will take practice to become fluent. You might need to refer to these materials or look up help repeatedly when using R on real-life datasets. That's normal.

This workshop is heavily focused on the tidyverse approach to R. The tidyverse is a particular philosophical approach to how to use R (more on that later). The other approach would be to use base R. This can incite violent debates in R communities on which approach is better. We will focus mainly on tidyverse and use some base R.

This workshop does not teach you how to use R Markdown. R Markdown is a package in R that enables you to write reproducible and dynamic reports with R that can be converted into Word documents, PDFs, websites, PowerPoint presentations, books, and much more. That will be covered in the intermediate workshop programme.

1.5 Where and when will the workshops take place?

The sessions will take place in **AMB-G035** (Psychology PC Suite). The schedule for the sessions is as follows:

- Feb 7th: Introduction to R and RStudio
- Feb 14th: R Programming (Part I)
- Feb 21st: R Programming (Part II)
- Feb 28th: Data Cleaning in R (Part I)
- March 6th: Data Cleaning in R (Part II)
- March 13th: Data Visualization

- March 20th: Running Inferential Statistical Tests in R (Part I)

- March 27th: Running Inferential Statistical Tests in R (Part II)

Each session is on a Wednesday and will run between 11:00 - 13:00.

1.6   Are there any prerequisites for taking this course?

None at all. This course is beginner-friendly. You also do not need to purchase anything (e.g., textbooks or software).

1.7   Do I need to bring a laptop to the class?

If you have a laptop that you work on, I strongly encourage you to bring it. That way, we can get R and RStudio installed onto your laptop, and you'll be able to run R outside of the classroom.

If you work with a desktop, don't worry. The lab space will have computers that you can sign in and work on and use R.

Chapter 2

Getting Started with R and RStudio

This workshop introduces the programming language R and the RStudio application. Today, we will download both R and RStudio, set up our RStudio environment, and write and run our first piece of R Code. This will set us up for the rest of the workshops.

2.1 What is R?

R is a statistical programming language that enables us to instruct our computer directly to perform tasks. Typically, when we use our computers, we do not speak to them directly; instead, we interact with "translators" (i.e., applications like SPSS) via button-click interfaces to communicate with our computers on our behalf. These interfaces record and translate our instructions to our computers, which then carry out the instructions and return the results to the application, which then translates those results back to us.

Applications like SPSS are convenient. They usually have a user-friendly button-click-based interface and take away the heavy lifting of communicating with our computer. This makes them significantly easier to learn in the short term compared to programming languages.

However, these apps also limit what we can do. For example, base SPSS is functional when it comes to creating visualizations, but it is difficult to make major changes to your graph (e.g., making it interactive). If we want to create such visualizations, we will likely need to look elsewhere for it. Similarly, we might also be financially limited in our ability to use such apps, as proprietary software like SPSS is not cheap (it can cost between $3830 - 25200 for a single licence depending on the version)!

In contrast, R is a free, open-source statistical programming language that enables us to conduct comprehensive statistical analysis and create highly elegant visualizations. By learning R, we can cut out the middleman.
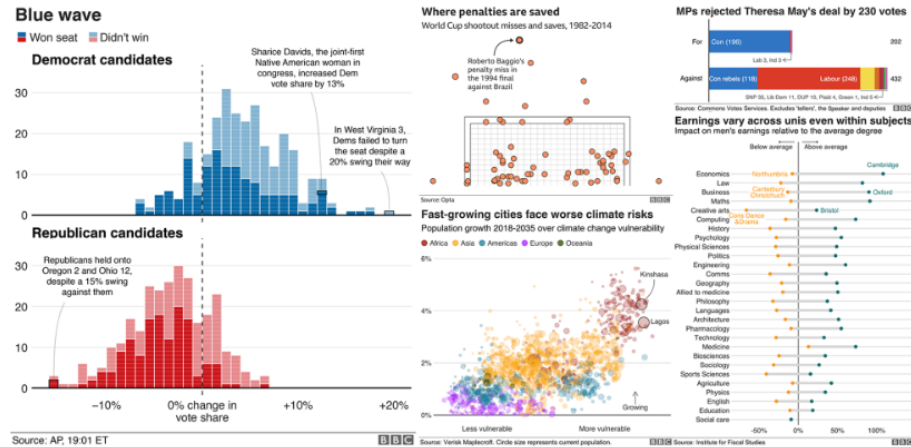
Figure 2.1: BBC graphs created in R.

But why should we learn R and not a different programming language? In contrast to other programming languages (Python, JavaScript, C), R was developed by statisticians. Consequently, R contains an extensive vocabulary to enable us to carry out sophisticated and precise statistical analysis. I have used R and Python to conduct statistical analysis, and anytime I wanted to use a less frequently used statistical test, there was significantly more support and information on how to conduct that analysis in R than in Python. For such reasons, R is typically used among statisticians, social scientists, data miners, and bioinformaticians - and will be used in this course[1].

2.2   Create a Posit Cloud Account.

In the next section, I am going to show you how to download R and RStudio on your desktop. But before we do that, I want you to set up a free account on Posit Cloud (formerly known as RStudio Cloud).

Posit Cloud enables you to use R and RStudio online for free, no need to install anything. There are limitations to this service (you only get so many hours on

---

[1]There are always tradeoffs in selecting a language. Many programming concepts are easier to grasp in Python than in R. Similarly, there is a lot of resources available for conducting machine-learning analysis in Python.

But if you are goal is conduct data cleaning, analysis, visualization, and reporting, then R is a excellent choice. The good thing is that once you achieve a certain level of competency in one programming language, you will find it significantly easier to pick up a following one.
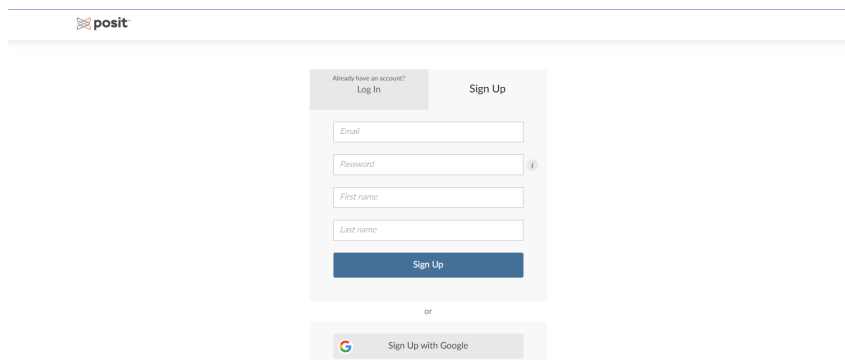
it with the free account), and I much rather you use your own computers in class. But it will be a handy back-up option in case any technical issues pop up. During class, I might not be able to solve that issue quickly and efficiently, so if it does occur, then you can sign in to Posit Cloud and keep following along with the session.

To create a Posit Cloud account, please follow the following instructions:

1. Go to their sign up page website and enter your details to create an account or Sign up with Google.

2. Once you have created an account and are in Posit Cloud, click "New Project" From the drop-down menu click "New RStudio Project". This should take a few seconds to set up (or "deploy")
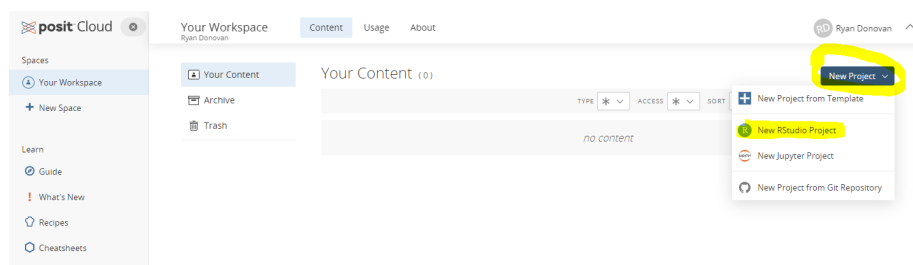
1. Once it is deployed, name your project at the top as ***rintro***
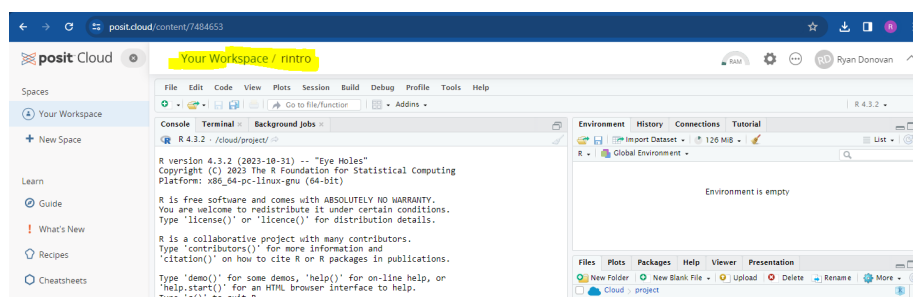
Don't worry about what anything on the screen means for now. We'll come back to that once we download RStudio on your computer. For now, you can sign out of Posit Cloud.

2.3 Downloading R on to your Computer

Please follow the following instructions to download R on either Windows or Mac.

2.3.1 Downloading R on Windows

1. Go to the website: https://cran.r-project.org/
2. Under the heading *Download and Install R,* click *Download R for Windows*
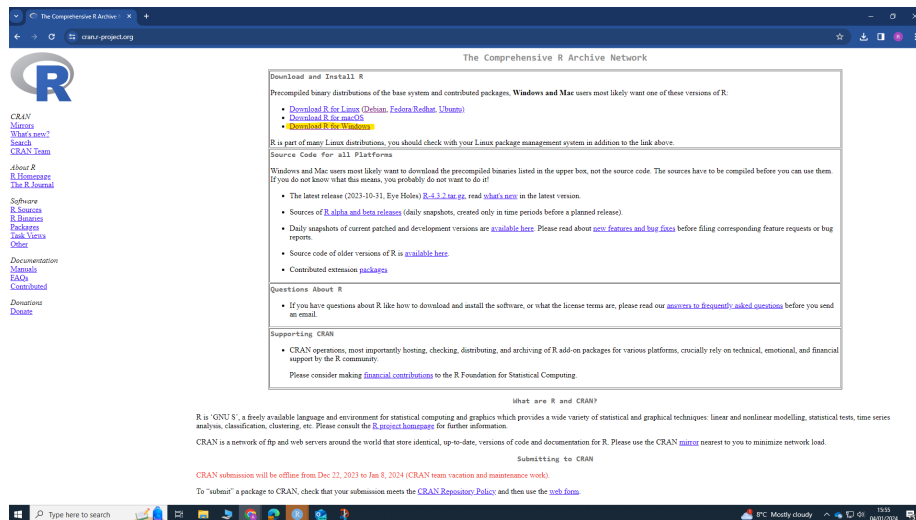
3. Click the hyperlink **base** or **install R for the first Time**
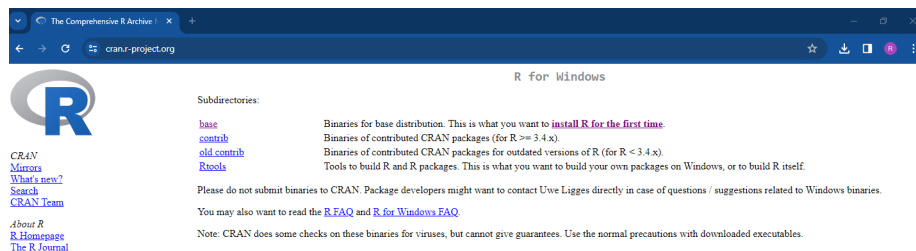
4. Click Download R-4.3.2 for Windows (depending on the date you accessed this, the version of R might have been been updated. That's okay, you can download newer versions). Let the file download.
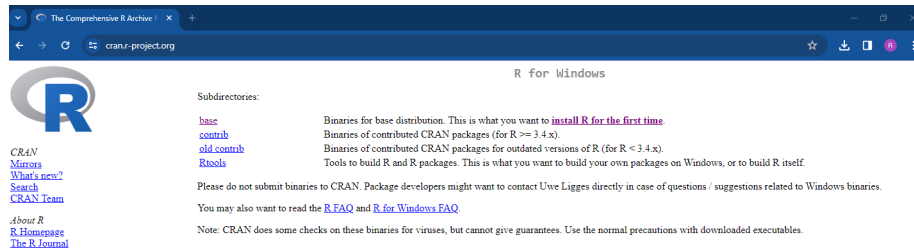
Figure 2.2: The R programming language is occasionally updated, so the specific version of R that you see might be different than mine. But that's okay!

5. Once the file has been downloaded, open it, and click "Yes" if you are asked to allow this app to make changes to your device. Then choose English as your setup language. The file name should be something like "R-4.3.2.-win". The numbers will differ depending on the specific version that was downloaded.

6. Agree to the terms and conditions and select a place to install R. It is perfectly fine to go with the default option.

2.3.2   Downloading R on Mac

The instructions are largely the same for Mac.

1. Go to the website: https://cran.r-project.org/

2. Click Download R for (Mac) OS X.
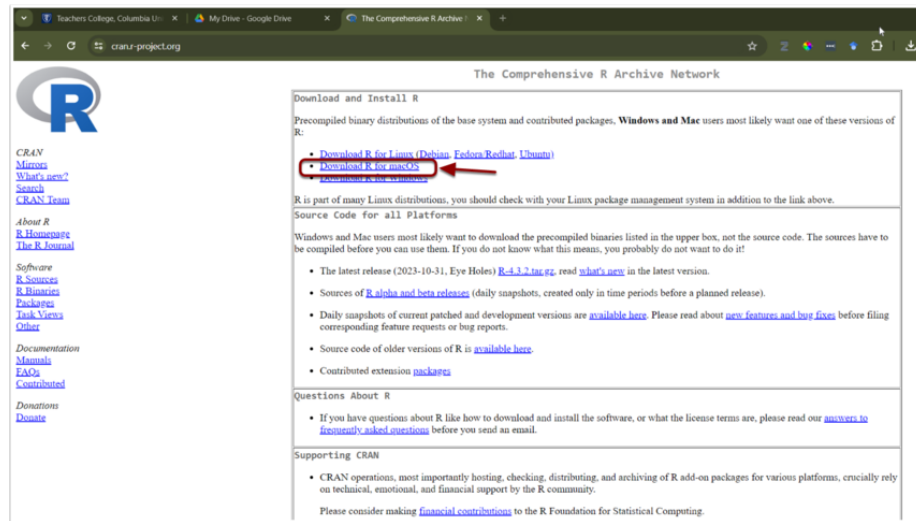
1. Check the Latest release: section for the appropriate version and follow the directions for download. If you are unsure about this, please ask me.
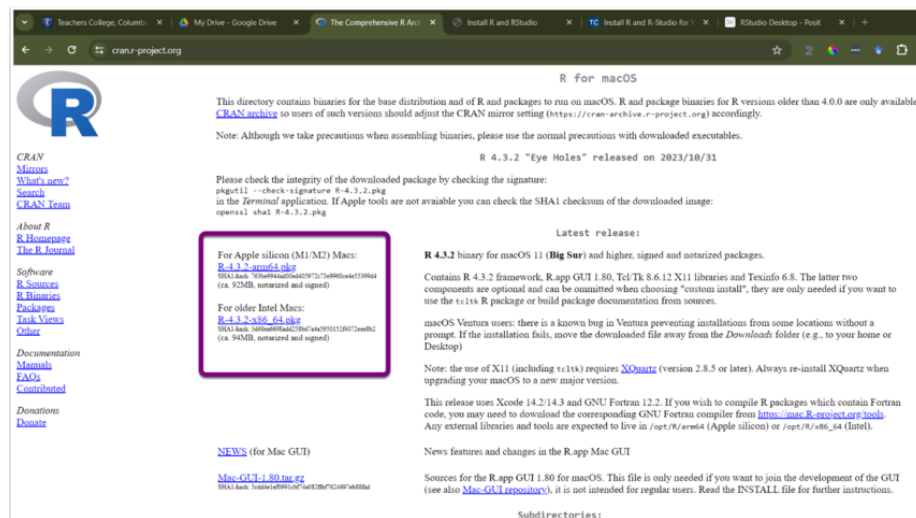
1. Once the file download is complete, click to open the installer. Click Continue and proceed through the installer, I recommend going with all default options.

1. Once the R installer has finished, click Close.
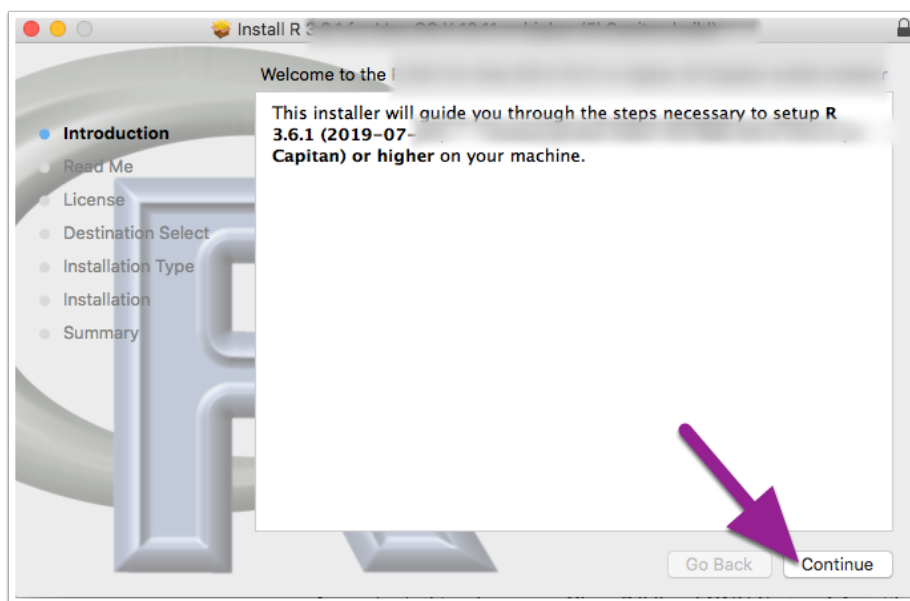
Figure 2.3: Depending on your version of Mac OS, this might look slightly different. But you should still be able to install it.
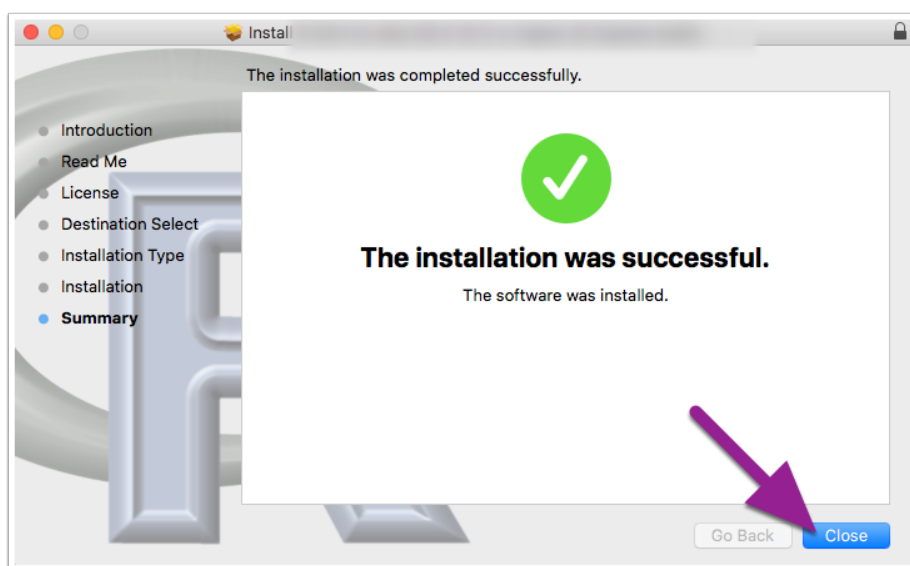
2.4   Install and Open R Studio

Once R is installed, we will install RStudio.

RStudio is a user-friendly front-end program for R, enhancing your R coding experience without sacrificing any capabilities. RStudio allows us to write and save R code, create plots, manage files, and perform other useful tasks. Think of RStudio as similar to Microsoft Word compared to a basic text editor; while you can write a paper in a text editor, it's much quicker and efficient in Word.

1. **NB:** Make sure that R is installed ***before*** trying to install R Studio.
2. Go to the RStudio website: https://posit.co/download/rstudio-desktop/
3. The website should automatically detect your operating system. Click the ***Download RStudio Desktop*** button.

1: Install R

RStudio requires R 3.3.0+. Choose a version of R that matches your computer's operating system.

DOWNLOAD AND INSTALL R

2: Install RStudio

DOWNLOAD RSTUDIO DESKTOP FOR WINDOWS

Size: 215.66 MB | SHA-256: 93C7F307 | Version: 2023.12.0+369
| Released: 2023-12-20

Once the file is downloaded, open it and allow it to make changes to your device. Then follow the instructions to download the program. I recommend using all the default options during installation.

After downloading both R and RStudio, open RStudio on your computer. You do not have to open R separately, as RStudio will work with R if everything is set up correctly.

When you first open RStudio, you will see three panes or "windows" in RStudio: "Console" (left), "Environment" (top right), and "Files" (bottom right).
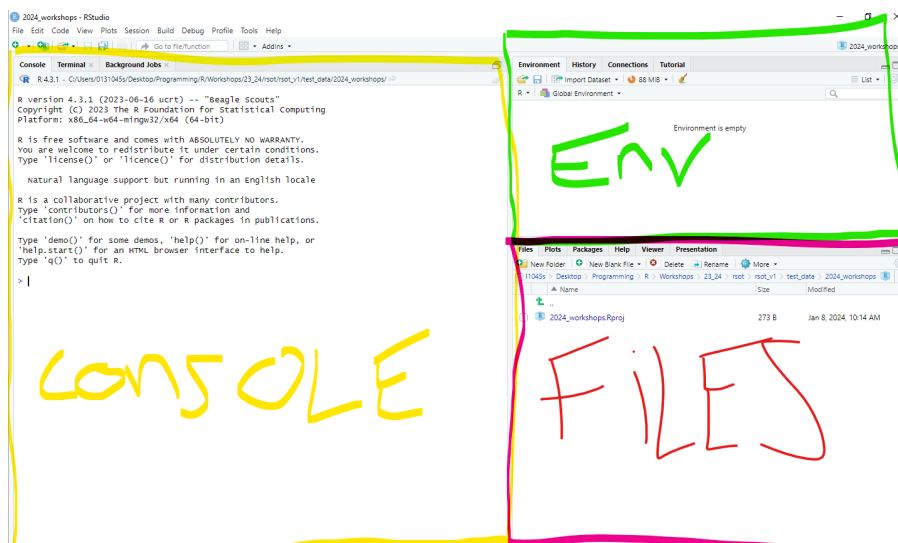
2.5   Creating an R Project

Our first step in RStudio is to create an *R Project.* R Projects are environments that group together input files (e.g., data sets), analyses on those files (e.g., code), and any outputs (e.g., results or plots). Creating an R Project will set up a new directory (folder) on your computer. Whenever you open that project, you are telling R to work within that specific directory.

Activity

Let's create an R Project that we will use during these workshops.

1. Click "File" in the top left hand corner of RStudio-> then click new "New Project"

2. The "New Project Wizard" screen will pop up. Click "New Directory" -> "New Project"

3. In the "Create New Project" screen, there are four options we are going to change.

Option 1: The "Directory name" options sets the name of the project and associated folder.

- You can set this to whatever you want. ***Just don't set it to "R",*** as this can create problems down the line.

- I *recommend* that you set the same directory name as me - ***rintro***

Option 2: The "Create project as sub-directory of" option selects a place to store this project on your computer.

- You can save it anywhere you like (e.g., your Desktop). Just ensure it's in a place you can easily find and where it won't be moved (e.g., if you save folders to your desktop but tend to relocate them later, avoid saving it on your desktop).

- My recommendation is to create a folder called "R_Programming" on your desktop and save your project inside this folder.

- Regardless of where you save your project, copy the location and keep it in a place you can check later (e.g., in a text file).

Option 3: The "Use renv with this project" option enables you to create a virtual environment for this project that will be separate to other R projects. Don't worry for now about what that means, it will be explained later on.

- Tick this option.

Option 4: The "Open in new session" just opens a new window on RStudio for this project.

- Tick this option.

Note on Github Repository: This will probably not appear on your RStudio project, but that's okay, you don't need it for this course.

You can see my example below. Once you're happy with your input for each option, click "Create Project" This will open up the project ***rintro***.

2.5.1   Navigating RStudio

In our new project, ***rintro***, we are going to open the "Source" pane, which we will often use for writing code, and viewing datasets.

There are a variety of ways to open the Source pane.

Button approach: Click the "File" tab in the top-left hand corner (not the File pane) -> Click "New File" -> "R Script"
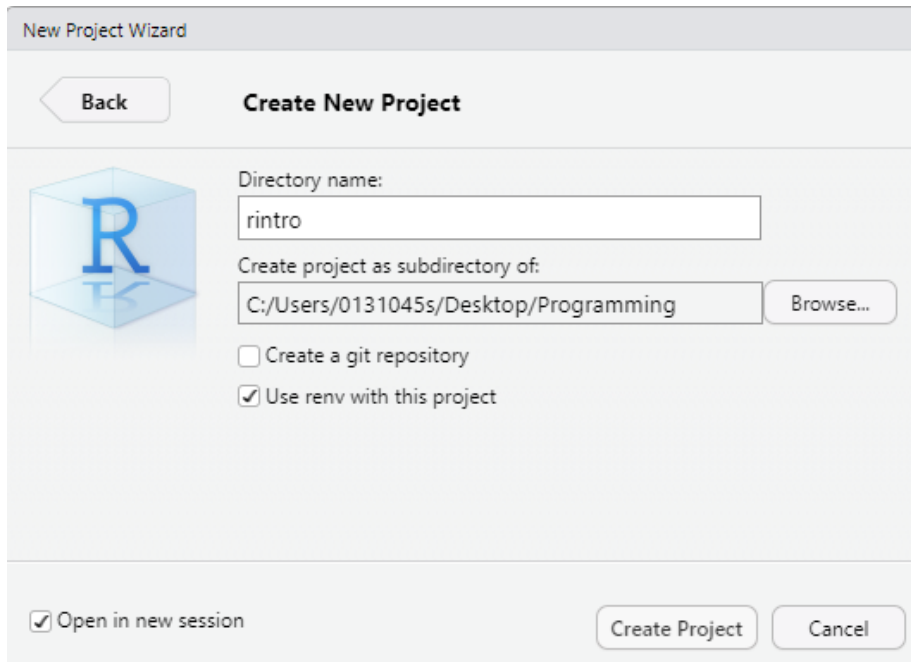
Figure 2.4: New Project Set Up
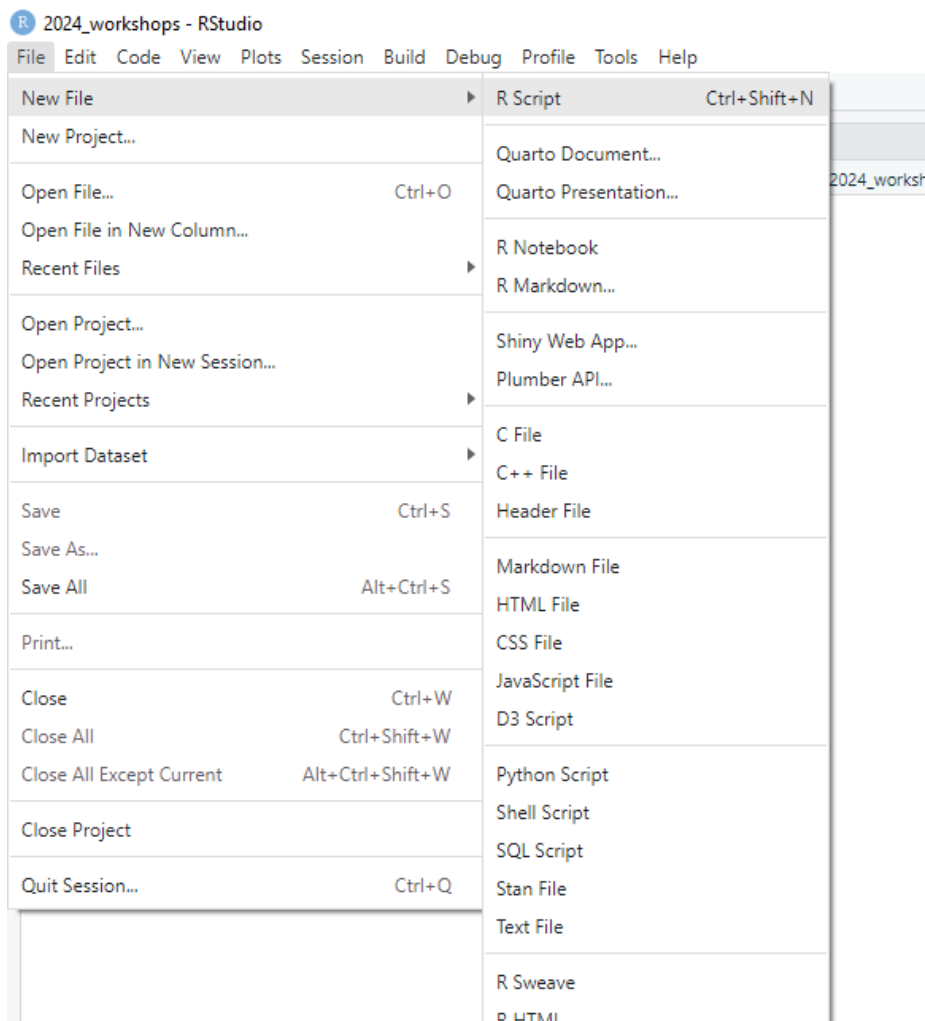
Button Shortcut: directly underneath the *File* tab, there is an icon of a white sheet with a green and white addition symbol. You can click that too.

Keyboard Shortcut: You can press "Ctrl" + "Shift" and "N" on Windows. Or "Cmd" + "Shift" + "N" on Mac.

Now you should see your four panes: Source, Console, Environment, and Files.
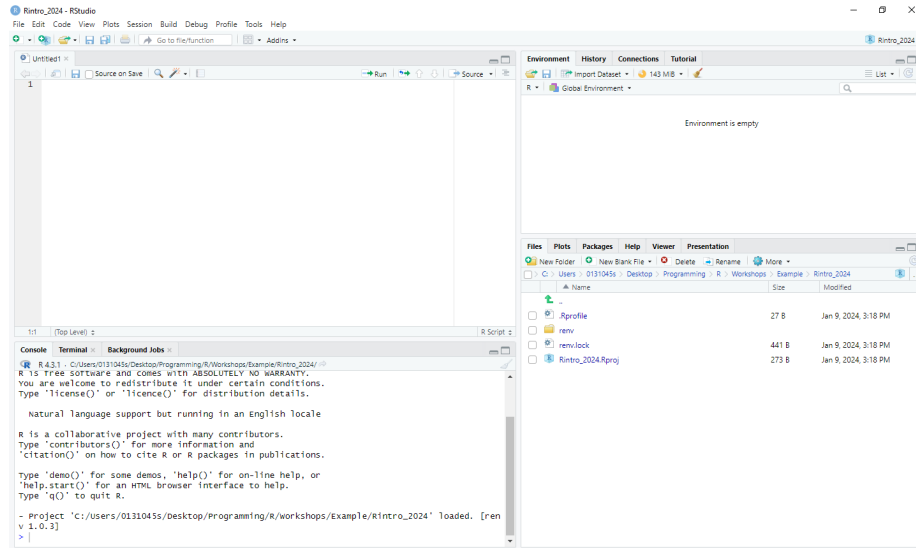
2.5.1.1   The RStudio Workspace

With each pane opened, let's briefly describe their purposes.

- The ***Source Pane*** is where you will write R scripts. R scripts enable you to write, save, and run R code in a structured format. For instance, you might have an R script titled "Descriptive," containing the code for computing descriptive statistics on your data set. Similarly, you might have another R script titled "Regression" for performing regression analyses in R.

- The ***Console Pane*** is where you can write R code or enter commands into R. The console is also where you can find various outputs from your R scripts. For example, if you create a script for running a t-test in R, the results will appear in the Console Pane. Any error or warning messages related to your code will also be highlighted in the console. In short, the console is where R actually runs.

- The ***Environment Pane*** contains information about data sets and variables imported or created in R within a specific R project. The "History" tab shows a history of the R code executed during the project. This pane

is helpful for getting an overview of a project, especially if you return to it after a long time or are reviewing someone else's code.

- The ***Files Pane*** includes your R project files (Files tab), the output of any plots you create (Plots tab), the status of downloaded packages (Packages tab), and information about R functions and packages (Help).

All four panes will be used extensively during these workshops.

2.5.2   Checking our Working Directory

Every time you open a project or file in RStudio, it's good practice to check the working directory. The working directory is the environment on your computer where R is currently operating.

Ideally, you want the working directory to match the location of your R project. This ensures that any files you import into RStudio or any files you export (datasets, results, graphs) can be easily found in your R project folder. Checking the working directory can help prevent many common R problems. To check the working directory, type the following into the console pane:

```r
getwd()
```

```
## [1] "C:/Users/0131045s/Desktop/Programming/R/Workshops/rintro"
```

This will display the current working directory where R is operating. Your working directory will likely differ from mine, which is normal. Just confirm that it matches the location you specified when creating your project (**Option 2**).

2.5.3   Setting up a new Working Directory

We are going to slightly change our working directory. In our R Project, we are going to create a folder for week1 of the workshop. Anything that we create in R will then be saved into this week1 folder.

- Click "Session" on your RStudio toolbar -> Set Working Directory -> Choose Directory
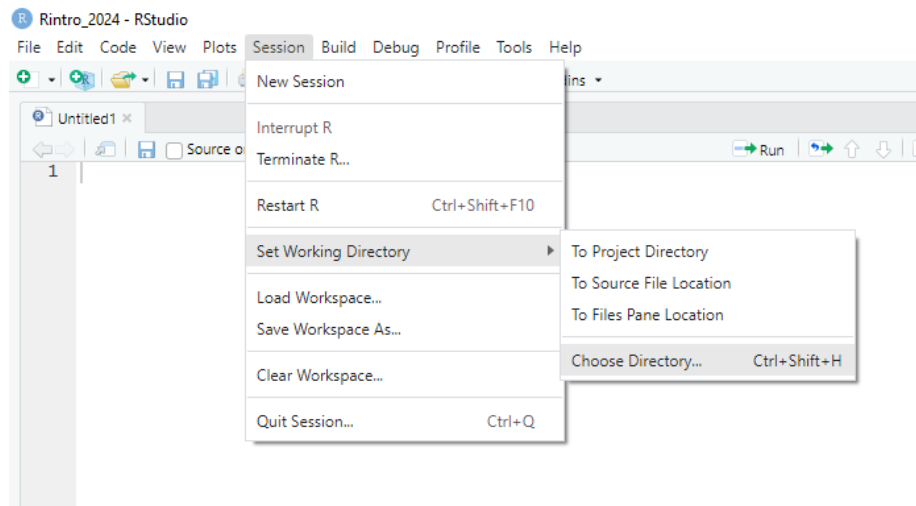
- By default you should be in your R Project (e.g., **rintro**).

- Within this R Project, create a new folder and call it "week1"

- Click "week1" and then click Open
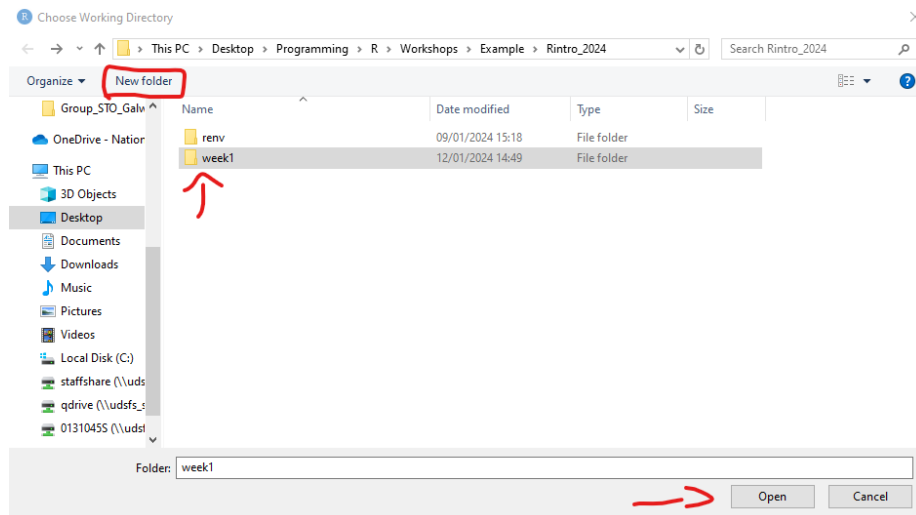
You should see something like the following in your console

```
> setwd("C:/Users/0131045s/Desktop/Programming/R/Workshops/rintro/week1")
```

Check whether this is actually the location you want to store your files for this course. If it is, we are good to go. If not, then let me know.

2.6 Writing our first R Code

Let's write our first line of R code in the console. The R console uses the prompt symbol > to indicate that it is ready for a new line of code.

Type in each of the following instructions (after the > operator) and press enter. Feel free to change the second line of code to add your own name.

```r
print("Hello World")
```

```
## [1] "Hello World"
```

```r
print("My name is Ryan and I am learning to code in R")
```

```
## [1] "My name is Ryan and I am learning to code in R"
```

Congratulations, you've written your first piece of code!

Let's describe what is going on here. We used a function called print() to print the words "Hello World" and "My name is Ryan, and I am learning to code in R" in the console. Functions are equivalent to verbs in the English language - they describe doing things. In this case, R sees the function print - then it looks inside the bracket to see what we want to print, and then it goes ahead and prints it. Pretty straightforward.

Functions are a very important programming concept, and there is a lot more going on under the hood than I have described so far - so we will be returning to functions repeatedly and filling you in with more information. But in essence, functions are verbs that enable us to tell our computer to carry out specific actions on objects.

2.7 Console vs Source Script

You might have noticed that I asked you to write code in the console rather than in the source pane. It's worth discussing here what the differences are between the console and the script when it comes to writing code.

The console is like the immediate chat with R. It's where you can type and execute single lines of code instantly. Imagine it as a friendly conversation where you ask R to perform a task, and it responds immediately. The console is great for experimenting and getting instant feedback. It's your interactive playground, perfect for spontaneous interactions with R.

The console is also really useful for performing quick calculations, testing functions or pieces of code, and for running code that should run once and only once.

However, the console is cumbersome to use if we want to write code that is several lines long and/or when we want to structure or save our code. This is where R scripts come in.

R scripts are text files where we can write R code in a structured manner. Scripts enable us to structure our code (e.g., with headings and instructions), write several pieces of code, and save and rerun code easily. If you think of your console as a draft, then your script is for the code that you want to keep.

From now on, whenever we write code, we are going to be using R scripts by default. For the times we will write code in the console, I will let you know beforehand.

2.8   Let's write some statistical code

Okay, we have talked a lot about R and RStudio. To finish off this session, let's write code that will take a data set, calculate some descriptive statistics, run an inferential test, generate a graph, and save our results. Don't worry if you don't understand all of the code provided below. Just follow along and type it yourself in the R script we opened up earlier (if it's not open, click "File" -> "New File" -> "RScript"). Once you have created this script, save it as "01-paired-t-tests".

When you download R, you will have automatic access to several functions (e.g., print) and data sets. One of these data sets is called sleep, which we are going to use right now. To learn more about the sleep data set, type **?sleep** into the console. You will find more information on the data sets in the Files pane, under the Help tab.

First, let's have a look at the sleep data set by writing the following code in the R script. To run scripts in R, select the code you have written and click the Run button with the green arrow in the top right corner of the script.

```
print(sleep)
```

```
##    extra group ID
## 1    0.7     1  1
## 2   -1.6     1  2
## 3   -0.2     1  3
## 4   -1.2     1  4
## 5   -0.1     1  5
## 6    3.4     1  6
## 7    3.7     1  7
## 8    0.8     1  8
## 9    0.0     1  9
## 10   2.0     1 10
## 11   1.9     2  1
```

```
## 12   0.8    2  2
## 13   1.1    2  3
## 14   0.1    2  4
## 15  -0.1    2  5
## 16   4.4    2  6
## 17   5.5    2  7
## 18   1.6    2  8
## 19   4.6    2  9
## 20   3.4    2 10
```

The **print()** function here prints out the sleep data set in the console. There are also other ways to view a data set, such as using the functions **head()**, **tail()**, **View()**, and **str()**. Type these in the console (make sure to put **sleep** inside the brackets) and see what results you get.

The result of **print(sleep)** shows us there are 20 observations in the dataset (rows), with three different variables (columns): extra (hours of extra sleep each participant had), group (which treatment they were given), and ID (their participant ID).

Now let's calculate some descriptive statistics. One way we can do this is by using the **summary()** function. This function takes in an object (e.g., like a data set) and summarizes the data. Write the following in your R script and press run.

```
summary(sleep)
```

```
##      extra          group        ID
##  Min.   :-1.600   1:10   1      :2
##  1st Qu.:-0.025   2:10   2      :2
##  Median : 0.950          3      :2
##  Mean   : 1.540          4      :2
##  3rd Qu.: 3.400          5      :2
##  Max.   : 5.500          6      :2
##                          (Other):8
```

Running **summary(sleep)** shows us descriptive statistics for each of our variables. We can see that the mean change in hours of sleep was +1.5, and that there were 10 participants in both the control and experimental condition.

But it's not exactly what we need. Firstly, we don't need summary descriptives on the participant ID. Secondly, it only tells us the mean of the entire sample, whereas we want the mean score for each treatment group. To get this information, we can use the **aggregate()** function, which enables us to split our data into subsets and then compute summary statistics per group. Remember to press run after you've written your code.

```r
#The code inside the aggregate bracket tells our computer to:
# data = sleep -> Go to the sleep data set

#extra ~ group -> Take the variable "extra" and split it into subsets based on the var

# FUN = mean -> Apply the mean() function (FUN) on each subset

aggregate(data = sleep, extra ~ group, FUN = mean)
```

```
##   group extra
## 1     1  0.75
## 2     2  2.33
```

That's more like it. Now we can see that there does seem to be a difference between treatment1 and treatment2. Participants slept an extra 2.33 hours on average when taking treatment 2, whereas they only slept 0.75 hours (e.g., 45 minutes) more on average when taking treatment 1. So, treatment 2 does seem more effective.

Let's run a paired-samples t-test to see if those differences are significant (I have assumed all parametric assumptions are correct).

```r
t.test(sleep$extra[sleep$group == 1], #this code extracts the group 1 scores
       sleep$extra[sleep$group == 2], # this code extracts group 2 scores
       paired = TRUE) #this code tells R to run a paired t-test, not between/independe
```

```
##
##  Paired t-test
##
## data:  sleep$extra[sleep$group == 1] and sleep$extra[sleep$group == 2]
## t = -4.0621, df = 9, p-value = 0.002833
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -2.4598858 -0.7001142
## sample estimates:
## mean difference
##           -1.58
```

Boom! We can see there is a statistically significant difference between the two groups. I know the code within the t-test might look a bit complicated, but we will break it down and explain it as we go on in further weeks.

Finally, let's visualize our data with the plot() function.

```r
plot(sleep$group, sleep$extra)
```
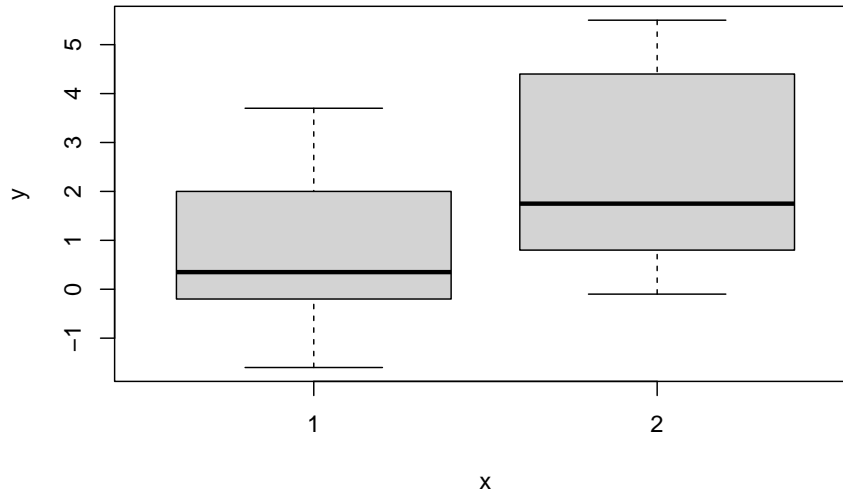

Figure 2.5: Generic Boxplot

The `plot()` function is an example of a generic function, which means it adapts to our code. In this case, the plot() function looks at the variables we want to plot and identifies that the box plot is the most appropriate way to plot it.

Now this plot is perfectly adequate for a first viewing, but let's make it a bit more instructive by adding labels to the x and y-axes, and by adding a title to it.

```r
#xlab = creates a label for the x-axis

#ylab = creates a title for the y-axis

#main = creates a title for the plot


plot(sleep$group, sleep$extra, xlab = "Treatment", ylab = "Hours of Sleep", main = "Effect of Tre
```

Now let's take this plot and save it to a PDF so that we can share our results with others. The standard way of doing this in R is a bit cumbersome. We have

Effect of Treament on Sleep Duration
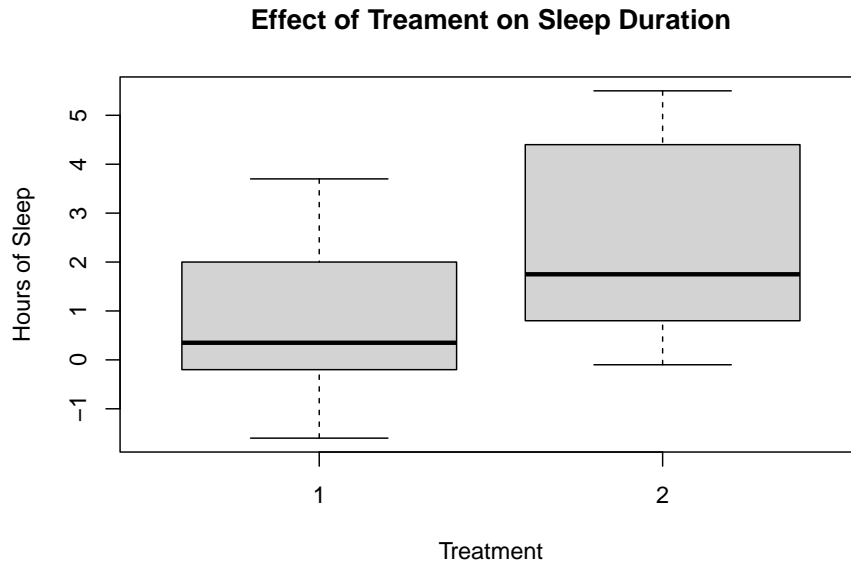
Figure 2.6: Generic Boxplot with appropriate labelling

to tell R that we are about to create a plot that we want to make into a PDF. Then we have to generate the plot. Then we have to tell R we are done with creating the PDF. We'll learn a MUCH simpler way to do this in future weeks, but this will do for now.

```r
pdf(file = "myplot.pdf") #Tells R that we will create a pdf file called "my_plot" in o

plot(sleep$group, sleep$extra, xlab = "Treatment", ylab = "Hours of Sleep", main = "Eff

dev.off() #this tells R that we are done with adding stuff to our PDF
```

```
## pdf
##   2
```

Go to the files pane, and open up the pdf "myplot.pdf". It should be in your working directory. Open it up the PDF and have a look at your graph[2].

---

[2]This is a fairly generic type of graph offered by base R. During the course we will looking at ways we can create "sexier" and more APA friendly type of graphs. But for one line of code, it's not bad!

2.8.1 Comments

One last concept before we finish. You might have noticed that I wrote several things with a **#** before them. These are known as comments. Comments are any piece of text that will be ignored by R (i.e., they will not be executed within the console). They are fundamental to writing clear code.

We create comments using the **#** symbol. This symbol tells R to ignore whatever comes directly ***afterwards***.

There are various reasons for using comments.
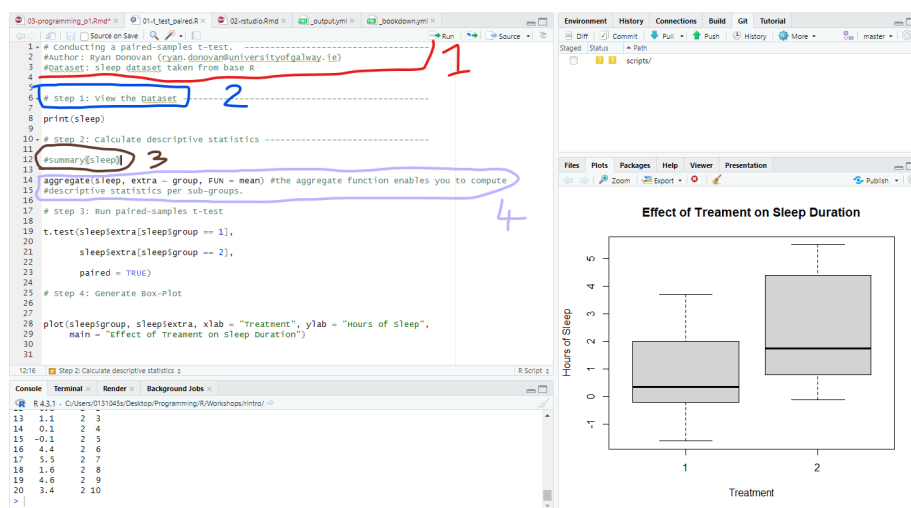
Figure 2.7: Four Examples of Comments Use

In the above figure, you'll see four different types of comments.

1. The first type of comment provides a quick introduction to the R script. It can be really useful here to provide clear information on what this script is trying to do (e.g., run a paired samples t-test), what data it is working on (the sleep dataset), and who wrote or developed this script. This makes it significantly easier for anyone who might be reviewing your work or trying to apply your code to their own work to understand what is going on.

2. The second type of comment structures the format of the script by providing headings or steps. Again, this just makes it easier to understand what is going on.

3. The third type of comment is placed before the summary. This means that the code **summary(sleep)** will not be executed in R. Why would we do this? If you remember last week, we wanted to compute the mean per each of our two treatment groups, which the summary function does not enable

us to do, so it's not part of our main analysis. So why keep it? Well, it still provides us with valuable information (e.g., mean, median, min, max for the entire sample), so rather than delete it, we'll just put a comment in front of it. And if any time we want to check these descriptives, we can just remove the **#** and run that line of code.

4. The fourth type of comment provides some context or information on what a specific line of code is doing, namely, what the **aggregate()** function does. Again, this is really useful, particularly if you are using functions that are not well-known.

Comments are extremely useful for orienting yourself to code. My advice would be to comment as much as your code as you. Anyone who has coded will have experienced the following situation - You spend days/weeks writing a piece of code to clean a messy data set and run a specialized type of analysis. Several months go by, and you need to return to your data set (pesky reviewer #2 wants you to change something). You open up your R script, and you are ***completely lost***. You have written no comments, so you have to spend days trying to remember what each piece of code was trying to do.

If you comment a lot, it will save you so much heartache in the future. And it will help you understand various code concepts better if you can explain them while you are using them. So comment, comment, comment!

2.9   Summary

There we have it! That completes our first session with R and RStudio. Today was more about getting to grips with the software R and RStudio, but we still got our first pieces of code written. Hopefully, it's given you a tiny glimpse into what R can do.

In the next two sessions, we will learn basic programming concepts and how to import data in R.

2.10   Glossary

This glossary defines key terms introduced in Chapter 2.

Term	Definition
Comment	Text in an R script that is ignored by R. Comments are preceded by the **#** symbol and are used to add explanations, headings, or disable code temporarily.

Term	Definition
Console	The interactive interface in RStudio where you can type and execute R commands and see their immediate output.
Environment Pane	The pane in RStudio that displays information about data sets, variables, and the history of R commands used in the current R session.
Files Pane	The pane in RStudio that displays the files and folders in your current working directory, as well as other useful tabs like Plots, Packages, and Help.
Function	A fundamental programming concept in R, representing a reusable block of code that performs a specific task. Functions are like verbs in English; they describe actions.
R	A programming language and environment for statistical analysis and data visualization.
R Project	An environment created in RStudio that groups together input files, code, and outputs. It helps organize and manage your work in a specific directory.
RStudio	An integrated development environment (IDE) for R, providing a user-friendly interface and tools for coding, data analysis, and visualization.
Script	A file containing a sequence of R commands that can be saved, executed, and reused.
Source Pane	The pane in RStudio where you can write and edit R scripts.
Term	Definition
Working Directory	The directory or folder on your computer where R is currently operating. It is important for managing file paths and organizing project files.

Chapter 3

R Programming (Part I)

Today, we are going to explore fundamental programming concepts in R. By the end of this session, you should be capable of the following:

- Running and troubleshooting commands in the R console.
- Understanding different data types and when to use them.
- Creating and using variables, and understanding best practices in naming variables.
- Grasping key data structures and how to construct them.

3.1   Activity 1: Set up your Working Directory

It's good practice to set your working directory when you first open RStudio. Remember that the working directory is the location where we want to store any resulting data files or scripts that you'll work on in a session. Last week I showed you how to do this using a button-and-click interface.

Using those instructions, create a folder called "Week2" in the **rintro** project folder and set it as your working directory. Your output should be something like this:

```
> setwd("C:/Users/0131045s/Desktop/Programming/R/Workshops/Example/Rintro_2024/week2")
```

Use the ´getwd()´ to check that it has been set as your working directory.

3.2   Using the Console

In the previous chapter, I made a distinction between the script and the console. I said that the script was an environment where we would write and run polished

code, and the R console is an environment for writing and running "dirty" quick code to test ideas, or code that we would run once.

That distinction is kinda true, but it's not completely true. In reality, when we write a script we are preparing **commands** for R to **execute** in the console. In this sense, the R script is equivalent to a waiter. We tell the waiter (script) what we want to order, and then the waiter hands that order to the chef (console).

It's important to know how to work the R console, even if we mostly use scripts in these workshops. We don't want the chef to spit on our food.

3.2.1   Typing Commands in the Console

We can command the R console to perform calculations. When following along in RStudio, there's no need to type the `>` operator; it simply indicates that R is ready to execute a new command, which can be omitted for clarity.[1]

```
> 10 + 20

[1] 30
```

```
> 20 / 10

[1] 2
```

When performing calculations in R, it's important to know that it follows the usual arithmetic convention of the order of operations (remember BEDMAS - Brackets, Exponents, Division, Multiplication, Addition, and Subtraction?).

```
> (20 + 10 / 10) * 4

[1] 84

> ((20 + 10) / 10) * 4

[1] 12
```

You may have noticed that the output of each code line we entered starts with a `[1]` before the actual result. What does this mean?

This is how R labels and organizes its responses. Think of it as having a conversation with R, where every question you ask gets an answer. The square brackets with a number, like `[1]`, serve as labels on each response, indicating which answer corresponds to which question. This is R **indexing** its answer.

---

[1]Including the ">" is a pain when formatting this book, so I won't include ">" in examples of code from this point forward.

In all the examples above, we asked R questions that have only 1 answer, which is why the output is always `[1]`. Look what happens when I ask R to print out multiple answers.

```r
print(sleep$extra) #this will print out the extra sleep column in the sleep dataset we used last
```

```
##  [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0  1.9  0.8  1.1  0.1 -0.1
## [16]  4.4  5.5  1.6  4.6  3.4
```

Here R tells us that the first answer (i.e., value) corresponds to `0.1`. The next label is `[16]`. which tells us that the 16th answer corresponds to 4.4.

But why does it only show the `[1]` and `[16]`th index? If you run this code in your console, you might actually see a different number than `[16]` depending on wide your console is on your device.

This is because R only prints out the index when a new row of data is needed in the console. If there were indexes for every single answer, it would clutter the console with unnecessary information. So R uses new rows as a method for deciding when to show us another index.

We'll delve deeper into indexing later in this session; it's a highly useful concept in R.

3.2.2 Console Syntax (Aka "I'm Ron Burgundy?")

3.2.2.1 R Console and Typos

One of the most important things you need to know when you are programming, is that you need to type *exactly* what you want R to do. If you make a mistake (e.g., a typo), R won't attempt to decipher your intention. For instance, consider the following code:

```r
> 10 = 20
```

```
## Error in 10 = 20: invalid (do_set) left-hand side to assignment
```

R interprets this as you claiming that 10 equals 20, which is not true. Consequently, R panics and refuses to execute your command. Now any person looking at your code would guess that since + and = are on the same key on our keyboards, you probably meant to type `10 + 20`. But that's because we have a strong theory of mind, whereas programming languages do not.

So be exact with your code or else be Ron Burgundy?.

On the grand scheme of mistakes though, this type of mistake is relatively harmless because R will tell us immediately that something is wrong and stop us from doing anything.

However, there are silent types of mistakes that are more challenging to resolve. Imagine you typed `-` instead of `+`.

```
> 10 - 20

[1] -10
```

In this scenario, R will runs the code and produces the output. This is because the code still makes sense; it is perfectly legitimate to subtract 20 away from 10. R doesn't know you actually meant to add `10` to `20`. All it can see is three objects `10`, `-`, and `20` in a logical order, so it executes the command. In this relationship, you're the one in charge.

In short calculations like this, it's clear what you've typed wrong. However, if you have a long block of connected code with a typo like this, the result can significantly differ from what you intended, and it might be hard to spot.

The primary way to check for these errors is to always review the output of your code. If it looks significantly different from what you expected, then this silent error may be the cause.

I am not highlighting these issues to scare you, it's just important to know that big problems (R code not running or inaccurate results) can often be easily fixed by tiny changes.

3.2.2.2   R Console and Incomplete Commands

I've been pretty mean to the console, but there are rare times it will be a good Samaritan. For example, if R thinks you haven't finished a command it will print out `+` to allow you to finish it.

```
> (20 + 10

+ )

[1] 30
```

So when you see "+" in the console, this is R telling you that something is missing. R won't let you enter a new command until you have finished with it.

```
(20 + 10
```

```
+ #if I press enter, it will keep appearing until I finish the code
+
+
+
+
```

If nothing is missing, then this indicates that your code might not be correctly formatted. To break out of the endless loops of "+", press the **Esc** key on your keyboard.

3.2.3 Exercises

1. Practice performing basic calculations in R console. Calculate the following:

   1. 25 multiplied by 4
   2. 72 divided by 8 3
   3. 0 multiplied by 4, and then divided by 2

2. Imagine you want to calculate the average/mean of the following 5 numbers 15, 22, 18, 30, and 25. Use the R console to find the average.
3. If I type the following code, then I get the + operator, how can I fix it?

```
> (60 / 100

+
```

3.3 Data Types

Our overarching goal for this course is to enable you to import your data into R, select the most relevant subset of data for analysis, conduct descriptive and statistical analysis, and create nice data visualizations. But it's important to consider **What is data and how is it stored in R?**

Data comes in various forms: numeric (integers and decimal values) or alphabetical (characters or lines of text). R has developed a system for categorizing this range of data into different data types.

3.4 Basic Data types in R

R has 4 basic data types that are used 99% of the time:

3.4.1 Character

A character is anything enclosed within quotation marks. It is often referred to as a *string*. Strings can contain any text within single or double quotation marks.

```r
#we can use the class() function to check the data type of an object in R

class("a")
```

```
## [1] "character"
```

```r
class("cat")
```

```
## [1] "character"
```

Numbers enclosed in quotation marks are also recognised as character types in R.

```r
class("3.14") #recognized as a character
```

```
## [1] "character"
```

```r
class("2") #recognized as a character
```

```
## [1] "character"
```

```r
class(2.13) #not recognised as a character
```

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

3.4.2 Numeric (or Double)

In R, the numeric data type represents all real numbers, with or without decimal value, such as:

```r
class(33)
```

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

```
class(33.33)
```

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

```
class(-1)
```

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

3.4.3 Integer

An integer is any real whole number without decimal points. We tell R to specify something as an integer by adding a capital "L" at the end.

```
class(33L)
```

```
## [1] "integer"
```

```
class(-1L)
```

```
## [1] "integer"
```

```
class(0L)
```

```
## [1] "integer"
```

You might wonder why R has a separate data type for integers when numeric/double data types can also represent integers. The very techincal and boring answer is that integers consume less memory in your computer compared to the numeric or double data types. `33 contains less information than 33.00`. So, when dealing with very large datasets (in the millions) consisting exclusively of integers, using the integer data type can save substantial storage space.

It's unlikely that you will need to use integers over numeric/doubles for your own research, but its good to be aware of just in case.

3.4.4   Logical (otherwise know as Boolean)

The Logical data type has two possible values: **TRUE** and **FALSE**. In programming, we frequently need to handle conditions and make decisions based on whether specific conditions are true or false. For instance, did a student pass the exam? Is a p-value below 0.05?

The Logical data type in R allows us to represent and work with these true or false values.

```r
class(TRUE)
```

```
## [1] "logical"
```

```r
class(FALSE)
```

```
## [1] "logical"
```

One important note is that it is case-sensitive, so typing any of the following will result in errors:

```r
class(True)    # Error: object 'True' not found
class(False)   # Error: object 'False' not found
class(true)    # Error: object 'true' not found
class(false)   # Error: object 'false' not found
```

The distinction between data types in programming is crucial because some operations are only applicable to specific data types. For example, mathematical operations like addition, subtraction, multiplication, and division are only meaningful for numeric and integer data types.

```r
11.00 + 3.23 #will work
```

```
[1] 14.23
```

```r
11 * 10 #will work
```

```
[1] 120
```

```r
"11" + 3 # gives error
```

```
Error in "11" + 3 : non-numeric argument to binary operator
```

This is an important consideration when debugging errors in R. It's not uncommon to encounter datasets where a column that should be numeric is incorrectly saved as a character. If you intend to perform a statistical operation on such a column (e.g., calculating the mean), you would first need to convert it to the numeric data type using the **as.numeric()** function.

```r
as.numeric("22")
```

```
## [1] 22
```

The following functions enable you to convert one data type to another:

```r
as.character()   # Converts to character
as.integer()     # Converts to integer
as.logical()     # Converts to logical
```

3.4.5 Exercises

1. Have a look at each of the following pieces of code and guess what data type it is. Check whether you are correct by using the **class()** function.

   1. `"Hello World!"`
   2. `43`
   3. `"42.34"`
   4. `FALSE`
   5. `44.4`

2. The following data types have been erroneously entered in R. Use the appropriate converting function to correct for it.

   1. Convert "42.34" from character to numeric.
   2. Convert "FALSE" from logical to character.
   3. Convert 2024 from numeric to string.
   4. Convert 1 from integer to logical (see what happens!). For bonus points, convert 0 from numerical to logical as well.

3.5 Variables

Until now, the code we've used has been disposable; once you type it, you can only view its output. However, programming languages allow us to store information in objects called ***variables.***

Variables are labels for pieces of information. Instead of running the same code to produce information each time, we can assign it to a variable. Let's say I have a character object that contains my name. I can save that character object to a variable.

```r
name <- "Ryan"
```

To create a variable, we specify the variable's name (**name**), use the assignment operator (**<-**) to inform R that we're storing information in **name**, and finally, provide the data (in this case, the string "Ryan"). Once we execute this code, every time R encounters the variable **name**, it will substitute it with "Ryan."

```r
print(name)
```

```
## [1] "Ryan"
```

Some of you might have seen my email and thought, *"Wait a minute, isn't your first name Brendan? You fraud!"* Before you grab your pitchforks, yes, you're technically correct. Fortunately, we can reassign our variable labels to new information.

```r
name <- "Brendan" #please don't call me this

print(name)
```

```
## [1] "Brendan"
```

We can use variables to store information for each data types.

```r
age <- 30L

height <- 175 #centimetre

live_in_hot_country <- FALSE

print(age)
```

```
## [1] 30
```

```r
print(height)
```

```
## [1] 175
```

```r
print(live_in_hot_country)
```

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

```r
paste("My name is", name, "I am", age, "years old and I am", height, "cm tall. It is", live_in_ho
```

```
## [1] "My name is Brendan I am 30 years old and I am 175 cm tall. It is FALSE that I was born in
```

We can use variables to perform calculations with their information. Suppose I have several variables representing my scores on five items measuring Extraversion (labeled **extra1** to **extra5**). I can use these variable names to calculate my total Extraversion score.

```r
extra1 <- 1
extra2 <- 2
extra3 <- 4
extra4 <- 2
extra5 <- 3

total_extra <- extra1 + extra2 + extra3 + extra4 + extra5

print(total_extra)
```

```
## [1] 12
```

```r
mean_extra <-  total_extra/5

print(mean_extra)
```

```
## [1] 2.4
```

Variables are a powerful tool in programming, allowing us to create code that works across various situations.

3.5.1 What's in a name? (Conventions for Naming Variables)

There are strict and recommended rules for naming variables that you should be aware of.

Strict Rules (Must follow to create a variable in R)

- Variable names can only contain uppercase alphabetic characters A-Z, lowercase a-z, numeric characters 0-9, periods `.`, and underscores `_`.

- Variable names must begin with a letter or a period (e.g., **1st_name** or **_1stname** is incorrect, while **first_name** or **.firstname** is correct).

- Avoid using spaces in variable names (`my name` is not allowed; use either `my_name` or `my.name`).

- Variable names are case-sensitive (`my_name` is not the same as `My_name`).

- Variable names cannot include special words reserved by R (e.g., if, else, repeat, while, function, for, in, TRUE, FALSE). While you don't need to memorize this list, it's helpful to know if an error involving your variable name arises. With experience, you'll develop an intuition for valid names.

Recommended Rules (Best practices for clean and readable code):

- Choose informative variable names that clearly describe the information they represent. Variable names should be self-explanatory, aiding in code comprehension. For example, use names like "income," "grades," or "height" instead of ambiguous names like "money," "performance," or "cm."

- Opt for short variable names when possible. Concise names such as `dob` (date of birth) or `iq` (intelligence quotient) are better than lengthy alternatives like `date_of_birth` or `intelligence_quotient`. Shorter names reduce the chances of typos and make the code more manageable.

- However, prioritize clarity over brevity. A longer but descriptive variable name, like `total_exam_marks`, is preferable to a cryptic acronym like `tem`.

- Avoid starting variable names with a capital letter. While technically allowed, it's a standard convention in R to use lowercase letters for variable and function names. Starting a variable name with a capital letter may confuse other R users.

- Choose a consistent naming style and stick to it. There are three common styles for handling variables with multiple words:

  1. **snake_case**: Words are separated by underscores (e.g., `my_age`, `my_name`, `my_height`). This is the preferred style for this course as it aligns with other programming languages.
  2. **dot.notation**: Words are separated by periods (e.g., `my.age`, `my.name`, `my.height`).
  3. **camelCase**: Every word, except the first, is capitalized (e.g., `myAge`, `myName`, `myHeight`).

For the purposes of this course, I recommend using `snake_case` to maintain consistency with my code. Feel free to choose your preferred style outside of this course, but always maintain consistency.

3.5.2 Exercises

1. Create a variable called `favourite_colour` and assign your favourite colour to this `favourite_colour`. What data type is this variable? Check it with `class()`.

2. Create two numeric variables, `num1` and `num2`, and assign them any two different numeric values.

3. Calculate the sum of `num1` and `num2` and store it in a new variable called `sum_result`

4. Print the value of `sum_result`.

5. Create a variable named `height_cm` and assign it your height in centimeters (a numeric value).

6. Create another variable named `height_m` and assign it the height in meters by dividing `height_cm` by 100.

7. Print the value of `height_m`.

3.6 Data Structures

So, we've talked about the different types of data that we encounter in the world and how R classifies them. We've also discussed how we can store this type of data in variables. However, in data analysis, we rarely work with individual variables. Typically, we work with large collections of variables that have a particular order. For example, datasets are organized by rows and columns.

This also holds true in R, which has several different types of **data structures** that organize and group together variables. Each data structure has specific rules and methods for creating or interacting with them. Today we are going to focus on the two main data structures we'll use in this course: **vectors** and **data frames**.

3.6.1 Vectors

The most basic and (probably) important data structure in R is ***vectors***. You can think of vectors as a list of data in R that are of the same data type.

For example, I could create a character vector with names of people in the class:

```r
rintro_names <- c("Gerry", "Aoife", "Liam", "Eva", "Helena", "Ciara", "Niamh", "Owen")


print(rintro_names)
```

```
## [1] "Gerry"  "Aoife"  "Liam"   "Eva"    "Helena" "Ciara"  "Niamh"  "Owen"
```

```
is.vector(rintro_names)
```

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

And I can create a numeric vector with their (totally randomly generated!)[2]

```
rintro_marks <- c(69, 65, 80, 77, 86, 88, 92, 71)

print(rintro_marks)
```

```
## [1] 69 65 80 77 86 88 92 71
```

And I can create a logical vectors that describes whether or not they were satisfied with the course (again randomly generated!):

```
rintro_satisfied <- c(FALSE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE, FALSE)

print(rintro_satisfied)
```

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

Technically, we have been using vectors the entire class. Vectors can have as little as 1 piece of data:

```
instructor <- "Ryan/Brendan"

is.vector(instructor)
```

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

However, we can't include multiple data types in the same vector. Going back to our numeric grades vector, look what happens when we try to mix in grades as characters:

```
rintro_grades <- c(69, 65, 80, 77, 86, 88, "A1", 71)


print(rintro_grades)
```

---

[2]I used the function `rnorm()` to generate these values. If you want to read more about this very handy function, type `?rnorm` into the console, or follow this link.

```
## [1] "69" "65" "80" "77" "86" "88" "A1" "71"
```

R has converted every element within the `rintro_grades` vector into a character. If R sees an object that is a vector but sees that its elements belong to different data types, it will try to convert every element to one data type. This is a strict rule in R - a vector can only be created if every single element (i.e., thing) inside that vector is of the same data type.

If we were to check the class of `rintro_marks` and `rintro_grades`, it will show us this conversion

```
class(rintro_marks)

[1] "numeric"
```

```
class(rintro_grades)

[1] "character"
```

Remember how I mentioned that you might download a dataset with a column that has numeric data but is actually recognized as characters in R? This is one scenario where that could happen. The person entering the data might have accidentally entered text into a cell within a data column. When R reads this column, it sees the text, and then R converts the entire column into characters.

3.6.1.1 Working with Vectors

We can perform several types of operations on vectors to gain useful information.

Numeric and Integer Vectors

We can run functions on vectors. For example, we can run functions like **mean()**, **median**, or **sd()** to calculate descriptive statistics on numeric or integer-based vectors:

```
mean(rintro_marks)
```

```
## [1] 78.5
```

```
median(rintro_marks)
```

```
## [1] 78.5
```

```r
sd(rintro_marks)
```

```
## [1] 9.724784
```

A useful feature is that I can sort my numeric and integer vectors based on their scores:

```r
sort(rintro_marks) #this will take the original vector and arrange from lowest to high
```

```
## [1] 65 69 71 77 80 86 88 92
```

The **sort()** function by default arranges from lowest to highest, but we can also tell it to arrange from highest to lowest.

```r
sort(rintro_marks, decreasing = TRUE)
```

```
## [1] 92 88 86 80 77 71 69 65
```

Character and Logical Vectors

We are more limited when it comes to operators with character and logical vectors. But we can use functions like **summary()** to describe properties of character or logical vectors.

```r
summary(rintro_names)
```

```
##    Length     Class      Mode
##         8 character character
```

```r
summary(rintro_satisfied)
```

```
##     Mode    FALSE     TRUE
## logical        4        4
```

The **summary()** function tells me how many elements are in the character vector (there are six names), whereas it gives me a breakdown of results for the logical vector.

> marks -> c(87, 92, 88, 77, 70, 80, 90, 75)

marks	87	92	88	77	70	80	90	75
index	1	2	3	4	5	6	7	8

Figure 3.1: Indexing for Numeric Vector

> names -> c("Ryan", "Gerry", "Aoife", "Ciara", "Eva", "Liam", "Niamh", "Owen")

name	"Ryan"	"Gerry"	"Aoife"	"Ciara"	"Eva"	"Liam"	"Niamh"	"Owen"
index	1	2	3	4	5	6	7	8

Figure 3.2: Indexing for Character Vector

> satisfied -> c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE, FALSE)

name	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE
index	1	2	3	4	5	6	7	8

Figure 3.3: Indexing for Logical Vector

3.6.1.2 Vector Indexing and Subsetting

A vector in R is like a list of items. To be more specific, vectors in R are actually *ordered* lists of items. Each item in that list will have a position (known as its index). When you create that list (i.e., vector), the order in which you input the items (elements) determines its position (index). So the first item is at index 1, the second at index 2, and so on. Think of it like numbering items in a shopping list:

This property in vectors means we are capable of extracting specific items from a vector based on their position. If I wanted to extract the first item in my list, I can do this by using `[]` brackets:

```r
rintro_names[1]
```

```
## [1] "Gerry"
```

Similarly, I could extract the 3rd element.

```r
rintro_marks[3]
```

```
## [1] 80
```

Or I could extract the last element.

```r
rintro_satisfied[8]
```

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

This process is called subsetting. I am taking an original vector and taking a sub-portion of its original elements.

I can ask R even to subset several elements from my vector based on their position. Let's say I want to subset the 2nd, 4th, and 6th elements. I just need to use `c()` to tell R that I am subsetting several elements:

```r
rintro_names[c(2, 4, 8)]
```

```
## [1] "Aoife" "Eva"   "Owen"
```

```r
rintro_marks[c(2, 4, 8)]
```

```
## [1] 65 77 71
```

```r
rintro_satisfied[c(2, 4, 8)]
```

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

If the elements you are positioned right next to each other on a vector, you can use : as a shortcut:

```r
rintro_names[c(1:4)] #this will extract the elements in index 1, 2, 3, 4
```

```
## [1] "Gerry" "Aoife" "Liam"  "Eva"
```

It's important to know, however, that when you perform an operation on a vector or you subset it, it does not actually change the original vector. None of these following code will actually change rintro_marks.

```r
sort(rintro_marks, decreasing = TRUE)

[1] 91 90 89 88 87 87

print(rintro_marks)
[1] 69 65 80 77 86 88 92 71

rintro_marks[c(1, 2, 3)]

[1] 87 91 87

print(rintro_marks)
[1] 69 65 80 77 86 88 92 71
```

You can see that neither the **sort()** function nor subsetting actually changed the original vector. They just outputted a result to the R console. If I wanted to actually save their results, then I would need to assign them to a variable label.

Here's how I would extract and save the top three exam marks:

```r
marks_sorted <- sort(rintro_marks, decreasing = TRUE)

marks_top <- marks_sorted[c(1:3)]

print(marks_top)
```

```
## [1] 92 88 86
```
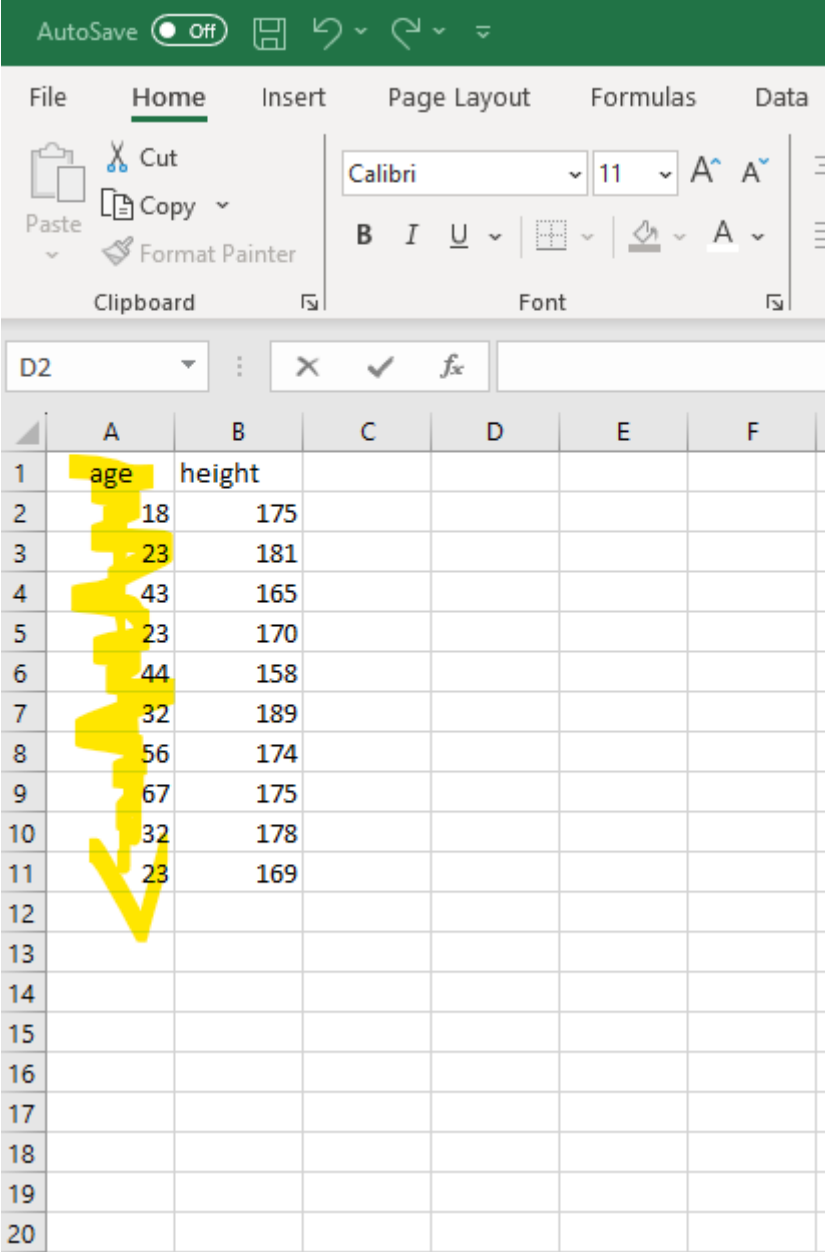
3.6.1.3   Vectors - making it a little less abstract.

You might find the discussion of vectors, elements, and operations very abstract. I certainly did when I was learning R. While the list analogy is helpful, it only works for so long - there's another data structure called **lists** (we'll talk more about it next week). That confused me.

But what helped me understand vectors was the realization that a vector is simply a "line of data." Let's say I was running a study and collected data on participants' age. When I open the Excel file, there will be a column called "age" with all the ages of my participants. That column is a vector of data with the variable label "age."

Creating that vector is the equivalent of creating a column in Excel:

```r
age <- c(18, 23, 43, 23, 44,32, 56, 67, 32, 23)
```

Similarly, rows are also lines of data going horizontally. If I add data to columns in Excel to a dataset, I am creating a new row (line) of data. In R, this is the equivalent of doing this:

```
p11 <- c(30, 175)
```
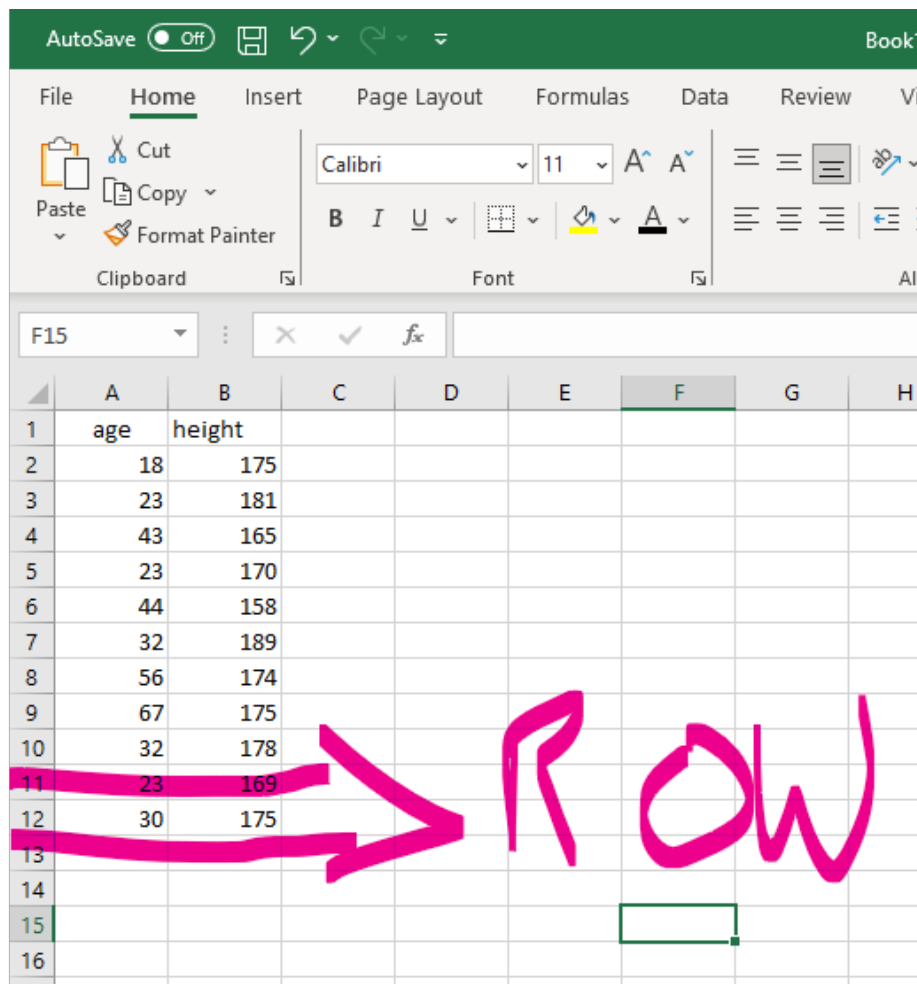

So whenever you think of a vector, just remember that it refers to a line of data that would either be a column or a row.

So what happens when we combine different vectors (columns and rows) together? We create a **data frame**.

3.6.2   Data frames

A data frame is a rectangular data structure that is composed of rows and columns. A data frame in R is like a virtual table or a spreadsheet in Excel:
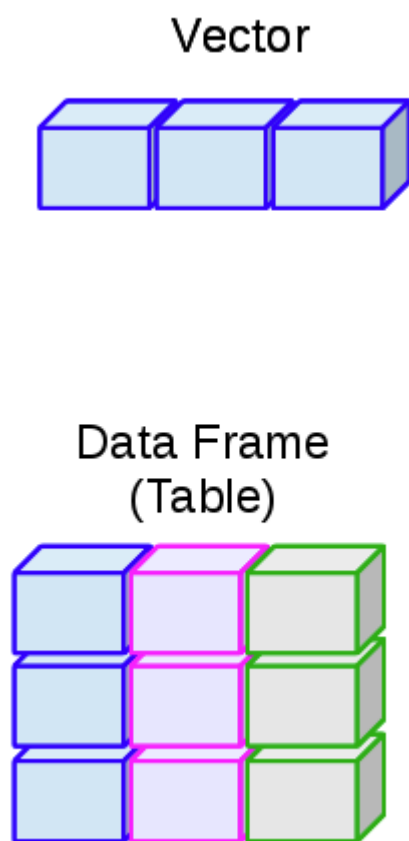
Figure 3.4: The relationship between data frames and vectors. The different colours in the data frame indicate they are composed of independent vectors

Data frames are an excellent way to store and manage data in R because they can store different types of data (e.g., character, numeric, integer) all within the same structure. Let's create such a data frame using the **data.frame()** function:

```r
my_df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"), #a character vector
  Age = c(25L, 30L, 22L), #an integer vector
  Score = c(95.65, 88.12, 75.33) #a numeric vector
)

my_df
```

```
##       Name Age Score
## 1    Alice  25 95.65
## 2      Bob  30 88.12
## 3 Charlie  22 75.33
```

3.6.2.1   Selecting Data from a Data Frame

Once you have created or imported a data frame, you'll often need to access it and perform various tasks and analyses. Let's explore how to access data within a data frame effectively.

3.6.2.1.1   **Selecting Columns**   Columns in a data frame represent different variables or attributes of your data. Often in data analysis, we want to select a specific column and then perform analyses on it. So how can we individually select columns? Well, in a data frame, every column has a name, similar to how each column in an Excel spreadsheet has a header. These column names enable you to access and manipulate specific columns or variables within your data frame.

We select columns based on their names via two tools:

The **$** Notation: You can use a dollar sign ($) followed by the column name to select **an individual column** in a data frame. For example, let's select the **Name** column in the **my_df** data frame:

```r
my_df$Name
```

```
## [1] "Alice"   "Bob"     "Charlie"
```

Square Brackets []: This is a similar approach to accessing elements from a vector. Inside the brackets, you can specify both the row and columns

that you want to extract. The syntax for selecting rows and columns is: **the dataframe[the rows we want, the columns we want]**.

So if we wanted to access the "Age" column of **my_df**, we could run the following code:

```
my_df[, "Age"]
```

```
## [1] 25 30 22
```

You'll notice that we left the "rows" part empty in the square brackets. This tells R "keep all the rows for this column."

We can also use this approach to access multiple columns using the **c()** function:

```
my_df[, c("Age", "Score")]
```

```
##   Age Score
## 1  25 95.65
## 2  30 88.12
## 3  22 75.33
```

3.6.2.1.2 Selecting Rows Rows in a data frame represent individual observations or records. You can access rows using indexing, specifying the row number you want to retrieve, following the syntax: **the dataframe[the rows we want, the columns we want]**.

To get the first row of your data frame (**my_df**), you can type the following:

```
my_df[1, ]
```

```
##    Name Age Score
## 1 Alice  25 95.65
```

This time I left the columns part blank; this tells R "please keep all the columns for each row."

To access the third row:

```
my_df[3, ]
```

```
##      Name Age Score
## 3 Charlie  22 75.33
```

If you want multiple rows, you can use the **c()** function to select multiple rows. Let's select the 1st and 3rd rows:

```
my_df[c(1, 3), ]
```

```
##      Name Age Score
## 1   Alice  25 95.65
## 3 Charlie  22 75.33
```

If you wanted to select a range of rows, you can use the : operator:

```
my_df[2:4, ]
```

```
##        Name Age Score
## 2       Bob  30 88.12
## 3   Charlie  22 75.33
## NA     <NA>  NA    NA
```

These methods allow you to extract specific rows or subsets of rows from your data frame.

3.6.2.1.3 Selecting Rows and Columns We can also select both rows and columns using **[]** and our syntax: **the dataframe[the rows we want, the columns we want]**.

For example, we could select the first and third rows for the **Age** and **Score** columns:

```
my_df[c(1,3), c("Age", "Score")]
```

```
##   Age Score
## 1  25 95.65
## 3  22 75.33
```

Similar to when we indexed vectors, this won't change the underlying data frame. To do that, we would need to assign the selection to a variable:

```
my_df2 <- my_df[c(1,3), c("Age", "Score")]

my_df2
```

```
##   Age Score
## 1  25 95.65
## 3  22 75.33
```

3.6.2.2 Adding Data to your Data Frame

3.6.2.2.1 Adding Columns You may often need to add new information to your data frame. For example, we might be interested in investigating the effect of **Gender** on the **Score** variable. The syntax for creating a new data frame is very straightforward:

```
existing_df$NewColumn <- c(Value1, Value2, Value3)
```

Using this syntax, let's add a **Gender** column to our **my_df** dataframe:.

```
my_df$Gender <- c("Female", "Non-binary", "Male")

#let's see if we have successfully added a new column in

my_df
```

```
##      Name Age Score     Gender
## 1   Alice  25 95.65     Female
## 2     Bob  30 88.12 Non-binary
## 3 Charlie  22 75.33       Male
```

Let's say I noticed I mixed up the genders, and that Bob is Male and Charlie is Non-Binary. Just like we can rewrite a variable, we can also rewrite a column using this approach:

```
my_df$Gender <- c("Female", "Male", "Non-binary")

#let's see if we have successfully rewritten the Gender Column

my_df
```

```
##      Name Age Score     Gender
## 1   Alice  25 95.65     Female
## 2     Bob  30 88.12       Male
## 3 Charlie  22 75.33 Non-binary
```

3.6.2.2.2 Adding Rows What about if we recruited more participants and wanted to add them to our data frame (it is pretty small at the moment!)? This is slightly more complicated, especially when we are dealing with data frames where each column (vector) is of a different data type.

What we need to do is actually create a new data frame that has the same columns as our original data frame. This new data frame will contain the new row(s) we want to add.

```r
new_row <- data.frame(Name = "John", Age = 30, Score = 77.34, Gender = "Male")
```

Then we can use the **rbind()** function to add the new row to your original data frame. **rbind** takes in two data frames and combines them together. The syntax is as follows:

```r
my_df <- rbind(my_df, new_row)

my_df
```

```
##        Name Age Score      Gender
## 1    Alice  25 95.65      Female
## 2      Bob  30 88.12        Male
## 3 Charlie  22 75.33 Non-binary
## 4     John  30 77.34        Male
```

3.6.3 Exercises

1. Create one vector of each data type:

   1. Create a character vector called `friends` with the name of 3 of your friends.

   2. Create an integer vector called `years` that describes the amount of years you have been friends (if it's less than 1 year, put 1).

   3. Create a numeric vector called `extra` with their extraversion scores (out of 5).

   4. Create a logical vector called `galway` that describes whether they live (`TRUE`) or don't live (`FALSE`) in Galway.

   5. Once you have created each vector, check whether it is the correct data type using the `class()` function.

2. Index the 2th, 4th, and 6th element for each of the following vectors.

   ```r
   vect1 <- c("Not this", "This", "Not This", "This", "Not This", "This")

   vect2 <- c(0, 1, 0, 1, 0, 1)

   vect3 <- c("FALSE", "TRUE", "FALSE", "TRUE", "FALSE")
   ```

3. How could we extract and save the bottom 3 results from the `rintro_marks`vector? Bonus Points: Calculate the mean of both the top 3 marks and bottom 3 marks.

4. Write code that adds a column to the `my_df` data frame called `Nationality`. The values for the column should be `"Irish"`, `"American"`, `"English"`, `"Irish"`.

5. Check whether that Nationality column has been successfully added by using the '$' notation. The output should look like this.

```
## [1] "English"  "American" "Irish"    "Irish"
```

5. What code could you write that would take the 'my_df' data frame and give you this output?

```
##      Name Age Nationality
## 1   Alice  25     English
## 3 Charlie  22        Irish
```

6. Write code that adds a row to the `my_df` data frame with your information for each of the columns (e.g., my data would be: `"Ryan"`, 30L, 100, `"Male"`). The `score` variable is a fake exam, so give yourself whatever score you want!

3.7 Summary

That concludes this session. Well done, we did a lot of work today. We learned more about the relationship between the console and the script and how we need to be precise when writing commands. We introduced the different types of data that R stores and how those data types can be stored in single lines of data in vectors or combined together in a table in a **data frame**.

Don't feel like you need to have mastered or even remember all the material that we covered today. Even though these concepts are labeled as "basic," that does not mean they are intuitive. It will take time for them to sink in, and that's normal. We'll drill these concepts a bit further next week. We'll also learn how to import **data frames**, which will set us up nicely for working with the type of data sets we see in Psychological research.

3.8 Glossary

This glossary defines key terms introduced in Chapter 3.

Term	Definition
Assignment	The process of assigning a value to a variable using the assignment operator (`<-` or `=`).
Character	A data type representing text or strings of characters.
Data Frame	A two-dimensional data structure in R that resembles a table with rows and columns. It can store mixed data types.
Data Type	The classification of data values into categories, such as numeric, logical, integer, or character.
Element	An individual item or value within a data structure, such as a character in a vector.
Index	A numerical position or identifier used to access elements within a vector or other data structures.
Indexing	The process of selecting specific elements from a data structure using their index values.
Integer	A data type representing whole numbers without decimals.
Logical	A data type representing binary values (TRUE or FALSE), often used for conditions and logical operations.
Numeric	A data type representing numeric values, including real numbers and decimals.
Object	A fundamental data structure in R that can store data or values. Objects can include vectors, data frames, and more.
Subsetting	The technique of selecting a subset of elements from a data structure, such as a vector or data frame, based on specific criteria.
Variable	A named storage location in R that holds data or values. It can represent different types of information.
Vector	A one-dimensional data structure in R that can hold multiple elements of the same data type.

3.9   Variable Name Table

Rule	Type	Incorrect Example	Correct Example
Variable names can only contain uppercase alphabetic characters A-Z, lowercase a-z, numeric characters 0-9, periods ., and underscores _.	Strict	1st_name	first_name
Variable names must begin with a letter or a period.	Strict	__1stname	.firstname
Avoid using spaces in variable names.	Strict	my name	my_name
Variable names are case-sensitive.	Strict	my_name == my_Name	my_Name == my_Name
Variable names cannot include special words reserved by R.	Strict	print	to_print
Choose informative variable names that clearly describe the information they represent.	Recommended	money	income
Opt for short variable names when possible.	Recommended	date_of_birth	dob
Prioritize clarity over brevity.	Recommended	tem	total_exam_marks
Avoid starting variable names with a capital letter.	Recommended	FirstName	firstName
Choose a consistent naming style and stick to it.	Recommended	myName, last_Name	my_name, last_name or myName and lastName

Chapter 4

R Programming (Part II)

Today, we are going to build upon the foundational concepts introduced last week and delve deeper into the world of R programming.

By the end of this session, you should be capable of the following:

- Understanding the logic of functions, including how and why they are created.

- Creating the "factor" data type and the list data structure.

- Being capable of enhancing your RStudio experience by installing and loading packages.

- Importing a dataset into R using both code and button-click interfaces from CSV and SPSS files.

- Exporting data to CSV and SPSS files.

4.1 Functions

In the previous two sessions, we have used several functions including: `print()`, `head()`, `View()`, `mean()`, `sd()`, `summary()`, `aggregate()`, `plot()`, `pdf()`, `t.test()`, `class()`, and `c()`. Each of these functions has served a particular purpose. All of them have taken in an input object (e.g., a variable, data type, and/or data structure), performed some operation on it, and produced some output.

But we haven't really talked about what functions actually are. I have told you they are similar to verbs in that they are "words" that do things. This makes them sound like some magical words.

You might assume that being good at programming is about learning as many functions as you can, and learning in detail what they can do, so that whenever you face a challenging situation in R, you know what tool to use.

There is some truth to this. You will inevitably learn more functions as you get better and more comfortable with R. This will make you more adept at using them. But what actually predicts becoming good at programming is your ability to understand the logic of functions and how they are created. If you grasp that, you'll be able to learn them quicker, use them more effectively, and even create your own functions.

This final point is critical. You can create your own functions. Let's create our own function to demonstrate what functions actually are.

4.1.1   The Syntax for Creating a Function

Functions are somewhat similar to variables. We create variable names as labels for information. That way when we want to access that information or perform operations on it we can use the variable name rather than recreating the information in R again.

Similarly, functions are like labels for code. We come up with a name for a function (e.g., `mean()`) and we assign code instructions to that function. So when we call a function, we can give it information (e.g., variables), and it will take information and run that code on it. This way we don't have to write out code instructions over and over again - we can just call the function. This increases the scalability of our code.

The syntax for creating a function looks like this:

```
my_function <- function(argument) {
  instruction_1
  instructions_2
  ....
  instruction_n
  return(output)
}
```

What's going on here?

1. First, we created a name, **my_function**, and used the assignment operator **<-** to tell R we are going to be storing some piece of information to that name.

2. Then, we wrote **function()** to tell R that we are going to be creating a function and storing it to our name **my_function**. Inside **function()**, we specified an **argument**, which is just a fancy word for **input**.

3. Inside the curly brackets **{}**, we write the code for **my_function**. This code comprises instructions (i.e., operations) that R will execute on our argument (i.e., input). We could have 1 instruction here, or we could have several hundred lines of instructions. That depends on the complexity of the function we are creating.

4. We want the function to provide us with some output information. To ensure that it does that, we tell R to **return()** the information (**output**) that we want.

4.1.2 Creating a Simple Function (1-Argument)

It may come as a shock to you to learn that I am not much of a chef. One of the reasons I'm not a chef is my irritation with reading recipes that include instructions like "1 cup," "10 ounces," or "17 eagle feet," or require preheating the oven to "1000 degrees Fahrenheit." What I could really use is a function that would help me convert those values automatically. Let's take the cup example. Let's create a function that will take in the number of cups we need and convert that to grams.

To do this, let's create a function called **cups_to_grams** (note: the naming conventions for functions are similar to the naming conventions for variables. The main rule is that your function name should describe the action it is carrying out.)

```r
cups_to_grams <- function(cups) {

}
```

Inside the **function()**, I have given it the argument **cups**. In this scenario, **cups** acts as a placeholder variable. Inside the function, we are going to write instructions on what to do with that variable. But we have not yet defined what that variable is yet. We do that when we use the function. So don't worry about that for now.

Now inside our function (i.e., inside the **{}**), we need to write instructions to enable R to convert cups to grams. When I googled this, several different answers popped up. But I am just going to follow the first website I found which said: "According to the metric system, there are 250 grams in 1 cup."

Let's write that instruction inside our function. We are going to save the result of that calculation to a variable called **grams**.

```r
cups_to_grams <- function(cups) {
  grams <- cups * 250
}
```

Now we are nearly done. But if we want R to provide us with the result of this function, we need to ask it to **return()** it to us. We can do that easily by:

```r
cups_to_grams <- function(cups) {
  grams <- cups * 250
  return(grams)
}
```

There we have it, we have created our first function! Now let's see if it works. In programming lingo, we say that we **call** a function when we use it. To call **cups_to_grams** it is the same process as the other functions we have used, we type out the name and then we insert our input inside the parentheses.

```r
cups_to_grams(cups = 1)
```

```
## [1] 250
```

It works! We can see here what I mean that **cups** is a placeholder variable. We want our function to be generalizable, so we don't tell it ahead of time what **cups** equals. All it knows is that it will receive some information that will equate to cups, and then it will multiply that information by 250.

This enables us to call our function several times with several different values.

```r
cups_to_grams(cups = 4)
```

```
## [1] 1000
```

```r
cups_to_grams(cups = 2)
```

```
## [1] 500
```

```r
cups_to_grams(cups = 1.5)
```

```
## [1] 375
```

```r
cups_to_grams(cups = 5L)
```

```
## [1] 1250
```

We can also define what **cups** is outside of the function.

```
cups = 2
cups_to_grams(cups)
```

```
## [1] 500
```

This is an example of a 1-argument function, as it only takes in 1 input. But we can also create functions that have multiple arguments.

4.1.3 Creating a Multi-Argument Function

You might have noticed previously that sometimes we put additional information inside functions, like `paired = TRUE` in `t.test()`, or `descending = FALSE` in `sort()`. This additional information represents other arguments that we can insert inside a function.

The process for creating a multi-argument function is the same as for a single-argument function. Let's create a function called `calculate_z_score` that calculates the z-score of a value.

```
calculate_z_score <- function(x, mean_val, sd_val) {
  # Calculate the z-score
  z_score <- (x - mean_val) / sd_val

  # Return the z-score
  return(z_score)
}
```

In this function:

- `x` is the value for which we want to calculate the z-score.

- `mean_val` is the mean score of the variable within our dataset.

- `sd_val` is the standard deviation of the variable within our dataset.

Inside the function, we calculate the z-score using the formula `(x - mean_val) / sd_val`. The calculated z-score is returned as the output of the function.

Just like before, none of the placeholder arguments (`x`, `mean_val`, and `sd_val`) are defined beforehand. We will define them in our script or when we call our function.

To test this function, let's use an example of an IQ score since we know the population mean (100) and standard deviation (15). Let's see what the z-score is for someone with an IQ of 130.

```r
calculate_z_score(x = 130, mean_val = 100, sd_val = 15)
```

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

Just as we would expect, a person with an IQ of 130 is 2 standard deviations away from the mean. This shows that our function is working as expected.

What if we had a vector of IQ scores? Could we use our function to calculate the z-score of each element in our vector? Absolutely!

```r
calculate_z_score(x = c(100, 130, 85), mean_val = 100, sd_val = 15)
```

```
## [1]  0  2 -1
```

Sticking with this example, let's say we had a data frame with participant IDs, their age, and their IQ scores. We could feed the vector **iq_scores** into our **calculate_z_score** function, calculate their z-scores, and create a column based on those scores.

```r
#first let's make that data frame

iq_df <- data.frame(
  ID = c(1, 2, 3, 4, 5, 6, 7, 8),
  age = c(22, 30, 41, 45, 18, 21, 23, 45),
  iq = c(100, 123, 111, 130, 90, 102, 88, 109)
)

#now let's feed that IQ vector into our function and save it to a variable

iq_z_scores <-  calculate_z_score(x = iq_df$iq, mean_val = 100, sd_val = 15) #this wil

#if we want to add that column, we use the syntax

#dataframe$newColumnName <- #new_vector

iq_df$iq_z_scores <- iq_z_scores

#now let's check our data frame

head(iq_df)
```

```
##   ID age  iq iq_z_scores
## 1  1  22 100   0.0000000
## 2  2  30 123   1.5333333
## 3  3  41 111   0.7333333
## 4  4  45 130   2.0000000
## 5  5  18  90  -0.6666667
## 6  6  21 102   0.1333333
```

While `calculate_z_score` is pretty handy, it's not perfect. It requires us to calculate the mean and standard deviation functions separately and then feed that into our function. That's okay if we are dealing with variables that have a known mean and standard deviation. Outside of those examples, we would need to do some extra work. But one of the virtues about functions is that it enables us to be lazy - we want to write functions that will automate boring tasks for us. So how could we improve this function? Well luckily, we can include functions inside functions.

4.1.4 Functions inside Functions

Let's say that we add a column in our `iq_df` dataframe that contains participants' mean scores on Beck's Depression Inventory. We'll call this column `total_depression_beck`.

```r
iq_df$total_depression_beck <- c(32, 36, 34, 46,
                                 30, 53, 40, 15) #adds the mean_depression beck vector to our data

head(iq_df) #check to see if it was added correctly.
```

```
##   ID age  iq iq_z_scores total_depression_beck
## 1  1  22 100   0.0000000                    32
## 2  2  30 123   1.5333333                    36
## 3  3  41 111   0.7333333                    34
## 4  4  45 130   2.0000000                    46
## 5  5  18  90  -0.6666667                    30
## 6  6  21 102   0.1333333                    53
```

Since we do not know the mean and standard deviation of the Beck Inventory, we will need to calculate them using the `mean()` and `sd()` functions. Luckily, we can use those functions within `calculate_z_score` to enable this for us. Let's add this to our function and call it.

```r
calculate_z_score <- function(x) {
  # Calculate the z-score
```

```
  z_score <- (x - mean_val) / sd_val
  mean_val <- mean(x)
  sd_val <- sd(x)

  # Return the z-score
  return(z_score)
}

calculate_z_score(x = c(100, 90, 110))
```

```
## Error in calculate_z_score(x = c(100, 90, 110)): object 'mean_val' not found
```

Uh-oh! Why is it telling us that the object **mean_val** was not found? The reason for this is that the order of your code within a function matters. The order of your code is the order in which R will compute that instruction. Currently, I have asked R to compute **z_score** before defining what **mean_val** or **sd_val** are.

So when R sees **mean_val**, it looks everywhere for what that value could mean, doesn't find anything, and then panics and stops working. Again, humans have a theory of mind, so we would assume that we could provide this information. But R needs to do everything literally step-by-step.

To rectify this, we just need to fix the order of our instructions inside R.

```
calculate_z_score <- function(x, mean_val, sd_val) {
  #compute mean_val, and sd_val first
  mean_val <- mean(x)
  sd_val <- sd(x)

  z_score <- (x - mean_val) / sd_val

  # Return the z-score
  return(z_score)
}

calculate_z_score(x = iq_df$total_depression_beck)
```

```
## [1] -0.33044366  0.02202958 -0.15420704  0.90321267 -0.50668028  1.52004083
## [7]  0.37450281 -1.82845491
```

Wahey, it worked! Try to add the z-scores for the **total_depression_beck** to the dataframe yourself (look at the end of the previous subsection for advice on how if you are stuck).

4.1.5 Returning Multiple Objects from a Function

What if we wanted to return not only the `z_score` variable from calculate_z_score, but also `mean_val` and `sd_val` as well?

Luckily, we can also tell our functions to return multiple different objects at the same time. We can do this by using lists.

We will discuss lists in more detail later in this chapter. For now, all you need to know is that lists are versatile data structures in R that can hold elements of different types. We can create a list within a function, populate it with the values we want to return, and then return the list itself.

We can create a variable within our function that is a list containing all the information we want to return. But since this changes the nature of the function, we are going to change its name to: 'calculate_mean_sd_z'

```r
calculate_mean_sd_z <- function(x, mean_val, sd_val) {
  #compute mean_val, and sd_val first
  mean_val <- mean(x)
  sd_val <- sd(x)

  z_score <- (x - mean_val) / sd_val

  results <- list(z_score, mean_val, sd_val)

  # Return the z-score
  return(results)
}

calculate_mean_sd_z(iq_df$total_depression_beck)
```

```
## [[1]]
## [1] -0.33044366  0.02202958 -0.15420704  0.90321267 -0.50668028  1.52004083
## [7]  0.37450281 -1.82845491
##
## [[2]]
## [1] 35.75
##
## [[3]]
## [1] 11.34838
```

This produces the results that we want, but the output leaves a lot to be desired. If someone else was calling our function, but was not aware of the instructions inside it, they might not know what each value from the output corresponds to. We can correct this by using the following syntax inside the list to label each value: `name_of_value = value`

```r
calculate_mean_sd_z <- function(x, mean_val, sd_val) {
  #compute mean_val, and sd_val first
  mean_val <- mean(x)
  sd_val <- sd(x)

  z_score <- (x - mean_val) / sd_val

  results <- list(z = z_score, mean = mean_val, sd = sd_val)

  # Return the z-score
  return(results)
}

calculate_mean_sd_z(iq_df$total_depression_beck)
```

```
## $z
## [1] -0.33044366  0.02202958 -0.15420704  0.90321267 -0.50668028  1.52004083
## [7]  0.37450281 -1.82845491
##
## $mean
## [1] 35.75
##
## $sd
## [1] 11.34838
```

Now if we wanted to extract certain features from the function, we can use the
'$' operator.

```r
scores <-  calculate_mean_sd_z(iq_df$total_depression_beck)

scores$mean
```

```
## [1] 35.75
```

```r
scores$sd
```

```
## [1] 11.34838
```

```r
scores$z
```

```
## [1] -0.33044366  0.02202958 -0.15420704  0.90321267 -0.50668028  1.52004083
## [7]  0.37450281 -1.82845491
```

4.1.6 Some Important Features about Functions

There are some important features about functions that you should know. Namely, the difference between Global and Local Variables and the ability to set and override Default Arguments.

4.1.6.1 Global vs Local Variables

It is important to note that R treats variables you define in a function differently than variables you define outside of a function. Any variable you define within a function will only exist within the scope of that function. Variables defined in a function are called local variables whereas variables defined outside of a function are called global variables.

```r
num1 <- 20 #this is a global variable

local_global <- function() {
  num1 <- 10#this is a local variable
  print(num1)  # This will print 10
}

local_global()
```

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

```r
print(num1)  # Error: object 'local_var' not found
```

```
## [1] 20
```

We start this code chunk by assigning the value 20 to the variable `num1`. This is a global variable, it will exist across our R environment unless we change it to a different value.

Within the `local_global()` function, we create another variable called `num1` and assign it the value of 10. But this is an example of a local variable, as it only exists inside of our function. When we run our function, it will print out 10, but the function will not change the global value of the variable.

4.1.6.2 Default Arguments

In R, functions can have default arguments, which are pre-defined values assigned to arguments in the function definition. Default arguments allow functions to be called with fewer arguments than specified in the function definition, as the default values are used when the arguments are not explicitly provided.

4.1.6.3   Syntax for Default Arguments

The syntax for defining default arguments in R functions is straightforward. When defining the function, you can assign default values to specific arguments using the `argument = default_value` format.

Imagine I wanted to write a function that greeted someone. I could write the following function, `greet`:

```r
# Function with default argument
greet <- function(name = "World") {
  print(paste("Hello,", name))
}
```

Within this function, I set the default value for the argument name as "World". So if I were to call the function but not specify the argument, the following would happen:

```r
# Calling the function without providing arguments
greet()
```

```
## [1] "Hello, World"
```

However, we can also override the default value of a function.

```r
greet(name = "Ryan") #please feel free to type in your own name
```

```
## [1] "Hello, Ryan"
```

Having default values in a function enables them to be called with fewer arguments, making code more readable. But since default values can be overridden, it also provides them with a degree of combustibility.

4.1.7   Exercises

1. Create a function named `fahrenheit_to_celsius` that converts a temperature from Fahrenheit to Celsius.

   Instructions:

   1. Define the function `fahrenheit_to_celsius` with an argument named `fahrenheit`.
   2. Inside the function, create a variable named `celsius` to store the converted temperature.

3. Calculate the Celsius temperature using the formula: **(fahrenheit - 32) / 1.8**.

4. Return the **celsius** variable.

2. Create a function called **calculate_discount** that calculates the discounted price of an item.

   1. Define the function **calculate_discount** with two arguments: **price** and **discount_percent**.

   2. Inside the function, create a variable named **discount_price** to store the discounted price.

   3. Calculate the discounted price using the formula: **price * (1 - discount_percent)**.

   4. Return the **discount_price**.

3. Add the z-scores for the Beck scale back into the iq_df data frame.

   Instructions:

   1. Use the **calculate_z_score** function you created earlier to calculate the z-scores for the Beck scale.

   2. Add the calculated z-scores to the **iq_df** data frame as a new column.

4. Look up help on the head() function - what is the default value for the number (n) of rows it prints argument?

   Instructions:

   1. Use the **?head** command to access the documentation for the **head()** function.

   2. Identify the default value for the **n** argument in the documentation.

   3. Call the head() function on the sleep dataframe, but override the default argument with a different number

4.2 The Factor Data Type

In Chapter 3, we introduced the four fundamental data types in R: character, integer, numeric, and logical. We learned that these data types can constitute vectors, which can then be aggregated in data frames. These data types cover a significant proportion of the information we encounter in psychological research. In terms of NOIR data types in Psychology, nominal data can be represented through the character type, ordinal data can be represented through character or integer/numeric data types, and scale data (interval and ratio) can be represented through the integer/numeric data type. Additionally, the logical data type can handle either/or cases.

However, there's an additional consideration. When dealing with datasets common in psychological research, particularly experimental or differential research, we often encounter columns consisting of categorical data representing our independent variable(s). The responses in these columns signify differences in categories for each participant (e.g., whether they were assigned to the control or experimental group, whether they belong to different categories across some domain). In software like SPSS, we typically label this categorical data using normal language (e.g., "control" or "experimental") or numerically (1 = "control", 2 = "experimental"). While we could categorize it as character data or numerically/integer data in R, this approach doesn't capture the fact that the data in this column represents something distinct - namely, that it is a ***factor*** used to understand differences across other variables.

Fortunately, R offers a data type called ***factors*** to address this need.

4.2.1   Creating a Factor

Let's say we run an experimental study investigating the effect of caffeine on sleep duration. Participants are randomly assigned to three experimental conditions: `No Caffeine`, `Low Caffeine (200 mg)`, and `High Caffeine (400 mg)`. Our data frame might resemble the following, where `1` = `No Caffeine`, `2 = Low Caffeine`, and `3 = High Caffeine`:

```r
caffeine_df <- data.frame(
  id = c(1:12),
  sleep_duration = c(7.89, 8.37, 7.46, 5.86, 5.25, 7.23, 6.05, 5.78, 6.77, 2.13, 5.78,
  group = c(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
)

head(caffeine_df)
```

```
##   id sleep_duration group
## 1  1           7.89     1
## 2  2           8.37     2
## 3  3           7.46     3
## 4  4           5.86     1
## 5  5           5.25     2
## 6  6           7.23     3
```

Although the `group` column is currently stored as numeric data, it actually represents categorical information. It wouldn't make sense to compute the mean, median, or standard deviation for this column. However, since it's numeric data, R allows us to do so.

```
mean(caffeine_df$group)
```

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

While this might not cause immediate problems, it could lead to issues later when conducting inferential statistical tests like ANOVAs or regressions, where R expects a factor data type.

To address this, we can use the **factor** function to convert the group column:

```
factor(caffeine_df$group)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3
## Levels: 1 2 3
```

Now, R recognizes that each distinct value in the column represents a different level of our independent variable. If we attempt to calculate the mean now, it will result in an erro:

```
mean(caffeine_df$group)
```

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

Whoops! What's wrong here? Well, when we called the **factor()** function, we never reassigned that information back to the 'group' vector in our **caffeine_df** data frame. So while R ran our command, it did not save it. To fix this, we just follow the syntax for creating a new column we learned last week **data.frame$newcolumn <- vector**.

```
caffeine_df$group <- factor(caffeine_df$group)
```

Now, attempting to compute the mean will result in an error, as expected:

```
mean(caffeine_df$group)
```

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

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

But I am not completely satisfied yet. Although right now we can remember that 1 = No Caffeine, 2 = Low Caffeine, and 3 = High Caffeine, we might forget that information in six months' time if we return to our dataset.

Luckily, we can label the levels of our factor through the **levels()** function.

```r
levels(caffeine_df$group) <- c("No Caffeine", "Low Caffeine", "High Caffeine")

print(caffeine_df$group)
```

```
##  [1] No Caffeine   Low Caffeine  High Caffeine No Caffeine   Low Caffeine
##  [6] High Caffeine No Caffeine   Low Caffeine  High Caffeine No Caffeine
## [11] Low Caffeine  High Caffeine
## Levels: No Caffeine Low Caffeine High Caffeine
```

That's better!

4.2.2 Using Factors to Sort Character Data

Factors also enable us to sort character data in an ordered manner that isn't alphabetical. For example, consider a vector called **degree** that denotes participants' highest level of education completed:

```r
degree <- c("PhD", "Secondary School", "Masters", "Bachelors", "Bachelors", "Masters",
```

Using the **table()** function, we can count the number of participants per category:

```r
count <- table(degree)
count
```

```
## degree
##        Bachelors          Masters              PhD Secondary School
##                3                2                1                4
```

And we can use the **barplot()** function to visualize those counts.

```r
barplot(count)
```
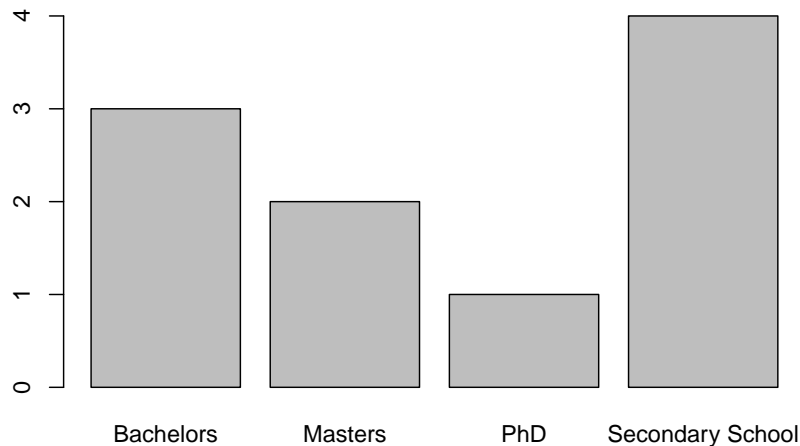
Now that gives us the information we need, but it's not ordered in an intuitive manner. Ideally, we would want the order of the bar plots to match the hierarchical order of the data, so that it would be: "Secondary School", "Bachelors", "Masters", and then "PhD". However, unless you specify the order of your levels, R will specify their order alphabetically.

There is an argument called `levels` in the `factor` function that we can use to rectify this. In the levels argument, we specify the order of the levels.

```
degree_ordered <- factor(degree, levels = c("Secondary School", "Bachelors", "Masters", "PhD"))
degree_ordered
```

```
##  [1] PhD             Secondary School Masters          Bachelors
##  [5] Bachelors        Masters          Bachelors        Secondary School
##  [9] Secondary School Secondary School
## Levels: Secondary School Bachelors Masters PhD
```

That's more like it. Now we can call the `table()` and `barplot()` functions again to visualise the number of participants per group.

```
count_ordered <- table(degree_ordered)
```

```
barplot(count_ordered)
```
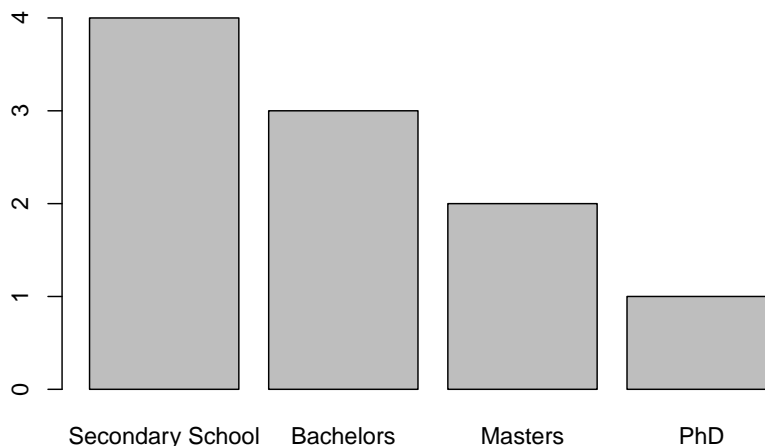
As you can see, our data looks much cleaner and more intuitive.

There is a lot more we can do with factors, but this covers the main points. Now let's talk about the last data type/structure we are going to be using in this course, which is the `list` data structure.

4.2.3   Exercises

Exercise 1 - Convert Numeric Data to Factor:

- Create a vector called **`personality`** containing personality types: "Introverted", "Extroverted", "Introverted", "Introverted", "Extroverted", "Extroverted".

- Convert the **`personality`** vector to a factor and store it in a new variable called **`personality_factor`**.

- Display the levels of the **`personality_factor`** variable.

Exercise 2 - Count and Visualize Factor Levels:

- Using the created **`personality`** vector from Exercise 1, create a table to count the number of occurrences of each Personality trait.

- Visualize the counts using a bar plot to show the distribution of genders.

Exercise 3 - Modify Factor Levels:

```r
# Treatment conditions vector
treatment <- c("Control", "Placebo", "Therapy", "Medication", "Therapy", "Placebo", "Control", "M
```

- Reorder the levels of the **treatment** vector so that "Therapy" becomes the first level, followed by "Control", "Placebo", and "Medication".

- Print the modified levels of the **treatment** variable to verify the order.

Exercise 4 - Factor Conversion with Levels:

- Create a vector called **coping_strategy** with the following values: "Problem-focused", "Emotion-focused", "Avoidant", "Problem-focused", "Emotion-focused", "Avoidant", "Problem-focused".

- Convert the **coping_strategy** vector to a factor with levels specified as "Problem-focused", "Emotion-focused", "Avoidant", and store it in a new variable called **coping_strategy_factor**.

- Display the levels of the **coping_strategy_factor** variable.

4.3 The List Data Structure

The two data structures we have discussed so far, vectors and data frames, are excellent ways to store data. However, each data structure has its limitations. Both vectors and data frames only allow us to store one object of data at a time.

What do I mean by one object of data? Well, when we create a vector, we are only storing information for one single vector in R. While we can store multiple vectors in one place in R by combining them into a data frame, this is only possible if each vector has the same number of data points. Based on what I have taught you so far, there is no way to store two independent and separate vectors in the same place in R.

Similarly, when I create a data frame, I am only storing information for one data frame. But sometimes we would want to keep multiple data frames saved in a similar location, similar to an Excel file that has multiple worksheets saved onto it.

Finally, what if I wanted to store a vector and a data frame together? For example, imagine I had a data frame that was a cleaned data set, and I wanted to store a vector with results from the analysis. Based on what I have taught you so far, there is no way to store both a vector and a separate data frame in a single space in R.

That's where the list data structure comes into play.

4.3.1   Understanding Lists

A list in R is a versatile data structure that can hold elements of different types, such as vectors, matrices, data frames, or even other lists. Think of it as a container that can store various objects together, similar to a bag where you can put in items of different shapes and sizes.

4.3.2   Creating Lists

Suppose we're conducting a study where we collect various information about each participant, such as their demographic details, test scores, and responses to questionnaires. We can store this information for each participant in a list. Here's how we can create a list for three participants:

```r
participant1 <- list(
  ID = 1,
  age = 25,
  gender = "Male",
  test_scores = c(80, 75, 90),
  questionnaire_responses = c("Agree", "Disagree", "Neutral", "Agree")
)

participant2 <- list(
  ID = 2,
  age = 30,
  gender = "Female",
  test_scores = c(85, 70, 88),
  questionnaire_responses = c("Neutral", "Agree", "Disagree", "Strongly Disagree")
)

participant3 <- list(
  ID = 3,
  age = 28,
  gender = "Non-binary",
  test_scores = c(78, 82, 85),
  questionnaire_responses = c("Disagree", "Neutral", "Agree", "Neutral")
)
```

The list for each participant is made up of separate vectors with varying lengths (e.g., there is only 1 element in ID, age, and gender, whereas there are 3 elements in test_scores and 4 in questionnaire_responses).

Let's print out the **participant1** list and break down the output:

```
print(participant1)
```

```
## $ID
## [1] 1
##
## $age
## [1] 25
##
## $gender
## [1] "Male"
##
## $test_scores
## [1] 80 75 90
##
## $questionnaire_responses
## [1] "Agree"    "Disagree" "Neutral"  "Agree"
```

When we print out the list, what R does in the console is print out the name of each object and then print out each element in that object. One thing that you might notice is the return of the $ which we use to access elements from vectors or columns from data frames. Luckily, we can also use the $ symbol to access objects and their elements from a list. So I wanted to extract `test_scores` from a list, I could type the following code:

```
participant1$test_scores
```

```
## [1] 80 75 90
```

I could also do this numerically using the [] notation we used when accessing vectors.

```
participant1[4]
```

```
## $test_scores
## [1] 80 75 90
```

Because `test_scores` is the fourth object within the list (the first three are ID, age, and gender), `test_scores[4]` is needed to extract it.

Regardless of which way you extract data, doesn't the output should look familiar? It is essential the same output when we print out a vector. That's not accidental, because we have in fact extracted a vector!

So what could we do if wanted to access the 1st and 3rd element from this vector? We just follow the same convention that we used last week `vectorname[elementswewant]`

```
participant1$test_scores[c(1, 3)]
```

```
## [1] 80 90
```

This illustrates a key point about lists. Once you first access an object (e.g., vector or dataframe) within that list, then you can use the specific indexing criteria for accessing elements within that object. For example, let's look at a list with a vector and a data frame.

```
df_v <- list(
  iq_df = iq_df,
  p_values = c(.001, .10, .05)
)

print(df_v)
```

```
## $iq_df
##   ID age  iq iq_z_scores total_depression_beck
## 1  1  22 100   0.0000000                     32
## 2  2  30 123   1.5333333                     36
## 3  3  41 111   0.7333333                     34
## 4  4  45 130   2.0000000                     46
## 5  5  18  90  -0.6666667                     30
## 6  6  21 102   0.1333333                     53
## 7  7  23  88  -0.8000000                     40
## 8  8  45 109   0.6000000                     15
##
## $p_values
## [1] 0.001 0.100 0.050
```

We can see that the first object in our list is the iq_df data frame, and the second contains (totally made up) p-values. So if wanted to access the `iq` and `total_depression_beck` columns from this data frame, and the first five rows, we can use the same subsetting techniques we learned last week: `dataframe[rows_we_want, columns_we_want]`

```
df_v$iq_df[1:5, c("iq", "total_depression_beck")]
```

```
##    iq total_depression_beck
## 1 100                    32
## 2 123                    36
## 3 111                    34
## 4 130                    46
## 5  90                    30
```

4.3.3   Lists within Lists (Indexing)

What happens if we combine lists together? Well, we can do that by putting lists inside lists. However, if we do that, the output might seem a bit overwhelming at first.

```r
participant_data <- list(participant1, participant2, participant3)

print(participant_data)
```

```
## [[1]]
## [[1]]$ID
## [1] 1
##
## [[1]]$age
## [1] 25
##
## [[1]]$gender
## [1] "Male"
##
## [[1]]$test_scores
## [1] 80 75 90
##
## [[1]]$questionnaire_responses
## [1] "Agree"    "Disagree" "Neutral"  "Agree"
##
##
## [[2]]
## [[2]]$ID
## [1] 2
##
## [[2]]$age
## [1] 30
##
## [[2]]$gender
## [1] "Female"
##
## [[2]]$test_scores
## [1] 85 70 88
##
## [[2]]$questionnaire_responses
## [1] "Neutral"           "Agree"             "Disagree"
## [4] "Strongly Disagree"
##
##
```

```
## [[3]]
## [[3]]$ID
## [1] 3
##
## [[3]]$age
## [1] 28
##
## [[3]]$gender
## [1] "Non-binary"
##
## [[3]]$test_scores
## [1] 78 82 85
##
## [[3]]$questionnaire_responses
## [1] "Disagree" "Neutral"  "Agree"     "Neutral"
```

Right now your eyes might be glazing over and that SPSS icon on your desktop has never looked so good. But just relax, while what you might be seeing here is **ugly**, I promise you it's not complicated.

Let's break down the first part of this output

```
[[1]]
```

This notation **[[1]]** extracts and indicates the first list within our combined list **participant_data** (in this case participant1). Similarly, **[[2]]** would indicate the second list (participant2), and so on. It's similar to how we index elements in vectors.

```
[[1]]$ID #this translates to from participant data, pick participant 1 and then extrac

[[1]]$age #this translates to from participant data, pick participant 1 and then extra

[[1]]$test_scores #this translates to from participant data, pick participant 1 and th

[[1]]$questionnaire_responses #this translates to from participant data, pick particip
```

Basically translates to "participant 1's data for ID, age, test_scores and questionnaire_responses".

There is a way we can make this code output neater. When we are creating a list, we can specify the index term for that list. The syntax for doing is: new index term = current_list

```r
participant_data <- list(p1 = participant1, #new index term = current list
                         p2 = participant2, #new index term = current list
                         p3 = participant3 #new index term = current list
                         )

print(participant_data)
```

```
## $p1
## $p1$ID
## [1] 1
##
## $p1$age
## [1] 25
##
## $p1$gender
## [1] "Male"
##
## $p1$test_scores
## [1] 80 75 90
##
## $p1$questionnaire_responses
## [1] "Agree"    "Disagree" "Neutral"  "Agree"
##
##
## $p2
## $p2$ID
## [1] 2
##
## $p2$age
## [1] 30
##
## $p2$gender
## [1] "Female"
##
## $p2$test_scores
## [1] 85 70 88
##
## $p2$questionnaire_responses
## [1] "Neutral"          "Agree"             "Disagree"
## [4] "Strongly Disagree"
##
##
## $p3
## $p3$ID
## [1] 3
```

```
##
## $p3$age
## [1] 28
##
## $p3$gender
## [1] "Non-binary"
##
## $p3$test_scores
## [1] 78 82 85
##
## $p3$questionnaire_responses
## [1] "Disagree" "Neutral"  "Agree"    "Neutral"
```

4.3.4 Summary

In summary, the list data structure in R is incredibly useful for organizing and managing diverse types of data in psychological research. Whether it's participant information, experimental conditions, or any other heterogeneous data, lists provide a flexible and efficient way to store and access this information.

4.4   R Packages

Throughout this course, we've been exploring the capabilities of base R, which offers a rich set of functions and data structures for data analysis and statistical computing. However, the power of R extends far beyond its base functionality, thanks to its vibrant ecosystem of user-contributed packages.

4.4.1   Understanding R Packages

R packages are collections of R functions, data, and documentation that extend the capabilities of R. These packages are developed and shared by the R community to address various needs in data analysis, visualization, machine learning, and more. By leveraging packages, R users can access a vast array of specialized tools and algorithms without reinventing the wheel.

4.4.2   Installing and Loading R Package

One of the most important things to know about R packages is that you first need to install them on your computer. Once installed, you will not need to install them again.

However, if you want to use a package, then you will need to load it while you are in RStudio. Every time you open RStudio after closing it, you will need to load that package again if you want to use it.

In this way, R packages are like applications on your phone. Once you download Spotify on your phone, then you won't need to install it again. But every time you want to use Spotify, you will need to open the application.

4.4.2.1   Installation

We are going to install three packages called Haven, praise, and fortunes. The praise package provides users with, well, praise. And the fortunes package will spit out statistic and programming quotes at you. Neither of them is particularly useful, other than demonstrating the process of loading packages.

The Haven package enables you to load SPSS data into R.

4.4.2.2   Installation using RStudio Interface

In the Files pane on the bottom right-hand corner of R, you will see a tab called "Packages". Click that tab. You will already see a list of packages that are currently installed. You will see three columns:

- Name (the name of the R package)

- Description (describes what the package does)

- Version (the version of the package currently installed on your computer)

To install a package, click the "Install" button above the "Name" column. Once you click that, you should see a small pop-up window. In the textbox, type in the word Haven. Make sure the box "Install dependencies" is clicked. After that, you can click Install. You should get something like the following output in the console, which looks scary with all the red text, but means it has worked correctly.

```
> install.packages("Haven")
trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-x86_64/contrib/4.3/rio_1.0.1.tgz'
Content type 'application/x-gzip' length 591359 bytes (577 KB)
==================================================
downloaded 577 KB


The downloaded binary packages are in
    /var/folders/h8/8sb24v_x2lg51cg2z7q8fk3w0000gp/T//RtmpvaY1Ue/downloaded_packages
```

4.4.2.3   Installation using Commands

If we want to install packages using the console, you can use the `install.packages()` function followed by the name of the package you wish to install. The syntax would looks like this

```r
install.packages("package name")
```

The important thing here is that whatever goes inside the parentheses is inside quotation marks. Let's use this syntax to install the `praise` and `fortunes` packages.

```r
install.packages(c("praise", "fortunes"))
```

```
trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-x86_64/contrib/4.3/praise_1.0.0
Content type 'application/x-gzip' length 16537 bytes (16 KB)
==================================================
downloaded 16 KB

trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-x86_64/contrib/4.3/fortunes_1.5
Content type 'application/x-gzip' length 208808 bytes (203 KB)
==================================================
downloaded 203 KB


The downloaded binary packages are in
    /var/folders/h8/8sb24v_x2lg51cg2z7q8fk3w0000gp/T//RtmpvaY1Ue/downloaded_packages
```

Again the output is rather scary but the sentences "package 'praise' successfully unpacked and MD5 sums checked" and "package 'fortunes' successfully unpacked and MD5 sums checked" mean that they are successfully installed onto your computer.

4.4.3   Loading Packages

Okay, now to actually use those packages, we will need to load them. Again, I will show you two ways to load packages.

4.4.3.1   Loading using RStudio Interface

Go back to the Packages tab in the File pane. On the left-hand side of the package name, you will see a tick box. If the box is ticked, that means the package is currently loaded. If it is unticked, it is not loaded.

Scroll down to find the package "praise" and load it by ticking the box.
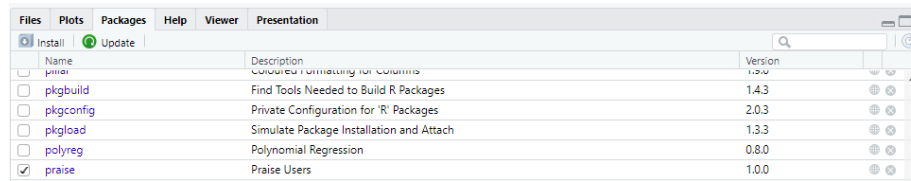
Figure 4.1: Loading Packages through RStudio Interface

You should see something like the following in your R console (don't worry if you get a warning message like mine, or if you don't receive a warning message)

```
> library(praise)
Warning message:
package 'praise' was built under R version 4.3.2
```

4.4.3.2 Loading using the R Console Command

We can use the same syntax from that R console output to load in packages there. To load in the `fortunes` and `Haven` packages, you can type in the following into the console or script one at a time:

```
library(fortunes)
library(haven)
```

```
## Warning: package 'haven' was built under R version 4.3.2
```

There is one significant difference between installing and loading packages through code. When you are installing packages, you can install multiple packages in one command. However, you can only load one package at a time

```
#will work
install.packages(c("package1", "package2", "package3"))


#will not work
library(c("package1", "package2", "package3"))

#the following will work
library(package1)
library(package2)
library(package3)
```

4.4.4   Testing our New Functions

To make sure the following functions are working, run the following code to check:

```
## Warning: package 'praise' was built under R version 4.3.2
```

```
## [1] "You are groovy!"
```

```
##
## I think that the formula language does allow expressions with '/' to represent
## nested factors but I can't check right now as there is a fire in the building
## where my office is located. I prefer to simply write nested factors as factor1
## + factor1:factor2.
##    -- Douglas Bates
##       R-SIG-Mixed-Models (February 2013)
```

```r
praise() #everytime you run this line of code it gives you a different line of praise
#so don't be worried if your result is different than mine

fortune() #this will print out something so incredibly nerdy
```

4.4.5   Error Loading Packages

If you ever encounter the following error when trying to load a package:

```r
library(madeuppackage)
```

```
## Error in library(madeuppackage): there is no package called 'madeuppackage'
```

This means that you have either made a typo in writing the name of the package or you have not installed the package. You need to install packages before R will be able to load them.

4.4.6   Package Conventions if Using Code

There are important rules to follow when writing code to install and load packages in R.

Firstly, any packages you install and load onto R should be placed at the top of the R script you are working on. This way, anyone following your analysis can easily spot the packages they will need.

Secondly, you should type the command **install.packages("package_name")** in the console rather than the script. If someone downloads your script and accidentally runs it, it will automatically install the packages on their computer. We want to avoid this as it might interfere with other aspects of their operating system (this is rare, but it's better to be cautious).

Thirdly, if you do include the command **install.packages()** in the script, make sure it is commented out. This way, if it is accidentally run, R won't execute that command.
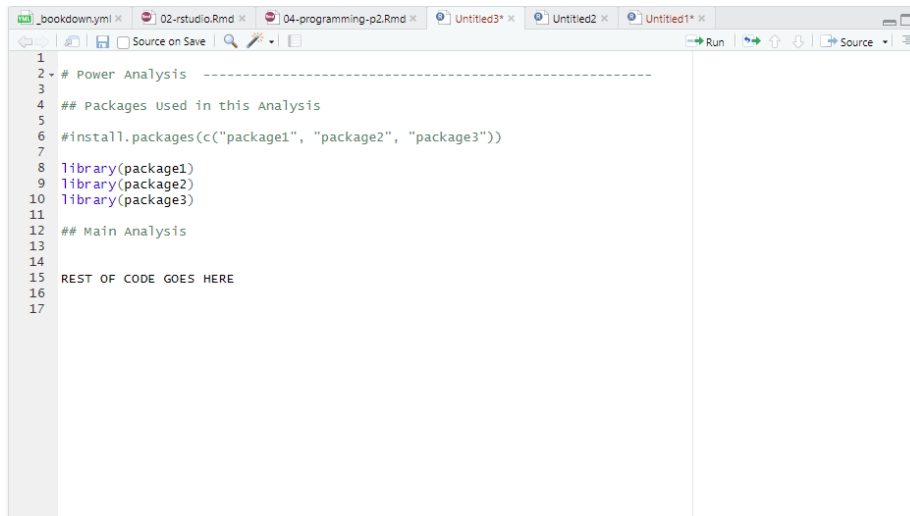
Figure 4.2: Conventions for Installing and Loading Packages in R Script

4.4.7 Summary

There you have it. You have successfully installed and loaded your first packages in R. Are they particularly useful packages? No! In the rest of this course, we will be loading packages more regularly, and these packages will make our lives significantly easier than if they were not around. In fact, we will use the `haven` package in the next section that will enable us to import SPSS data.

4.5 Importing and Exporting Data

While creating data frames and lists in R is valuable, the majority of the data you'll work with in R will likely come from external sources. Therefore, it's essential to know how to import data into R. Similarly, once you've processed and analyzed your data in R, you'll often need to export it for further use or sharing.

In this section, we'll explore how to import and export data using the graphical user interface (GUI) provided by RStudio. The GUI offers a user-friendly and intuitive way to manage data files without requiring you to write any code. Let's delve into the process of importing and exporting data using the RStudio GUI.

First, you'll need to download two files: `psycho.csv` and `burnout.sav`. Both files are available in the rintro Teams channel. If you don't have access to the Teams channel, please contact me at ryan.donovan@universityofgalway.ie.

4.5.1   Importing CSV files.

Comma-Separated Values (CSV) files are a prevalent format for storing tabular data. Similar to Excel files, data in CSV files is organized into rows and columns, with each row representing a single record and each column representing a different attribute or variable.

CSV files are plain text files, making them easy to create, edit, and view using a simple text editor. This simplicity and universality make CSV files a popular choice for data exchange across various applications and platforms.

In a CSV file, each value in the table is separated by a comma (,), hence the name "comma-separated values." However, depending on locale settings, other delimiters such as semicolons (;) or tabs (\t) may be used instead.

One of the key advantages of CSV files is their compatibility with a wide range of software and programming languages, including R. They can be effortlessly imported into statistical software for analysis, making them a versatile and widely adopted format for data storage and sharing.

To import the "psycho.csv" file, please follow these steps:

1. Open RStudio.

2. Navigate to the Environment pane (top right hand corner) and make sure the Environment tab is clicked.

3. Click on the "Import Dataset" button.

4. Select "From Text (base)".

5. Browse your folder and select the "psycho.csv" to import.

6. Click "Open".

7. A pop-up window should now open. There is a lot of different information here, so let's unpack it.

   - The "Name" text box enables you to specify the variable name assigned to the data frame. Leave it as "psycho."

- Most of the time, you can leave the options with the default settings.
- The "Input File" box shows a preview of the CSV file being imported.
- The "Data Frame" box displays a preview of how the data frame will look once the CSV is imported into R.

```r
library(knitr)

include_graphics("img/04-import-csv.png")
```


Figure 4.3: Pop-up window for importing CSV files

8. Click Import. A tab in the source pane called "psycho" should now open showing the data frame. Additionally, you should also see the data frame in the Environment pane. You should see under value that it has "60 obs. of 3 variables" which means there are 60 rows of data and 3 columns of data.

9. Once you have imported your data in R, it is always good practice to briefly check that it has been imported successfully and you have imported the correct data set. Two ways you can do this is by calling the `head()` and `summary()` functions on the data frame.

```r
head(psycho) #this will print out the first six rows
```

```
##   Participant_ID Treatment Neuroticism
## 1              1   Placebo    39.39524
## 2              2   Placebo    42.69823
## 3              3   Placebo    60.58708
## 4              4   Placebo    45.70508
## 5              5   Placebo    46.29288
## 6              6   Placebo    62.15065
```

```r
summary(psycho) #print out summary stats for each column
```

```
##  Participant_ID    Treatment           Neuroticism
##  Min.   : 1.00   Length:60          Min.   :25.33
##  1st Qu.:15.75   Class :character   1st Qu.:41.75
##  Median :30.50   Mode  :character   Median :49.44
##  Mean   :30.50                      Mean   :48.99
##  3rd Qu.:45.25                      3rd Qu.:54.61
##  Max.   :60.00                      Max.   :76.69
```

If your results match mine, it means you have correctly imported the data.

4.5.2   Importing SPSS (.sav files).

SPSS (Statistical Package for the Social Sciences) is another popular software used for statistical analysis, particularly in the social sciences. SPSS data files are typically saved with a `.sav` extension. These files can contain data, variable names, variable labels, and other metadata.

To import an SPSS file (`burnout.sav`), follow these steps:

1. Open RStudio.

2. Navigate to the Environment pane (top right-hand corner) and make sure the Environment tab is clicked.

3. Click on the "Import Dataset" button.

4. Select "From SPSS".

5. A pop-up window will open.

```r
include_graphics("img/04-import-sav.png")
```
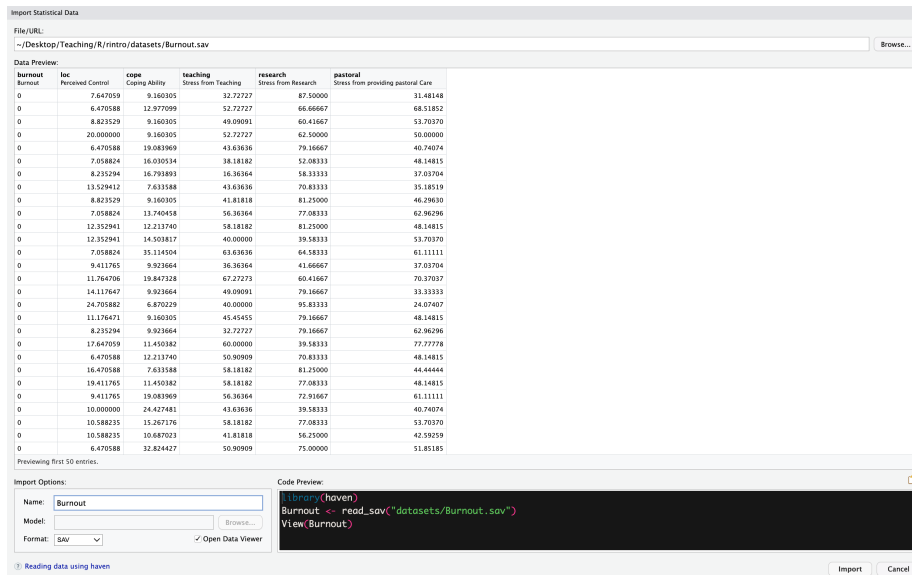

6. Browse your folder and select the **burnout.sav** file to import.

7. Under Import Options, you can set the name of the data frame variable in R for the SPSS data set. Leave it as **burnout.sav**.

8. The Code Preview shows you the written code commands that you could type out to import the data frame instead of using the RStudio interface.

9. To import the dataframe, click "Import".

After importing, a tab in the source pane called "burnout" should open, displaying the data frame. Additionally, you should see the data frame in the Environment pane. Under the "value" column, it should indicate "467 obs. of 6 variables," indicating there are 467 rows and 6 columns of data.

Once imported, use the **head()** and **summary()** functions to check that the data has been imported correctly:

```r
head(Burnout)
```

```
## # A tibble: 6 x 6
##   burnout            loc  cope teaching research pastoral
##   <dbl+lbl>        <dbl> <dbl>    <dbl>    <dbl>    <dbl>
## 1 0 [Not Burnt Out] 7.65  9.16     32.7     87.5     31.5
```

```
## 2 0 [Not Burnt Out]  6.47 13.0     52.7     66.7     68.5
## 3 0 [Not Burnt Out]  8.82  9.16    49.1     60.4     53.7
## 4 0 [Not Burnt Out] 20     9.16    52.7     62.5     50
## 5 0 [Not Burnt Out]  6.47 19.1     43.6     79.2     40.7
## 6 0 [Not Burnt Out]  7.06 16.0     38.2     52.1     48.1
```

```
summary(Burnout)
```

```
##    burnout            loc              cope             teaching
##  Min.   :0.0000   Min.   :  6.471   Min.   :  3.817   Min.   : 16.36
##  1st Qu.:0.0000   1st Qu.:  9.412   1st Qu.: 12.214   1st Qu.: 47.27
##  Median :0.0000   Median : 14.118   Median : 19.084   Median : 54.55
##  Mean   :0.2548   Mean   : 17.900   Mean   : 23.919   Mean   : 55.43
##  3rd Qu.:1.0000   3rd Qu.: 22.353   3rd Qu.: 31.298   3rd Qu.: 61.82
##  Max.   :1.0000   Max.   :100.000   Max.   :100.000   Max.   :100.00
##    research          pastoral
##  Min.   : 20.83   Min.   : 18.52
##  1st Qu.: 52.08   1st Qu.: 46.30
##  Median : 62.50   Median : 53.70
##  Mean   : 61.91   Mean   : 55.25
##  3rd Qu.: 72.92   3rd Qu.: 62.96
##  Max.   :100.00   Max.   :100.00
```

If your results match mine, then you have successfully imported the SPSS data into R.

4.5.3   Exporting Datasets in R

After analyzing and processing your data in R, you may need to export the results to share them with others or use them in other applications. R provides several functions for exporting data to various file formats, including CSV, Excel, and R data files. In this section, we'll explore how to export datasets using these functions.

4.5.3.1   Exporting to CSV Files

To export a dataset to a CSV file, you can use the write.csv() function:

```
# Export dataset to a CSV file using the following syntax
write.csv(my_dataset, file = "output.csv")
```

The argument `file` will create the name of the file and enable you to change the location of the file. The way this is currently written, it will save your file

to your working directory. If you need a reminder on how to set and check your working directory click here. Make sure it is set to the location you want your file to go.

Let's write the `iq_df` as a CSV file.

```
write.csv(iq_df, file = "iq_df.csv")
```

In your working directory (check the Files pane), you should see the file `iq_df.csv`. If you go to your file manager system on your computer, find the file, and open it, the file should open in either a text or Excel file.

4.5.3.2 Exporting to SPSS Files

To export a dataset to an SPSS file, you can use the **write.foreign()** function from the **foreign** package. First, make sure you have installed and loaded the **foreign** package:

```
#install.packages("foreign")
library(foreign)
```

Then, use the following syntax to export data frames to SPSS:

```
write.foreign(my_dataset, #the dataset you are exporting
              datafile = "output.sav", #the location you will export your data
              codefile = "output.sps", #this creates the syntax for converting your file to SPSS
              package = "SPSS") #this specifies the package the exported file will work for
```

You might be wondering why there is both a datafile and a codefile argument. Let's break each down:

- datafile: This argument specifies the file path where the data will be saved in the desired format (e.g., SPSS data file). The datafile argument is used to store the actual data values from the R dataset.

- codefile: This is where the instructions on how to use your data are saved. When you specify a file path for the codefile argument, R will create a file containing commands or instructions that another program (like SPSS) can use to understand and import your data correctly. It's like writing down step-by-step directions for SPSS to follow.

So, in simple terms, the datafile is where your data is saved, while the codefile contains the instructions for using that data in another program. They

work together to make sure your data can be easily used in different software programs.

Now let's export the `psycho` data frame (we imported from CSV) to SPSS!

```r
write.foreign(psycho, #the dataset you are exporting
              datafile = "psycho.sav", #the location you will export your data
              codefile = "psycho.sps", #this creates the syntax for converting your fi
              package = "SPSS") #this specifies the package the exported file will wor
```

Again this will save the file in your working directory. If you go to your file explorer system and open up the file, it should open SPSS for you.

4.6   Summary

Congratulations, you've made it through Programming Part I and II! We've covered a lot of useful (but let's be honest, not exactly riveting) concepts in programming with R. Throughout these sections, we've learned how R categorizes data, stores it in data structures, converts data types, and creates variables and functions. Additionally, we've explored how to install and load packages to enhance R's capabilities, and how to import and export data.

With this foundation, we're now well-equipped to move on to the next phase: data processing. In the upcoming week, we'll dive into methods for cleaning our data, setting the stage for more advanced analyses.

4.7   Glossary

Term	Definition
CSV	Comma-Separated Values: a common file format for storing tabular data, where each value is separated by a comma.
SPSS	Statistical Package for the Social Sciences: software commonly used for statistical analysis, often associated with .sav files.
Dataframe	A two-dimensional data structure in R that resembles a table with rows and columns. It can store mixed data types.
Importing	The process of bringing data from external sources into R for analysis or manipulation.
Exporting	The process of saving data from R to external files or formats for use in other applications.
write.csv()	A function in R used to export a dataset to a CSV file.
write.foreign()	A function in R used to export a dataset to other formats, such as SPSS files.

Chapter 5

Data Wrangling and Cleaning (Part I)

In this session, we are going to learn how to use key packages and functions that enable you to conduct data cleaning in R.

By the end of this session, you should be capable of the following:

- Understand the concept of `tidy data` and `tidy principles`

- Using the functions `select()`, `mutate()`, `rename()`, and `relocate` to perform key operations on columns.

- Using the functions `filter()`, `arrange()`, and `distinct()` to perform key operations on rows

- Understand how to group information and perform calculations on those groupings.

- Understand how to pipe together functions to enable efficient data cleaning analysis.

5.1   What is Data Wrangling and Cleaning?

If you read the literature on data science and data analytics, you will see terms like `data cleaning`, `data munging`, `data wrangling`, `data transformation`, `data preprocessing`, and `data transponding`. Often, when I see these words, I feel like Monica Bing in that one episode of Friends and scream "THAT'S NOT EVEN A WORD!".

I am going to break down three of these terms:

1. **Data cleaning** refers to the process of identifying and correcting errors in your data set. This could involve fixing errors such as duplicates or typos, correcting formatting errors, and handling missing values.

2. **Data wrangling** refers to preparing raw data for statistical analysis. It includes data cleaning but also may involve changing the structure of your data, merging different data sets together, creating new variables, and getting your data in a structure that enables you to conduct whatever statistical analysis you intend to carry out.

3. **Data munging** is mostly synonymous with data wrangling. You "munge" data when you take it from a raw form and transform it into a format that enables data analysis.

4. **Data preprocessing** is an umbrella term for each of these processes. It encompasses anything you do to your data before you analyze it.

To give you a concrete example: If you download data from Qualtrics or Gorilla Research, it is not ready for statistical analysis right away. It will have rows and columns that you won't need and will interfere with your data analysis (e.g., when you download SPSS data from Qualtrics, it will contain both data collected from when you previewed the study and when it went live).

The process of changing that data into a format that you can use (e.g., by removing preview rows, removing columns, changing column names) is data wrangling. However, if you run your descriptive and inferential statistical analysis and you notice that you have missing data in columns, or some rows are duplicated, or there are typos (e.g., Mal instead of Male) and you fix those errors, then that is data cleaning.

Data wrangling takes raw materials and builds a specific structure (e.g., like taking wood and cement and building a house you can live in). Data cleaning ensures that structure looks its absolute best.

5.1.1   Naming Conventions in this Book

I will often say `data cleaning` as a catch-all term for any process involved in getting data ready for statistical analysis. While it's good to know the different meanings of these words when you are searching for specific information, it doesn't affect your day-to-day analysis if you use them `incorrectly`. This might make a heretic in the data science community, but I can live with that.

So we will use `data cleaning` from this point out.

5.1.2   Why is Data Cleaning Important?

It's estimated that 80% of your time working with data is actually focused on data cleaning. That's partially because once your data is ready for statistical

analysis, it only takes a few lines of code/button clicks to run the analysis. Since most of your time working with data will be on cleaning it, it's really worthwhile to do it effectively and efficiently. By doing so, you will decrease the amount of time cleaning data (which let's be honest, is dull) and spend more time on the interesting part - interpreting your results.

5.1.3   How can R help you with Data Cleaning?

Before I learned R, I used to clean my data manually in Excel. This was not ideal for two reasons, namely:

1. Mistakes were easy to make but difficult to spot. It's easy to accidentally overwrite a value in a cell, delete rows, or make some small mistake when doing it by hand - and not even notice it.

2. I would need to repeat the process over and over again whenever I reran the study, collected more data, or noticed a mistake in my data but could not identify when or how I made the mistake.

This made the process excruciating time-consuming and stressful.

R can significantly increase the speed at which you can conduct data cleaning. You can write code instructions to import the raw data (e.g., from Gorilla or Qualtrics), clean it step-by-step, and save the cleaned data in a consistent manner. This significantly reduces the number of errors you can make. Additionally, if you collect any more participants, you don't even need to look at the Excel files. Just download them into your working directory, run the same R script, and you will have your cleaned data. Finally, in times you notice that you did make a mistake (e.g., you excluded an important variable column) you just make that change in your R script and rerun the analysis.

5.1.4   Tidyverse (and Base R)

The **tidyverse** package is a comprehensive collection of R packages designed to facilitate consistent and intuitive data cleaning. At its core are three fundamental principles known as **tidy data** principles:

1. Every variable should occupy a separate column.

2. Every observation (e.g., participant) should occupy a separate row.

3. Each value in the dataset should have its own cell.
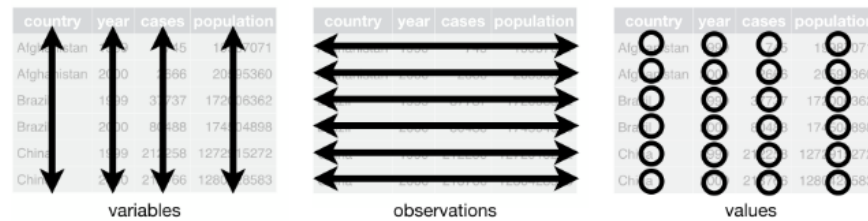
Figure 5.1: Tidy Data principles

Tidyverse is probably the most well-known package in the R community, but there is controversy behind it. High-profile people in the R community debate about whether **tidyverse** is actually appropriate for beginners to learn and whether it really is more effective and efficient than base R. Similarly, some of the R community dislike the idea that the **tidyverse** is the only "real" way to conduct data cleaning.

We will be using both base R and tidyverse in this course. The main thing that you should know is that tidyverse and Base R are like two different dialects in R. If in the future you search for help on R code, the code might be written from one perspective or the other. If the code looks more like what you saw in Chapters 2-4, then it is in base R. If you search for help and looks more like the code in the rest of the course, it's probably tidyverse.

Understanding both approaches will empower you to choose the one that aligns with your preferences and needs, ensuring you can find the right solution more effectively. Whether you are more comfortable with base R or tidyverse, knowing both approaches will enable you to navigate and utilize the wealth of resources available in the R community.

5.2   Let's Get Set Up

Okay, that's enough conceptual information for now - let's learn how to actually clean some data. First, we are going to need to get our RStudio or PositCloud environment ready. We'll do this by setting up our working directory, downloading and importing our data files, and then installing and loading the **tidyverse** packages.

5.2.1   Activity 1.1: Set up your WD

Remember that the working directory is the location where we want to store any resulting data files or scripts that you'll work on in a session. In Chapter 2

I showed you how to do this using a button-and-click interface.

Using those instructions, create a folder called "Week4" in the `rintro` project folder (or whatever folder you created) and set it as your working directory. Use the ´getwd()´ to check that it has been set as your working directory. Your output should be something like this:

```
> setwd("C:/Users/0131045s/Desktop/Programming/R/Workshops/Example/Rintro_2024/week4")
```

5.2.2 Activity 1.2: Import your CSV files and R script

You need to import several files for this activity:

1. **raw_remote_associations.csv**

2. **data_cleaning_script.R**

To download these files, navigate to the Teams channel for this course and access the "Week 4 - Data Cleaning (Part I)" channel. Once downloaded, use your file management system (File Explorer on Windows or Finder on Mac) to copy and paste these files into the "Week4" folder you created in Activity 1.1.

If you're using RStudio, you should see these files in your working directory within the Files pane. Open the R script (**data_cleaning_script.R**).

To import the raw_remote_associations.csv dataset into R, do the following:

1. Click Environment in the Environment Pane -> Import Dataset -> From Text(base) -> Select **raw_remote_associations.csv ->** change its name to **df_raw**
2. See last week's section on importing data for more information

Alternatively, you can run the following command within the **data_cleaning_script.R**

```
df_raw <- read.csv("datasets/raw_remote_associations.csv")
```

5.2.3 Activity 1.3: Install and load the tidyverse package

A good practice in R is to load packages at the start of your script. In the downloaded R script, you'll find the following lines commented out. Copy and paste the code **install.packages("tidyverse")** into your console and press enter to download the tidyverse package.

```
#install.packages("tidyverse")
#library(tidyverse)
```


Figure 5.2: Tidyverse Installation

Once installed, uncomment and run the **library(tidyverse)** in your script to load the package. If installed correctly, you should see the tidyverse package and its dependencies listed in the console. Since tidyverse contains multiple packages, it will load them all at once. So if you see the following then you are good to go.

```
library(tidyverse)
```


You might also get the following warnings when you load in R. If you do, that just means you are using an older version of R. At the beginning of the class, we downloaded version 4.2 onto our own computers, but in the meantime version 4.3 has been released. We can stick with our current version for now.

```
Warning: package 'tidyverse' was built under R version 4.3.2
Warning: package 'ggplot2' was built under R version 4.3.2
Warning: package 'tibble' was built under R version 4.3.2
Warning: package 'tidyr' was built under R version 4.3.2
Warning: package 'readr' was built under R version 4.3.2
Warning: package 'purrr' was built under R version 4.3.2
Warning: package 'dplyr' was built under R version 4.3.2
Warning: package 'stringr' was built under R version 4.3.2
Warning: package 'forcats' was built under R version 4.3.2
Warning: package 'lubridate' was built under R version 4.3.2
```

5.3 Cleaning the Remote Associates Data set.

5.3.1 Context and Viewing it in RStudio

The dataset we imported and are cleaning today comes from a study investigating the effect of mood induction on convergent thinking. Participants were required to read three emotional vignettes (one positive, one negative, and one neutral) before completing a remote associations test. In this test, 41 participants had to identify the link between given words (e.g., Square / Cardboard / Open, with the answer being "Box"). The order of the vignettes was counterbalanced across participants. Additionally, participants completed five items on Openness from the Big Five Aspects, as it is known to positively correlate with convergent thinking.

The first step after importing a dataset is to check it to ensure that we have imported the correct data. We can do this using the **head()** function:

```
#View(df_raw) to open a new tab in Source and look at the entire dataset
head(df_raw) #this will load the first six rows into the console
```

```
##   X Participant.Private.ID Local.Timestamp Experiment.Status
## 1 1               10168827     1.70601e+12           preview
## 2 2               10192092     1.70601e+12           preview
## 3 3               10205485     1.70601e+12           preview
## 4 4               10208522     1.70601e+12           preview
## 5 5               10218310     1.70601e+12           preview
## 6 6               10225898     1.70601e+12           preview
##   remote.assocotations.1 remote.assocotations.2 order Remote.association.3
## 1                      6                     13   ABC                   22
## 2                      9                     18   ABC                   27
## 3                      6                      6   BCA                    6
## 4                      3                     10   ABC                   18
## 5                      9                     18   CAB                   27
## 6                      3                      7   ABC                   11
##   Response Participant.OS response1 open1 open2 open3 open4 open5 age gender
## 1      END    Windows 10        50     3     3     3     3     3  34   male
## 2    BEGIN    Windows 10        63     3     4     3     3     3  34 female
## 3     bank    Windows 10        73     2     4     4     3     3  18   male
## 4      day    Windows 10        58     3     3     2     2     3  29 female
## 5    black    Windows 10        68     3     3     2     3     2  46   male
## 6   common    Windows 10        54     2     3     3     2     3  40 female
```

Ouch! Now that is a data set only a parent could love. Here is what each column is telling us about each participant

Column	Description
X	An empty column that counts the row number.
Participant.Private.ID	Experiment ID.
Local.Timestamp	Timestamp indicating when participants took part in the study.
Experiment.Status	Indicates whether the study was published (live) or not (preview) for each participant.
Remote-Associations1-3	Scores on each remote associations task.
order	Order of watching the vignettes (A = "positive", B = "negative", C = "neutral").
Response	Participants' answers on the remote associations task.
Participant.OS	Operating system used by participants.
response1	Aggregated mood score.
open1-open5	Scores on Openness items.
age and gender	Participants' age and gender.

However, several issues are evident in this dataset:

1. **Pointless columns:** Columns such as "Local.Timestamp" and "Participant.OS" are unnecessary for our analysis.

2. **Awkward, inconsistent, or misspelled column names:** Columns like "Participant.Private.ID" and "remote.assocotations.2" have awkward or misspelled names.

3. **Column order:** The order of columns is not ideal, with important columns scattered throughout the dataset.

4. **Improper scoring:** The maximum score on each remote association task in this study was 9. However, we have several scores that exceed this maximum value.

5. **Mismatch in participant count:** The dataset reports a different number of participants who completed the experiment (52) than what was expected (41).

6. **Presence of preview data:** The dataset contains responses from both the preview and live phases of the study, which need to be addressed by removing preview data.

5.3.2 Key Functions That Will Help You Clean Most Datasets

The following functions are extremely useful for cleaning data frame in R. They all come from the `tidyverse` package. The following table defines each pack-

age and what it does. We will use each function in this chapter to clean the `remote_associations` data frame, so you will get to practice each one.

Function	Description
select()	Include or exclude certain variables (columns)
filter()	Include or exclude certain observations (rows)
mutate()	Create new variables (columns)
arrange()	Change the order of observations (rows)
group_by()	Organize the observations (rows) into groups
summarise()	Create summary variables for groups of observations
disinct()	Select unique observations (rows)
rename()	Rename variables (columns)

5.4 Choosing our Columns of Interest with `Select()`

When you are first cleaning a data set, the first thing you should do is identify the columns that you might need in your analysis or hold important information. In our cases, these columns would be `Participant.Private.ID`, `Experiment.Status`, the response, remote associations, openness, and demographic columns. We can select those columns by using the `select()` function.

The formula for this function is: `select(dataframe, c(col1name, col2name, col3name))`

```
df_select <- select(df_raw, c(Participant.Private.ID, Experiment.Status, order,
                        remote.assocotations.1, remote.assocotations.2,   Remote.associatio

head(df_select)
```

```
##   Participant.Private.ID Experiment.Status order remote.assocotations.1
## 1               10168827           preview   ABC                      6
## 2               10192092           preview   ABC                      9
## 3               10205485           preview   BCA                      6
```

```
## 4                    10208522                 preview   ABC                        3
## 5                    10218310                 preview   CAB                        9
## 6                    10225898                 preview   ABC                        3
##   remote.assocotations.2 Remote.association.3 Response response1 open1 open2
## 1                     13                   22      END        50     3     3
## 2                     18                   27    BEGIN        63     3     4
## 3                      6                    6     bank        73     2     4
## 4                     10                   18      day        58     3     3
## 5                     18                   27    black        68     3     3
## 6                      7                   11   common        54     2     3
##   open3 open4 open5 age gender
## 1     3     3     3  34   male
## 2     3     3     3  34 female
## 3     4     3     3  18   male
## 4     2     2     3  29 female
## 5     2     3     2  46   male
## 6     3     2     3  40 female
```

Instantly, that is immediately better. Now there is a couple of things you might
have noticed in my code.

The first is that the order in which I put columns in `select()` is the order in
which the columns appear in the `df_select`. This is a very useful feature of
`select()` as it means we can reorder our dataframe and select the columns at
the same time. Just to demonstrate this further.

```
df_select <- select(df_select, Participant.Private.ID, Experiment.Status, age, gender,

head(df_select)
```

```
##   Participant.Private.ID Experiment.Status age gender order
## 1               10168827           preview  34   male   ABC
## 2               10192092           preview  34 female   ABC
## 3               10205485           preview  18   male   BCA
## 4               10208522           preview  29 female   ABC
## 5               10218310           preview  46   male   CAB
## 6               10225898           preview  40 female   ABC
##   remote.assocotations.1 remote.assocotations.2 Remote.association.3 Response
## 1                      6                     13                   22      END
## 2                      9                     18                   27    BEGIN
## 3                      6                      6                    6     bank
## 4                      3                     10                   18      day
## 5                      9                     18                   27    black
## 6                      3                      7                   11   common
##   response1 open1 open2 open3 open4 open5
## 1        50     3     3     3     3     3
```

```
## 2           63     3     4     3     3     3
## 3           73     2     4     4     3     3
## 4           58     3     3     2     2     3
## 5           68     3     3     2     3     2
## 6           54     2     3     3     2     3
```

The second thing you might notice is that inside the `c` function I go from naming each column one at a time to using the `:` operator. If you remember from Chapters 3 & 4, when you are selecting elements from a data structure, the `:` operator enables you to select anything between those elements. So the code `remote.assoctations.1:open5` selects every column starting from `remote.assoctations1` and ending with `open5`. This saves us from having to type out every column.

What if we wanted to remove one or two columns? Is there an easier way to remove them then by specifying the columns we want? Yes! We can use `select()` to remove columns. All we need to do is put the `-` operator before the column we want to remove.

If you look at the `Response` column, it doesn't really provide us with any genuinely useful information - we already have their performance on the task. So let's get rid of it.

```
df_select <- select(df_select, -Response) #this will remove the #response column
head(df_select)
```

```
##   Participant.Private.ID Experiment.Status age gender order
## 1               10168827           preview  34   male   ABC
## 2               10192092           preview  34 female   ABC
## 3               10205485           preview  18   male   BCA
## 4               10208522           preview  29 female   ABC
## 5               10218310           preview  46   male   CAB
## 6               10225898           preview  40 female   ABC
##   remote.assocotations.1 remote.assocotations.2 Remote.association.3 response1
## 1                      6                     13                   22        50
## 2                      9                     18                   27        63
## 3                      6                      6                    6        73
## 4                      3                     10                   18        58
## 5                      9                     18                   27        68
## 6                      3                      7                   11        54
##   open1 open2 open3 open4 open5
## 1     3     3     3     3     3
## 2     3     4     3     3     3
## 3     2     4     4     3     3
## 4     3     3     2     2     3
## 5     3     3     2     3     2
## 6     2     3     3     2     3
```

5.5 Renaming our Columns of Interest with `rename()`

Our data set is definitely looking cleaner after having shedding those columns, but good grief are those names still ugly. Luckily, we can change their name using the `rename()` function. The syntax for this function is slightly counter intuitive in its order, as it goes like this: `rename(df, newcolumnane = oldcolumnname)`

```r
df_rename <- rename(df_select,
                    ID = Participant.Private.ID, #newcolname = oldcolname
                    status = Experiment.Status,
                    remote_pos = remote.assocotations.1,
                    remote_neg = remote.assocotations.2,
                    remote_neut = Remote.association.3,
                    total_mood = response1,
                    condition = order)

head(df_rename)
```

```
##          ID  status age gender condition remote_pos remote_neg remote_neut
## 1 10168827 preview  34   male       ABC          6         13          22
## 2 10192092 preview  34 female       ABC          9         18          27
## 3 10205485 preview  18   male       BCA          6          6           6
## 4 10208522 preview  29 female       ABC          3         10          18
## 5 10218310 preview  46   male       CAB          9         18          27
## 6 10225898 preview  40 female       ABC          3          7          11
##   total_mood open1 open2 open3 open4 open5
## 1         50     3     3     3     3     3
## 2         63     3     4     3     3     3
## 3         73     2     4     4     3     3
## 4         58     3     3     2     2     3
## 5         68     3     3     2     3     2
## 6         54     2     3     3     2     3
```

5.6 Creating new Columns using the `mutate()` function

Okay, so we have gotten rid of superfluous columns and cleaned up the names of the ones we have left. But there is still a column "missing". At the moment, we have participants' scores on individual items for Openness to Experience. But unless we are running a reliability or factor analysis, we actually want participants' total level of openness.

The **mutate()** function will help us create that column. This function takes existing column(s) and performs operations on them to create new columns in our data set.

Let's use **mutate()** to create a column called **total_openness**. The syntax for this function is: **mutate(df, new_column_name = instructions on what to do with current columns)**

```
df_mutate <- mutate(df_rename,
                    total_openness = open1 + open2 + open3 + open4 + open5)
                    #new_col_name = operation on current columns

df_mutate$total_openness
```

```
##  [1] 15 16 16 13 13 13 16 14 14 14 14 15 14 15 14 14 16 14 15 15 18 14 17 14 16
## [26] 15 16 15 16 15 16 16 15 15 16 17 15 17 15 15 15 17 16 14 13 15 17 14 14 14
## [51] 17
```

If we wanted to calculate the mean of these items, then the process is slightly more complicated. First, we would need to tell R that we want the mean per observation rather than per column (e.g., we need the mean score of open1 to open5 per participant, rather than across the entire dataset) by using the **rowMeans()** function. Then we would need to **select()** the columns that we want the row means from.

Let's do this and save the operation to a new column called **mean_openness**

```
df_mutate <- mutate(df_mutate,
                    #we use `select()` to pick the columns we want
                    mean_openness = rowMeans(select(df_mutate,
                                                    c(open1, open2, open3,
                                                      open4, open5))))


head(df_mutate)
```

```
##          ID  status age gender condition remote_pos remote_neg remote_neut
## 1 10168827 preview  34   male       ABC          6         13          22
## 2 10192092 preview  34 female       ABC          9         18          27
## 3 10205485 preview  18   male       BCA          6          6           6
## 4 10208522 preview  29 female       ABC          3         10          18
## 5 10218310 preview  46   male       CAB          9         18          27
## 6 10225898 preview  40 female       ABC          3          7          11
##   total_mood open1 open2 open3 open4 open5 total_openness mean_openness
## 1         50     3     3     3     3     3             15           3.0
```

```
## 2            63      3     4     3     3     3                16                3.2
## 3            73      2     4     4     3     3                16                3.2
## 4            58      3     3     2     2     3                13                2.6
## 5            68      3     3     2     3     2                13                2.6
## 6            54      2     3     3     2     3                13                2.6
```

If we do not need the individual items, then there is an argument in `mutate()` called `.keep`. This specifies what should be done with the columns that were operated on to create the new column. If this argument is set to `unused`, then R will only keep columns that were not used to create the new column. In other words, it will remove any columns used to calculate the new column.

```r
df_mutate <- mutate(df_rename,
                    total_openness = open1 + open2 + open3 + open4 + open5,
                    .keep = "unused")

head(df_mutate)
```

```
##          ID  status age gender condition remote_pos remote_neg remote_neut
## 1 10168827 preview  34   male       ABC          6         13          22
## 2 10192092 preview  34 female       ABC          9         18          27
## 3 10205485 preview  18   male       BCA          6          6           6
## 4 10208522 preview  29 female       ABC          3         10          18
## 5 10218310 preview  46   male       CAB          9         18          27
## 6 10225898 preview  40 female       ABC          3          7          11
##   total_mood total_openness
## 1         50             15
## 2         63             16
## 3         73             16
## 4         58             13
## 5         68             13
## 6         54             13
```

5.6.1   Rewriting existing columns using `mutate()`

If we have a column that contains mistakes, we can use `mutate()` to rewrite that column's values. The syntax for this is the same as when you are creating new columns with `mutate`: `mutate(df, existing_column_name = instructions on what to do with current columns)`

If you look at our columns `remote_pos`, `remote_neg`, and `remote_neut`, there is a mistake. Each column should represent the participants' scores (out of a maximum score of 9) on the remote associations task after engaging with either positive, negative, or neutral stimuli. However, while `remote_pos` scoring

looks correct, the scores for **remote_neg** are going up to 18, and the scores for **remote_neut** are going up to 27. So what is going on?

What happened here is that Gorilla Research added participants' scores on each task to each other. So the **remote_neg** column is really the participants' correct answers on **remote_pos** plus their correct answers on **remote_neg**. Similarly, the **remote_neut** contains their scores on the **remote_neg** column plus their scores on the **remote_neut** task.

To fix this, the formula we need to use is the following:

1. remote_neg = remote_neg - remote_pos

2. remote_neut = remote_neut - remote_neg

We can fix this error using the **mutate()** function

```
df_mutate_ra <- mutate(df_mutate,
                   remote_neg = remote_neg - remote_pos,
                   remote_neut = remote_neut - remote_neg)


head(df_mutate_ra)
```

```
##          ID  status age gender condition remote_pos remote_neg remote_neut
## 1 10168827 preview  34   male       ABC          6          7          15
## 2 10192092 preview  34 female       ABC          9          9          18
## 3 10205485 preview  18   male       BCA          6          0           6
## 4 10208522 preview  29 female       ABC          3          7          11
## 5 10218310 preview  46   male       CAB          9          9          18
## 6 10225898 preview  40 female       ABC          3          4           7
##   total_mood total_openness
## 1         50             15
## 2         63             16
## 3         73             16
## 4         58             13
## 5         68             13
## 6         54             13
```

Hold on, what happened here? The **remote_neut** scores are still 9+? Well, we first changed the column **remote_neg** to get our correct scores on this task. However, the new values of **remote_neg** are then inserted on the next line of the formula. To fix for this error, we need to put **remote_neut** first

```
df_mutate_ra_fixed <- mutate(df_mutate,
                   remote_neut =  remote_neut - remote_neg,
```

```
                    remote_neg = remote_neg - remote_pos)

df_mutate_ra_fixed$remote_neut
```

```
## [1]  9  9  0  8  9  4  9  5  9  2  1  9  6  7  9 28  4  8  9  7  0  4  9  9  9
## [26]  8  5  9  6  6  0  9  0  4  5  0  7  1  1  5  6  9  9  0  8  3  0  9  8  9
## [51]  1
```

Better!

5.7   Checking in on our data set

We've done a good bit of cleaning so far. We have removed unnecessary columns, we have fixed their names and their order, we have created new columns, and we have fixed errors in existing columns. Our data set is looking nearly ready for analysis.

However, there is still data that needs to be removed. We should only have 41 participants in our sample, but if you look at your dataframe, we actually have 51 participants.

```
nrow(df_mutate_ra_fixed)
```

```
## [1] 51
```

So where are these extra 10 participants coming from? If you look in the **status** column in our data set, you'll notice that there exists both **preview** and **live** data. We can use the **table()** function to count the number of participants who contributed **preview** versus **live** data.

```
table(df_mutate_ra_fixed$status)
```

```
##
##    live preview
##      42       9
```

Okay, so 9 of our extra 10 participants are coming from preview data. That means we have one extra participant than we should have from when we published the study. The first thing we should do is check what is going on here.

After that, we'll need to remove the 9 preview participants from the dataframe.

The following operations will enable us to clean our datasets based on participant scores on a row-by-row basis rather than on a column basis.

5.8 Removing Duplicates using `distinct()`

What could explain the extra participant? Well, there may have been an error when we downloaded the data and rows got duplicated. Luckily, there is a relatively easy way to do this through the **duplicated()** function. We call the function and give it the dataframe we want to check. The function then checks each row to see if it is a duplicate of another row.

```
duplicated(df_mutate_ra_fixed)
```

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

The function prints out logical data types **TRUE** or **FALSE**. Each answer corresponds to a row in the **df_mutate_ra_fixed** data frame. We can see that the last row is the only one with a value of **TRUE**, which means it is a duplicate of another row. If we want to extract and see the duplicated rows, we can follow the syntax we discussed in Chapter 3 about extracting values from a dataframe: **dataframe[rows_we_want, columns_we_want]**

```
df_mutate_ra_fixed[duplicated(df_mutate_ra_fixed), ]
```

```
##           ID status age gender condition remote_pos remote_neg remote_neut
## 51 10230547   live  28 female       CAB          1          1           1
##    total_mood total_openness
## 51         64             17
```

We can see that the participant with the **ID = 10230547** has a duplicated value. To remove these values we can use the **distinct()** function. This function takes in a data frame or a column and only keeps the rows that are unique.

```
df_distinct <- distinct(df_mutate_ra_fixed)
```

We can use the **table()** function to check whether we have removed the extra participant.

```
table(df_distinct$status)
```

```
##
##    live preview
##      41       9
```

It looks like we have. We can also double-check this by calling the Similarly, if we used the **duplicated()** function again to see if any values return as `TRUE`

```
duplicated(df_distinct)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE
```

There we go, we have successfully removed the duplicate from our data frame. Now let's handle the data that was collected when the study was being `previewed`.

5.9   Removing Rows using the `filter()` Function

We can use the **filter()** function to selectively retain or discard rows in a dataset based on specified conditions. Its syntax is structured as follows: **filter(dataframe, condition)**, where the condition specifies which rows to retain.

In the case of our status column, suppose we want to retain only the rows where the status is "live" and remove those where it is "preview". We achieve this with the following code:

```
df_filter <- filter(df_distinct, status == "live")
```

The condition **status == "live"** instructs R to retain rows where the status column equals **"live"**. Rows that do not meet this condition are filtered out (e.g., where **status == "preview"**). We can use the **table()** function to check that we only have rows where **status == "live"**

```
table(df_filter$status)
```

```
##
## live
##   41
```

We can also use negative conditions in **filter()**. By that, I mean we can tell R to keep rows that ***do not*** meet a certain condition. For example, I will get the same resulting data set if I tell R to keep rows that are not equal to **preview** in the **status** column.

```
df_filter_neg <- filter(df_distinct, status != "preview")

table(df_filter_neg$status)
```

```
##
## live
##   41
```

Here, `!=` denotes "not equal to", allowing us to exclude rows where the status is `"preview"`. This can be really useful if you have multiple answers within a column and it's easier to tell R what not to keep rather than specify everything it needs to keep.

Since we have removed the **preview** scores from our **status** column, do we actually need it anymore? Probably not as it won't be used in our final analysis. So let's remove it using the **select()** function and **-** operator.

```
df_filter <- select(df_filter, -status)
```

5.9.1 Removing Rows Using Multiple Conditions with `filter()`

We can also use **filter()** to check whether rows meet several conditions at the same time. For example, let's look at our columns **remote_pos**, **remote_neg**, and **remote_neut** which corresponds to different times they completed the remote associations task. Participants could score a maximum of 9 and a minimum of 0 on each task.

We might expect that it is reasonable that any participant could score a 0 on a single remote association task. Similarly, we might conclude that it is reasonable that a participant scored a 0 on two remote associations tasks. But we might be suspicious if they scored 0 on all three remote associations tasks. This could be evidence that the participant wasn't paying attention and was just motoring through the study as quickly as possible.

We can use the **arrange()** function to check whether participants scored **0** on these columns. To use the **arrange()** function you specify the data frame, and then you specify the column or columns that you want to arrange by. The syntax is: **arrange(df, col1, col2, col3)**

The function arranges in ascending fashion by default (this can be overridden). If you specify multiple columns, it will arrange in the order of the columns you enter. The following code will arrange our dataframe first by the values of **remote_pos**, then by **remote_neg**, and then by **remote_neut**.

```
df_filter <- arrange(df_filter, remote_pos, remote_neg, remote_neut)

head(df_filter)
```

```
##            ID age gender condition remote_pos remote_neg remote_neut total_mood
## 1 10229575  47 female       BCA          0          0           0         70
## 2 10230546  43 female       ABC          0          0           0         88
## 3 10230541  46   male       BCA          0          0           0         63
## 4 10229436  26   male       CAB          0          2           4         61
## 5 10230533  36 female       CAB          0          4           4         62
## 6 10230659  45 female       ABC          0          5           0         74
##   total_openness
## 1             15
## 2             17
## 3             14
## 4             16
## 5             15
## 6             17
```

We can see that there are three participants scored a **0** on all three tasks. How could we remove these participants? Well, we can use **filter()** to individually check whether participants' scores on the relevant column were greater than 0 using the sign **>**. We could then save the results to different data frames.

```
df_filter_pos <- filter(df_filter, remote_pos > 0) #it will only keep rows in remote_p

df_filter_neg <- filter(df_filter_pos, remote_neg > 0) #this code will only keeps row

df_filter_neut <- filter(df_filter_neg, remote_neut > 0) #this code will only keep row
```

This gets the job done, but it is a tedious approach as we need to type out the same command three times. It makes it significantly more likely we will make a mistake somewhere too.

Luckily, we can do this all in the one line using the | operator. The | operator translates to the logical condition **OR**.

```
df_filter_multi <- filter(df_filter, remote_pos > 0 | remote_neg > 0 | remote_neg > 0)
```

This code translates to "Create a new variable called **df_filter_multi** by taking the **df_filter** dataframe and keep only the rows where participants score higher than 0 on either remote_pos, remote_neg, and remote_neut. If a participant scores 0 on ALL of these columns, please remove them."

This removes any participants from our dataset that did not get a single correct answer on any of the three remote associations tasks. The **nrow()** function shows us that our manipulation was successful.

```
nrow(df_filter_multi)
```

```
## [1] 38
```

What if we wanted to be more strict? What if we wanted to remove any participant that scored a **0** on *any* of **remote_pos**, **remote_neg**, **remote_neut**? In this case, we can use the **&** operator, which translates to the logical condition **AND**.

```
df_filter_multi_strict <- filter(df_filter, remote_pos > 0 & remote_neg > 0 & remote_neg > 0)
```

This code translates to: "Create a new variable called **df_filter_multi_strict** by taking the **df_filter** dataframe and keeping only the rows that have a score that is greater than or equal (>=) to 0 on remote_pos, remote_neg, and remote_neut. If a participant has a score of 0 on ANY of these columns, please remove them."

If we check the number of rows again, we can see that our data frame has shrunk.

```
nrow(df_filter_multi_strict)
```

```
## [1] 33
```

What if we wanted to be stricter again and say that we want to remove participants if they scored less than a 2 on any of the remote association columns? Well, we use the less than or equal to operator **>=**

```
df_filter_multi_stricter <- filter(df_filter, remote_pos >= 2 & remote_neg >= 2 & remote_neg >= 2
```

This code will only keep participants who scored a 2 or higher in all three columns. If we check **nrow()**, we will see that our data frame has shrunk even further

```
nrow(df_filter_multi_stricter)
```

```
## [1] 26
```

I am not a big fan of either the **df_filter_multi_strict** or **df_filter_multi_stricter** columns. So let's carry the **df_filter_multi** data frame to the next section.

5.10   Identifying the Factors in our Data Frame

In the last chapter, we talked about the **factor** data type, which enables us to identify different levels of a variable. This would be the equivalent of identifying which variables are **nominal** or **ordinal** in SPSS. It is important to specify which of your columns are factors when you are cleaning your data, as certain inferential tests require the factor data type as an input. If we look at our current columns, none of them are the **factors**.

```
glimpse(df_filter_multi)
```

```
## Rows: 38
## Columns: 9
## $ ID            <int> 10229436, 10230533, 10230659, 10428518, 10229522, 10229~
## $ age           <int> 26, 36, 45, 37, 30, 36, 28, 47, 46, 35, 18, 20, 33, 28,~
## $ gender        <chr> "male", "female", "female", "male", "female", "female",~
## $ condition     <chr> "CAB", "CAB", "ABC", "BCA", "CAB", "ABC", "CAB", "ABC",~
## $ remote_pos    <int> 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4~
## $ remote_neg    <int> 2, 4, 5, 9, 1, 1, 1, 1, 1, 2, 5, 0, 6, 8, 3, 5, 7, 4, 4~
## $ remote_neut   <int> 4, 4, 0, 9, 0, 1, 1, 1, 2, 28, 6, 0, 6, 8, 3, 5, 7, 4, ~
## $ total_mood    <int> 61, 62, 74, 68, 59, 63, 64, 75, 48, 52, 65, 70, 64, 48,~
## $ total_openness <int> 16, 15, 17, 14, 16, 14, 17, 15, 14, 14, 15, 18, 14, 14,~
```

Let's say for this study, we were interested in investigating the effect of **gender** and **condition** on participants' performance. We would need to convert these two columns to factors. How could we do this? Well, if you remember, we can use **as.factor()** to convert a vector into a factor, and we can use **mutate()** to rewrite existing columns. Let's combine these two approaches now to turn **gender** and **condition** into factors.

```
df_factor <- mutate(df_filter_multi,
                    gender = as.factor(gender),
                    condition = as.factor(condition))
```

Now if we call **glimpse()**, we will see that these columns are now factors.

```
glimpse(df_factor)
```

```
## Rows: 38
## Columns: 9
## $ ID            <int> 10229436, 10230533, 10230659, 10428518, 10229522, 10229~
## $ age           <int> 26, 36, 45, 37, 30, 36, 28, 47, 46, 35, 18, 20, 33, 28,~
## $ gender        <fct> male, female, female, male, female, female, female, mal~
```

```
## $ condition    <fct> CAB, CAB, ABC, BCA, CAB, ABC, CAB, ABC, CAB, CAB, ABC, ~
## $ remote_pos   <int> 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4~
## $ remote_neg   <int> 2, 4, 5, 9, 1, 1, 1, 1, 1, 2, 5, 0, 6, 8, 3, 5, 7, 4, 4~
## $ remote_neut  <int> 4, 4, 0, 9, 0, 1, 1, 1, 2, 28, 6, 0, 6, 8, 3, 5, 7, 4, ~
## $ total_mood   <int> 61, 62, 74, 68, 59, 63, 64, 75, 48, 52, 65, 70, 64, 48,~
## $ total_openness <int> 16, 15, 17, 14, 16, 14, 17, 15, 14, 14, 15, 18, 14, 14,~
```

5.11 Summarising our Data by Groups using `group_by()` and `summarise()`

Now our data frame is looking substantially nicer than when we first started.

```
head(df_filter_multi)
```

```
##          ID age gender condition remote_pos remote_neg remote_neut total_mood
## 1 10229436  26   male       CAB          0          2           4         61
## 2 10230533  36 female       CAB          0          4           4         62
## 3 10230659  45 female       ABC          0          5           0         74
## 4 10428518  37   male       BCA          0          9           9         68
## 5 10229522  30 female       CAB          1          1           0         59
## 6 10229414  36 female       ABC          1          1           1         63
##   total_openness
## 1             16
## 2             15
## 3             17
## 4             14
## 5             16
## 6             14
```

At this point, I am happy to say that the data is cleaned. We do not have any superfluous columns or rows. Our variable names are clear. We have created important new columns and fixed incorrect values in original columns. At this point, I would rename the dataframe to something like **df_clean** to make it clear to future me (or to others), which data frame is ready for analysis.

```
df_clean <- df_factor
```

Now we can actually start summarizing our data. There are two key functions that enable us to do this: **summarise()** and **group_by()** functions.

The **summarise()** function enables you to calculate specific descriptive statistics for columns in your data frame. The syntax for summarise is: **summarise(df, output_statistic = statistic function(column))**. On the left-hand side

of `=`, you name the column that you will be creating and on the right-hand side, you specify the descriptive statistic function that you want to call.

Let's use **summarise()** to display the number of participants and the mean and SD scores for our numeric variables.

```
sum_df <- summarise(df_clean,

        n = n(), #the n() function counts the number of rows in your sample

        mean_remote_pos = mean(remote_pos),
        sd_remote_pos = sd(remote_pos),

        mean_remote_neg = mean(remote_neg),
        sd_remote_neg = sd(remote_neg),

        mean_remote_neut = mean(remote_neut),
        sd_remote_neut = sd(remote_neut),

        mean_openness = mean(total_openness),
        sd_openness = sd(total_openness),

        mean_mood = mean(total_mood),
        sd_mood = sd(total_mood))

sum_df
```

```
##    n mean_remote_pos sd_remote_pos mean_remote_neg sd_remote_neg
## 1 38        4.210526      3.129268        5.710526      2.967462
##   mean_remote_neut sd_remote_neut mean_openness sd_openness mean_mood   sd_mood
## 1         6.526316       4.717631      15.15789    1.127694  61.55263 8.916118
```

The output of the **summarise()** function is a data frame. It's somewhat useful to use when you are calculating descriptive statistics for your entire sample, but we'll learn quicker and easier ways to compute these statistics later on in the course. Where **summarise()** really shines is when you use it in conjunction with **group_by()**.

The **group_by()** function enables you to group together observations in your data frame based on different categories within a column(s). If you combine **group_by** with **summarise**, then you can calculate specific statistics per each group within your data frame.

Let's first do this with gender.

```r
group_gender <- group_by(df_clean, gender)

sum_gen_df <- summarise(group_gender,
                        n = n(),

                        mean_remote_pos = mean(remote_pos),
                        sd_remote_pos = sd(remote_pos),

                        mean_remote_neg = mean(remote_neg),
                        sd_remote_neg = sd(remote_neg),

                        mean_remote_neut = mean(remote_neut),
                        sd_remote_neut = sd(remote_neut))

sum_gen_df
```

```
## # A tibble: 2 x 8
##   gender      n mean_remote_pos sd_remote_pos mean_remote_neg sd_remote_neg
##   <fct>   <int>           <dbl>         <dbl>           <dbl>         <dbl>
## 1 female     20            4            2.94            5.55          2.84
## 2 male       18            4.44         3.40            5.89          3.18
## # i 2 more variables: mean_remote_neut <dbl>, sd_remote_neut <dbl>
```

We could also do it based on the **condition**.

```r
group_condition <- group_by(df_clean, condition)

sum_condition_df <- summarise(group_condition,
                        n = n(),

                        mean_remote_pos = mean(remote_pos),
                        sd_remote_pos = sd(remote_pos),

                        mean_remote_neg = mean(remote_neg),
                        sd_remote_neg = sd(remote_neg),

                        mean_remote_neut = mean(remote_neut),
                        sd_remote_neut = sd(remote_neut))

sum_condition_df
```

```
## # A tibble: 3 x 8
##   condition      n mean_remote_pos sd_remote_pos mean_remote_neg sd_remote_neg
##   <fct>      <int>           <dbl>         <dbl>           <dbl>         <dbl>
```

```
## 1 ABC           13          3.92          3.25          5.38          3.15
## 2 BCA           11          4.55          2.66          7.09          1.92
## 3 CAB           14          4.21          3.53          4.93          3.27
## # i 2 more variables: mean_remote_neut <dbl>, sd_remote_neut <dbl>
```

In each example, the **group_by()** function tells R to pay attention to the participant's score on the specified columns when calculating descriptive statistics. We can also tell R to **group_by** two variables at the same time.

```
group_multi <- group_by(df_clean, condition, gender)

multi_df <- summarise(group_multi,
                  n = n(),

                   mean_remote_pos = mean(remote_pos),
                   sd_remote_pos = sd(remote_pos),

                   mean_remote_neg = mean(remote_neg),
                   sd_remote_neg = sd(remote_neg),

                   mean_remote_neut = mean(remote_neut),
                   sd_remote_neut = sd(remote_neut))
```

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

```
multi_df
```

```
## # A tibble: 6 x 9
## # Groups:   condition [3]
##    condition gender     n mean_remote_pos sd_remote_pos mean_remote_neg
##    <fct>     <fct>  <int>           <dbl>         <dbl>           <dbl>
## 1 ABC        female     6            4             3.74            5.67
## 2 ABC        male       7            3.86          3.08            5.14
## 3 BCA        female     6            5.17          2.14            7.33
## 4 BCA        male       5            3.8           3.27            6.8
## 5 CAB        female     8            3.12          2.85            4.12
## 6 CAB        male       6            5.67          4.08            6
## # i 3 more variables: sd_remote_neg <dbl>, mean_remote_neut <dbl>,
## #   sd_remote_neut <dbl>
```

Running this block of code will produce the warning: **summarise() has grouped output by 'condition'. You can override using the .groups argument.** This basically means that R will summarize by **condition** first.

That's why in the output you see the groups clumped together (e.g., ABC - female, ABC - male; BCA - female...).

There we have it! We have successfully cleaned our first data frame in R. Well done.

5.12 Chaining it All Together with the Pipe Operator

The way we cleaned our data so far is perfectly legitimate. The only annoying thing is that it does clutter up your environment tab, as we needed to create several intermediate data frames on the way to getting to `df_clean`. This process of creating intermediate data frames is (ironically) a bit of a messy workflow. If you come back to this data frame in a few months time, you might forget the difference between `df_factor`, `df_mutate_ra_fixed`, `df_filter`. You might wonder if there is a cleaner way to clean data.

Yes! We can streamline our cleaning process by using tidyverse's pipe operator (%>%), which enables us to chain together multiple functions into a single, cohesive block of code. The pipe operator functions as a "AND THEN" connector, allowing us to sequentially apply different data manipulation functions to our data frame. By using the pipe operator, we can avoid the need to create multiple intermediate data frames and instead perform all cleaning steps in one concise code block.

Let's say I wanted to create a data frame (`df_demo`) with only demographic information and with variable `gender` changed to `sex`. In the way I have shown you so far, I would need to do this:

I want to rename the `gender` column to `sex`, instead of doing this:

```
df_demo_raw <- select(df_clean, ID, age, gender)
head(df_demo_raw)
```

```
##          ID age gender
## 1 10229436  26   male
## 2 10230533  36 female
## 3 10230659  45 female
## 4 10428518  37   male
## 5 10229522  30 female
## 6 10229414  36 female
```

```
df_demo <- rename(df_demo_raw, sex = gender)
head(df_demo)
```

```
##          ID age    sex
## 1 10229436  26    male
## 2 10230533  36 female
## 3 10230659  45 female
## 4 10428518  37    male
## 5 10229522  30 female
## 6 10229414  36 female
```

I can use the pipe operator to connect these functions into one code chunk, like this

```
df_demo_pipe <- df_clean %>% #create a variable called df_demo_pipe. To create it, tak
  select(ID, age, gender) %>%  #select its columns ID, age, gender AND THEN
  rename(sex = gender) #rename gender to sex

head(df_demo_pipe)
```

```
##          ID age    sex
## 1 10229436  26    male
## 2 10230533  36 female
## 3 10230659  45 female
## 4 10428518  37    male
## 5 10229522  30 female
## 6 10229414  36 female
```

Using the pipe operator, we can chain together the functions select() and rename() in one code block. This makes our code more concise and readable compared to the traditional approach of creating separate intermediate data frames.

5.12.1   Using a Pipe Operator on the Remote Associations Data Frame

If we wanted to clean our `remote_associations` data frame using `%>%`, we can run the following code:

```
df_clean_pipe <- df_raw %>%  #we take our raw dataframe AND THEN ( %>% )
  select(Participant.Private.ID, Experiment.Status, age, gender, order, response1, remo

  rename(ID = Participant.Private.ID, #newcolname = oldcolname
         status = Experiment.Status,
         remote_pos = remote.assocotations.1,
         remote_neg = remote.assocotations.2,
```

```r
        remote_neut = Remote.association.3,
        total_mood = response1,
        condition = order) %>% #We RENAME our columns AND THEN

  mutate(total_openness = open1 + open2 + open3 + open4 + open5,
                  .keep = "unused") %>% #we create the total_openess column and fix the remote

  mutate(remote_neut = remote_neut - remote_neg,
        remote_neg = remote_neg - remote_pos,) %>%

  distinct() %>%  #remove duplicates

  filter(status == "live" & (remote_pos > 0 | remote_neg > 0 | remote_neg > 0)) %>%


  mutate(gender = as.factor(gender),
        condition = as.factor(condition))



df_clean_pipe %>%
  group_by(condition, gender) %>%
  summarise(
          n = n(),

          mean_remote_pos = mean(remote_pos),
          sd_remote_pos = sd(remote_pos),

          mean_remote_neg = mean(remote_neg),
          sd_remote_neg = sd(remote_neg),

          mean_remote_neut = mean(remote_neut),
          sd_remote_neut = sd(remote_neut))
```

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

## # A tibble: 6 x 9
## # Groups:   condition [3]
##   condition gender     n mean_remote_pos sd_remote_pos mean_remote_neg
##   <fct>     <fct>  <int>           <dbl>         <dbl>           <dbl>
## 1 ABC       female     6            4             3.74            5.67
## 2 ABC       male       7            3.86          3.08            5.14
## 3 BCA       female     6            5.17          2.14            7.33
## 4 BCA       male       5            3.8           3.27            6.8
```

```
## 5 CAB       female    8          3.12         2.85              4.12
## 6 CAB       male      6          5.67         4.08              6
## # i 3 more variables: sd_remote_neg <dbl>, mean_remote_neut <dbl>,
## #   sd_remote_neut <dbl>
```

The pipe operator enables us to produce the same output as before, but with significantly cleaner code. By chaining together commands, we reduce the need for intermediate data frames and make our data cleaning process more efficient.

The Pipe is an extremely useful operator. But I actually encourage you to first clean your data frames the first way by creating intermediate data frames and calling each function separately. Even though it is more cumbersome, there are two advantages to this approach when you are a beginner. Firstly, it forces you to test each line of code that you run when you are cleaning a data frame, thereby encouraging you to think about what cleaning steps you are taking. Secondly, if you make a mistake in using a function, you will get immediate feedback from R on what the error is. When you use the pipe, R will run the entire code chunk at once and the error message might not tell you exactly where your code broke down. Fixing this can be very frustrating for any programmer, but it can be incredibly disheartening when you are just starting out.

My advice, clean your data frames first using the approach we used here. Once it is clean, try and recreate it using pipes. After a while you should start to feel comfortable using pipes from the start - that's the time to make the transition.

5.13   Activity: Clean the Flanker Dataset

5.13.1   Cleaning Flanker Test Dataset

This exercise involves cleaning the Flanker Test Dataset, which was collected as part of a study investigating the effect of alcohol consumption on inhibiting incongruent responses (e.g., flanker effect). The dataset includes responses from 50 participants who took part in the study when it was made live. Participants answered questions related to their age, gender, and neuroticism. Additionally, participants were split into two conditions: control (no alcohol) and experiment (high alcohol).

Download the files `cleaning-exercise.R` and `flanker.csv` to run this exercise.

5.13.2   Load the "Tidyverse" Package

```
library(tidyverse)
```

5.13.3   Reset Your Environment

Before starting, it's recommended to clear your environment to prevent any accidental use of variables from previous tasks. You can do this by clicking the brush icon in the environment tab.

5.13.4   Import the Data

```r
df_raw <- read.csv("flanker.csv") #make sure you have this in your working directory
# Check the structure and first few rows of the dataframe

View(df_raw)

head(df_raw)

(df_raw)
```

Now that we've loaded the necessary package and imported our data, we can proceed with cleaning the data set.

5.13.5   Cleaning Instructions

Instructions

1. **Select Relevant Columns:**

   - Create a variable called **df_select**.
   - Use the **select()** function to choose the columns relating to participant ID, experiment status (live or preview), age, gender, study condition (control versus experiment), the flanker tasks, and neuroticism items.

2. **Rename Columns:**

   - Create a variable called **df_rename**.
   - Rename the messy columns to the following names:
     - ID
     - status
     - age
     - gender
     - condition

- flanker_congruent
- neuro1
- neuro2
- neuro3
- neuro4
- neuro5

3. **Create Total Neuroticism Score:**

   - Create a variable called **df_mutate**.

   - Use the **mutate()** function to create a variable called **neuro_total** that is the sum of each neuroticism item. Remove the neuro items using **.keep = "unused"**.

4. **Calculate Flanker Effect:**

   - Create a variable called **df_mutate_flanker**.

   - Use the **mutate()** function to create a variable called **flanker_effect**. The formula for the flanker effect is: **flanker_effect = flanker_congruent - flanker_incongruent**.

5. **Filter Participants:**

   - Create a variable called **df_filter**.

   - Use the **filter()** function to select only the participants that took the live experiment AND (&) are older than 18.

6. **Remove Duplicates:**

   - Create a variable called **df_distinct**.

   - Use the **distinct()** function to remove any duplicate participants.

7. **Convert Factors:**

   - Create a variable called **df_factor**.

   - Use the **mutate()** function to convert the columns relating to experimental condition and gender to factors using **as.factor()**.

8. **Finalize Cleaning:**

   - Create a variable called **df_clean** with the following code: **df_clean <- df_factor**.

9. **Chaining with Pipe Operator:**

   - Create a variable called **df_clean_pipe** and chain your code together using the pipe operator (**%>%**).

5.14 Summary

Well done. You have cleaned your first data sets in R. The functions that we used today will cover 70-80% of the functions that you would need to handle most data cleaning problems. In the next chapter, we will look at situations where we need to transform the shape of our data, handle missing values, join together datasets, and identify problematic patterns in our data.

Chapter 6

Data Wrangling and Cleaning (Part II)

In this session, we are going to learn how to clean more challenging data than what we encountered in Chapter 5. In contrast to the last chapter, this section is more of a reference guide than an end-to-end cleaning example. That's because the tools you learn here might not always pop up, or at least are unlikely to all pop up in the one data frame. Nonetheless, when you combine the functions you learn here to clean data with the functions from last week, you will be able to handle an impressive amount (probably around 80%) of any data cleaning challenges you will encounter in your research.

By the end of this session, you should be capable of the following:

- Understand the concepts of **wide** and **long** data and be capable of `pivoting` (i.e., transforming) from one format to another.

- Know how to merge together separate data frames into one file that you can clean.

- Identifying and handling missing data (`NA` values).

6.1   Let's Get Set Up

Similar to the last chapter, first we need to set up RStudio and download some data files.

6.1.1   Activity 1.1: Set up your WD

Remember that the working directory is the location where we want to store any resulting data files or scripts that you'll work on in a session. In Chapter 2, I showed you how to do this using a button-and-click interface.

Using those instructions, create a folder called "Week5" in the **rintro** project folder (or whatever folder you created) and set it as your working directory. Use the **getwd()** to check that it has been set as your working directory. Your output should be something like this:

```
> setwd("C:/Users/0131045s/Desktop/Programming/R/Workshops/Example/Rintro_2024/week5")
```

6.1.2   Activity 1.2: Import your CSV files and R script

You need to import several files for this activity:

1. **background.csv**

2. **flanker_task1.csv**

3. **flanker_task2.csv**

4. **flanker_task3.csv**

5. **demographics.csv**

6. **reaction_time.csv**

7. **raw_remote_clean.csv**

To download these files, navigate to the Teams channel for this course and access the "Week 5 - Data Cleaning (Part II)" channel. Once downloaded, use your file management system (File Explorer on Windows or Finder on Mac) to copy and paste these files into the "Week5" folder you created in Activity 1.1.

f you're using RStudio, you should see these files in your working directory within the Files pane. Create an R script and save it as (**data_cleaning_ii_script.R**).

To import the raw_remote_associations.csv dataset into R, do the following:

1. Click Environment in the Environment Pane -> Import Dataset -> From Text(base) -> Select **demographics.csv ->** change its name to **df_dem**
2. Click Environment in the Environment Pane -> Import Dataset -> From Text(base) -> Select **flanker_task1.csv ->** change its name to **df_flanker1**

3. Click Environment in the Environment Pane -> Import Dataset -> From Text(base) -> Select **flanker_task2.csv ->** change its name to **df_flanker2**
4. Click Environment in the Environment Pane -> Import Dataset -> From Text(base) -> Select **flanker_task3.csv ->** change its name to **df_flanker3**
5. Follow the same instructions to import the **demographics.csv**, **reaction_time.csv,** and **raw_remote_clean.csv** files. Change their name to **df_demographics, df_rt, and df_clean** respectively.
6. See Chapter 4's section on importing data for more information

Alternatively, you can write and run the following commands within the **data_cleaning_ii_script.R**. Just make sure they are in the Week 5 folder and that you have set that folder as your working directory.

```r
df_background <- read.csv("background.csv")

df_flanker1 <- read.csv("flanker_task1.csv")

df_flanker2 <- read.csv("flanker_task2.csv")

df_flanker3 <- read.csv("flanker_task3.csv")

df_demographics <- read.csv("demographics.csv")

df_rt <- read.csv("reaction_time.csv")
```

6.1.3   Activity 1.3: Load the tidyverse package

A good practice in R is to load packages at the start of your script. Write the following in your R script to load in tidyverse.

```r
library(tidyverse)
```

Okay, now we are ready to get cleaning!

6.2   Data Formats (Long and Wide Data)

Psychological research often involves working with data stored in tables, commonly referred to as **Data Frames**. These tables can take two primary formats: **wide** or **long**. Depending on the research software used, the raw data downloaded from a study might be in either **wide** or **long** table format.

Understanding the differences between these formats is crucial because each format facilitates certain tasks more easily. Similarly, knowing how to effectively and efficiently `pivot` (i.e., transform or convert) between these formats is essential for performing specific tasks on the data. Fortunately, R and the tidyverse package are well-equipped to handle both types of data and to convert between them seamlessly.

In this section, we will first define `wide` and `long` data formats and discuss the advantages of each format over its counterpart. We will then explore how to pivot between these formats using R.

6.2.1   Defining Long and Wide Data

Wide Data

In wide data, each row represents a unique participant, and each column represents a separate variable. Table 6.1 shows an example of data in wide format. Each row contains all the information on a specific participant across each variable collected. For example, in one row of information, we can observe that participant 2 is 25 years old, 165 centimeters tall, weighs 60kg, and has a BMI score of 22.

ID	Age	Height	Weight	BMI
1	30	175	76	24.8
2	25	165	60	22
3	35	185	80	23.4

If you are like most psychologists, you are used to seeing data in wide formats. Data is often inputted in wide format in software like Excel or SPSS, as it easier for humans to read. We are used to scanning data horizontally (left-right) rather than vertically (up-down). Because each participant is in a single row, repetition in the data frame is minimized, again making it easier for us to read..

In terms of statistical analysis, wide data is useful for calculating descriptive statistics (e.g., mean, standard deviations) on variables. Certain statistical tests like ANOVA, Linear Regression, and Correlation are easier to compute in R when the data is in wide format.

Long Data

In long data, each row contains a participant's single response to a single variable. The table below illustrates data in long format. Instead of having a column for each variable, there is one column that identifies the measured variable, and another column contains the participant's response to that variable. If multiple variables are collected, each participant has several rows, with each row representing a single response to a single variable.

ID	Variable	Value
1	Age	30
1	Height	175
1	Weight	76
1	BMI	24.8
2	Age	25
2	Height	165
2	Weight	60
2	BMI	22
3	Age	35
3	Height	185
3	Weight	80
3	BMI	23.4

Each row in Table 6.2 represents a participant's response to a single variable. For example, in row 1, we see that participant 1 reported their age (**Variable**) as 30 (**Value**). But I have to look to other rows to see this participant's score on other variables.

It is more difficult to scan long data to quickly capture the information that we need. However, it is often easier for computers and programming languages to work with long data. This is one of the reasons why the concept of `Tidy Data` discussed in the previous chapter prefers data in the long format - every row contains the minimum amount of information needed rather than "cluttering" rows with lots of information.

This preference for long data isn't only stylistic, long-data format is more suitable for certain forms of analyses. Long data is often more suitable if you are analyzing data in R that involves repeated measures or longitudinal designs, basically any test where we are interested in within-subject variability over time. Similarly, a lot of the packages/functions developed to enable high quality data visualizations were built with the assumption that your data is in long-format.

6.3 Converting the Format of Our Data

While it's important to know the differences between wide and long data formats, do not feel you have to memories every detail. If you are running a statistical test it'll be pretty easy to find out what type of format your data needs to be in. If the data is not in the correct format, then the tidyverse package makes it straightforward to convert one format to another, thanks to two functions called: **pivot_longer()** and **pivot_wider()**.

6.3.1 Pivoting from Wide to Long

The `pivot_longer()` function converts a wide data frame into long format. Typing `?pivot_longer` into the console provides detailed information about this function in RStudio through the Help tab in the Files Pane[1].

```
?pivot_longer
```

There is a lot of information that will appear in the help section. I want to draw your attention to the **Usage** section, which contains the arguments (inputs) that we can specify in the `pivot_longer()` function.

There is a lot of potential inputs we can throw in, but I want to highlight the key arguments that you will use most of the time when you use this function

Argument	Meaning
`data`	Here you specify the wide data frame that you want to convert to long format
`cols`	The column(s) that will be moved or altered when you pivot the data frame.
`names_to`	The names of each variable identified in cols will be stored in a new column in our long data frame. The names_to argument specifies the name(s) of that new column(s).
`values_to`	The values associated with each variable identified in cols will be stored in a new column in our long data frame. The values_to argument specifies the name of that new column.

Let's create an example data frame to use `pivot_longer()` longer with. I recommend you copy and paste the code below to your R script and run it to create the data frame

```r
#set.seed(123) ensures that the data my R generates will be the same as the data your

set.seed(123)


wide_df <- data.frame(
```

---

[1]You can use this syntax with every function in R. We haven't used so far in the course because I personally think the "helpful information" that R gives you is absolute GARBAGE if you are beginner. It tends to be highly technical, minimal, and will often confuse more people than it will help inform.

```
  ID = 1:10,
  Wellbeing = round(rnorm(n = 10, mean = 4, sd = 0.8), 2),
  Extraversion = round(rnorm(n = 10, mean = 3, sd = 0.8), 2),
  Neuroticism = round(rnorm(n = 10, mean = 3, sd = 0.8), 2),
  Conscientiousness = round(rnorm(n = 10, mean = 3, sd = 0.8), 2),
  Openness = round(rnorm(n = 10, mean = 3, sd = 0.8), 2)
)

#rnorm() randomly generates a set of numbers (n) that have a certain mean and standard deviation.

head(wide_df)
```

```
##   ID Wellbeing Extraversion Neuroticism Conscientiousness Openness
## 1  1      3.55         3.98        2.15              3.34     2.44
## 2  2      3.82         3.29        2.83              2.76     2.83
## 3  3      5.25         3.32        2.18              3.72     1.99
## 4  4      4.06         3.09        2.42              3.70     4.74
## 5  5      4.10         2.56        2.50              3.66     3.97
## 6  6      5.37         4.43        1.65              3.55     2.10
```

So we've created a wide data frame with 10 participants (5 male, 5 female) with scores on each of the Big Five personality traits. We know the data frame is wide because every variable is a separate column and each row tells us a participant's response to each variable.

Let's see how we can convert this data frame from wide to long using pivot_longer(). We'll call the long data frame long_df:

```
long_df <- pivot_longer(
  wide_df,
  cols = Wellbeing:Openness, #we will pivot everything except ID
  names_to = "Variable",
  values_to = "Response"
)

head(long_df)
```

```
## # A tibble: 6 x 3
##      ID Variable          Response
##   <int> <chr>                <dbl>
## 1     1 Wellbeing             3.55
## 2     1 Extraversion          3.98
## 3     1 Neuroticism           2.15
## 4     1 Conscientiousness     3.34
```

```
## 5      1 Openness              2.44
## 6      2 Wellbeing             3.82
```

Now we have the same data frame but in a different format.

The figure below gives an example of the pivot process. We typically do not pivot the ID column, because that enables us to identify which participant's score each variable. Let's look at what happens if we do include ID in the `cols` argument

```r
pivot_longer(
  wide_df,
  cols = ID:Openness, #pivot everything
  values_to = "Response"
)
```

```
## # A tibble: 60 x 2
##    name              Response
##    <chr>                <dbl>
##  1 ID                       1
##  2 Wellbeing             3.55
##  3 Extraversion          3.98
##  4 Neuroticism           2.15
##  5 Conscientiousness     3.34
##  6 Openness              2.44
##  7 ID                       2
##  8 Wellbeing             3.82
##  9 Extraversion          3.29
## 10 Neuroticism           2.83
## # i 50 more rows
```

Now we have lost our record for identifying which participant contributed to which data point. This identifies a key about using `pivot_longer()` in that not EVERYTHING needs to pivoted, it depends on our analytical needs are. Let's go through a slightly more complicated data frame to illustrate what I mean by this.

6.3.2 Pivoting our Remote Associations Data Frame

Last week, we cleaned the **raw_remote_associations.csv** data frame and stored it in a variable called **df_clean**. If we use **head()**, we'll see that it's in wide format.

Figure 6.1: Visual representation of what is pivoted

```
#if you do not have df_clean in your environment, download the dataset `raw_remote_clean.csv` fro

#Run the following code to load it in
#df_clean <- read.csv("raw_remote_clean.csv")

head(df_clean)
```

```
##            ID condition age gender remote_pos remote_neg remote_neut total_mood
## 1 10229436        CAB  26   male          0          2           4         61
## 2 10230533        CAB  36 female          0          4           4         62
## 3 10230659        ABC  45 female          0          5           0         74
## 4 10428518        BCA  37   male          0          9           9         68
## 5 10229522        CAB  30 female          1          1           0         59
## 6 10229414        ABC  36 female          1          1           1         63
##    total_openness mean_openness
## 1             16           3.2
## 2             15           3.0
## 3             17           3.4
## 4             14           2.8
## 5             16           3.2
## 6             14           2.8
```

Let's convert **df_clean** to the long data format, and let's call it **df_clean_long**. However, we are not going to follow the same protocol as the last example, where we pivoted everything except ID. For this data frame, we are going to include every variable inside the **cols** argument except for ID, condition, and gender.

```
df_clean_long <- pivot_longer(df_clean,
  cols = c(age, remote_pos:mean_openness), #we select age, and then we select everything from rem
  names_to = "Variable", #this creates a column called `variables` that will tell us the the vari
  values_to = "Response"#this creates a column called `answer` that will tell us participants act
)
```

```
print(df_clean_long)
```

```
## # A tibble: 266 x 5
##           ID condition gender Variable       Response
##        <int> <chr>     <chr>  <chr>             <dbl>
##  1 10229436 CAB       male   age                  26
##  2 10229436 CAB       male   remote_pos            0
##  3 10229436 CAB       male   remote_neg            2
##  4 10229436 CAB       male   remote_neut           4
##  5 10229436 CAB       male   total_mood           61
##  6 10229436 CAB       male   total_openness       16
##  7 10229436 CAB       male   mean_openness         3.2
##  8 10230533 CAB       female age                  36
##  9 10230533 CAB       female remote_pos            0
## 10 10230533 CAB       female remote_neg            4
## # i 256 more rows
```

There we have it! Now our data is in long format. Each row contains a partici-
pant's individual score on a particular variable. But this time, the participant's
information on condition and gender also gets replicated for each row created
for that participant.

But why did we not include the variables condition and gender in our conversion?
There are both technical and analytical reasons for this decision. Let's address
the technical reason first.

Technically, we actually can't create the `Answer` column by combining partic-
ipants responses on variables like `condition` and `gender` with their answers
on the other variables. This is because the data type for both `condition` and
`gender` are `factors`, whereas the data type for every other variable is `numeric`.

If you remember from our vector discussion (and remember everything that
every column is just a lucky vector who found a home) we mentioned that
vectors are lines of data where everything in the line is of the same data type.
You can have character vectors, factor vectors, numerical vectors, logical vectors,
integer vectors, but you cannot have a single vector with multiple data types.

So if we try `pivot_longer` on our `df_clean` data frame, including the `gender`
and `condition` columns, we get the following error:

```
pivot_longer(df_clean,
  cols = c(condition:mean_openness), #we try to select everything except ID
  names_to = "Variable", #this creates a column called `variables` that will tell us t
  values_to = "Response"#this creates a column called `answer` that will tell us parti
)
```

```
## Error in 'pivot_longer()':
## ! Can't combine 'condition' <character> and 'age' <integer>.
```

If you're stubborn and you insist on pivoting everything, then you would need to convert all of our columns to the same data type. There is an argument in the **pivot_longer()** function that enables us to do this, called **values_transform**. The easiest solution would be to transform everything that will go into our **Response** vector/column into a character.

```
pivot_longer(df_clean,
  cols = condition:mean_openness,
  names_to = "Variable",
  values_to = "Response",
  values_transform = list(Response = as.character) #takes everything that will be put into the Re
  )
```

```
## # A tibble: 342 x 3
##          ID Variable      Response
##       <int> <chr>         <chr>
##  1 10229436 condition     CAB
##  2 10229436 age           26
##  3 10229436 gender        male
##  4 10229436 remote_pos    0
##  5 10229436 remote_neg    2
##  6 10229436 remote_neut   4
##  7 10229436 total_mood    61
##  8 10229436 total_openness 16
##  9 10229436 mean_openness 3.2
## 10 10230533 condition     CAB
## # i 332 more rows
```

That creates an example of a long data frame, which looks neater than our earlier attempt. However, I don't recommend this approach. Since everything inside **Response** is not a character, we can't conduct any quantitative analysis, defeating the purpose.

This leads me on to the analytical reason why we don't want to pivot our **condition** and **gender** columns. Since **condition** and **gender** are factors, we'' want to investigate the extent to which participants scores on **Wellbeing** and our Big Five traits are influenced by that factor. In other words, we want to investigate the effect of our independent variables on our dependent variables. If you look at **df_clean_long**, the data frame keeps a record of a participant's score on each dependent variable in relation to our two independent variables. This sets us up nicely for conducting statistical analysis.

```
## # A tibble: 10 x 5
##          ID condition gender Variable       Response
##       <int> <chr>     <chr>  <chr>             <dbl>
##  1 10229436 CAB       male   age                  26
##  2 10229436 CAB       male   remote_pos            0
##  3 10229436 CAB       male   remote_neg            2
##  4 10229436 CAB       male   remote_neut           4
##  5 10229436 CAB       male   total_mood           61
##  6 10229436 CAB       male   total_openness       16
##  7 10229436 CAB       male   mean_openness         3.2
##  8 10230533 CAB       female age                  36
##  9 10230533 CAB       female remote_pos            0
## 10 10230533 CAB       female remote_neg            4
```

6.3.3   Pivoting from Long to Wide

Now that we've covered converting data frames from wide to long, how can we convert from long to wide? We can use the `pivot_wider()` function. The figure below shows the results of typing `?pivot_wider` into the console. The table below it shows the key arguments of this function.

Argument	Meaning
`data`	The long data frame that you want to convert to wide format
`id_cols`	The columns that help identify each participant. This is often the values that are repeated in each row within a long data frame (e.g., like ID or any independent variables)
`names_from`	When we pivot from long to wide, we will be creating new columns for each variable that we collected data on. We need to tell R where to find the names for those variables.
`values_from`	We need to tell R where to find the values for the new columns that we are creating.

Let's use `pivot_wider()` to convert our `long_df` back into wide format.

```
pivot_wider(long_df,
            id_cols = ID,
            names_from = Variable,
            values_from = Response)
```

```
## # A tibble: 10 x 6
```
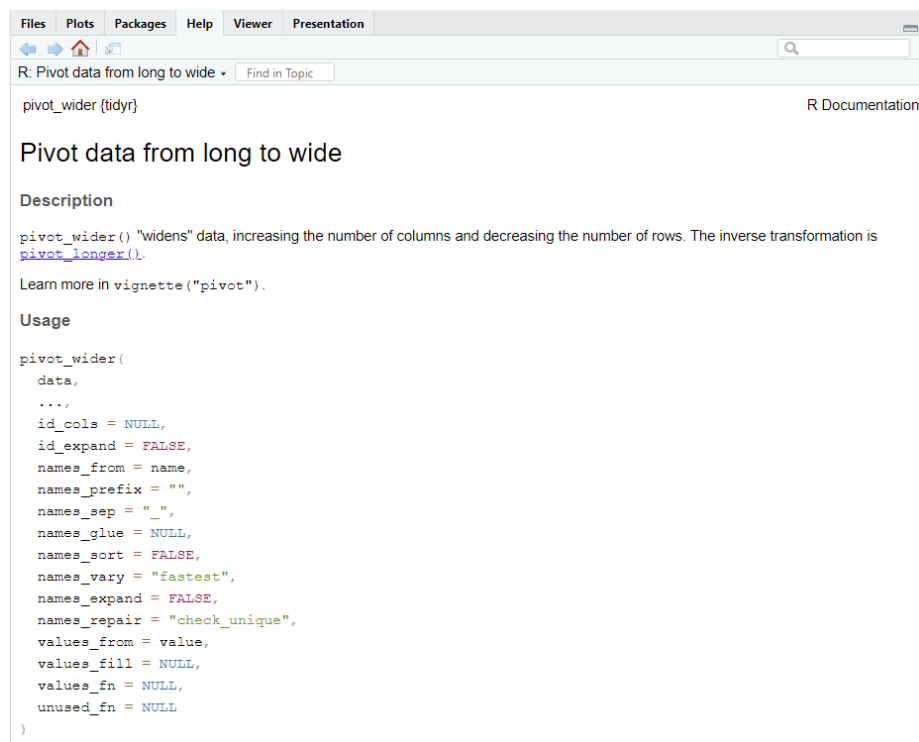
Figure 6.2: The arguments that we can pass to the pivot_wider() function

```
##          ID Wellbeing Extraversion Neuroticism Conscientiousness Openness
##       <int>     <dbl>        <dbl>       <dbl>             <dbl>    <dbl>
## 1      1       3.55         3.98        2.15              3.34     2.44
## 2      2       3.82         3.29        2.83              2.76     2.83
## 3      3       5.25         3.32        2.18              3.72     1.99
## 4      4       4.06         3.09        2.42              3.7      4.74
## 5      5       4.1          2.56        2.5               3.66     3.97
## 6      6       5.37         4.43        1.65              3.55     2.1
## 7      7       4.37         3.4         3.67              3.44     2.68
## 8      8       2.99         1.43        3.12              2.95     2.63
## 9      9       3.45         3.56        2.09              2.76     3.62
## 10    10       3.64         2.62        4                 2.7      2.93
```

Look familiar? If you compare it to our original `wide_df`, you'll notice they look exactly the same.

Now let's do the same thing with the long version of `remote_associations` data frame.

```
df_clean_wide <- pivot_wider(df_clean_long,
                             id_cols = ID:gender,
                             names_from = Variable,
                             values_from = Response)

head(df_clean_wide)
```

```
## # A tibble: 6 x 10
##           ID condition gender   age remote_pos remote_neg remote_neut total_mood
##        <int> <chr>     <chr> <dbl>      <dbl>      <dbl>       <dbl>      <dbl>
## 1 10229436 CAB       male     26          0          2           4         61
## 2 10230533 CAB       female   36          0          4           4         62
## 3 10230659 ABC       female   45          0          5           0         74
## 4 10428518 BCA       male     37          0          9           9         68
## 5 10229522 CAB       female   30          1          1           0         59
## 6 10229414 ABC       female   36          1          1           1         63
## # i 2 more variables: total_openness <dbl>, mean_openness <dbl>
```

If you compare that to the head of `df_clean`, you'll see that its back to the format we cleaned it to last week.

6.3.3.1   Summary

That covers basic pivoting from long to wide and wide to long using `pivot_long()` and `pivot_wide()`. However, both functions enable you to

handle much more complicated data frames and clean them up as you go. We won't cover that in this version of the textbook, but I highly recommend reading the `Advanced Pivoting` section from the `Data Wrangling` Standford ebook if your pivoting needs are more complicated than the examples here.

6.4 Handling Missing Values

In psychological research, dealing with missing data is a common challenge that requires careful consideration to ensure the integrity and validity of analyses. In this section, we'll explore how to handle missing values (NA) in R using the tidyverse package. We'll cover techniques for identifying missing values, strategies for handling them, and best practices for addressing missing data in psychological research studies.

6.4.1 Introduction to Missing Values

Missing values, often represented as NA in R, occur when data is not available or cannot be recorded for certain observations. This can occur due to various reasons such as non-response in surveys, data entry errors, or incomplete data collection processes. It's essential to understand and address missing values appropriately to avoid biased or misleading results in data analysis.

6.4.2 Identifying Missing Values

Detecting missing values in datasets is the first step towards handling them effectively. In R, we can use functions like `is.na()` to identify missing values.

```r
# Example dataset with missing values
missing_data <- data.frame(
  id = 1:10,
  age = c(25, NA, 30, 28, 35, NA, 40, 22, NA, 29),
  gender = c("Male", "Female", "Male", NA, "Female", "Male", NA, "Male", "Female", NA)
)

#check for missing values
missing_values <- summarise_all(missing_data, ~sum(is.na(.)))

#inside summarise_all, we tell R to pick the missing_data and then to sum up the number of missin

print(missing_values)
```

```
##   id age gender
## 1  0   3      3
```

In this example, we've created a dataset **missing_data** with variables **age** and **gender**, some of which contain missing values. We then used **summarise_all()** along with **is.na()** to count the number of missing values in each column.

We can see that there are three missing values in both the **age** and **gender**nder columns.

6.4.3   Removing Missing Values

To remove rows or columns with missing values, we can use functions like **drop_na()** or **na.omit()** in the tidyverse. Here's how we can remove rows with missing values in the **age** column:

```r
# Remove rows with missing values in the age column
removed_data <- drop_na(missing_data)

print(removed_data)
```

```
##   id age gender
## 1  1  25   Male
## 2  3  30   Male
## 3  5  35 Female
## 4  8  22   Male
```

```r
#filter

removed_data <- na.omit(missing_data)

print(removed_data)
```

```
##   id age gender
## 1  1  25   Male
## 3  3  30   Male
## 5  5  35 Female
## 8  8  22   Male
```

6.5   Merging Data (i.e., Joining Different Datasets Together)

In psychological research, we'll often encounter situations where data from multiple sources or studies need to be combined for analysis. We might have collected demographic information separately from participants answers on experimental tasks. We may have collected data using a variety of platforms (e.g.,

survey data using Qualtrics and response data using PsychoPy). Additionally, the research software tools we use might modulate the data. For example, if we run a study in Gorilla Research, then each separate task and questionnaire gets downloaded as separate files.

Whatever the reason, you will often need to merge or join data together from different sources in order to conduct the analysis you need. Luckily, R is quite capable at facilitating data merging. In this subsection, we will look at ways you can merge different data frames together.

6.5.1 Introduction to Merging Data

Merging data involves combining data frames based on their common variables. Let's image we have two data frames called `df_demographics` and `df_rt`. These data frames contain information on both participants demographic information and their reaction time on a specific task and which condition they were randomly assigned to. Let's load both of these data frames into R (make sure you have downloaded them and put them into your working directory before running the following code).

```
head(df_demographics)
head(df_rt)
```

Ideally, we would have one merged data frame that would contain a participant's response on all our variables. However, there are some complications with these two data frames. If you check the number of rows in each data frame, we can see there are differing number of participants.

```
nrow(df_demographics)
```

```
## [1] 60
```

```
nrow(df_rt)
```

```
## [1] 42
```

There are 60 participants in the demographics data frame whereas there are only 42 participants in the reaction time data frame. If the study was online, maybe participants gave up after completing the demographic information, maybe there was connection issues, maybe the data did not save correctly. Whatever the reason for this mismatch in participants, we need to account for this when we merge these data frames together.

Luckily, there are a multitude in ways we can do this through the `tidyverse` `package`.

These types of join are:

- **Inner Join**: Includes only the rows that have matching values in both datasets. This type of join retains only the observations that exist in both datasets, excluding unmatched rows.

- **Left Join**: Includes all rows from the left dataset and matching rows from the right dataset. Unmatched rows from the right dataset are filled with NA values.

- **Right Join**: Includes all rows from the right dataset and matching rows from the left dataset. Unmatched rows from the left dataset are filled with NA values.

- **Outer Join (or Full Join)**: Includes all rows from both datasets, filling in missing values with NA where there are no matches.

Using our `df_demographics` and `df_rt` data frames, lets show you the result of each of these joins and why/when you would use them.

6.5.2   Inner_Join

The `inner_join()` function joins together two data frames, but it will only keep the rows that have matching values in both data frames.

When we use `inner_join()` we need to specify the value(s) that we want to match across both data frames. Once we do, then in the case of the `df_demographics` and `df_rt` data frames, what this means is that only the participants who match on that specified value(s) in both data frames will merged together.

Let's create a merged data frame using `inner_join` and call it `df_inner`. The syntax for inner_join is: `inner_join(df1, df2, by = join_by(column(s))`

```
df_inner <- inner_join(df_demographics, df_rt, by = "ID")

head(df_inner)
```

```
##           ID     gender age   condition mean_rt
## 1 EoYncPX1QK     Female  24 no caffeine     269
## 2 Zn1yzAeYA4 Non-Binary  21 no caffeine     206
## 3 B4iCIhzgPi       Male  24 no caffeine     187
## 4 9sJnqQMOlo     Female  23 no caffeine     217
## 5 FPSgiOjwA7 Non-Binary  23 no caffeine     160
## 6 OgOAFLyHCe       Male  24 no caffeine     255
```

We can see that our `df_inner` has combined the `gender` and `age` columns from `df_demographics` with the `condition` and `mean_rt` columns from `df_rt`. When we use `inner_join` the order in which specify the data frames is the order in which the columns will be added. So if we wanted the `condition` and `mean_rt` columns to come first, then we can change the order:

```
inner_join(df_rt, df_demographics, by = join_by(ID))
```

```
##             ID    condition mean_rt     gender age
## 1  EoYncPX1QK   no caffeine     269     Female  24
## 2  Zn1yzAeYA4   no caffeine     206 Non-Binary  21
## 3  B4iCIhzgPi   no caffeine     187       Male  24
## 4  9sJnqQMOlo   no caffeine     217     Female  23
## 5  FPSgiOjwA7   no caffeine     160 Non-Binary  23
## 6  OgOAFLyHCe   no caffeine     255       Male  24
## 7  hlmrG4AyLu   no caffeine     145     Female  22
## 8  oOUz7EpZDf   no caffeine     240 Non-Binary  24
## 9  QhvvMEVq1X   no caffeine     212       Male  24
## 10 WHdme1YyZv   no caffeine     207     Female  22
## 11 yFTywINVDl   no caffeine     236 Non-Binary  24
## 12 E4TDInCFgc   no caffeine     157       Male  23
## 13 w15ouKhYjX   no caffeine     139     Female  24
## 14 PRHjvltTq9   no caffeine     185 Non-Binary  23
## 15 T1INTYDoxW  low caffeine     261       Male  23
## 16 wQgCowzLTF  low caffeine     229     Female  22
## 17 gAPefpxFu2  low caffeine     314 Non-Binary  23
## 18 kJRT78syM1  low caffeine     150       Male  24
## 19 zigHVms3ZU  low caffeine     205     Female  23
## 20 MdaNDDZypx  low caffeine     353 Non-Binary  24
## 21 1kVtNcCJZR  low caffeine     201       Male  21
## 22 vWP6tk42hT  low caffeine     245     Female  23
## 23 uSbQmTf4hR  low caffeine     242 Non-Binary  24
## 24 F5JS8nOpw8  low caffeine     281       Male  24
## 25 GNNjyhr8Oi  low caffeine     203     Female  22
## 26 gg587jx1wz  low caffeine     260 Non-Binary  22
## 27 QG48CYjO1m  low caffeine     113       Male  22
## 28 QkyZzgywzF  low caffeine     162     Female  23
## 29 t8x4iOKwnT high caffeine     204 Non-Binary  22
## 30 fAazPW5qCz high caffeine     170       Male  23
## 31 Aug7OzOtfX high caffeine     211     Female  22
## 32 DECqK4tIyT high caffeine     114 Non-Binary  23
## 33 GbdjGeOyh2 high caffeine     227       Male  24
## 34 yuSrPEfg8O high caffeine     175     Female  24
## 35 VqOSBONOgt high caffeine     177 Non-Binary  23
## 36 rCqGbWbmxW high caffeine     192       Male  22
```

```
## 37 crbKl4mP8f high caffeine     171     Female  22
## 38 PPL7VpIJAA high caffeine     114 Non-Binary  23
## 39 I8wcEVbUwc high caffeine     230       Male  22
## 40 i8MDEDiJHv high caffeine     201     Female  23
## 41 5ROBC8QMSK high caffeine      48 Non-Binary  22
## 42 QGslHuqlDI high caffeine     255       Male  23
```

If we check the number of rows, we will see that it matches the number of rows in `df_rt` rather than `df_demographics`.

```
nrow(df_inner)
```

```
## [1] 42
```

6.5.3   Left_join

The function `left_join` keeps every participant (row) in the first data frame we feed it. It then matches participants responses in the second data frame and joins them together, once we specify a value that needs to be matched. If there is not a match on that column, then it fills the results with `NA` values.

Let's create the data frame `df_left` using `left_join()`. The syntax for this function is: `left_join(df1, df2, by = join_by(ID))`.

```
df_left <- left_join(df_demographics, df_rt, by = join_by(ID))
```

```
head(df_left)
```

```
##           ID     gender age    condition mean_rt
## 1 EoYncPX1QK     Female  24 no caffeine     269
## 2 Zn1yzAeYA4 Non-Binary  21 no caffeine     206
## 3 B4iCIhzgPi       Male  24 no caffeine     187
## 4 9sJnqQMOlo     Female  23 no caffeine     217
## 5 FPSgiOjwA7 Non-Binary  23 no caffeine     160
## 6 OgOAFLyHCe       Male  24 no caffeine     255
```

```
tail(df_left) #prints out the last six rows of a data frame
```

```
##           ID     gender age condition mean_rt
## 55 TNfJ2PV63D     Female  24      <NA>      NA
## 56 VZjyLYJOyd Non-Binary  23      <NA>      NA
## 57 oZv0WxMU7K       Male  21      <NA>      NA
```

```
## 58 T1UopshE5K    Female  24       <NA>       NA
## 59 MZV79pikQY Non-Binary  24       <NA>       NA
## 60 D8j28H49Lt      Male  23       <NA>       NA
```

```
nrow(df_left)
```

```
## [1] 60
```

We can see that every participant in the `df_demographics` is included inside
the `df_left` data frame. If that participant does not have scores on `condition`
and `mean_rt`, then `NA` is substituted in.

The function is called `left_join()` because it joins whatever is put first (i.e.,
left) in the function is given priority over what comes second (i.e., right). The
next merging function we'll discuss does the opposite.

6.5.4   Right_Join

The function `left_join` keeps every participant (row) in the second data frame
we feed it. It then matches participants responses in the first data frame and
joins them together, once we specify a value that needs to be matched. If there
is not a match on that column, then it fills the results with `NA` values.

Let's create the data frame `df_left` using `left_join()`. The syntax for this
function is: `right_join(df1, df2, by = join_by(ID))`

```
df_right <- right_join(df_demographics, df_rt, by = join_by(ID))
```

```
head(df_right)
```

```
##           ID      gender age    condition mean_rt
## 1 EoYncPX1QK      Female  24 no caffeine     269
## 2 Zn1yzAeYA4 Non-Binary  21 no caffeine     206
## 3 B4iCIhzgPi        Male  24 no caffeine     187
## 4 9sJnqQMOlo      Female  23 no caffeine     217
## 5 FPSgiOjwA7 Non-Binary  23 no caffeine     160
## 6 OgOAFLyHCe        Male  24 no caffeine     255
```

```
tail(df_right)
```

```
##           ID      gender age      condition mean_rt
## 37 crbKl4mP8f      Female  22 high caffeine     171
## 38 PPL7VpIJAA Non-Binary  23 high caffeine     114
```

```
## 39 I8wcEVbUwc      Male  22 high caffeine       230
## 40 i8MDEDiJHv    Female  23 high caffeine       201
## 41 5ROBC8QMSK Non-Binary  22 high caffeine        48
## 42 QGslHuqlDI      Male  23 high caffeine       255
```

```r
nrow(df_right)
```

```
## [1] 42
```

In this case, because every participant ID in `df_rt` has a matching response in `df_demographics`, we do not see any `NA` values.

You might be wondering why the hell would you want both a `left_join()` and a `right_join()` function. Couldn't we have just the one function, and just specify which order we want to join things together?

We could, but having the option of having `left_join()` and `right_join()` becomes handy when we have complicated and deeply nested code using the pipe `%>%` operator.

But nonetheless you may never have a need for both functions, but in case you do, you know it's there.

6.5.5   Outer Join

The **outer join**, also known as a **full join**, combines rows from both datasets, including all observations from both data frames and filling in missing values with NA where there are no matches. This type of join ensures that no data is lost, even if there are unmatched rows in either dataset.

Let's demonstrate the outer join using two new data frames: **df_scores** and **df_survey**. These data frames contain information on participants' test scores and survey responses, respectively. The **df_scores** data frame will have scores for participants 1-5, whereas the **df_survey** data frame will have scores for participants 3-7. So there will be some overlap in data, but also some areas where there is not matching scores.

```r
# Creating sample data frames
df_scores <- data.frame(
  ID = c(1, 2, 3, 4, 5),
  Test_Score = c(85, 92, 78, 90, 88)
)

df_survey <- data.frame(
  ID = c(3, 4, 5, 6, 7),
  Satisfaction = c("High", "Medium", "Low", "High", "Medium")
```

```
)
```

```
# Displaying the sample data frames
head(df_scores)
```

```
##   ID Test_Score
## 1  1         85
## 2  2         92
## 3  3         78
## 4  4         90
## 5  5         88
```

```
head(df_survey)
```

```
##   ID Satisfaction
## 1  3         High
## 2  4       Medium
## 3  5          Low
## 4  6         High
## 5  7       Medium
```

Now, let's join them together. The syntax for **full_join()** is: **full_join(df1, df2, by = join_by(column))**

```
# Performing outer join
df_outer <- full_join(df_scores, df_survey, by = "ID")
```

```
df_outer
```

```
##   ID Test_Score Satisfaction
## 1  1         85         <NA>
## 2  2         92         <NA>
## 3  3         78         High
## 4  4         90       Medium
## 5  5         88          Low
## 6  6         NA         High
## 7  7         NA       Medium
```

In the resulting **df_outer** data frame, all rows from both **df_scores** and **df_survey** are included, regardless of whether there was a match on the specified column (**ID**). Rows with no matching values are filled with NA.

The outer join is particularly useful when you want to retain all information from both datasets, even if there are inconsistencies or missing values between them. This ensures that you have a complete dataset for analysis, with all available information from each source preserved.

6.5.6   Summary

Your choice of join ultimately depends on your research questions, the nature of your data, and the analysis you intend to perform. But hopefully at this point you have an appreciation of the variety of ways you can merge data in R.

6.6   Data Wrangling Example (Demographic and Flanker Task)

In the last chapter, I asked you to clean a Flanker task. However, this flanker task was actually composed of separate flanker.csv files. This week we are going to take those three separate flanker files (the `df_flanker1`, `df_flanker2`, and `df_flanker3` data frames we loaded in earlier) and merge them together. Additionally, we are going to clean the associated demographic file (`df_background`).

First let's clean the `df_background file`, then we will clean the three `flanker_files`. At the end then we will merge them all together.

6.6.1   Part I: Cleaning the `df_background` file

First, let's have a look at the `df_background` data frame.

```r
head(df_background)
```

```
##   Event.Index UTC.Timestamp UTC.Date.and.Time Local.Timestamp Local.Timezone
## 1           1       1.71e+12  23/01/2024 12:36        1.71e+12              0
## 2           2       1.71e+12  23/01/2024 12:36        1.71e+12              0
## 3           3       1.71e+12  23/01/2024 12:36        1.71e+12              0
## 4           4       1.71e+12  23/01/2024 12:36        1.71e+12              0
## 5           5       1.71e+12  23/01/2024 12:36        1.71e+12              0
## 6           6       1.71e+12  23/01/2024 12:36        1.71e+12              0
##   Local.Date.and.Time Experiment.ID Experiment.Version      Tree.Node.Key
## 1    23/01/2024 12:36        161196                 3 questionnaire-ja96
## 2    23/01/2024 12:36        161196                 3 questionnaire-ja96
## 3    23/01/2024 12:36        161196                 3 questionnaire-ja96
## 4    23/01/2024 12:36        161196                 3 questionnaire-ja96
## 5    23/01/2024 12:36        161196                 3 questionnaire-ja96
```

```
## 6     23/01/2024 12:36        161196              3 questionnaire-ja96
##   Repeat.Key Schedule.ID Participant.Public.ID Participant.Private.ID
## 1        NA    34404119                 BLIND               10168827
## 2        NA    34404119                 BLIND               10168827
## 3        NA    34404119                 BLIND               10168827
## 4        NA    34404119                 BLIND               10168827
## 5        NA    34404119                 BLIND               10168827
## 6        NA    34404119                 BLIND               10168827
##   Participant.Starting.Group Participant.Status Participant.Completion.Code
## 1                         NA           complete                          NA
## 2                         NA           complete                          NA
## 3                         NA           complete                          NA
## 4                         NA           complete                          NA
## 5                         NA           complete                          NA
## 6                         NA           complete                          NA
##   Participant.External.Session.ID Participant.Device.Type Participant.Device
## 1                              NA                computer  Desktop or Laptop
## 2                              NA                computer  Desktop or Laptop
## 3                              NA                computer  Desktop or Laptop
## 4                              NA                computer  Desktop or Laptop
## 5                              NA                computer  Desktop or Laptop
## 6                              NA                computer  Desktop or Laptop
##   Participant.OS Participant.Browser Participant.Monitor.Size
## 1     Windows 10    Chrome 120.0.0.0                 1280x720
## 2     Windows 10    Chrome 120.0.0.0                 1280x720
## 3     Windows 10    Chrome 120.0.0.0                 1280x720
## 4     Windows 10    Chrome 120.0.0.0                 1280x720
## 5     Windows 10    Chrome 120.0.0.0                 1280x720
## 6     Windows 10    Chrome 120.0.0.0                 1280x720
##   Participant.Viewport.Size Checkpoint Room.ID Room.Order    Task.Name
## 1                 1280x559         NA      NA         NA Demographics
## 2                 1280x559         NA      NA         NA Demographics
## 3                 1280x559         NA      NA         NA Demographics
## 4                 1280x559         NA      NA         NA Demographics
## 5                 1280x559         NA      NA         NA Demographics
## 6                 1280x559         NA      NA         NA Demographics
##   Task.Version order.kx46 Randomise.questionnaire.elements.    Question.Key
## 1            1        ABC                                No BEGIN QUESTIONNAIRE
## 2            1        ABC                                No             Sex
## 3            1        ABC                                No    Sex-quantised
## 4            1        ABC                                No        Sex-text
## 5            1        ABC                                No             Age
## 6            1        ABC                                No    Age-quantised
##   Response
## 1
## 2    Female
```

```
## 3         1
## 4
## 5     18-24
## 6         1
```

Again, not exactly a data frame to write home to your parents about. There is a lot of cleaning we need to do here. Let's go through it step-by-step

First thing we need to do is select our columns as most of the default columns are unnecessary. The columns we need are:

- `Participant.Private.ID` - Participant's ID
- `Question.Key` - The question they were being asked.
- `Response` - Their response to that question.

Let's select those columns using `select()`. Remember that the syntax is: `select(dataframe, columns we want)`

```r
df_background_select <- select(df_background,
                               Participant.Private.ID,
                               Question.Key,
                               Response)

#remember to check it with head()

head(df_background_select)
```

```
##    Participant.Private.ID        Question.Key Response
## 1                10168827 BEGIN QUESTIONNAIRE
## 2                10168827                 Sex   Female
## 3                10168827        Sex-quantised        1
## 4                10168827            Sex-text
## 5                10168827                 Age    18-24
## 6                10168827       Age-quantised        1
```

Okay that's a lot easier to look at. If you inspect the values in the `Participant.Private.ID` column, you'll notice that there is a lot of repeated values. This is because this data is in long format rather than wide.

We are going to change that in a couple of steps. But the next thing we are going to do is fix our column names using the **rename()** function. Remember that the syntax is: **rename(df, new_column_name = old_column_name)**

```
df_background_rename <- rename(df_background_select,
                               ID = Participant.Private.ID,
                               Question = Question.Key)

head(df_background_rename)
```

```
##         ID               Question Response
## 1 10168827 BEGIN QUESTIONNAIRE
## 2 10168827                   Sex    Female
## 3 10168827         Sex-quantised         1
## 4 10168827              Sex-text
## 5 10168827                   Age     18-24
## 6 10168827         Age-quantised         1
```

Using just two functions we have significantly cleaned up our data frame by reducing its size and enhancing its readability.

Now we need to think about cleaning our rows. If you look at the values under the `Question` and `Response` columns, you will see relatively strange responses. Let's talk through the values in Question and their associated Response.

Question	Response
BEGIN QUESTIONNAIRE	This "response" is something Gorilla records in the data frame to help us quickly identify the start and end of participants' responses to a questionnaire. We do not need this response in our clean dataset, so we will be getting rid of it.
Sex	Participants were asked to select a drop-down choice to identify their sex as either male or female. This records that selection.
Sex-quantised	This translates the participant's response to a numerical value. The numbers correspond to the order in which they were displayed answers. Female was displayed first, so it gets recorded as a 1. Male was displayed second, so it gets recorded as a 2. We will get rid of this column because we just want the male or female response.
Sex-text	No idea, to be honest. But we don't need it either way.
Age	Participants asked to select a drop-down choice to identify their age across the categories 18-24, 25-30 up to 41-50. This records that selection.
Age-quantised	Again, this translates the participant's response to a numerical value in relation to the order options were displayed. Not needed.

Question	Response
Age-text	Again, no clue. But we do not need it.

We only need the rows where Question is equal to Age or Sex. Let's select only these rows by using the `filter()` function and the | operator.

```
df_background_filter <- filter(df_background_rename, Question == "Age" | Question == "
```

```
head(df_background_filter)
```

```
##          ID Question Response
## 1 10168827      Sex   Female
## 2 10168827      Age    18-24
## 3 10192092      Sex   Female
## 4 10192092      Age    31-40
## 5 10205485      Sex
## 6 10205485      Age    18-24
```

Now let's pivot our data frame from long to wide using our new friend the `pivot_wider()` function.

```
df_background_wide <- pivot_wider(df_background_filter,
                                  id_cols = ID,
                                  names_from = Question,
                                  values_from = Response)
```

```
head(df_background_wide)
```

```
## # A tibble: 6 x 3
##         ID Sex      Age
##      <int> <chr>    <chr>
## 1 10168827 "Female" 18-24
## 2 10192092 "Female" 31-40
## 3 10205485 ""       18-24
## 4 10208522 "Female" 18-24
## 5 10218310 "Female" 18-24
## 6 10225898 "Female" 18-24
```

Okay, we are not quite done yet. You will have noticed that there is a missing value for participant `10205485`, but it is not turning up as an `NA`. What is going on here?

To get a better idea, let's print out the values of both the `Sex` and `Age` columns.

```
df_background_wide$Sex
```

```
##  [1] "Female" "Female" ""       "Female" "Female" "Female" "Female" "Female"
##  [9] "Male"   "Male"   ""       "Female" "Female" "Male"   "Male"   "Female"
## [17] "Male"   "Female" "Female" "Female" "Female" "Male"   "Female" "Female"
## [25] "Male"   "Female" "Female" "Female" "Female" "Female" "Female" "Male"
## [33] "Female" "Female" "Female" "Female" "Female" "Female" "Female" "Female"
## [41] "Female" "Female" "Female" "Female" "Male"   "Female" "Female" "Female"
## [49] "Male"   "Female"
```

```
df_background_wide$Age
```

```
##  [1] "18-24" "31-40" "18-24" "18-24" "18-24" "18-24" "18-24" "18-24" "18-24"
## [10] "41-50" "18-24" "18-24" "18-24" ""      "18-24" "18-24" "18-24" "18-24"
## [19] "18-24" "18-24" "18-24" "18-24" "18-24" "18-24" "18-24" "18-24" "18-24"
## [28] "41-50" "31-40" "18-24" "18-24" "18-24" "31-40" "18-24" "18-24" "18-24"
## [37] "25-30" "18-24" "18-24" "18-24" "18-24" "18-24" "18-24" "18-24" "18-24"
## [46] "18-24" "18-24" "18-24" "18-24" "18-24"
```

If you look at values [3] and [11] in our Sex vector and [14] in our Age vector, what's happened is that Gorilla saved these data points as an empty character data (" "). Even an empty character is still counted as a character in R. We can remove these values using the `filter()` function and the & (AND) operator.

```
df_background_clean <- filter(df_background_wide, Age != "" & Sex != "")

head(df_background_clean)
```

```
## # A tibble: 6 x 3
##         ID Sex    Age
##      <int> <chr>  <chr>
## 1 10168827 Female 18-24
## 2 10192092 Female 31-40
## 3 10208522 Female 18-24
## 4 10218310 Female 18-24
## 5 10225898 Female 18-24
## 6 10228586 Female 18-24
```

Grand job, our background data frame is clean. Let's move on to cleaning our `df_flanker1`, `df_flanker2`, and `df_flanker3` data frames.

6.6.2   Cleaning our Flanker Tasks

If you are cleaning multiple data frames in R, one way you can do this is to clean each one individually and then merge the clean ones together. However, if multiple data frames are very similar in structure, it's better to merge these data frames together first and then clean it all in one go.

If you look at the structure of `df_flanker1`, `df_flanker2`, and `df_flanker3` you will notice that they all have the same variable names and types.

```
str(df_flanker1)
```

```
## 'data.frame':    376 obs. of  5 variables:
##  $ participant..id: int  10168827 10168827 10168827 10168827 10168827 10168827 10168
##  $ trial          : chr  "START" "Congruent" "Incongruent" "Congruent" ...
##  $ reaction.time  : num  NA 627 621 604 287 ...
##  $ stimulus       : chr  "positive" "positive" "positive" "positive" ...
##  $ condition      : chr  "ABC" "ABC" "ABC" "ABC" ...
```

```
str(df_flanker2)
```

```
## 'data.frame':    376 obs. of  5 variables:
##  $ participant..id: int  10168827 10168827 10168827 10168827 10168827 10168827 10168
##  $ trial          : chr  "START" "Congruent" "Incongruent" "Congruent" ...
##  $ reaction.time  : num  NA -2.6 206.9 625.3 309.9 ...
##  $ stimulus       : chr  "negative" "negative" "negative" "negative" ...
##  $ condition      : chr  "ABC" "ABC" "ABC" "ABC" ...
```

```
str(df_flanker3)
```

```
## 'data.frame':    376 obs. of  5 variables:
##  $ participant..id: int  10168827 10168827 10168827 10168827 10168827 10168827 10168
##  $ trial          : chr  "START" "Congruent" "Incongruent" "Congruent" ...
##  $ reaction.time  : num  NA 889 748 547 910 ...
##  $ stimulus       : chr  "neutral" "neutral" "neutral" "neutral" ...
##  $ condition      : chr  "ABC" "ABC" "ABC" "ABC" ...
```

Each data frame has the same number of columns (5 variables) with the same variable names and the same number of participants (rows). So rather than clean each one individually, let's combine them all into the one data frame and clean it there.

We can do that using `full_join()` since we want to keep all observations in each data frame. The only thing is that `full_join` only allows us to merge

two `dataframes` at a time, so we will need to first join `df_flanker1` with `df_flanker2` and call the result `df_flanker_proxy`. Then we will merge `df_flanker_proxy` with `df_flanker_total`.

```
## Joining with 'by = join_by(participant..id, trial, reaction.time, stimulus,
## condition)'
## Joining with 'by = join_by(participant..id, trial, reaction.time, stimulus,
## condition)'
```

```r
df_flanker_proxy <- full_join(df_flanker1, df_flanker2)

df_flanker_total <- full_join(df_flanker_proxy, df_flanker3)
```

When you do this, you might get a scary-looking message like this:

```
Joining with `by = join_by(participant..id, trial, reaction.time, task)`Joining with `by = join_b
```

That just means it is joining everything together.

Now let's look at our `df_flanker_total`

```r
head(df_flanker_total, n = 10) #n = 10, changes the number of rows printed out to 10
```

```
##      participant..id       trial reaction.time stimulus condition
## 1           10168827       START            NA positive       ABC
## 2           10168827   Congruent       627.183 positive       ABC
## 3           10168827 Incongruent       621.178 positive       ABC
## 4           10168827   Congruent       604.430 positive       ABC
## 5           10168827 Incongruent       287.365 positive       ABC
## 6           10168827   Congruent       284.131 positive       ABC
## 7           10168827 Incongruent       592.393 positive       ABC
## 8           10168827         END            NA positive       ABC
## 9           10192092       START            NA positive       ABC
## 10          10192092   Congruent       661.832 positive       ABC
```

Now, we can start cleaning the merged data frame. Here's a breakdown of the cleaning steps:

1. **Rename Columns**: We start by renaming columns for clarity and consistency.

```r
#We're renaming some columns in our combined data frame to give them better names that
# For example, changing "participant..id" to just "ID" and "reaction.time" to "rt" for

df_flank_rename <- rename(df_flanker_total,
                          ID = participant..id,
                          rt = reaction.time)
```

2. **Filter Rows**: We remove unnecessary rows with the values **START** and
   END.Again these values are Gorillas way to indicate to us when a partici-
   pant stated the task. But we do not need them in our clean data set.

```r
# We're only interested in rows where the trial is either "Congruent" or "Incongruent"

df_flank_filter <- filter(df_flank_rename,
                          trial == "Congruent" | trial == "Incongruent")
```

3. **Group Data**: We will need to calculate mean reaction times score for
   congruent and incongruent trials. But first we need to tell R that we want
   to group scores based on participants ID and the trial they were in

```r
df_flank_group <- group_by(df_flank_filter, ID, trial)
```

4. **Calculate Mean Reaction Time**: We calculate the mean reaction time
   for each participant-trial combination using the `mutate()` function.

```r
#Here, we're calculating the average (mean) reaction time for each participant and tri
# This will give us a better idea of how participants performed in different trial con

df_flank_rt_average <- mutate(df_flank_group,
                              mean_rt = mean(rt),
                              .keep = "unused")
```

5. **Remove Duplicates**: Now, we're removing any duplicate rows in our
   data frame to make sure we're not double-counting any participants.

```r
df_flank_distinct <- distinct(df_flank_rt_average)
```

6. **Reshape Data to Wide Format**: We pivot the data frame from long
   to wide format to make it easier on the eye.

```r
df_flank_wide <- pivot_wider(df_flank_distinct,
  id_cols = c(ID, condition),
  names_from = c(trial, stimulus),
  values_from = mean_rt,
  values_fn = list(mean = mean)
)
```

7. **Calculate Flanker Effect**: Finally, we calculate the Flanker effect for each participant. We can use mutate again here. The flanker effect is the congruent trial minus the incongruent trials

```r
df_flank_effect <- mutate(df_flank_wide,
                    flanker_effect_pos = Congruent_positive - Incongruent_positive,
                    flanker_effect_neg = Congruent_negative - Incongruent_negative,
                    flanker_effect_neut = Congruent_neutral - Incongruent_neutral,
                    .keep = "unused")
```

8. Remove **Missing Values (NA)**: If you `View(df_flank_clean)` you will see some missing values, we can remove them using the `na.omit()` function.

```r
df_flank_clean <- na.omit(df_flank_effect)
```

Boom, there we have it, the cleaned version of our `flanker` data frames.

6.6.3 Merging the Data Frames

For our final step, we can merge our two data frames together into `clean_df`

```r
clean_df <- inner_join(df_background_clean, df_flank_clean, by = join_by(ID))

head(clean_df)
```

```
## # A tibble: 6 x 7
##          ID Sex    Age   condition flanker_effect_pos flanker_effect_neg
##       <int> <chr>  <chr> <chr>                  <dbl>              <dbl>
## 1 10168827 Female 18-24 ABC                    -35.2              -35.2
## 2 10192092 Female 31-40 ABC                      5.37               5.37
## 3 10208522 Female 18-24 ABC                    116.               116.
## 4 10218310 Female 18-24 ABC                    -99.4              -99.4
## 5 10225898 Female 18-24 ABC                   -184.              -184.
## 6 10228586 Female 18-24 ABC                    -72.0              -72.0
## # i 1 more variable: flanker_effect_neut <dbl>
```

6.7   Summary

That concludes our two sessions on data cleaning. You have covered a lot in these two sessions. The lessons you have learn here are applicable to cleaning the majority of the datasets you will encounter in your research. Well done.

In the next session, we will learn how we can use R to create nice data visualisations. See you then.

Chapter 7

Data Visualisation in R

In this session, we are going to learn how to generate APA style plots in R. In particular, we are going to learn about the `ggplot2` package and it's associated function `ggplot()`. This package has been used to create plots for publications like the BBC and the Economist. By the end of this session you should be capable of:

- Understanding the logic of the ggplot method for drawing plots.
- Generating and customising elegant Box Plots, Violin Plots, Bar Charts, Scatterplots, Histograms, and Line Charts.
- Making your plots APA ready.
- Arranging and faceting (grouping together) your plots.
- Export your plots to PDFs.

7.1   Let's Get Set Up

Open up RStudio or Posit Cloud and complete the following activities to get set up for today.

7.1.1   Activity: Set Up Your Working Directory

In your folder for this class, create a new folder called `week6`. Set this folder as your working directory. To do this, click `Session` -> `Set Working Directory` -> `Choose Directory`, then find your new folder, select it and click `Open`.

In your console, the path to that folder should now be printed. It should look something like this:

```
setwd("C:/Users/0131045s/Desktop/Programming/R/Workshops/rintro/week6") #this specific
```

7.1.2   Activity: Prepare your R Script

Now in RStudio (or posit cloud), create an R script (try the keyboard shortcut: Ctrl-Shift-N on Windows, or command–shift–N on Mac) called `06-data-visualisation.R`.

7.1.3   Activity: Install and Load Your Packages

We will be using the `ggplot2`, `jtools`, and `patchwork` in today's session. Luckily, ggplot2 comes with Tidyverse, so we won't need to install it if you have installed Tidyverse already. We will need to install `patchwork`. Copy and paste the following code to your R script. Copy and paste the code (minus the #) `install.packages("patchwork")` into your console first and press enter. Once that is installed, you can run the following code.

```
#install.packages("tidyverse") if tidyverse does not load for you, then you will need
library(tidyverse)

#install.packages(c("jtools", "patchwork"))
library(jtools) #this package enables us to make APA themed plots
library(patchwork) #this package enables us to arrange plots we have created
```

7.1.4   Activity: Download and Import Your Files

We are going to need the following files for today's session. You will find them in the `Teams Channel under Channel 6 - Data Visualisation`. The files are:

- `06-data-visualisation.R`
- `personality_RD.csv`

Download these files onto your computer.

Now let's load in our data file. Make sure all your files are in your `week6` folder. Once they are, copy and paste the following code.

```
read.csv("personality_RD.csv")
```

7.2 Introduction to ggplot2

We analysed the `sleep` data frame in our first session. We created and exported a plot using the `base R plot()` function

```r
plot(sleep$group, sleep$extra,
     xlab = "Treatment Group",
     ylab = "Number of Extra Hours of Sleep",
     main = "Effect of Treament on Sleep Duration")
```

Effect of Treament on Sleep Duration
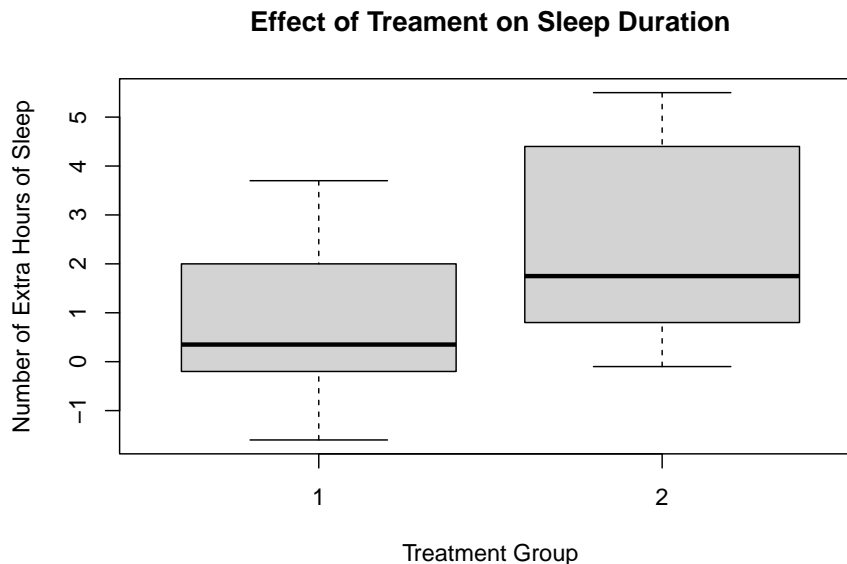
This is a perfectly fine plot, but one of the major advantages of using R over other programming languages or other statistical software is that it enables you to develop customisable, elegant, and even interactive plots. This is thanks to the `ggplot` package and the `ggplot2()` function.

What does the gg in ggplot stand for?

The **gg** stands for the *G*rammar of *G*raphics. This is because ggplot is built upon a logical system of how to draw a plot. This system involves a logical order where you incrementally add different layers to your plot. By understanding these system, you'll be able to create excellent plots in no-time. Luckily, this structure is relatively straightforward to understand.

7.3   How to Draw a Plot

We are going to recreate the plot we made in the first session using `ggplot`. After that, I am going to show you how we can use this function to make our plot more visually striking and informative using the same function. We will be using the `sleep` data frame again, but I am going to refer to as `df` for short.

```
df <- sleep
```

7.3.1   First we set up the Canvas

The first thing we do when we want to create a plot is call the `ggplot()` function and tell it what data frame we are working from. In this case, we are using the `sleep` data frame. For short, I am going to refer it as `df` and tell call `ggplot()`

```
ggplot(df)
```
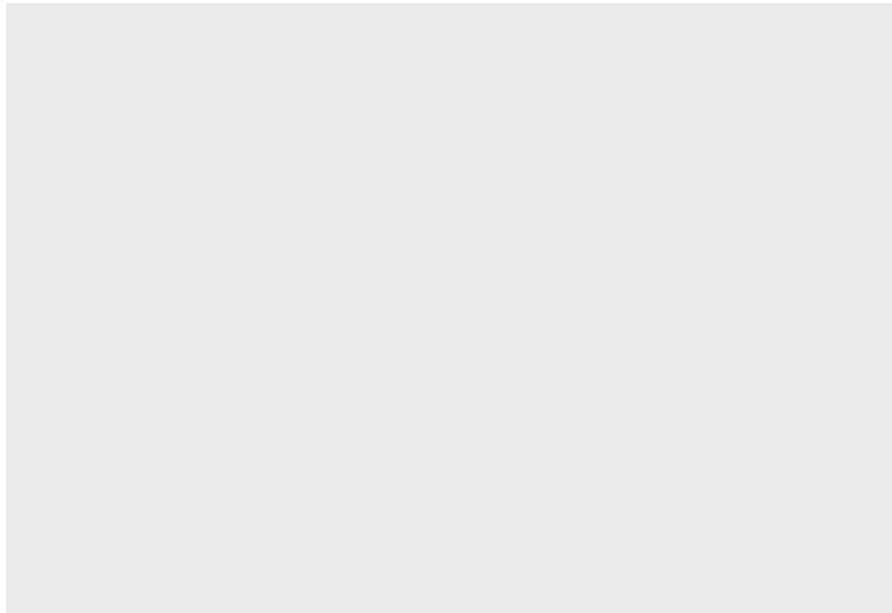
This creates a grey canvas where we can draw our plot on. The default R canvas is grey, but we can the appearance of the canvas later on.

Now that our canvas is set up, we will want to specify some aesthetic properties to our plot, like the y-axis and x-axis. To achieve this, we need to tell R to *map* a variable in our `df` to the **x-axis** and another variable to the **y-axis**. In ggplot, there is an argument called `mapping = aes()` that enables us to do this, where

aes is short for aesthetics. Let's may the **group** variable to the x-axis and the **extra** variable to the y-axis.

```
ggplot(df, mapping = aes(x = group, y = extra))
```
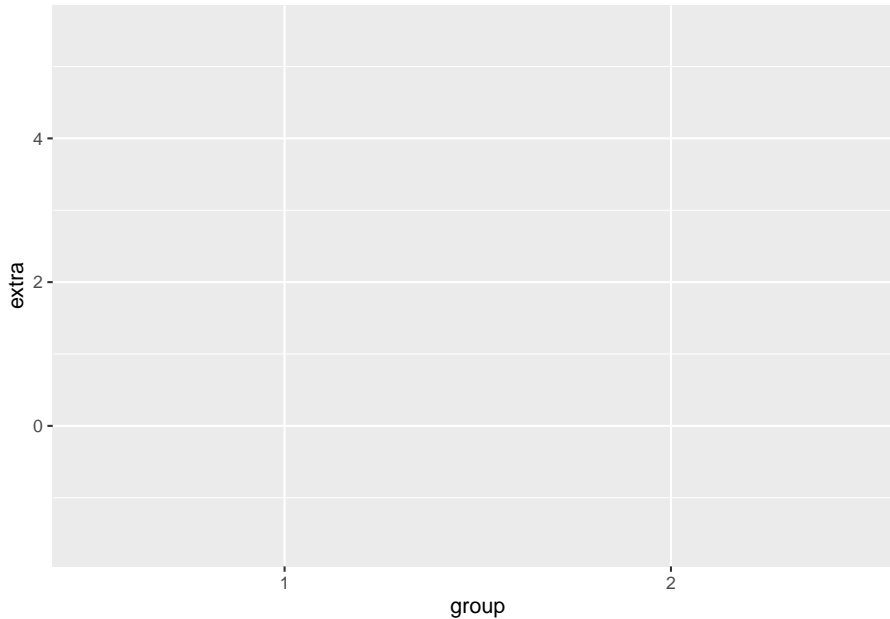

Now we can see that our x-axis is mapped to the two values in our **group** variable, whereas the y-axis is mapped to the range of values in the **extra** variable.

7.3.2 Creating our Box Plot

This sets up the shape of our canvas, the next thing we need to do is specify what **type** of plot we want to create. In ggplot, this means draw a **geom** (i.e., geometrical shape) onto our plot. There are dozens of geoms (see table at end of the chapter) that we can draw to our plot and we can draw multiple at the same time.

Since we are creating a boxplot, we'll use the **geom_boxplot** geom.

```
ggplot(df, mapping = aes(x = group, y = extra)) +
  geom_boxplot()
```
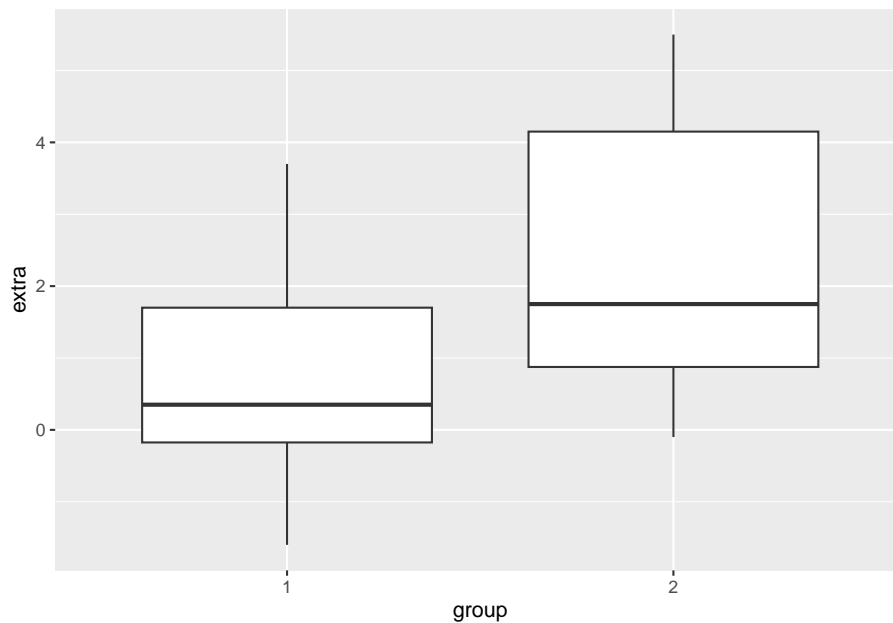
We can see that R has now drawn box plots for each of our groups. The default style in `ggplot()` is not to add the whiskey horizontal lines (e.g., the T) at the top and end of each boxplot. Generally, I am happy enough with the default option, but since we are recreating our first boxplot, let's add these whisker lines.

To do this, we need to tell R to create a shape based on statistical properties of our data. In particular, we need to create a statistical `errorbar` for a box plot. We can do this through adding the following line of code in our plot.

```r
ggplot(df, mapping = aes(x = group, y = extra)) +
  stat_boxplot(geom ='errorbar') +
  geom_boxplot()
```
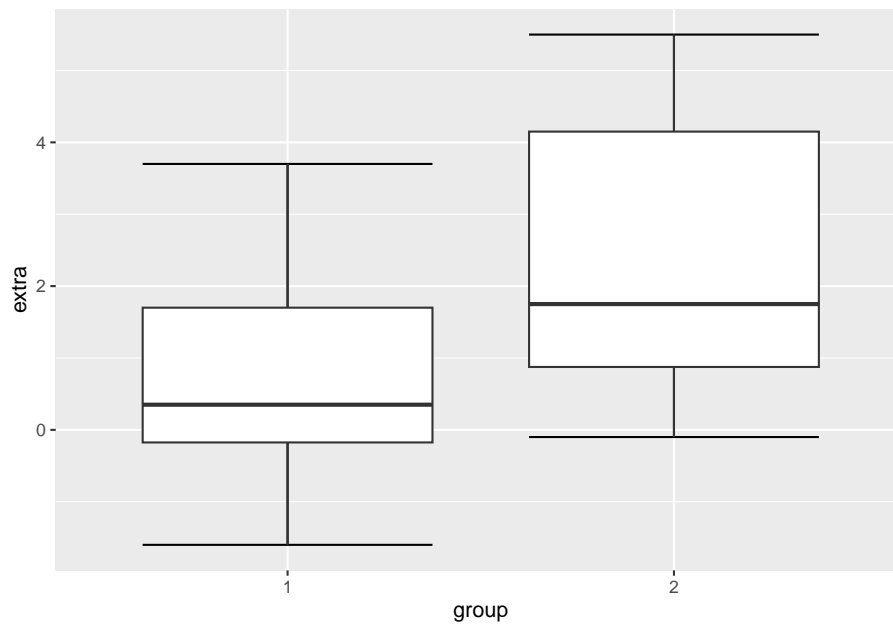
Now we have our whisker lines. Now I am not perfeclty happy with the width of these error lines. We can change that by specifying `width` in our `stat_boxplot()` function

```
ggplot(df, mapping = aes(x = group, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot()
```
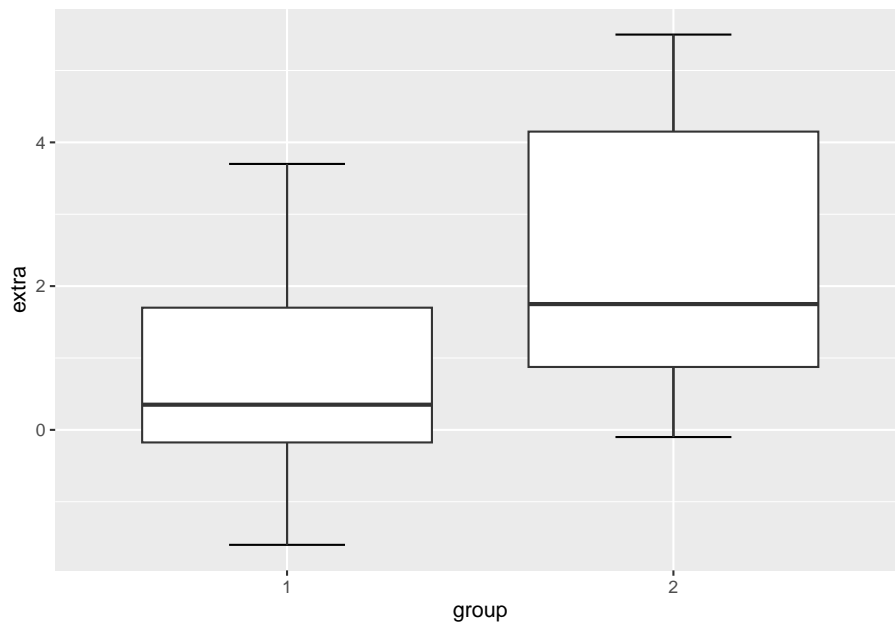
7.3.3   Changing the Name of Our X-Axis and Y-Axis

Okay, our plot is looking better. The next thing we will want to do is add informative labels to our x and y-axis. We can do this by using the ggplot functions `scale_x_discrete` and `scale_y_continous` to draw our labels.

```r
ggplot(df, mapping = aes(x = group, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot() +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep")
```
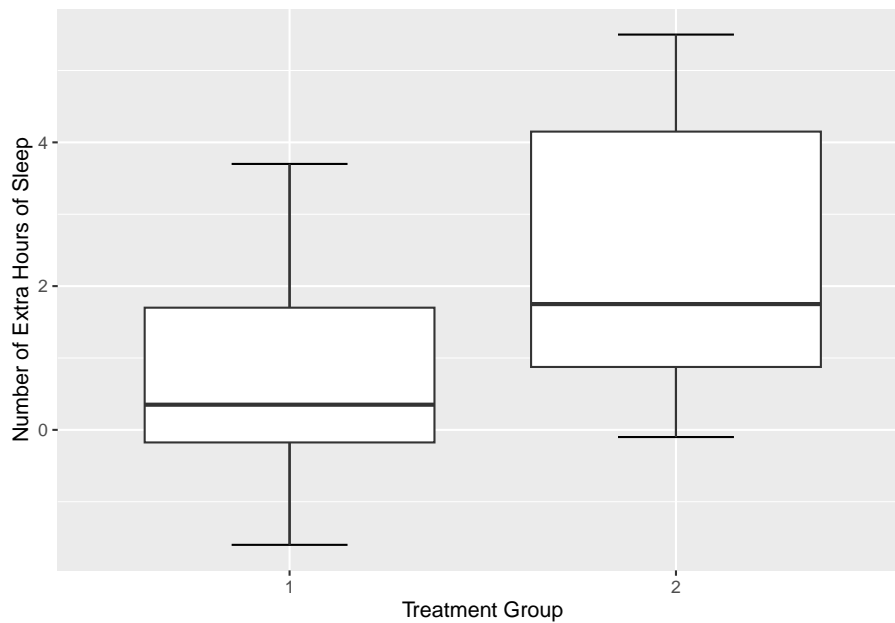
There now we have our x and y-labels. One thing that is bothering me is that our treatment group is labelled as `1` and `2`. The sleep data frame does not provide us with any information on what each treatment group received. So I am going to take artistic liberties and say that `1` means `Xanax` and `2` means `Ambien`.

There are two aproaches we can take to add this to our plot.

The first approach would be to add labels to the x-axis, in `scale_x_discrete()`.

```
ggplot(df, mapping = aes(x = group, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot() +
  scale_x_discrete(name = "Treatment Group",
                   labels = c("1"  = "Xanax", #this changes the 1 in the x-axis to Xanax
                              "2" = "Ambien")) +
  scale_y_continuous(name = "Number of Extra Hours of Sleep")
```
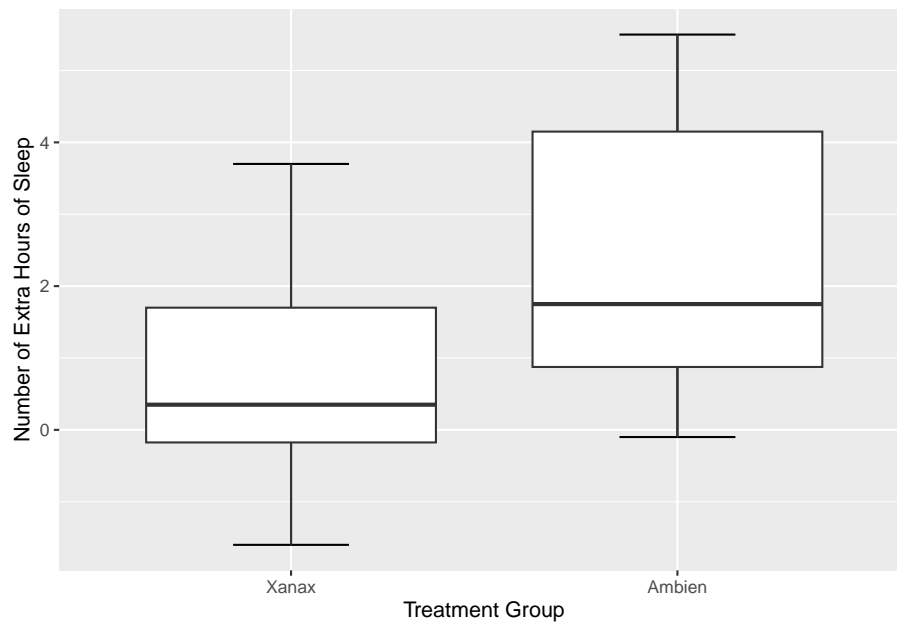
The second approach would be to first change the data frame itself. We can
do this using our old friend `mutate()`. I am going to create a variable called
`treatment` that `recode()` the values in the `group` variable.

```r
df <- mutate(df, treatment = recode(group, #recode changes variable values
                                    `1` = "Xanax",
                                    `2` = "Ambien"))

df$treatment
```

```
##  [1] Xanax  Xanax  Xanax  Xanax  Xanax  Xanax  Xanax  Xanax  Xanax  Xanax
## [11] Ambien Ambien Ambien Ambien Ambien Ambien Ambien Ambien Ambien Ambien
## Levels: Xanax Ambien
```

Now we can recreate our plot, but this time put `treatment` in the `x-axis`
instead of group.

```r
ggplot(df, mapping = aes(x = treatment, y = extra)) + #substitute treatment for group
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot() +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep")
```
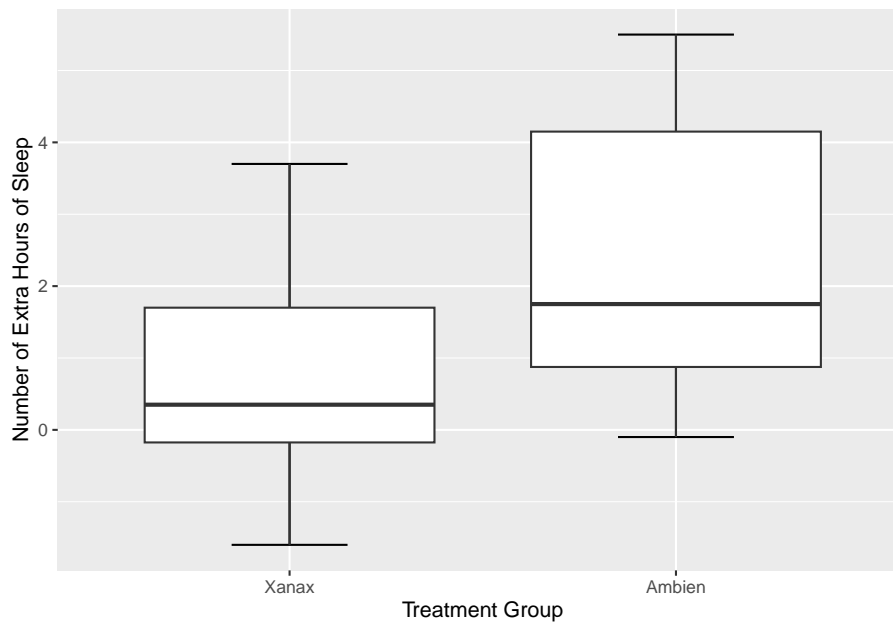
This produces the same plot. In this example, I prefer the second method. But I will explain why a little bit later.

Now, let's change the look of our canvas.

7.3.4   Changing the Look (Theme) of Our Canvas

One of the nice features of ggplot() is can change the `theme` of our canvas. There are several themes that we can use (see table at the end of the chapter for more information). The current theme we are using is `theme_gray`, which is the default theme.

```
ggplot(df, mapping = aes(x = treatment, y = extra)) + #substitute treatment for group
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot() +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_gray()
```
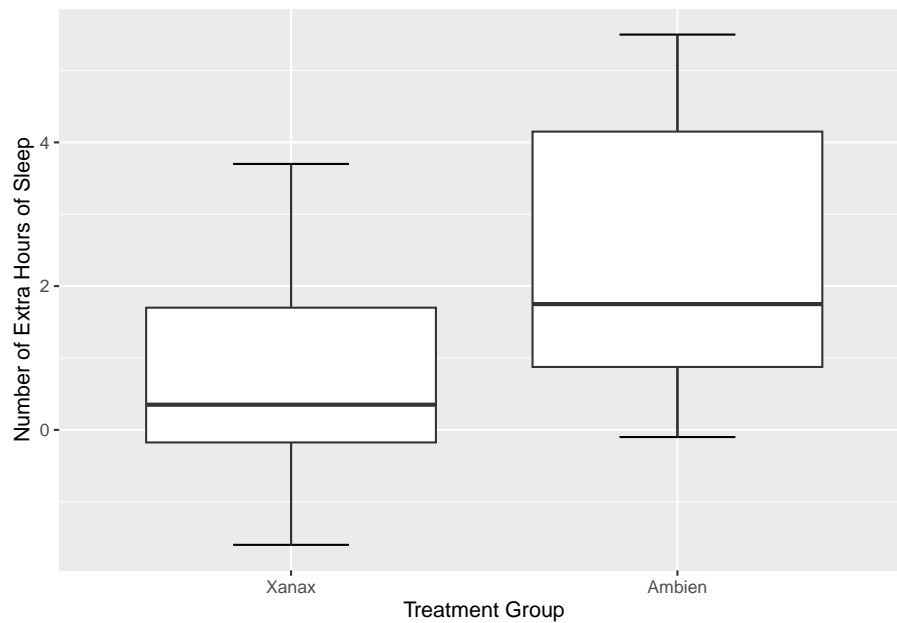
That is perfectly fine, but I dislike the grey. We could also set it to `theme_bw`
(white background and black gridlines).

```
ggplot(df, mapping = aes(x = treatment, y = extra)) + #substitute treatment for group
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot() +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_bw()
```
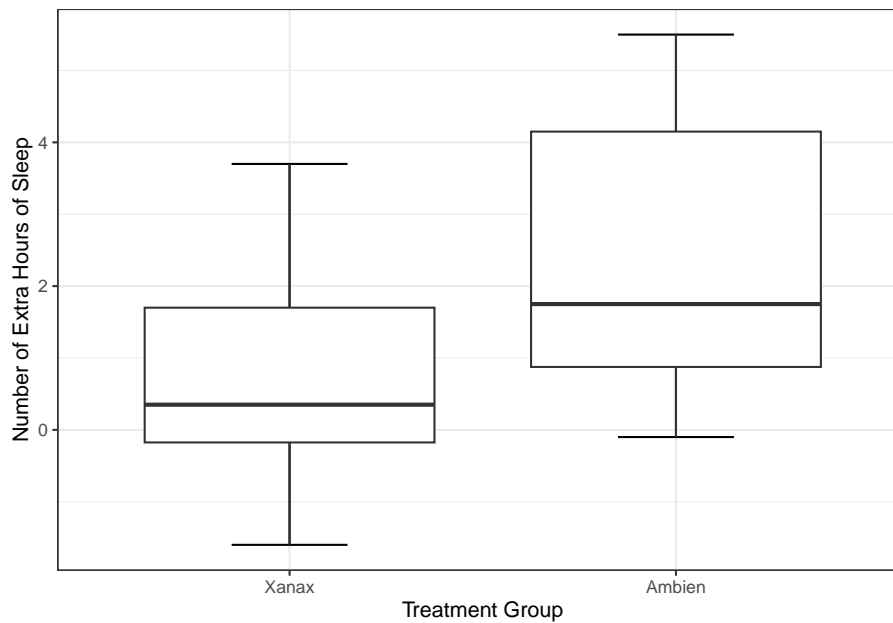
We could set it to a dark theme, using `theme_dark()`

```r
ggplot(df, mapping = aes(x = treatment, y = extra)) + #substitute treatment for group
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot() +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_dark()
```
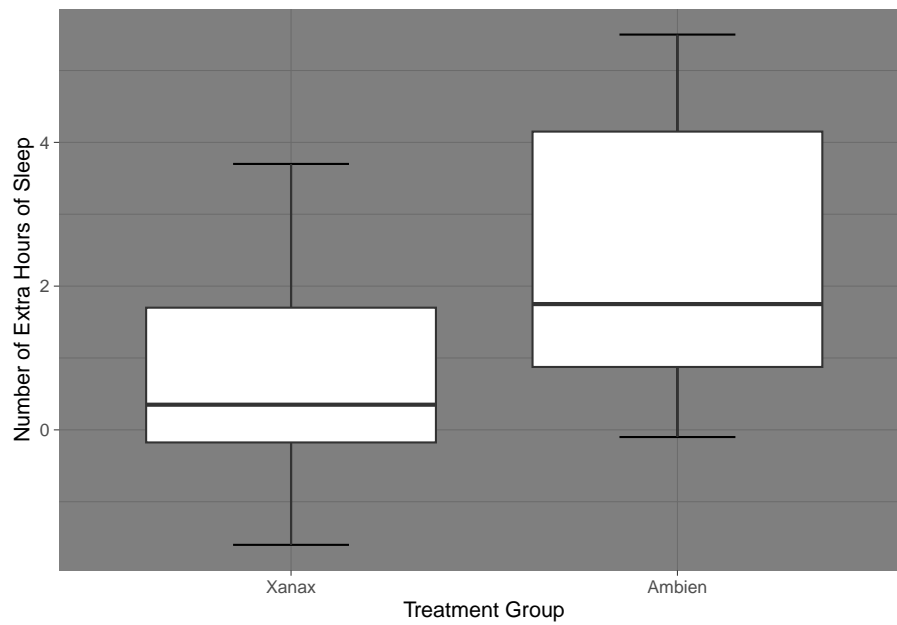
Or we could remove the grid lines and have a more `classic` approach, using `theme_classic()`

```
ggplot(df, mapping = aes(x = treatment, y = extra)) + #substitute treatment for group
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot() +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_classic()
```
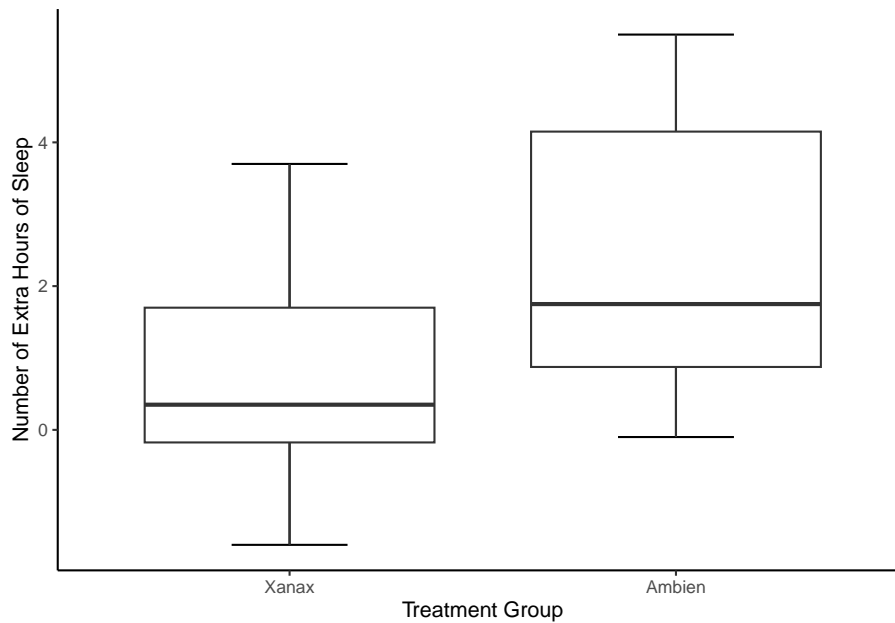
Since we are psychologists, we will mostly looking for plots in **APA** style. When you download `ggplot()` there is no pre-installed theme that matches this style. However, the **jtools** package we installed and loaded comes with an `apa_theme()`. Make sure that is loaded before running the following code:

```
ggplot(df, mapping = aes(x = treatment, y = extra)) + #substitute treatment for group
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot() +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_apa()
```
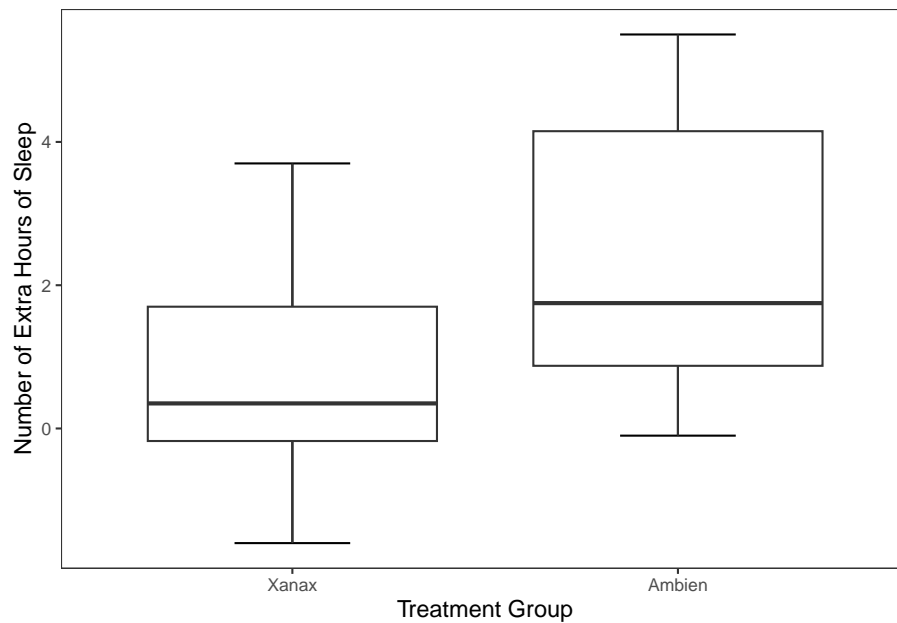
And now we have a pretty nice looking plot. Let's change the colour, to match our original plot we created in week 1 by adding `fill = "grey"` inside `geom_boxplot()`.

```r
ggplot(df, mapping = aes(x = treatment, y = extra)) + #substitute treatment for group
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(fill = "grey") +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_apa()
```
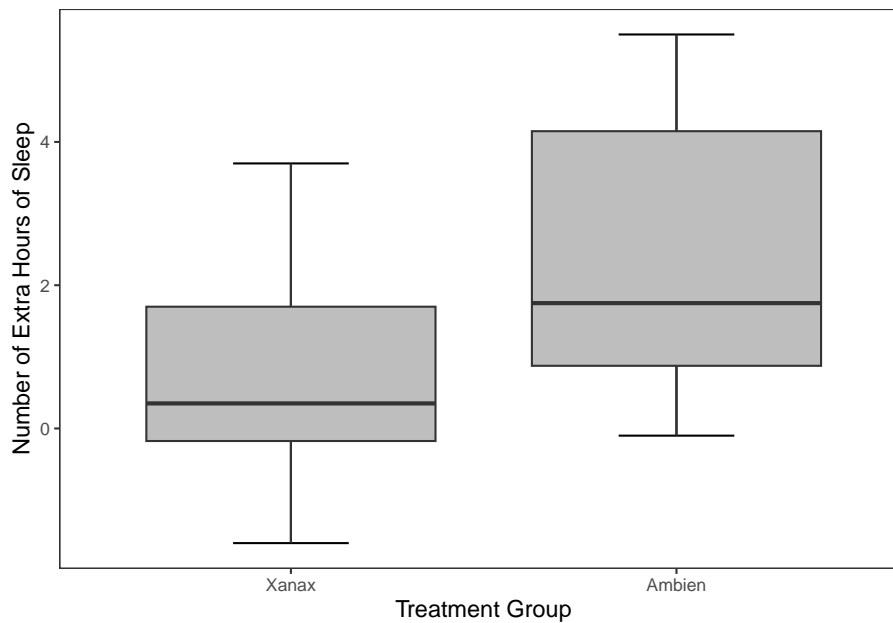
7.4 The Real Power of ggplot - Customisation

You might be wondering right now how useful is `ggplot` really. I mean, if you compare the two code chunks to create essentially the same plot, you'll notice that approach we took in week 1 is significantly shorter.

```r
#base R approach

plot(sleep$group, sleep$extra,
     xlab = "Treatment Group",
     ylab = "Number of Extra Hours of Sleep",
     main = "Effect of Treament on Sleep Duration")

#ggplot approach

ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(fill = "grey") +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_apa()
```

The real power of ggplot is the ability to customize our graphs to make them

more striking and informative. We have seen glimpses of this already with the
ability to add labels, colour, and themes to our plots. In this section, I am going
to show you more ways we can customise our plot.

7.4.1   Mapping Aesthetic Properties (like Colour and Fill) to Our Variables

In the last section, I used the argument `fill = grey` to specify the colour of
boxplots. If I wanted multiple colours, I could have used the `c()` function in
that argument:

```
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(fill = c("green", "orange")) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_apa()
```
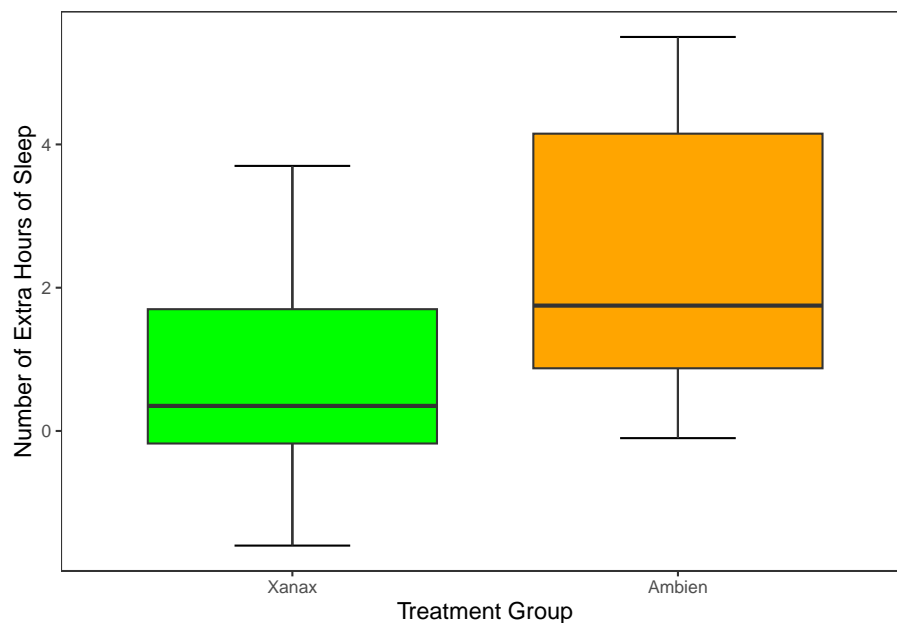

This approach is okay if are only specifying a limited number of colours, but
if there are several colours we need to specify, it is cumbersome. Luckily, we
can ask R to map the colours of boxplot to specific values in our data frame.
We do this through a similar approach used in `ggplot()` where we add the
argument `mapping = aes()` to our `geom_boxplot()` function. This time inside
the `aes()` argument, we specify that we want the `fill` (the colour inside our
boxplots) to map to the variable `treatment`.

```
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep") +
  theme_apa()
```
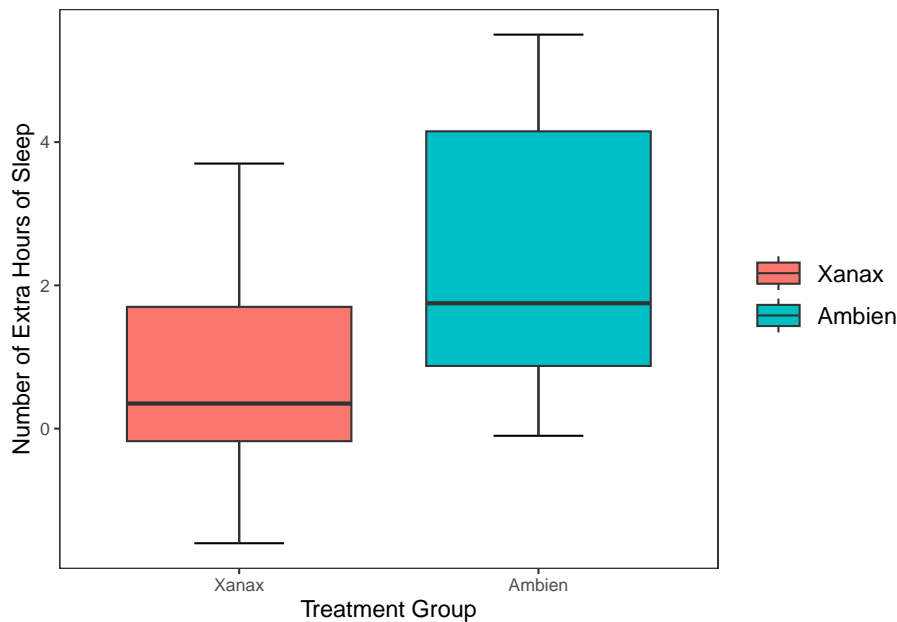

R will choose colours that are visually distinct from each other. Additionally, it will add a legend to our graph. If you want to remove the legend, add `show.legend = FALSE` to the `geom_boxplot()` function.

7.4.2   Changing the Value of Our Y-Axis

I can tell R to specify the number of breaks on the y-axis. At the moment, it is only showing breaks in increments of two. R will try find a straightforward solution to the number of points on the y-axis. We can override this by using the `breaks()` argument in `scale_y_continous`, which will add a break between each number specified.
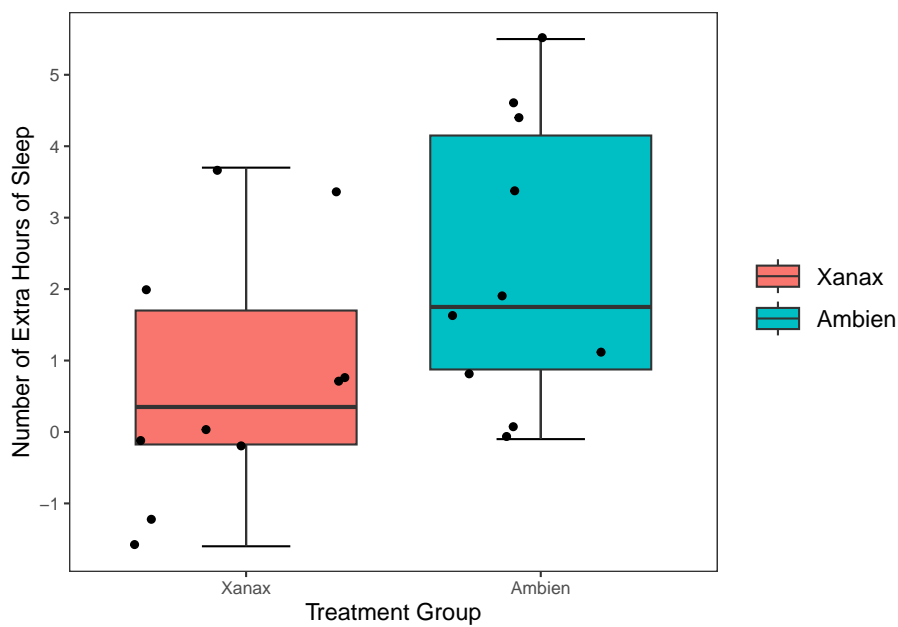
```
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_x_discrete(name = "Treatment Group") +
```

```
scale_y_continuous(name = "Number of Extra Hours of Sleep",
                   breaks = c(-2:6) #this will add a break for each value between -2
                   ) +
theme_apa()
```
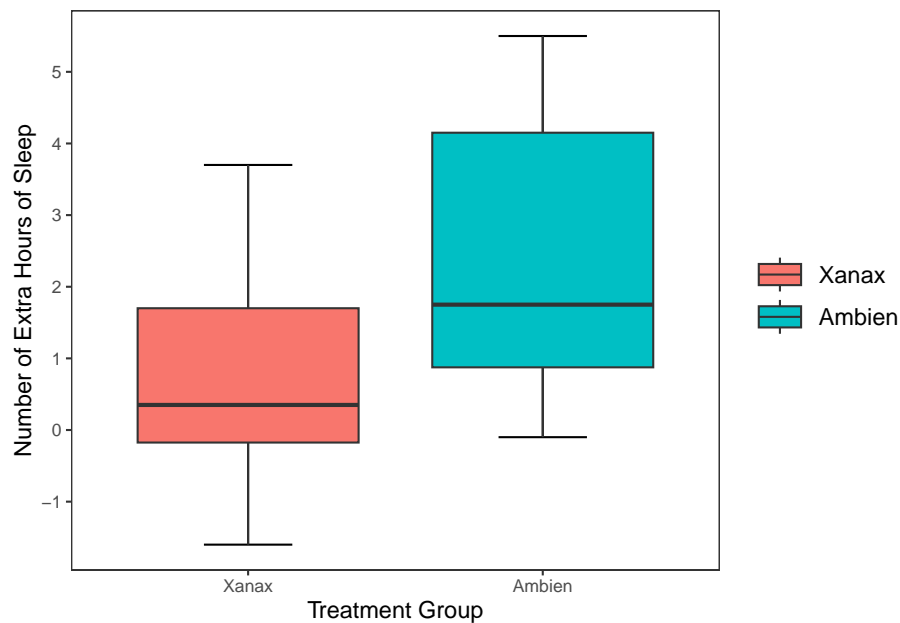

Changing the Orientation

We can also change the orientation of our graph in `ggplot()`. All we need to
do is change the `x` and `y` values in the `ggplot()` call. And then we just need to
change `scale_y_continuous` to `scale_y_discrete`, and `scale_x_discrete` to
`scale_x_continuous`.

```
ggplot(df, mapping = aes(x = extra, y = treatment)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_y_discrete(name = "Treatment Group") +
  scale_x_continuous(name = "Number of Extra Hours of Sleep",
                     breaks = c(-2:6) #this will add a break for each value between -2
                     ) +
  theme_apa()
```
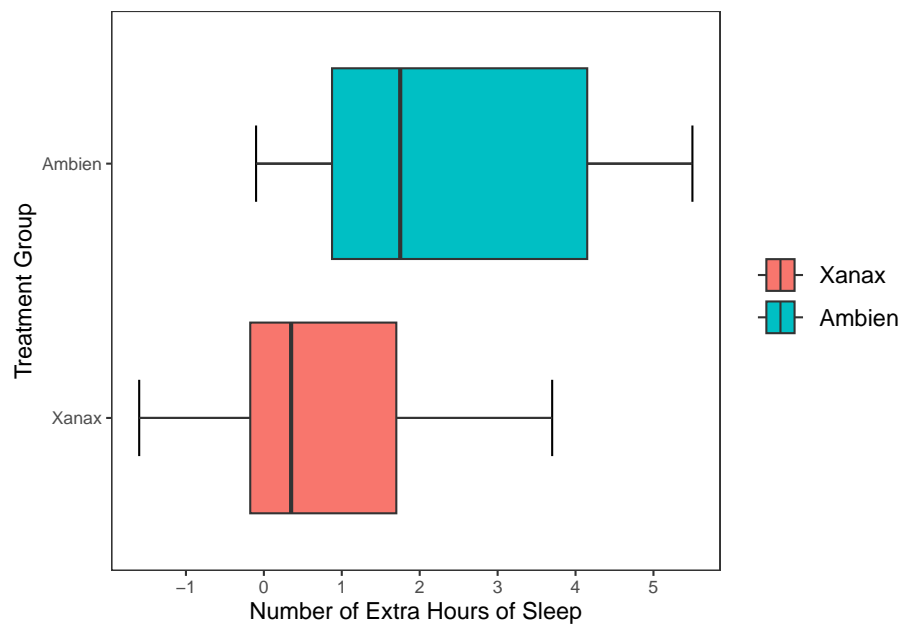
This option is really handy if you have lots of different categorical groups and your struggling to fit your graph onto the page. Let me demonstrate with the `mpg` data set that is already installed when you download R.

```
#This first plot if really squished together on the x-axis

ggplot(mpg, mapping = aes(x = manufacturer, y = cty)) + #cty = number of cylinders
  geom_boxplot()
```
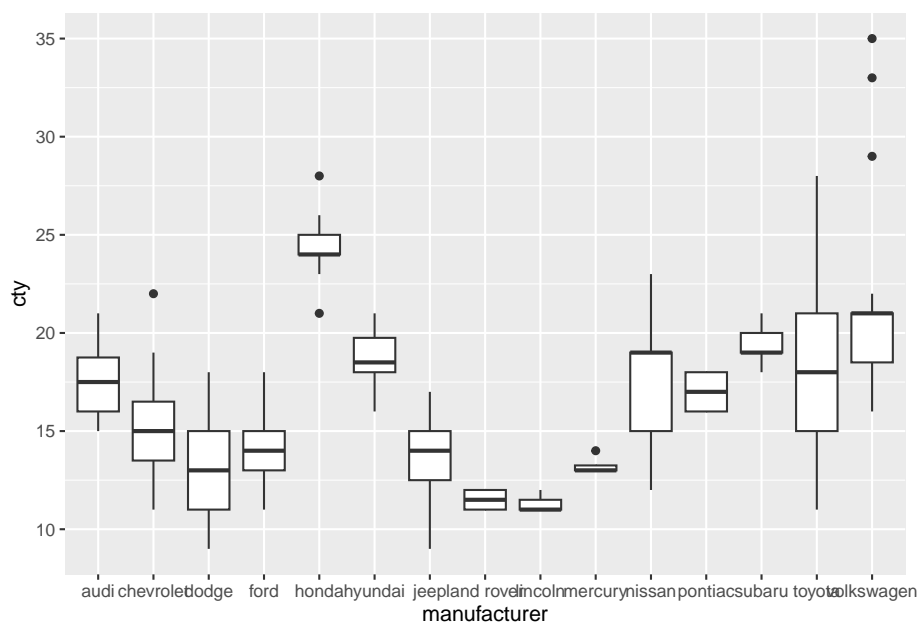
```
#but if we pivot it, then it looks much nicer

ggplot(mpg, mapping = aes(x = cty, y = manufacturer)) +
  geom_boxplot()
```
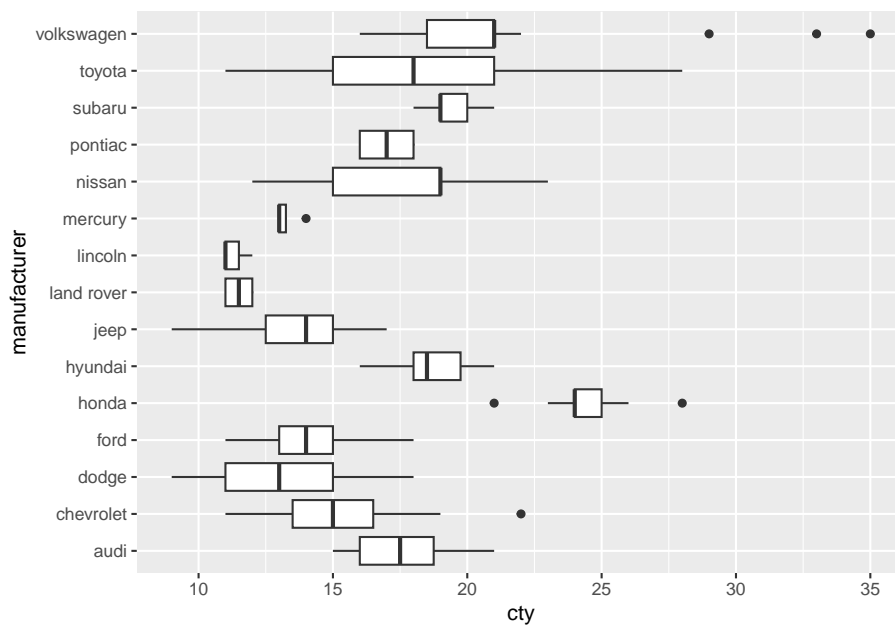
7.4.3 Plotting our Data Points in the Graph

What if I wanted to add individual data points to our graph? To provide more information on the scatter of scores? There are two options I can use. The first option is to the use the `geom_point()`, which will each participant's data point to the graph. Since there are only two possible observations in the `x-axis`, all data points will be printed in a straight line for each observation.

```
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep",
                     breaks = c(-2:6) #this will add a break for each value between -2 and +6
                     ) +
  theme_apa() +
  geom_point() #will add individual scores onto to the graph
```
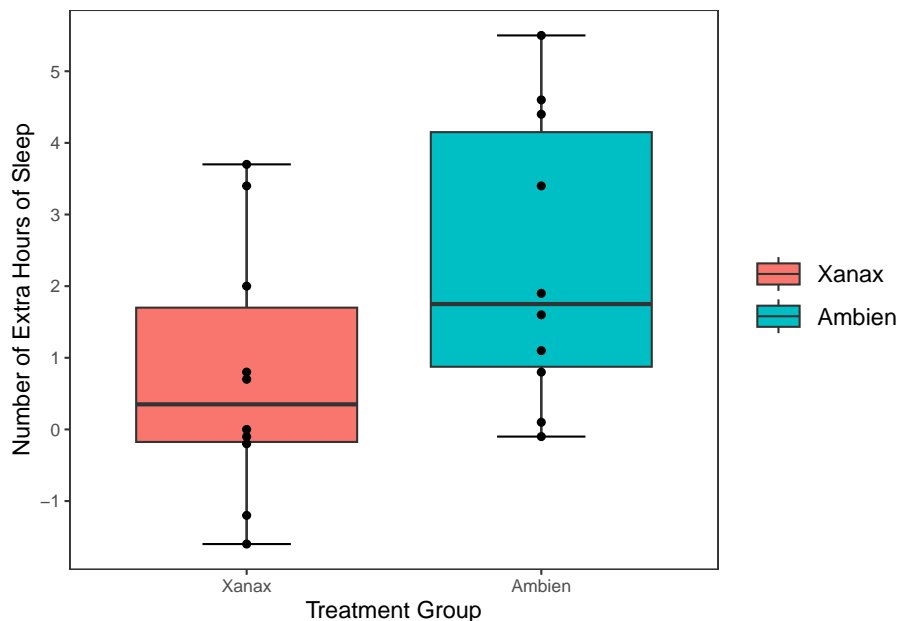

This is a perfectly legitimate approach to take. There is not a lot of data, so we can make our each individual point, even if there is some overlap. However, we can use another approach called `geom_jitter()`. This will plot each individual point just like `geom_point()` does, but it will add some random movement (i.e. a jitter) to each point. This can prevent overplotting of individual points.

```r
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep",
                     breaks = c(-2:6) #this will add a break for each value between -2
                     ) +
  theme_apa() +
  geom_jitter() #will add individual scores onto to the graph and give them space away
```


The added space left or right for each data point is randomly generated. But
we can reduce the upper and lower bounds of that random generation. Let's do
that for our current plot.

```r
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep",
                     breaks = c(-2:6) #this will add a break for each value between -2
                     ) +
  theme_apa() +
  geom_jitter(width = .20) #changes the horizontal jitter
```
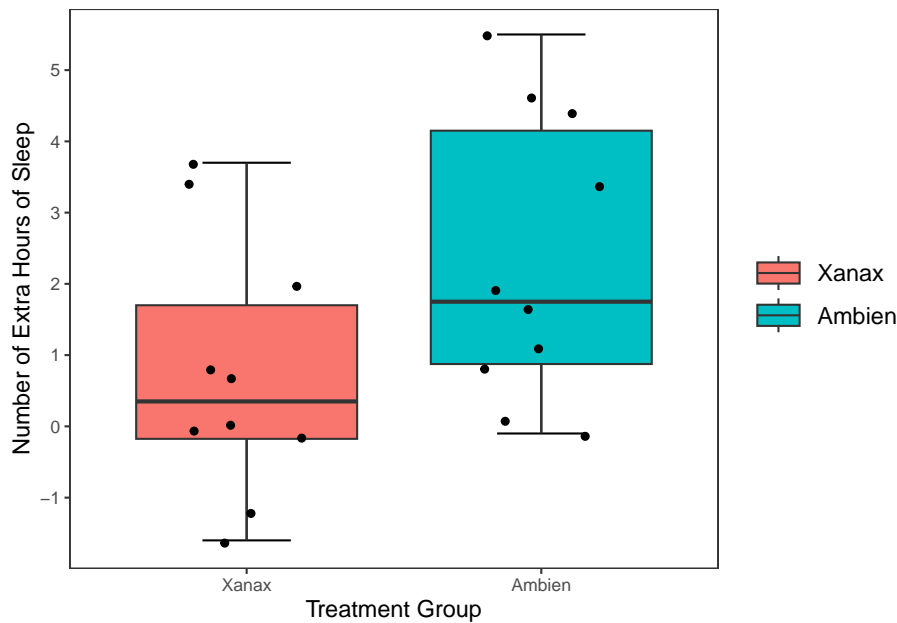
7.4.4   Adding Statistical Information to Our Plot

We can also add statistical summary information to our plot. Right now our boxplot tells us about individual scores, the median score, and the range of values. What if we wanted it to visualise the mean score treatment group?

No problem. To do this, we need to tell R to draw a `geom` shape in the position of the mean score. The easiest `geom` to do this with is `geom_point()`.

```
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep",
                    breaks = c(-2:6) #this will add a break for each value between -2 and +6
                    ) +
  theme_apa() +
  geom_jitter(width = .20) +
  geom_point(stat = "summary", fun = "mean", size = 5, colour = "black")
```
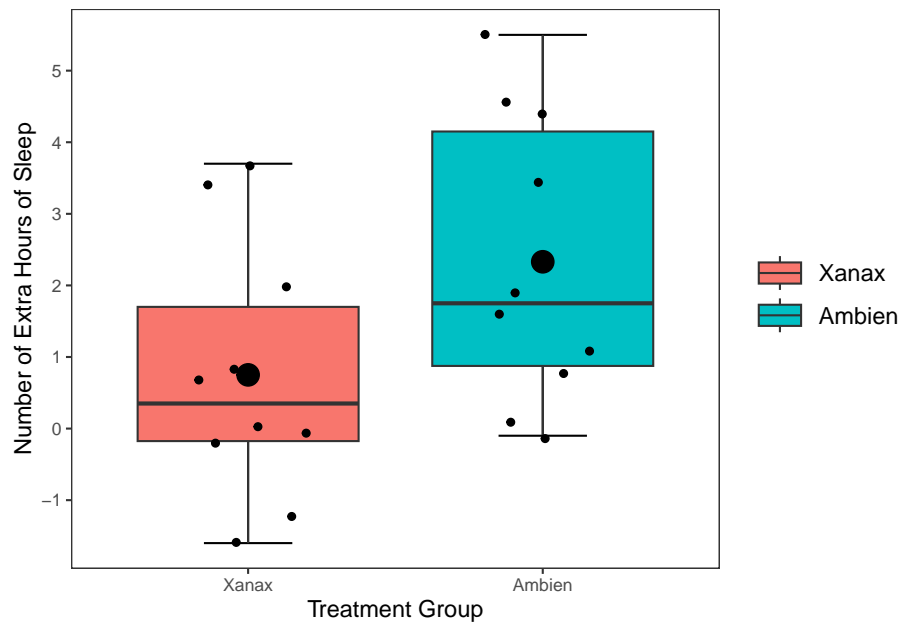
This draws a dot exactly where the mean value falls for both the `Xanax` and `Ambien` treatment groups fall. I changed the size of the point to make it more visible salient than the other data points. We could have also change the colour (try it!)

7.4.5   Adding Text to Our Plot

We can also add the results of our t-test we ran in the first session to the plot. We do this by using the `annotate()` function.

```
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep",
                     breaks = c(-2:6) #this will add a break for each value between -2
                     ) +
  theme_apa() +
  geom_jitter(width = .20) +
  geom_point(stat = "summary", fun = "mean", size = 5, colour = "black") +
  annotate("text",
           label = "t(9) = 4.061, p = .002**.",
           x = "Ambien",
           y = 5.5,
           hjust = 2.2,
```

```
        vjust = 1,
        size = 4)
```
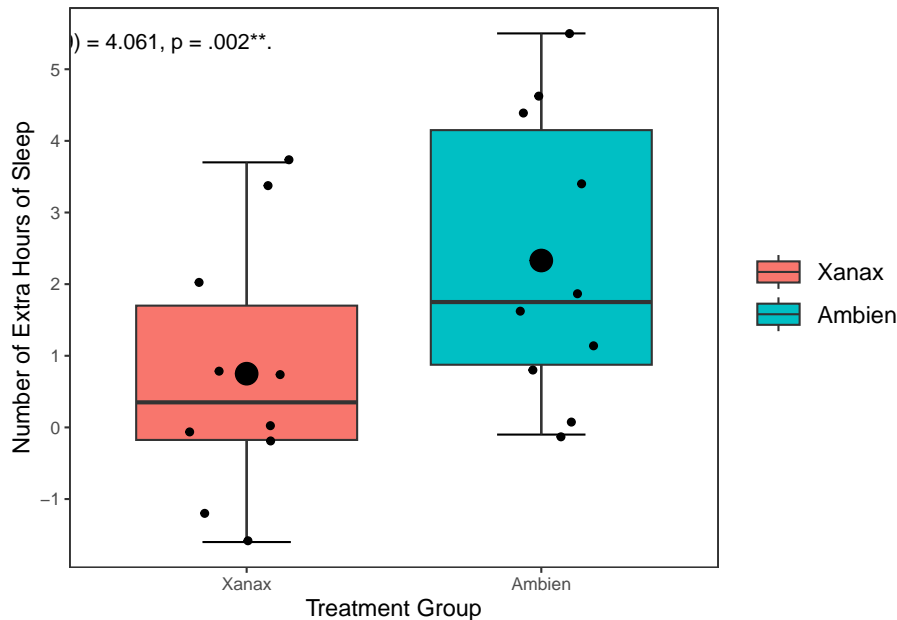

7.4.6   Exporting our Plot

We can export our plot easily using the `ggsave()` function. Inside the function, you specify the file name. It will save the file into your working directory.

By default, this function will export the last plot that you displayed. That is why it always best this function directly underneath the plot you made in your code.

```
ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment)) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep",
                     breaks = c(-2:6) #this will add a break for each value between -2 and +6
                     ) +
  theme_apa() +
  geom_jitter(width = .20) +
  geom_point(stat = "summary", fun = "mean", size = 5, colour = "black") +
  annotate("text",
           label = "t(9) = 4.061, p = .002**.",
```

```
        x = "Ambien",
        y = 5.5,
        hjust = 2.2,
        vjust = 1,
        size = 4)


ggsave("sleep_boxplot.pdf")
```

You should find the file `sleep_boxplot.pdf` in your working directory now. Open it up and have a look.

7.5   Drawing a Scatter Plot

Okay, so we talked a lot step by step how to create a box plot. Let's talk in somewhat less detail about how to create a scatter plot. After we have covered those two charts, the rest of this chapter will serve as a reference guide for creating other charts you might be interested in making (e.g., bar charts, line charts, histograms, violin plots).

7.5.1   Context

In one of my PhD studies, I investigated the relationships between basic emotional states (Anger, Disgust, Fear, Joy, Sadness, and Surprise) and the Big Five personality traits and their sub-traits. I was interested in knowing whether personality traits make one more or less likely to a) experience certain emotions and b) be more sensitive to those emotions. To achieve this, I collected data on the personality traits along with participant's daily experience of basic emotional states (baseline) and their reactive emotional experience after watching a series of emotionally provocative video clips (post-stimulus)

Let's load in my data frame and see what it looks like:

```
#View(df_personality)


str(df_personality)
```

```
## 'data.frame':    203 obs. of  35 variables:
##  $ hash             : chr  "00df03f53d53a2e3e32052b1c5a6a34f958773d6" "04a579b9da(
##  $ Openness_experience: num  3.55 3.45 2 3.65 3.2 3.35 3.95 3.1 4.25 4.45 ...
##  $ Intellect        : num  4.2 3.4 1.9 4.2 2.8 3.8 3.6 3.5 4.1 4.3 ...
##  $ Openness         : num  2.9 3.5 2.1 3.1 3.6 2.9 4.3 2.7 4.4 4.6 ...
```

```
##  $ Conscientiousness  : num   3.4 2.9 4.2 2.75 2.9 3.95 3.2 3.8 3.55 2.85 ...
##  $ Industriousness    : num   3.9 3.4 4.4 3 3 4.3 2.3 4.3 3.8 2.9 ...
##  $ Orderliness        : num   2.9 2.4 4 2.5 2.8 3.6 4.1 3.3 3.3 2.8 ...
##  $ Extraversion       : num   3.1 3.25 4.3 3.2 3.35 3.7 3.3 3.55 2.75 3.7 ...
##  $ Assertiveness      : num   3.3 3.3 4.6 2.9 3.1 3.1 2.6 3.2 3 3.6 ...
##  $ Enthusiasm         : num   2.9 3.2 4 3.5 3.6 4.3 4 3.9 2.5 3.8 ...
##  $ Agreeableness      : num   3.85 3.65 1.95 3.85 3.7 3.95 4.45 4.2 4 4.2 ...
##  $ Compassion         : num   3.7 3.7 2.1 4 3.5 3.9 5 4.1 3.9 4.6 ...
##  $ Politeness         : num   4 3.6 1.8 3.7 3.9 4 3.9 4.3 4.1 3.8 ...
##  $ Neuroticism        : num   1.9 3.05 3.05 3.35 2.7 1.95 4.4 1.8 3.1 3.1 ...
##  $ Volatility         : num   1.8 3.6 3.6 3 2.3 2.1 4.3 1.6 3 3.1 ...
##  $ Withdrawal         : num   2 2.5 2.5 3.7 3.1 1.8 4.5 2 3.2 3.1 ...
##  $ Anger_reaction     : num   2.17 1.33 2.17 1.33 2.17 ...
##  $ Disgust_reaction   : num   2.67 1.67 2.17 1.33 3.33 ...
##  $ Fear_reaction      : num   2.5 1.33 1.67 1.5 3.17 ...
##  $ Joy_reaction       : num   2.17 3.17 2.17 2.17 2.67 ...
##  $ Sadness_reaction   : num   2 2 1.33 2.17 3.33 ...
##  $ Surprise_reaction  : num   3.67 3 2.33 1.83 4.67 ...
##  $ Anger_baseline     : int   1 1 3 1 1 1 3 1 1 2 ...
##  $ Disgust_baseline   : int   1 1 4 1 1 1 2 1 1 1 ...
##  $ Fear_baseline      : int   1 1 1 2 2 1 2 1 2 1 ...
##  $ Joy_baseline       : int   2 2 3 3 4 3 2 3 3 3 ...
##  $ Sadness_baseline   : int   1 1 1 2 3 1 4 1 3 2 ...
##  $ Surprise_baseline  : int   1 3 1 2 1 2 1 1 2 2 ...
##  $ Country_Birth      : chr   "Ireland" "Pakistan" "United Kingdom" "United Kingdom" ...
##  $ Gender             : chr   "Male" "Male" "Female" "Male" ...
##  $ Education          : chr   "Bachelor's Degree" "Bachelor's Degree" "Master's Degree" "Bachel
##  $ Language           : chr   "English" "English" "English" "English" ...
##  $ Nationality        : chr   "Irish" "British" "English" "English" ...
##  $ Publication        : chr   "Yes" "No" "No" "No" ...
##  $ Age                : int   50 54 31 36 42 24 29 30 65 41 ...
```

You'll notice that is a large data frame (or at least large than ones we have been dealing with so far), with 35 variables and 203 participants. Feel free to have a thorough look at it using `View()`. But that each emotion is measured twice, once at the start of the study(e.g., `Anger_baseline`) and once as an average reaction to several video clips (`Anger_reaction`).

One of my hypotheses was that Extraverts are more sensitive to experiencing Joy than Introverts. Many researchers claim that one of the driving differences between Extraverts and Introverts is that Extraverts are more sensitive to experiencing positive emotion, making them more excitable and sociable. If this is true, then I would expect there to be a positive relationship between my `Extraversion` and `Joy_reaction` variables.

7.5.2   Drawing our Plot

Let's visualize this relationship by creating a scatterplot in R. There are several steps we need to take to do this.

First, let's call the ggplot() function, mapping Extraversion to the x-axis and Joy_reaction to the y-axis with the `mapping = aes()` call.

```
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction))
```
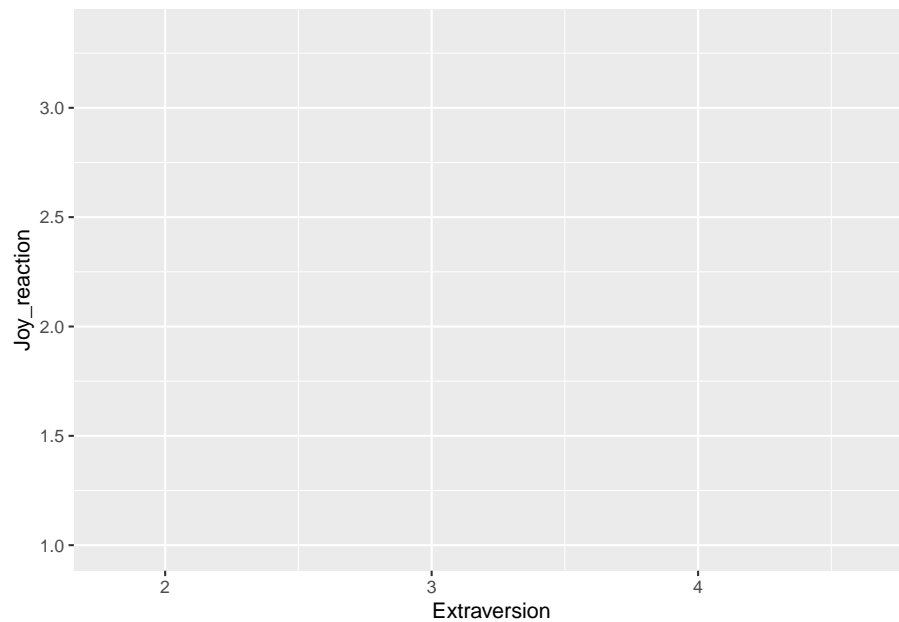

Now let's add our geometrical shape. For scatter plots, this is our old friend `geom_point()`.

```
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point()
```
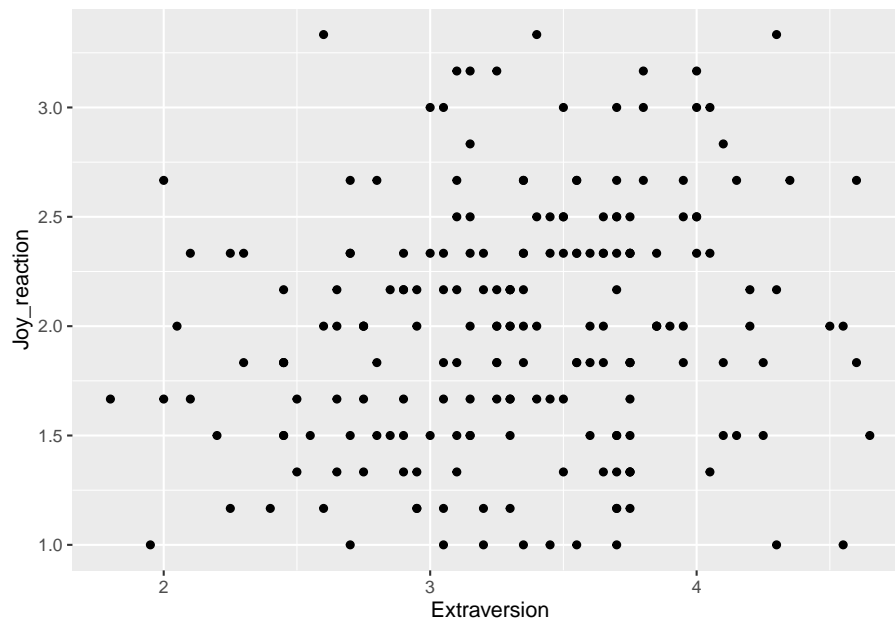
I am happy with the x-axis, but I would like the make the y-axis look more professional. So let's use `scale_y_continous()` to change its label.

```
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point() +
  scale_y_continuous(name = "Mean Experience of Joy (Reaction)")
```
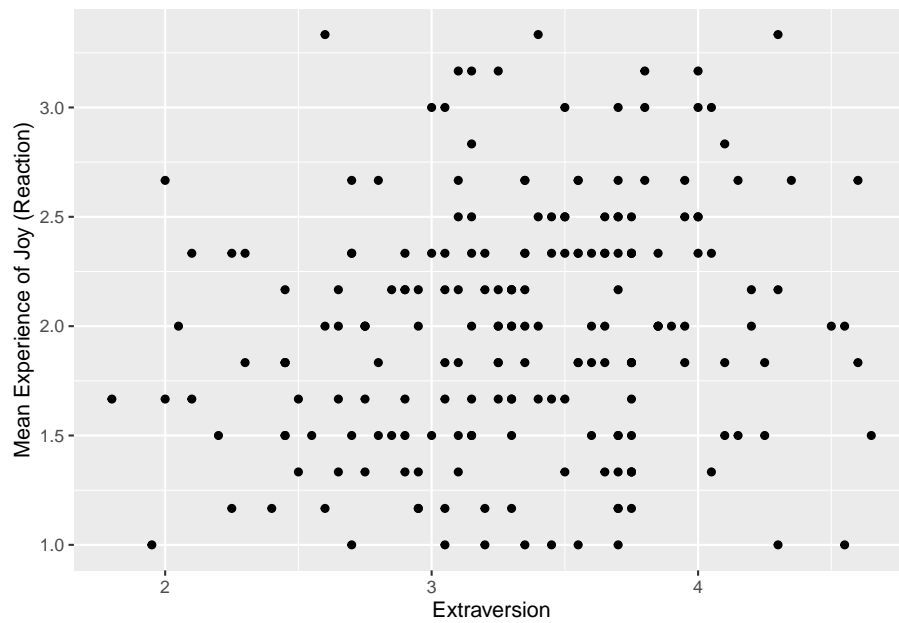
Let's make our plot prettier by adding the APA theme.

```r
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point() +
  scale_y_continuous(name = "Mean Experience of Joy (Reaction)") +
  theme_apa()
```
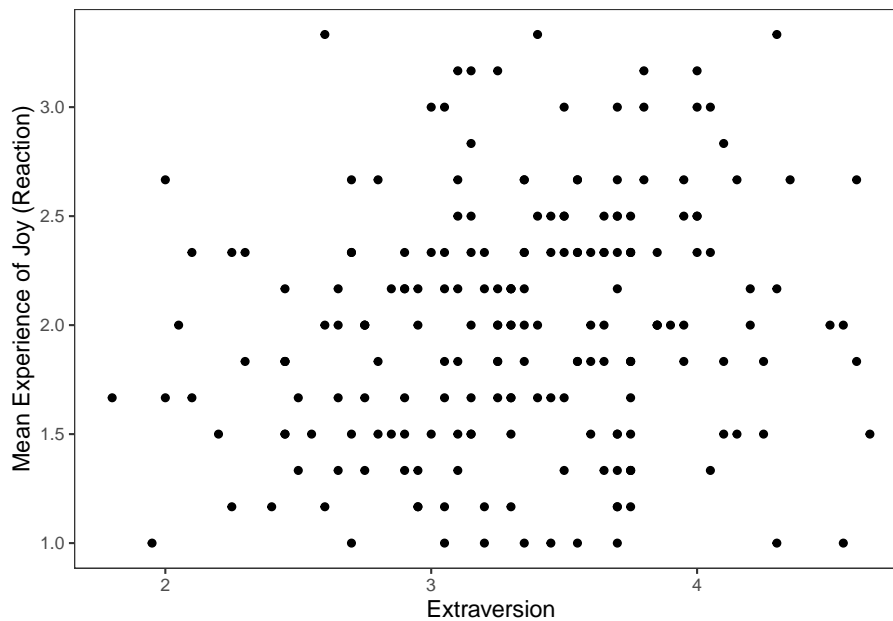
It's good to provide some information on the relationship between two variables on a scatter plot. We can do this by adding a regression line that best fits their relationship. To do this in R, we add a `geom` called `geom_smooth`, where we specify the model (method) we want to fit onto our data.

```
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point() +
  scale_y_continuous(name = "Mean Experience of Reaction Joy", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = FALSE)
```

```
## 'geom_smooth()' using formula = 'y ~ x'
```
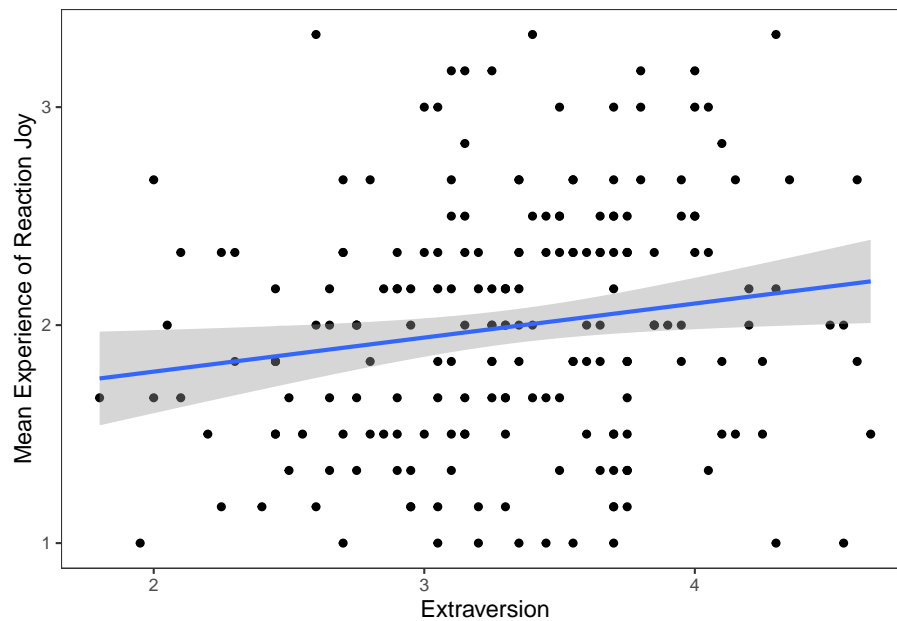
The function `geom_smooth` fits a model onto our data. When you specify `method = lm`, this mean that you are fitting a linear regression onto your data. I also set `show.legend = FALSE` because it create an annoying figure that we don't need.

We'll learn more running regressions in the next two workshops, but give you a preview, we compute a simple linear regression through the following code:

```r
joy_ext <- lm(Joy_reaction ~ Extraversion, data = df_personality)

summary(joy_ext)
```

```
##
## Call:
## lm(formula = Joy_reaction ~ Extraversion, data = df_personality)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.18510 -0.39468 -0.00537  0.38658  1.45300
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.47399    0.22543   6.539 5.02e-10 ***
## Extraversion 0.15629    0.06679   2.340   0.0203 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 0.5584 on 201 degrees of freedom
## Multiple R-squared:  0.02652,    Adjusted R-squared:  0.02167
## F-statistic: 5.475 on 1 and 201 DF,  p-value: 0.02027
```

Based on our plot and our linear regression model, we can see there is a small positive relationship between Extraversion and Joy (reaction) that is statistically significant. Let's add this information to our plot using the `annote()` function.

```r
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point() +
  scale_y_continuous(name = "Mean Experience of Reaction Joy", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F) +
  annotate("text", x = 2.5, y = 3.6,
           label = "F(1, 201) = 5.48, p = .02, R^2 = 0.02")
```

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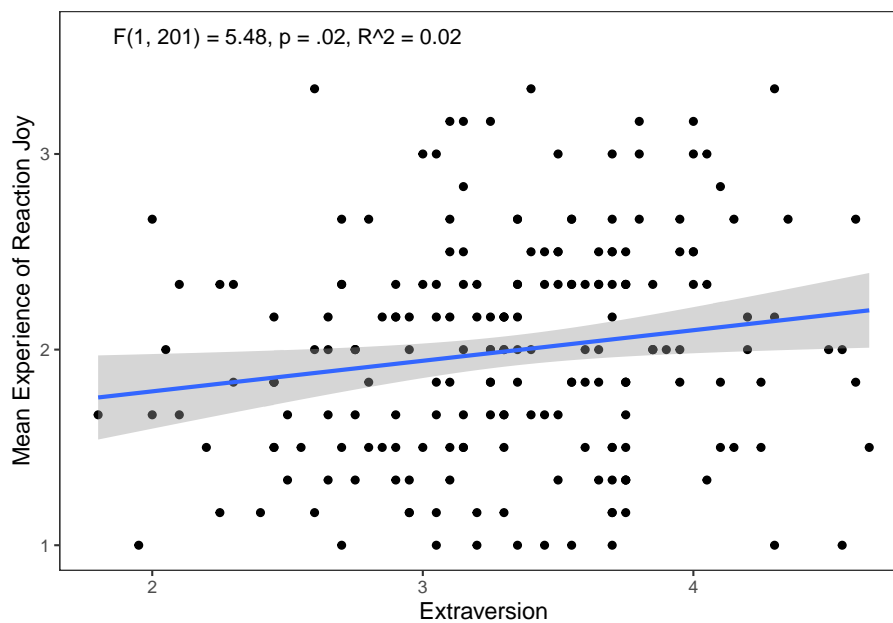

While this gets the message across, it is annoying that `F`, `p`, and `R` are not italicised. Additionally, how can we superscript the `2` in `R^2`? We can use the `expression()` function inside `label`. The syntax is a bit clunky, but it will get the job done.

```r
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point() +
  scale_y_continuous(name = "Mean Experience of Reaction Joy", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F) +
  annotate("text", x = 2.5, y = 3.6,
           label = expression(italic("F")(1, 201) == 5.48~","~italic("p") == .02~","~i
```

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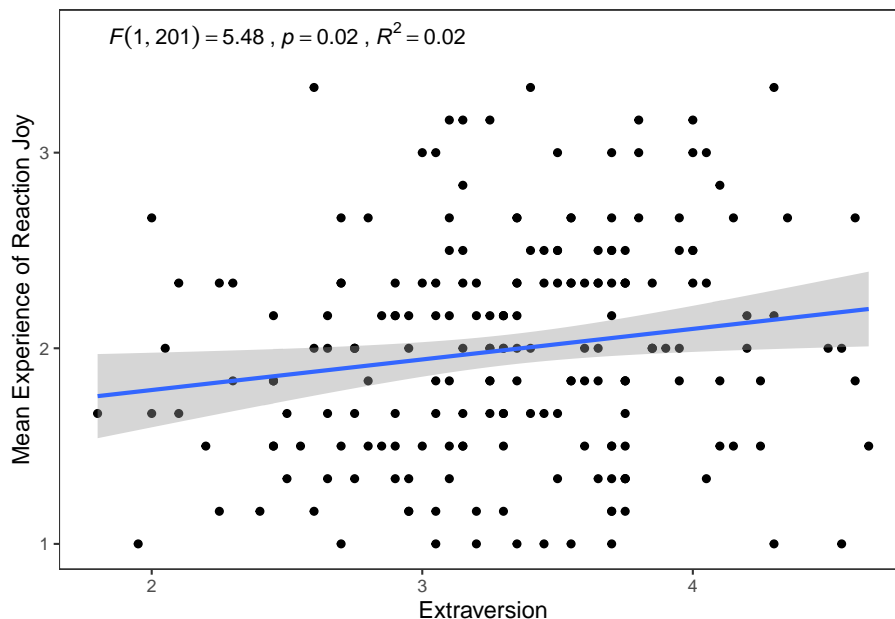

That looks nice and our scatter plot is ready to be exported.

7.5.3   Customising our Scatterplot based on Gender

But before we move on, I want to show you a few other things we can do when we create scatter plots. For example, what if we were interested in checking whether the relationship between Joy (Reaction) and Extraversion was similar for both males and females? How could we visualize this?

We can map the `colour` of the points and their `shape` to the variable `Gender`. There are two ways we can do this, through the `ggplot()` function or in the `geom_point()`. The way you do will have ramifications for how the data visualisation will appear. It's easier to show this than to explain. So let's first change

the colour and shape in the `ggplot()` function by mapping these properties to `Gender`.

```
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction, colour = Gender, shape =
  geom_point() +
  scale_y_continuous(name = "Mean Experience of Reaction Joy", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F) +
  annotate("text", x = 2.5, y = 3.6,
           label = expression(italic("F")(1, 201) == 5.48~","~italic("p") == .02~","~italic("R")^
```

`geom_smooth()` using formula = 'y ~ x'
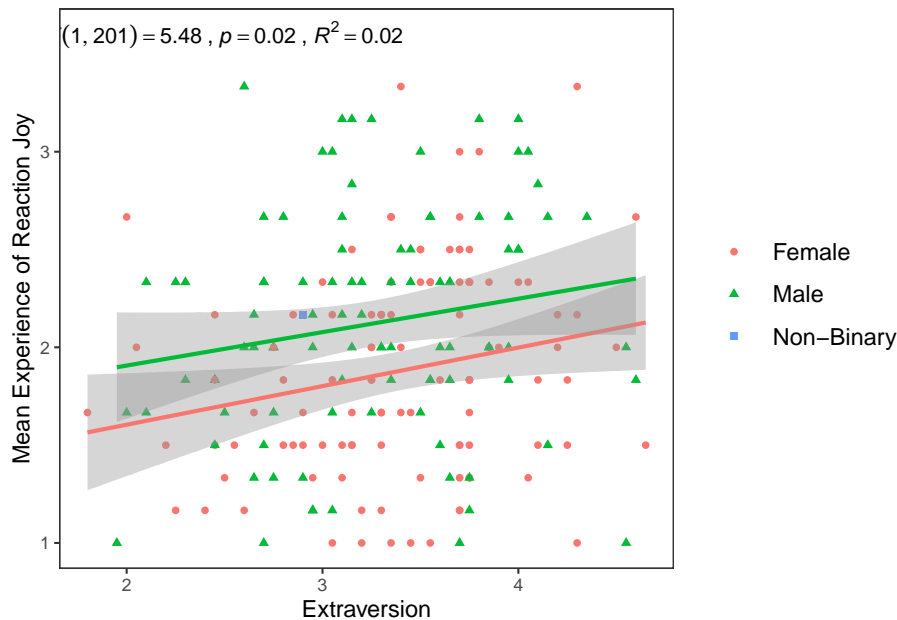
This changes the colour and shape of the data points depending on whether the participant was male, female, or non-binary. It also adds a regression line for the male participants scores and the female participants scores. A regression line was not added for non-binary participants because this was only 1 participant (can't draw a line of best fit with just one data point!). So we can see that male participants experienced more Joy than female participants during the study, but the nature of the relationship is very similar for both males and females (e.g. small positive relationship).

If we wanted to go with this approach, we could also map the appearance of the linear regression lines to `Gender`.

```r
ggplot(df_personality, mapping = aes(x = Extraversion,
                                     y = Joy_reaction,
                                     colour = Gender,
                                     shape = Gender,
                                     linetype = Gender)) +
  geom_point() +
  scale_y_continuous(name = "Mean Experience of Reaction Joy", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F) +
  annotate("text", x = 2.5, y = 3.6,
           label = expression(italic("F")(1, 201) == 5.48~","~italic("p") == .02~","~it
```

`geom_smooth()` using formula = 'y ~ x'
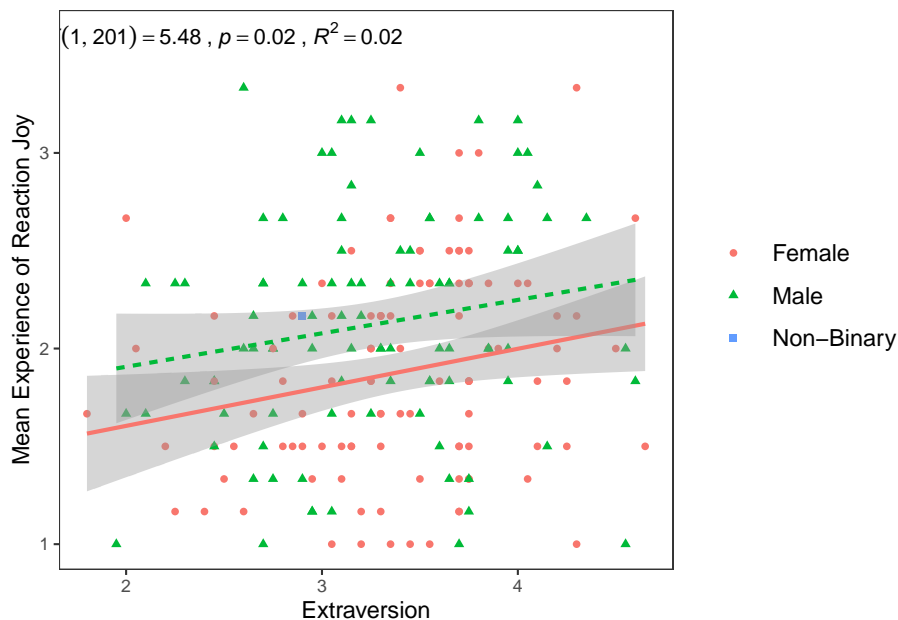
The second way we can map the `shape` and `colour` aesthetics is within the
`geom_point()` function.

```r
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point(mapping = aes(colour = Gender, shape = Gender)) +
  scale_y_continuous(name = "Mean Experience of Reaction Joy", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F) +
```

```
annotate("text", x = 2.5, y = 3.6,
         label = expression(italic("F")(1, 201) == 5.48~","~italic("p") == .02~","~italic("R")`
```

`geom_smooth()` using formula = 'y ~ x'
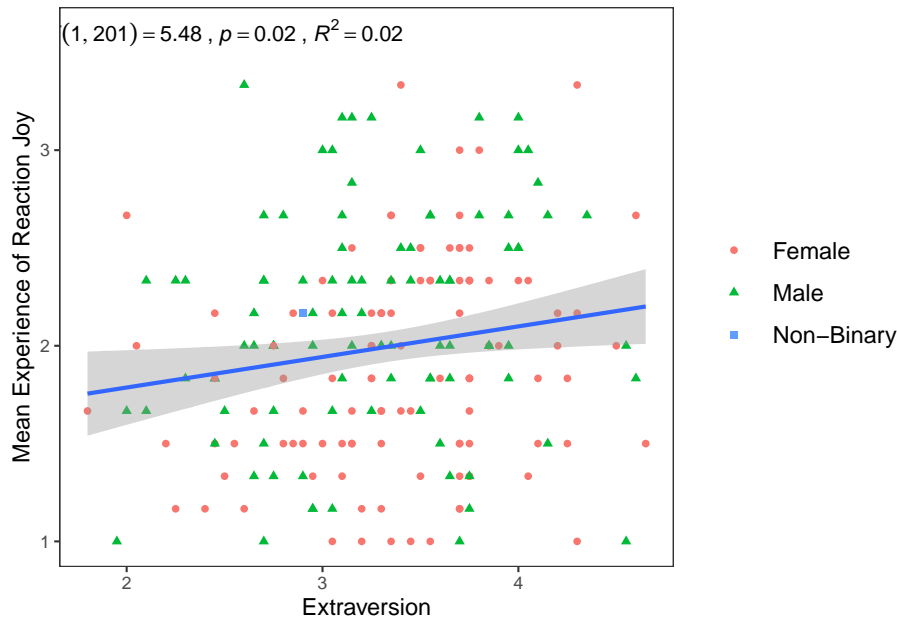
If you compare this plot to the previous approach, you'll notice that the colours and shape chosen are the same. The only difference is that there is one linear regression line now for the entire data set, rather than two for male and female participants.

This difference happens because of the structure of the `ggplot()` package. Basically, any aesthetic properties you map in the `ggplot()` function will be taken into account with everything you add to the plot. So when we map `shape`, `colour`, and `linetype` in `ggplot()` to `Gender`, the `geom_smooth()` function recognizes that we want separate visualizations for males, females, and non-binary participants and adds seperate lines accordingly.

However, when we map aesthetic properties outside of `ggplot()` and in a separate `geom()`, this will be restricted to that `geom`.

To sum it up. If we map our variables to aesthetics in `ggplot()`, these are global changes. If we map variables to aesthetics outside of `ggplot()`, this will only make local changes. This ability to choose local or global changes means that the `ggplot` package system provides a high level of control in creating the plot we want.

7.5.4   Faceting

What if I wanted to create three plots. A plot for only male scores, a plot for only female scores, and a plot for only non-binary scores? Well we can use the `facet_wrap()` function which will make a seperate graph based on different values on a specified variable.

The syntax for `facet_wrap()` is: `facet_wrap(~variable you are splitting the graph on)`

```r
ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point(mapping = aes(colour = Gender, shape = Gender)) +
  scale_y_continuous(name = "Mean Experience of Reaction Joy", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F) +
  facet_wrap(~Gender)
```

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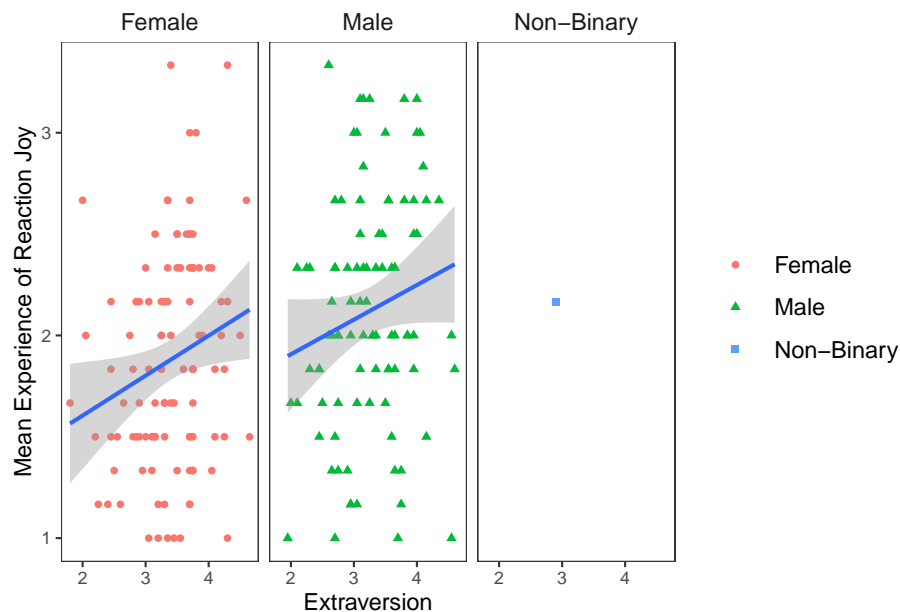

This can be a really useful tool in data exploration when you want to see differences in relationships or effects between different categories or scores. For the purposes of brevity, I will not go into details here about how to add individual annotations to each chart, but please check out the answer given by Pedro Aphalo and Kamil Slowikowski here if you are interested in doing this.

7.5.5 Combining Charts

Another hypotheses I wanted to test was if there is a positive relationship between Conscientiousness and Disgust sensitivity. Previous research and some high-profile scholars have made the claim that people who are high in Conscientiousness are more likely to feel disgust, which motivates their tendency to be structured, diligent, and orderly. If this is true, then I would expect there to be a positive relationship between `Conscientiousness` and `Disgust_reaction` variable

```
ggplot(df_personality, mapping = aes(x = Conscientiousness, y = Disgust_reaction)) +
  geom_point(mapping = aes(colour = Gender, shape = Gender)) +
  scale_y_continuous(name = "Mean Experience of Reaction Disgust", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F) +
  annotate("text", x = 2.75, y = 4,
          label = expression(italic("F")(1, 201) == 0.05~","~italic("p") == .81~","~italic("R")^
```

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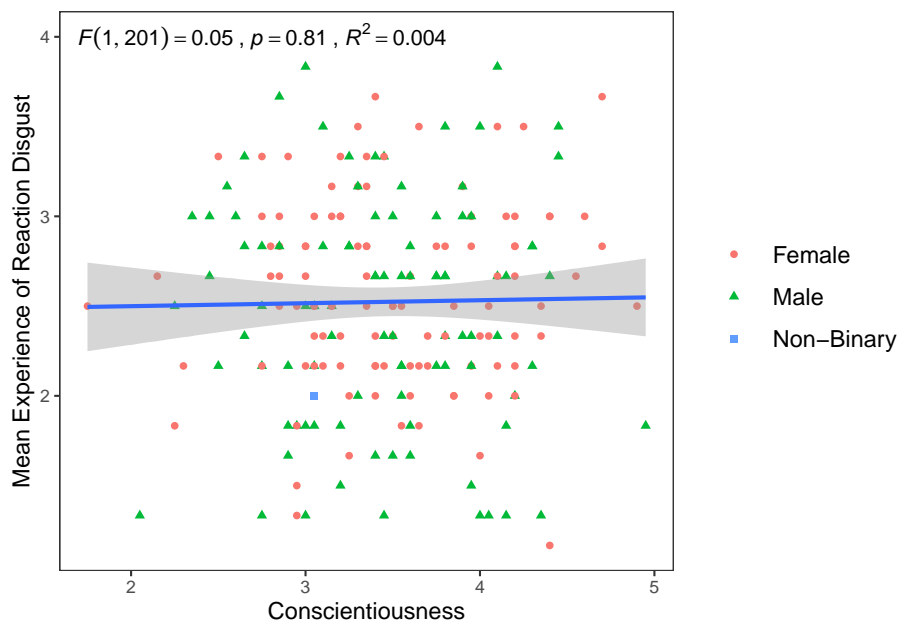

Well that's as a definite "NO!" that you're going to see in a scatterplot.

If you wanted to display and export these three plots together (for the sake of it, I am going to add the box plot in here as well), you can use the `patchwork`

package. Once it is installed, all you need to do is assign your plots to variable names, and then use the `+`, `|` and `/` operators together. For more information on how to use `patchwork`, then see this web page.

One thing to note is that you may need to change the appearance of each graph to make it easier to to combine them together. For the following graphs, I removed the legends from the scatter plots (by putting `show.legend = FALSE` in `geom_smooth` for both `p1` and `p2` and in `geom_boxplot` in `p3`) and changed the labels on the y-axis. Then I toyed around with their layout to find the layout that was the most informative (feel free to play around with diferent configurations)

```r
p1 <- ggplot(df_personality, mapping = aes(x = Extraversion, y = Joy_reaction)) +
  geom_point(mapping = aes(colour = Gender, shape = Gender), show.legend = FALSE) +
  scale_y_continuous(name = "Reaction Joy", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F)


p2 <- ggplot(df_personality, mapping = aes(x = Conscientiousness, y = Disgust_reaction)
  geom_point(mapping = aes(colour = Gender, shape = Gender),  show.legend = FALSE) +
  scale_y_continuous(name = "Reaction Disgust", breaks = c(1:5)) +
  scale_x_continuous(breaks = c(1:5)) +
  theme_apa() +
  geom_smooth(method = lm, show.legend = F)

p3 <- ggplot(df, mapping = aes(x = treatment, y = extra)) +
  stat_boxplot(geom ='errorbar', width = .3) +
  geom_boxplot(mapping = aes(fill = treatment), show.legend = FALSE) +
  scale_x_discrete(name = "Treatment Group") +
  scale_y_continuous(name = "Number of Extra Hours of Sleep",
                     breaks = c(-2:6) #this will add a break for each value between -2
                     ) +
  theme_apa() +
  geom_jitter(width = .20) +
  geom_point(stat = "summary", fun = "mean", size = 5, colour = "black")


p1 + p2



## `geom_smooth()` using formula = 'y ~ x'
## `geom_smooth()` using formula = 'y ~ x'
```
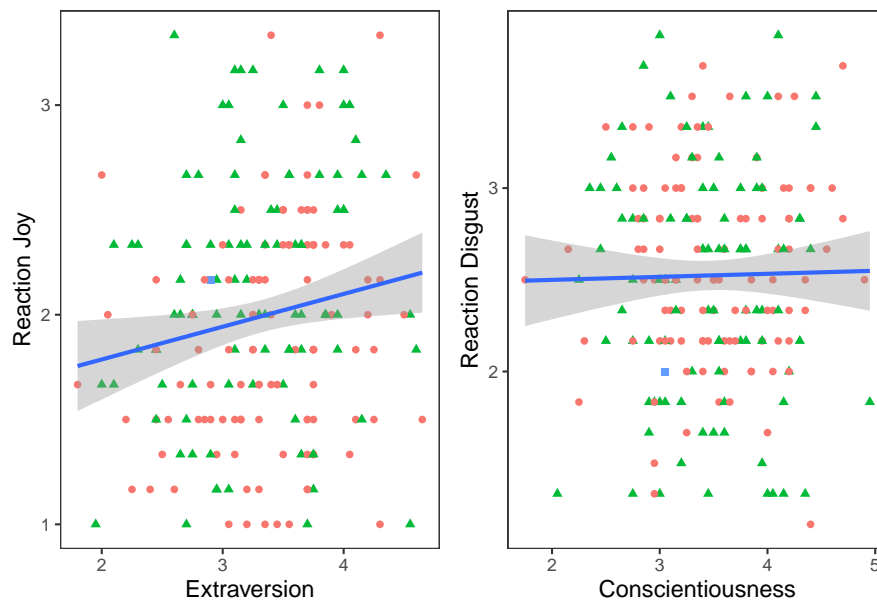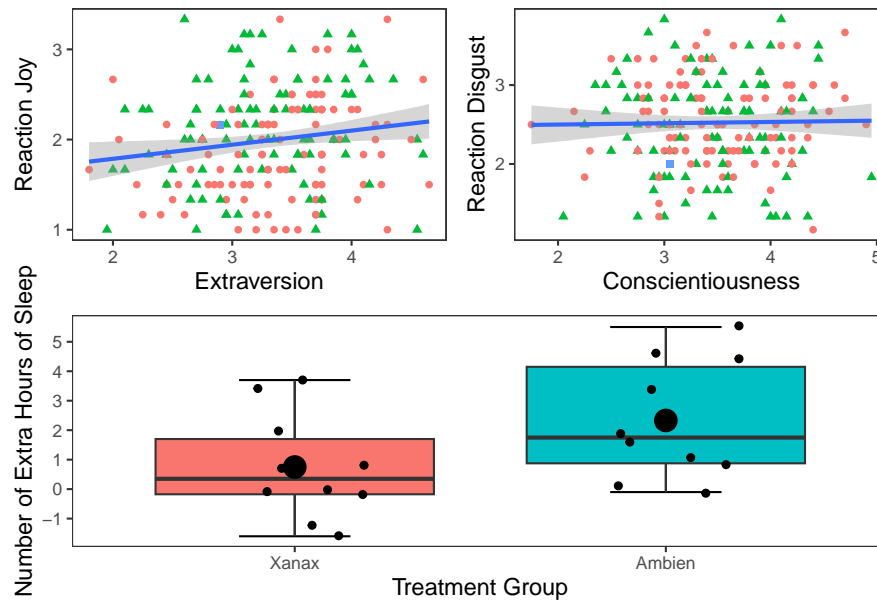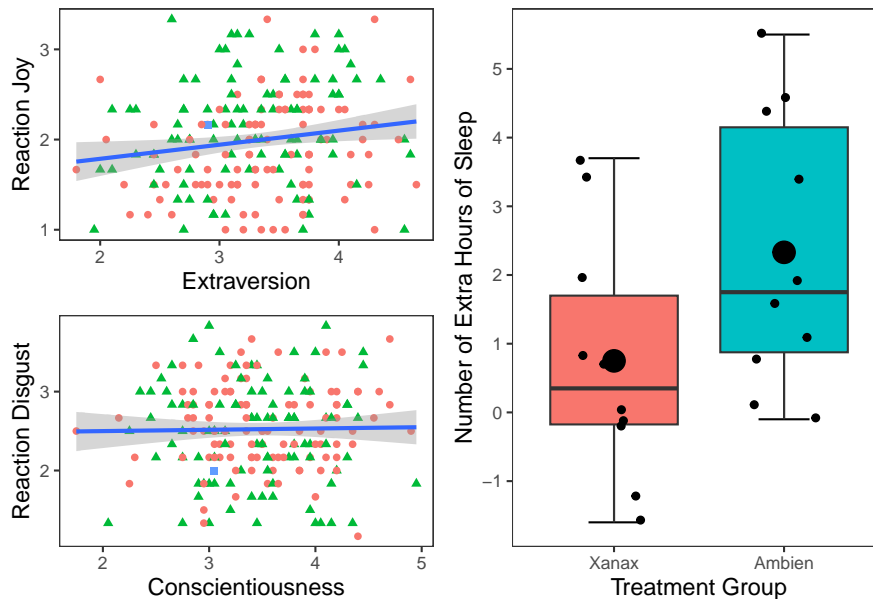
```
(p1 | p2) / p3
```

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

```
(p1 / p2) | p3
```

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

```
ggsave("combined.plots.pdf") #saving this configuration to my working directory
```

```
## Saving 6.5 x 4.5 in image
## 'geom_smooth()' using formula = 'y ~ x'
## 'geom_smooth()' using formula = 'y ~ x'
```

7.6   Violin Charts

A violin plot is a method to depict the distribution of numeric data across
different categories. It combines the features of a box plot and a kernel density
plot, offering a more comprehensive view of the data's distribution. To create
a violin plot we use the `geom_violin()` function. Here is the basic syntax for
creating a violin plot.

```r
df_violin <- data.frame(
  category = rep(c("A", "B", "C"), each = 100),
  value = c(rnorm(100), rnorm(100, mean = 1.5), rnorm(100, mean = 2))
)

# Create a violin plot
ggplot(df_violin, aes(x = category, y = value, fill = category)) +
  geom_violin() +
```

```r
geom_boxplot(width = 0.1, fill = "white") +
labs(
  x = "Category",
  y = "Value",
  title = "Violin Plot: Distribution of Values by Category"
) +
theme_apa()
```


Violin Plot: Distribution of Values by Category
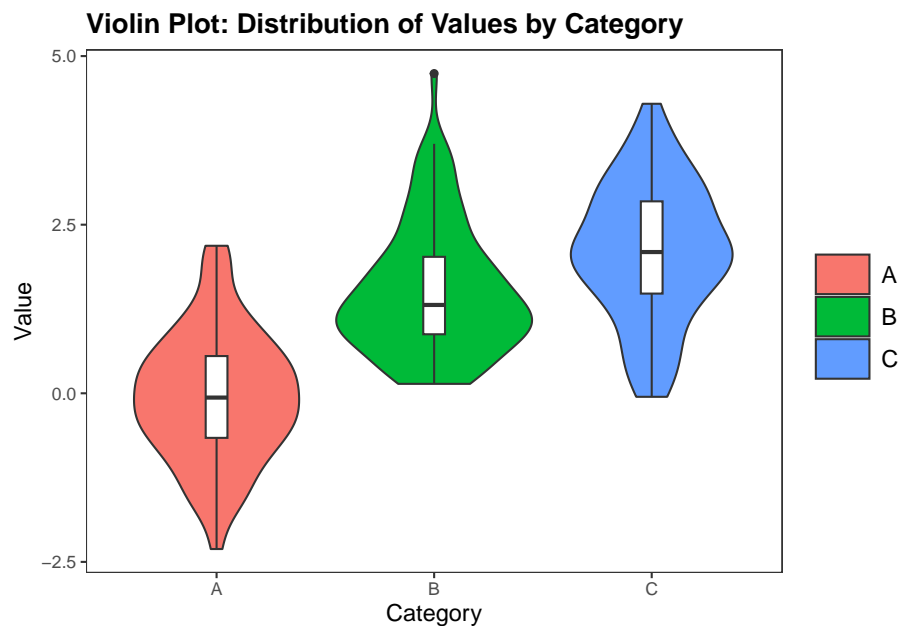
Let's do this with the `df_personality` data frame. We will put `Gender` on the x-axis and `Agreeableness` on the `y-axis`

```r
#I am going to remove the non-binary participant for this chart, as it does not make s

df_personality_binary <- df_personality %>%
  filter(Gender != "Non-Binary")

ggplot(df_personality_binary, aes(x = Gender, y = Agreeableness, fill = Gender)) +
  geom_violin() +
  geom_boxplot(width = 0.1, fill = "white") +
  labs(#this is another approach to naming x and y-axis, only use if you do not need t
    x = "Gender",
    y = "Agreeableness",
    title = "Violin Plot: Distribution of Agreeableness by Gender" #depending on APA s
  ) +
  theme_apa()
```

**Violin Plot: Distribution of Agreeableness by Gender**
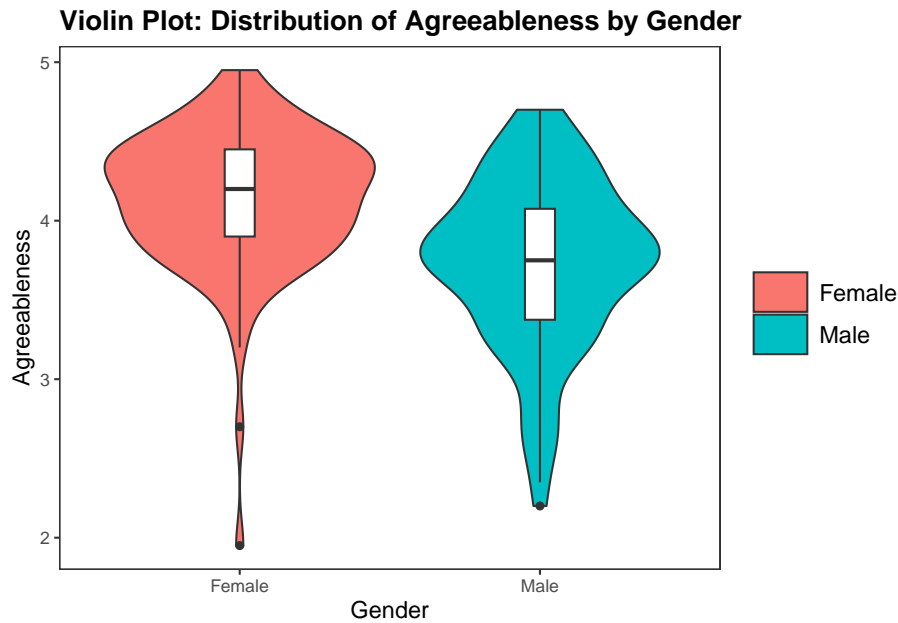
Bar Chart

A bar chart is great for displaying categorical data. It consists of rectangular bars with lengths proportional to the values they represent. Bar charts are effective for comparing the frequency, count, or proportion of different categories.

```r
df_bar <- data.frame(
  category = c("A", "A", "B", "C", "D", "E")
)

# Create a bar chart
ggplot(df_bar, aes(x = category, fill = category)) +
  geom_bar() + #
  labs(
    x = "Category",
    y = "Count",
    title = "Bar Chart: Distribution of Counts by Category"
  ) +
  theme_apa()
```

Bar Chart: Distribution of Counts by Category
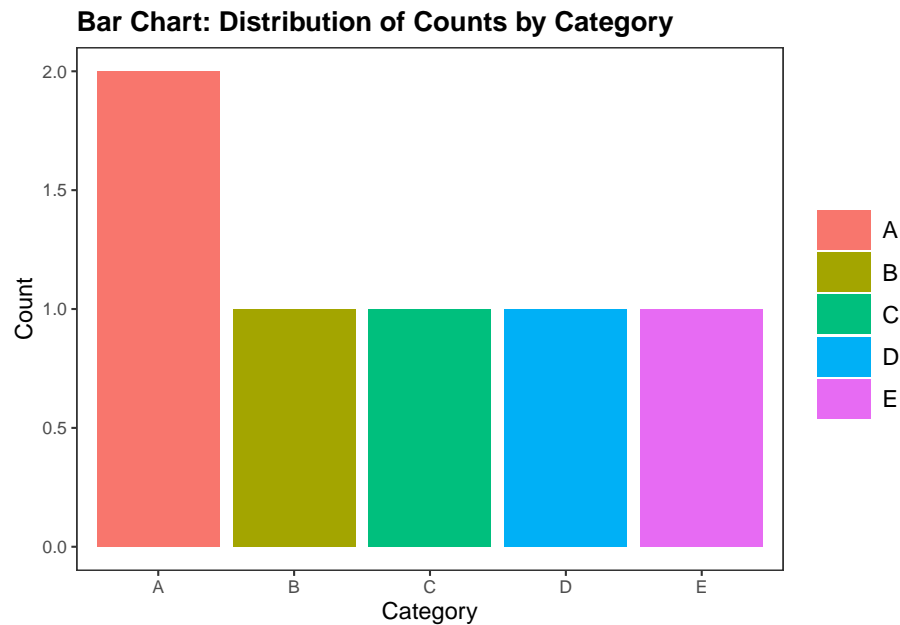
In `df_personality`, there is a variable called `Publication`. This variable captures whether participants were okay with having their recorded video and audio data potentially published in publications (for context, the participants facial expressions were recorded while they watched each video clip. After each video clip, they were asked to speak about their experiences and this was also recorded).

Let's use a `geom_bar` to visualise the count for each variable.

```
ggplot(df_personality, aes(x = Publication, fill = Publication)) +
  geom_bar() + #
  labs(
    x = "Education Level",
    y = "Count",
    title = "Bar Chart: Distribution of Counts by Education Level"
  ) +
  theme_apa()
```

**Bar Chart: Distribution of Counts by Education Level**
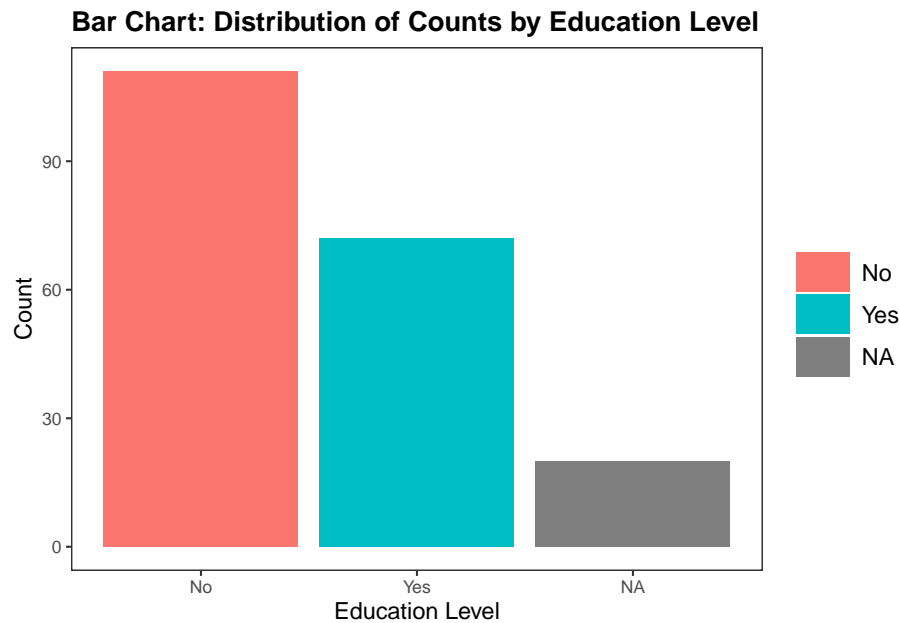
7.7 Histograms

Histograms are a type of bar plot that displays the distribution of a continuous variable. They partition the data into bins or intervals along the x-axis and then use the height of the bars to represent the frequency or density of observations within each bin.

Creating a histogram in ggplot2 is straightforward. We use the geom_histogram() function and map a continuous variable to the x-axis. Here's an example:

```
# Create a histogram of a continuous variable
ggplot(df_personality, aes(x = Age)) +
  geom_histogram()+
  theme_apa()
```

```
## 'stat_bin()' using 'bins = 30'. Pick better value with 'binwidth'.
```
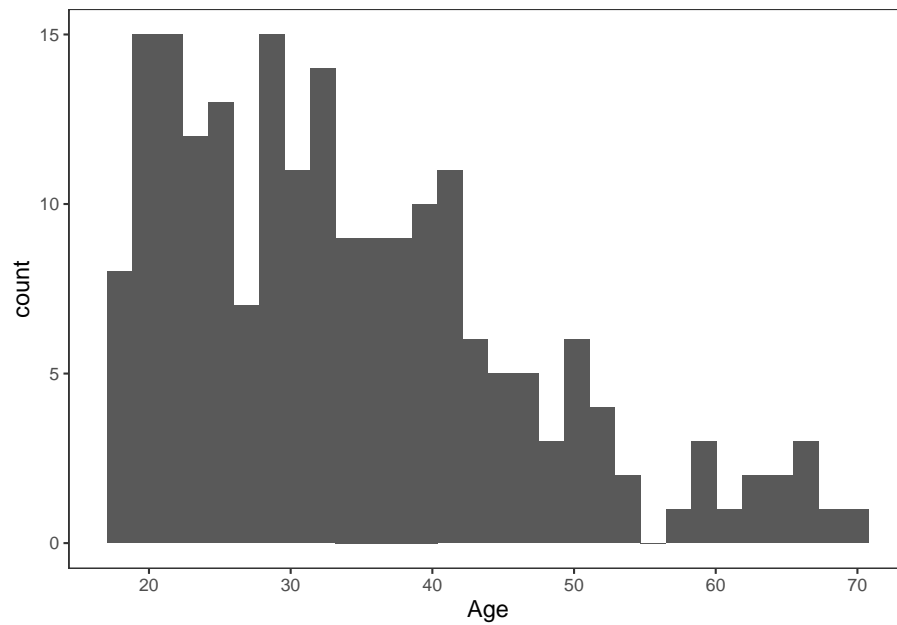
We can change the bin width by specify it in the `geom_histogram()` function.
Often you might need to play around with this to get the value you want.

```
ggplot(df_personality, aes(x = Age)) +
  geom_histogram(binwidth = 2,
                 colour = "white")+
  labs(
    x = "Age",
    y = "Count"
  ) +
  theme_apa()
```
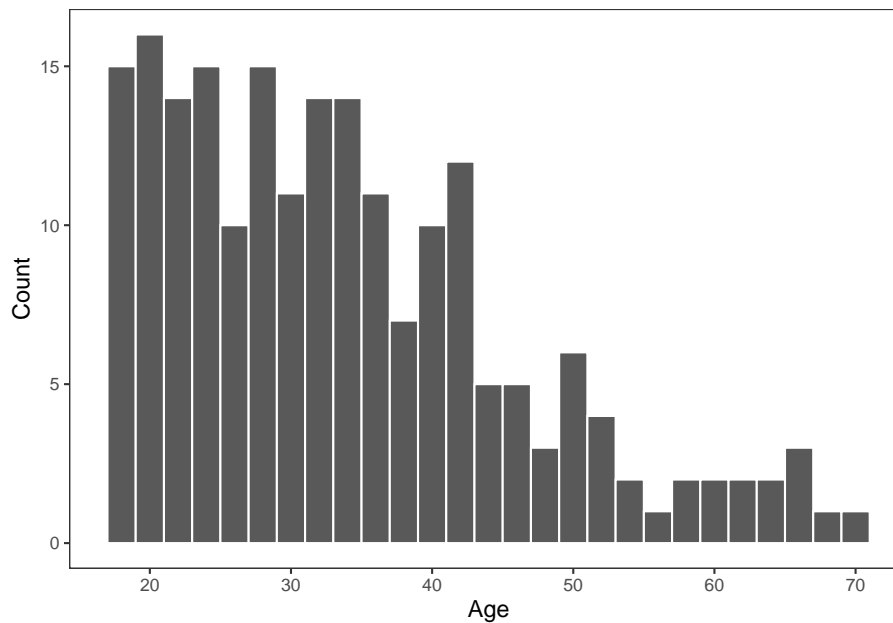
If we want to fit a normal distribution line, then we need to calculate the mean and standard deviation first.

7.8 Line Chart

Line charts are commonly used to visualize trends over time or any continuous variable. They are particularly useful for showing how a variable changes in relation to another variable, such as time.

Creating a line chart in ggplot2 involves mapping a continuous variable to the x-axis and a dependent variable to the y-axis. Here's an example:

```
# Create a line chart of a continuous variable over time
ggplot(data = economics, aes(x = date, y = unemploy)) +
  geom_line(color = "red") +
  labs(title = "Unemployment Rate Over Time", x = "Date", y = "Unemployment Rate") +
  theme_apa()
```

Unemployment Rate Over Time
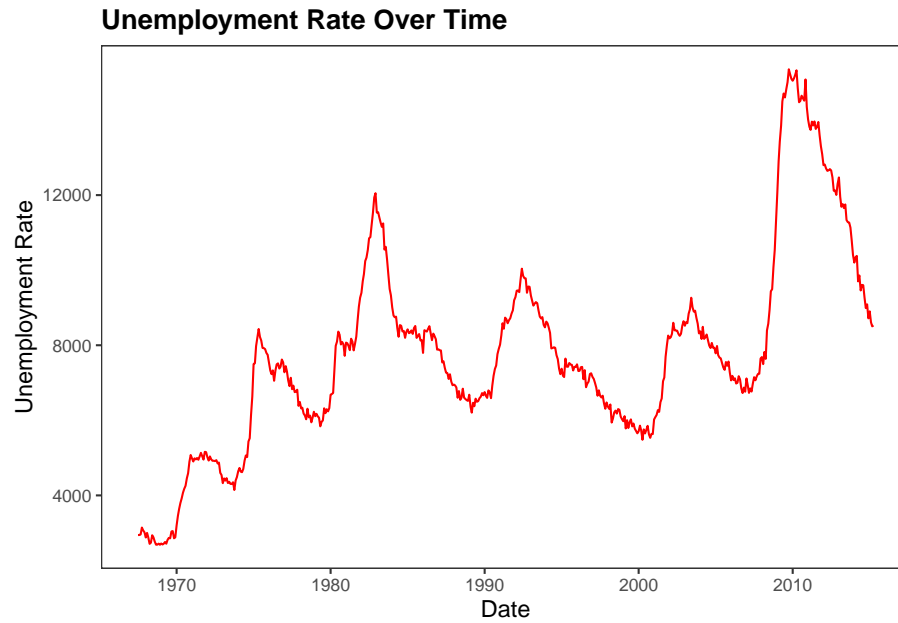
Summary

7.9   Geoms

Geom	Description	Similar Plot
`geom_point()`	Adds points to the plot. Useful for scatter plots to visualize the relationship between two continuous variables.	Scatter Plot
`geom_line()`	Connects points with lines. Often used to represent trends or changes over time in continuous data.	Line Plot
`geom_bar()`	Displays bars representing counts or frequencies of categorical data. Useful for comparing categories.	Bar Chart
`geom_histogram()`	Similar to geom_bar(), but used for continuous data. Creates bins of data and displays bars showing frequency distribution.	Histogram
`geom_boxplot()`	Represents the distribution of a continuous variable through quartiles, median, and outliers. Useful for identifying outliers and comparing distributions.	Box Plot

Geom	Description	Similar Plot
`geom_area()`	Fills the area below the line in a line plot. Useful for highlighting the cumulative effect of changes in a variable over time.	Area Plot
`geom_smooth()`	Adds a smoothed line to the plot, often useful for visualizing trends or patterns in noisy data.	Trend Line
`geom_text()`	Adds text labels to the plot, allowing annotation of specific points or adding additional information.	Text Annotation
`geom_label()`	Similar to geom_text(), but adds labels with a background, making them more prominent and readable.	Labeled Text
`geom_hline()`	Draws horizontal lines across the plot, useful for highlighting specific reference points or thresholds.	Horizontal Line
`geom_vline()`	Draws vertical lines on the plot, similar to geom_hline(), but for vertical lines.	Vertical Line
`geom_polygon()`	Draws polygons based on provided coordinates, useful for creating custom shapes or highlighting areas on a plot.	Polygon Plot
`geom_errorbar()`	Adds error bars to the plot, indicating uncertainty or variability in the data.	Error Bar Plot
`geom_jitter()`	Adds random noise to points, useful for avoiding overplotting in dense scatter plots.	Jittered Scatter Plot
`geom_tile()`	Creates a heatmap by filling rectangles with color based on a continuous variable. Useful for visualizing patterns in 2D data.	Heatmap
`geom_path()`	Similar to geom_line(), but does not close the path, useful for plotting trajectories or paths.	Path Plot

7.10 Themes

Theme	Description
`theme_gray()`	Default theme with a gray background.
`theme_bw()`	Theme with a white background and black gridlines.
`theme_minimal()`	Minimalistic theme with a light gray background and no gridlines.
`theme_light()`	Theme with a light gray background and gridlines.
`theme_dark()`	Theme with a dark gray background and white gridlines.
`theme_classic()`	Classic theme resembling base R plots.
`theme_void()`	Theme with no background, gridlines, or axis elements.

Theme	Description
`theme_linedraw()`	Theme resembling hand-drawn plots, with a white background and black gridlines.

Chapter 8

Appendix - Understanding Boolean Operators in the Context of `filter()`

In the Chapter 5, I brushed over what the operators `<` and `&`, and `|` meant and what they were actually doing in R. Each of these operators are known as Boolean operators in R.

Boolean operators are logical operators used in programming to combine or modify conditions, resulting in either **TRUE** or **FALSE** outcomes. In the context of data cleaning in R, Boolean operators help us construct conditions to select specific rows from a data frame based on certain criteria. When using `filter()`, R evaluates each row in the data frame against the specified conditions, determining whether each row meets the criteria and should be retained or not.

8.1 How Boolean Operators Work:

- **Logical AND (`&`):** The `&` operator evaluates to **TRUE** only if both conditions on either side of the operator are **TRUE**. In the context of `filter()`, rows are retained if they satisfy **all** conditions joined by `&`. For example, `filter(df, x > 5 & y < 10)` will retain rows where `x` is greater than `5` **and** `y` is less than `10`.

- **Logical OR (`|`):** The `|` operator evaluates to **TRUE** if **at least one** of the conditions on either side of the operator is **TRUE**. In the context of `filter()`, rows are retained if they satisfy **any** of the conditions joined

by `|`. For example, `filter(df, x > 5 | y < 10)` will retain rows where **x** is greater than **5 or y** is less than **10**.

- **Logical NOT (!):** The `!` operator negates a condition, converting **TRUE** to **FALSE** and vice versa. In the context of `filter()`, `!` can be used to exclude rows that satisfy a particular condition. For example, `filter(df, !(x == 5))` will exclude rows where **x** equals **5**.

8.1.1   Understanding R's Evaluation Process:

When applying `filter()` with Boolean operators, R sequentially evaluates each row in the data frame against the specified conditions. If a row satisfies **all** conditions joined by **&**, or **any** condition joined by `|`, it is retained in the filtered data frame. Rows that do not meet the specified criteria are excluded from the output.

The following table summarises the main Boolean operators we would use in R

Operator/Condition	Description	Example
==	Checks if two values are equal.	`x == 5` evaluates to `TRUE` if `x` equals `5`.
!=	Checks if two values are not equal.	`x != 5` evaluates to `TRUE` if `x` does not equal `5`.
>	Checks if one value is greater than another.	`x > 5` evaluates to `TRUE` if `x` is greater than `5`.
<	Checks if one value is less than another.	`x < 5` evaluates to `TRUE` if `x` is less than `5`.
>=	Checks if one value is greater than or equal to another.	`x >= 5` evaluates to `TRUE` if `x` is greater than or equal to `5`.
<=	Checks if one value is less than or equal to another.	`x <= 5` evaluates to `TRUE` if `x` is less than or equal to `5`.

Operator/Condition	Description	Example
&	Logical AND operator; evaluates to `TRUE` only if both conditions are `TRUE`.	`(x > 5) & (y < 10)` evaluates to `TRUE` if `x` is greater than `5` AND `y` is less than `10`.
|	Logical OR operator; evaluates to `TRUE` if at least one condition is `TRUE`.	`(x > 5) | (y < 10)` evaluates to `TRUE` if `x` is greater than `5` OR `y` is less than `10`.
!	Logical NOT operator; negates a condition.	`!(x == 5)` evaluates to `TRUE` if `x` is not equal to `5`.