Introduction to R for Advanced Research Methods (PS6183)

Contents

Chapter 1

Overview

Authors: Dr Ryan Donovan and Dr Ciara Egan.

This is the textbook for the Advanced Research Methods module (PS6183). This textbook will describe how to use R programming language to import, clean, process, and visualise psychological data. No prior knowledge of R or any other programming language or statistical software is required to successfully complete this module.

This textbook is still undergoing development and is not the final product. Consequently, all textbook materials are used for educational purposes only and should only be shared within the University of Galway's School of Psychology. Any issues or errors with the textbook should be reported to Ryan Donovan (ryan.donovan@universityofgalway.ie).

1.1   What will I learn in R?

In terms of specific R skills, students will learn how to:

- Import, export, and create datasets.

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

- Apply key techniques for data cleaning to enable statistical analysis.

- Run descriptive and inferential statistics.

- Create APA-standard graphs.

- Deal with errors or bugs with R code.

1.2   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 desk a few times to figure out a way to clean that dataset entirely in 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 - so do not be discouraged when it happens.

This textbook mainly uses 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 textbook 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. There are a lot of excellent resources available to learn how to use R markdown, but it's far more important to learn how to use R first.

1.3   Why are we learning 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 just an abstract ideal; several top-tier journals require that you submit R scripts along with manuscripts. If you don't know how to use R (or at least no one in your lab does), then this puts you at a disadvantage. The upside to this is that if you do know how to use R, then you will be at an advantage in your future career prospects.

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 significant 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! We will talk more in class about the advantages of using R over existing propriety statistical software.

1.4 Where and when will the classes take place?

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

Week	Date	Lecture (10am-11am)	Workshop (11am-1pm)
1	16/01/25	Introduction & NHST	Introduction to R
2	23/01/25	Research Design	R Programming 1
3	30/01/25	Statistical Power and Open Science	R Programming 2
4	06/02/25	Correlation & t-tests	T-tests in R
5	13/02/25	Regression	Correlation in R
6		Reading Week	
7	27/02/25	ANOVA	Regression in R
8	06/03/25	Moderation & Mediation	ANOVA in R
9	13/03/25	Data Visualisation	Data Visualisation in R
10	20/03/25	Data Wrangling Masterclass	
11	27/03/25	Course specific content	
12	03/04/25	Course specific content	

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

If you have a laptop that you work on, please 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.

1.6   Office hours and why they are important for everyone (R Support - Ryan Donovan).

Programming concepts can take a while to sink in and it is naturally to need additional help.
But in my experience, students often worry about asking for help, often from fears of being judged by their instructor. This fear is exacerbated in courses involving mathematics and programming, as students are quick to label themselves as "*no good*" at these subjects.

This is a completely unwarranted belief. My job is to teach you R. Consequently, if you are reading the material or listening in class, and it's not sinking in at all, then that is **_my fault, not yours._** If this is happening, then it is important that this feedback is being relayed. The most valuable teaching hours I've had are while talking with a student who is struggling to understand the material, because it forces me to see the errors in how I am explaining that material and adjust. That way, I learn better ways to explain a particular concept and the student comes with away with a greater understanding of said concept. We all win.

So pretty please, do not be a silent martyr and let me know if you are struggling with any of the R-based material in this course.

Staff Member	Office Hours	Location
Ryan Donovan	Wednesdays, 2.30-3.30pm. (In-person or online option). Time does not suit? Just get in touch, we can arrange a different time.	2065A, which is on the top floor of the Psychology building. It is in the corridor right next to the elevator.
ryan.donovan@university.goalsway.link	Teams Link: click here	

Chapter 2

Getting Started with R and RStudio

This session 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 course.

2.1 What is R?

R is a statistical programming language that enables us to directly communicate with our computers and ask it perform tasks. Typically we rely on button-click applications (e.g. SPSS, Excel, Word) to communicate with our computers on our behalf. These applications translate our button-click commands into a language that our computer can understand. Our computer then carries out those commands, returns the results to the application, which then translates those results back to us.

Applications like SPSS are convenient. They usually have a user-friendly interface that makes it easy to communicate with our computer. Consequently, this means one can become highly competent in a short amount of time with such applications.

However, these applications also limit what we can do. For example, base SPSS is functional when it comes to creating data 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 use an alternative application. 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.
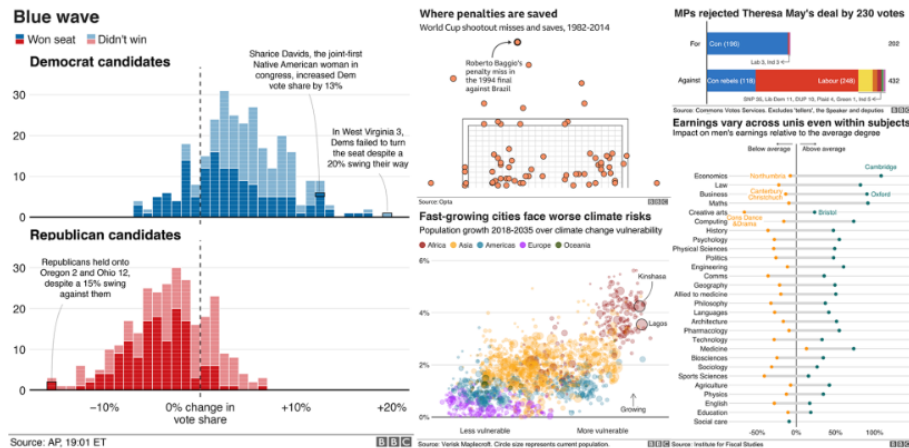

Figure 2.1: BBC graphs created in R.

But why R and not a different programming language? R was developed by statisticians for the purpose of conducting statistic analysis. In contrast, other popular programming languages (Python, JavaScript, C) were designed for different purposes. Consequently, R contains an extensive vocabulary that enables us to carry out sophisticated and precise statistical analysis. I have both used R and Python (often used in Data Science) to conduct statistical analysis, and I have found it significantly easier in R to run a wide range of statistical tests. Similarly, there is extensive support available online to enable you to run statistical analyses in R. This explains why R is typically used among statisticians, social scientists, data miners, and bioinformaticians. For these reasons, we will be using R 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 onto your laptop. 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 run R and RStudio online for free, with no need to install anything.

---

[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.

If you are using the desktops in the lab, then please only use Posit Cloud in all of our R and RStudio classes. It is much easier to run R and RStudio on Posit Cloud than on the absolute tragic disaster of an operating system they use on the desktop Windows machines. Please, please, please do not use the RStudio version on the desktop computers in the lab, or else your R life will be brutal and painful.

If you are using your own laptop, then use Posit Cloud as a backup option in case any technical issues pop up. During class, we might not be able to solve those issues quickly and efficiently (in a large classroom, one must always account for Murphy's Law). Rather than being hamstrung by technical difficulties, you can sign into Posit Cloud and keep following along with the session.
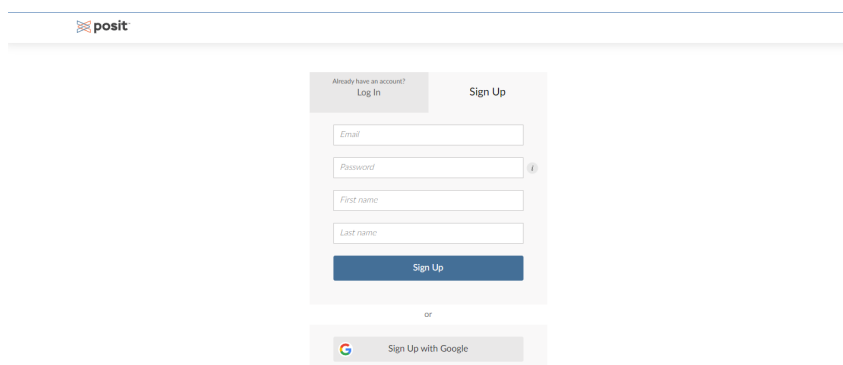
You might be wondering, why not just use Posit Cloud? The reason is that there are some restrictions to the free version of Posit Cloud, namely that you can only use 25 hours per month and are limited in the amount of data you can use during that time. It's highly unlikely that we will hit those restrictions in the next 10 weeks, but if you primarily use your laptop and Posit Cloud secondarily, then we do not have to worry about that.

For those of you on the desktop computers, if you do hit the max limit, then get in touch with Ryan (email: ryan.donovan@universityofgalway.ie). The simplest solution is just to create a second Posit Cloud account.

2.2.1 Instructions for Creating a Posit Cloud Account

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")
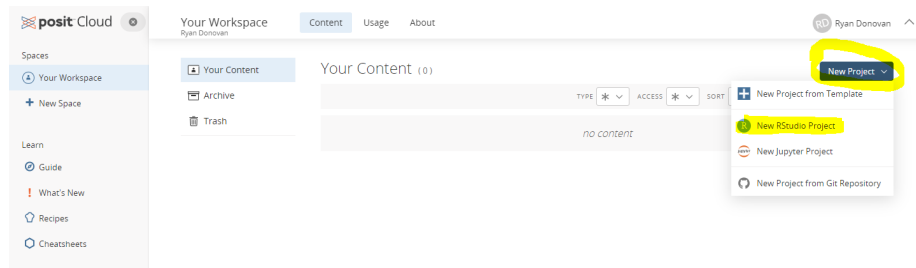
3. Once it is deployed, name your project at the top as ***rintro***
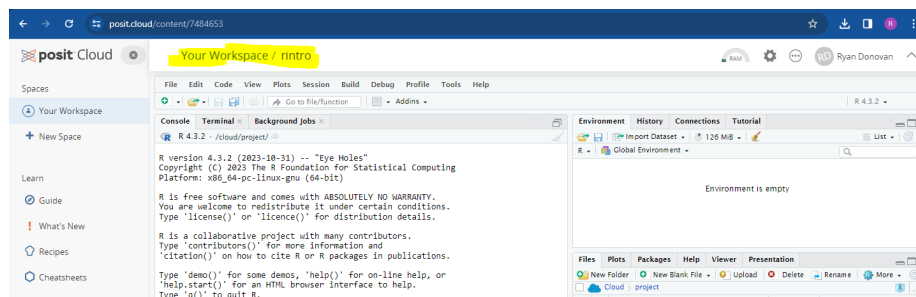
Don't worry about what anything on the screen means for now. We'll come back to that in the section **Creating an RProject (2.5).**

2.3   Downloading R on to your Computer (personal laptop or desktops only)

If you are using your own personal laptop or desktop (if at home), then please follow the following instructions to download R on either Windows or Mac.

If you are using the Desktops in the lab, do not follow these instructions, as we will be using Posit Cloud

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-4.4.2 for Windows (depending on the date you access this, the version of R might have been been updated. That's okay, you can download the newer version). Let the file download.
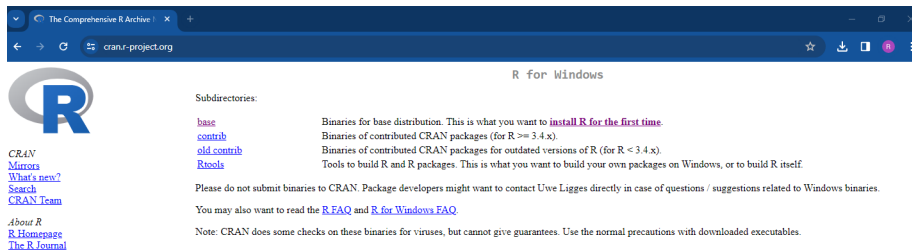
Figure 2.2: The R programming language is occasionally updated, so the specific version of R that you see might be different than mine. But that's okay!

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. Choose English as your setup language. The file name should be something like "R-4.4.2.-win" (numbers will differ depending on the version downloaded).

6. Agree to the terms and conditions and select a place to install R. The default option is fine.

2.3.2   Downloading R on Mac

The instructions are largely the same for Mac.

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

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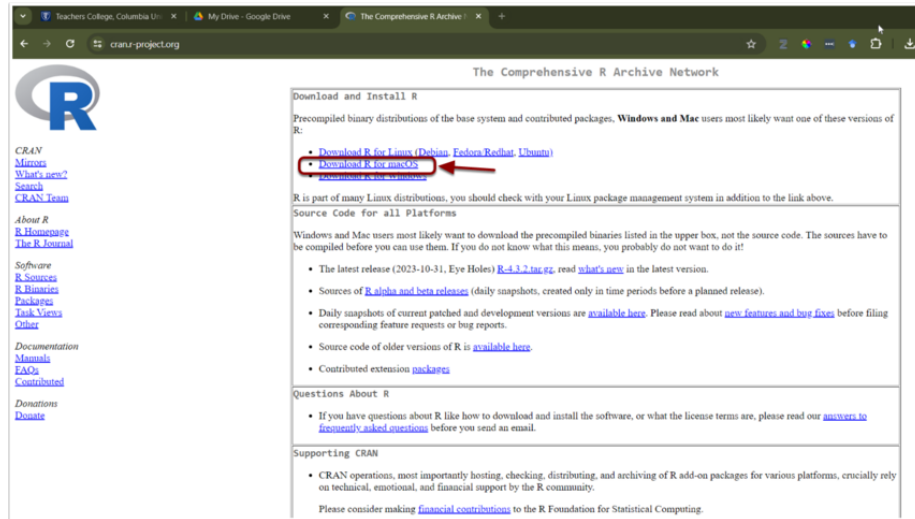
- Check the **Latest release** section for the appropriate version and follow the directions for download. If you are unsure, please ask me.
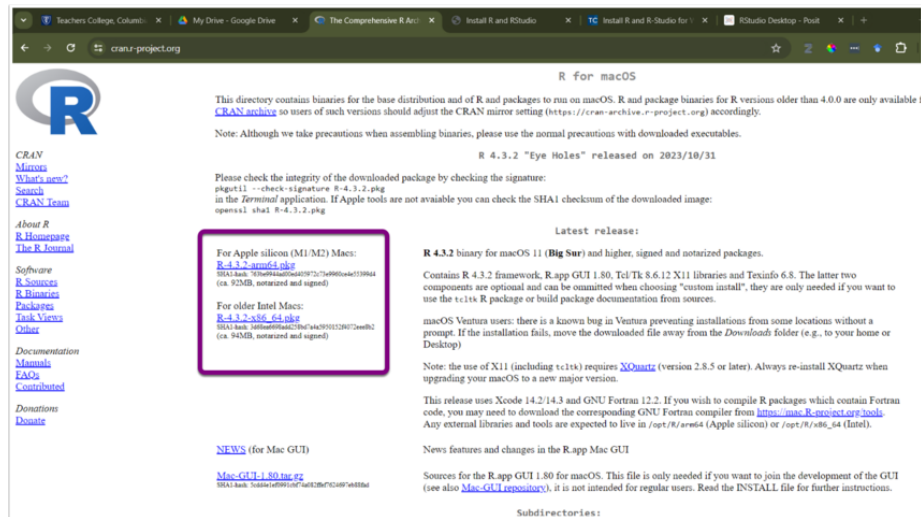
- Once the file download is complete, click to open the installer. Click **Continue** and proceed through the installer. I recommend going with all default options.

Figure 2.3: Depending on your version of Mac OS, this might look slightly different. But you should still be able to install it.

- Once the R installer has finished, click **Close.**
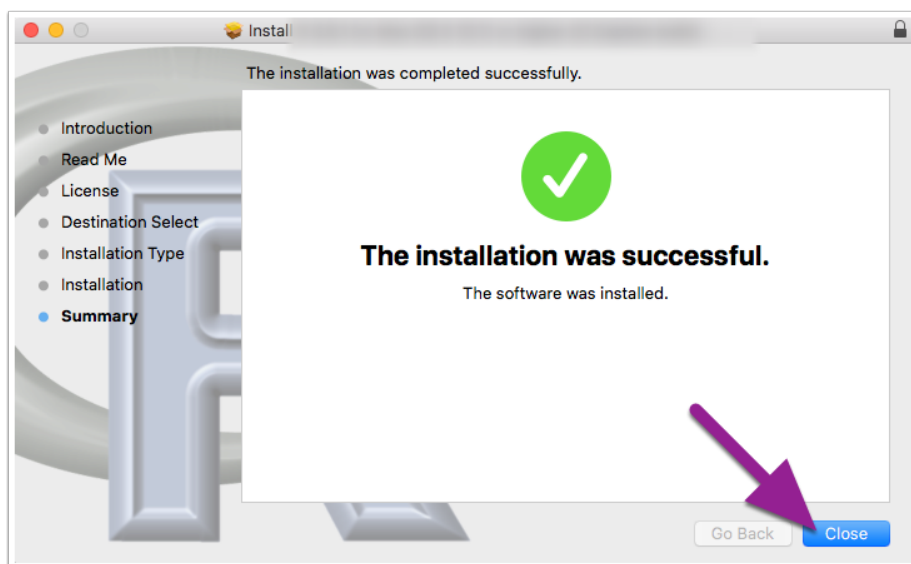
2.4   Install and Open RStudio

Once R is installed, we will install RStudio. ***Again, if you are using the desktops in the lab, you do not need to install RStudio—just make sure you have followed the Posit Cloud instructions.***

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 more efficient in Word.

1. **NB:** Make sure that R is installed ***before*** trying to install RStudio.

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. Follow the instructions to install 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).
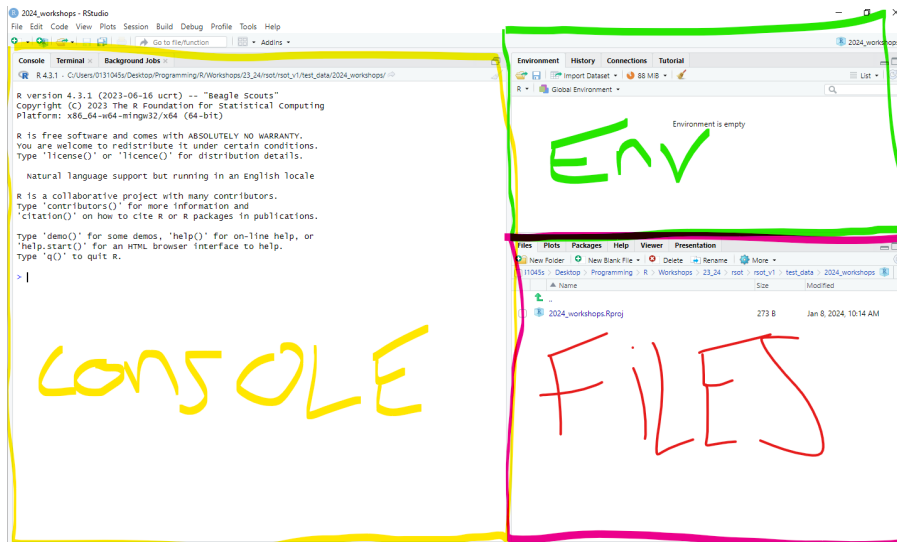
2.5 Creating an R Project

***Note: If you are using PositCloud instructions from 2.2.1, you have already created a project on PositCloud. Only follow this section if you are using your own computer or laptop.

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 (in this context, "directory" is just a fancy word for a folder).

Activity - Create an R Project (Personal Computer or Laptop)

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

1. Click "File" in the top left-hand corner of RStudio → then click "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.

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

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

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 later.

- My recommendation is to create a folder called "PS6183" on your desktop and save your project inside this folder.

- Regardless of where you save your project, make a note of the location on your computer and keep it handy (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***.

2.6   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.
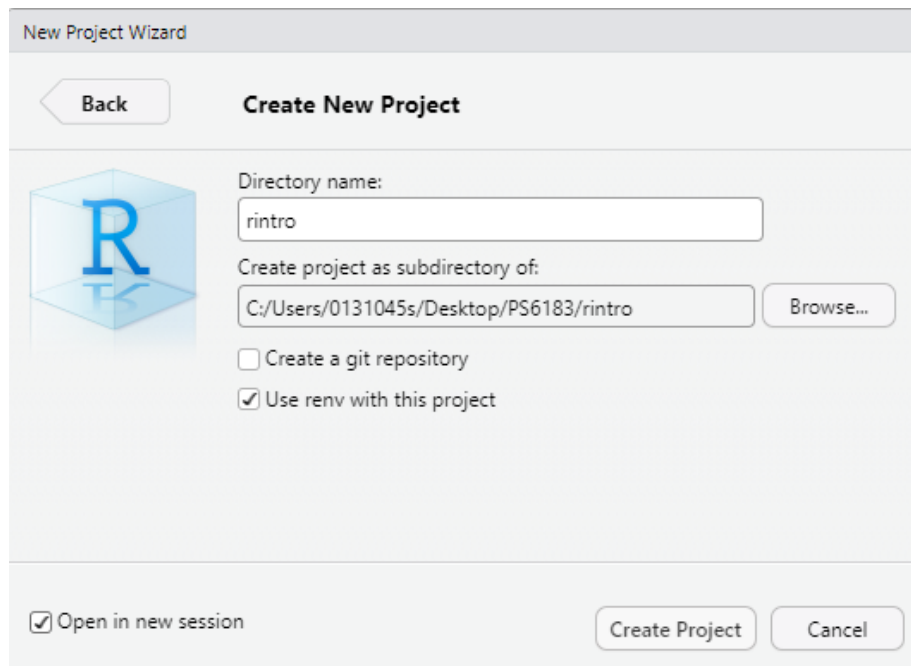
Figure 2.4: New Project Set Up

2.6.1   Opening the Source Pane

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

- **Button approach**: Click the "File" tab in the top-left corner → Click "New File" → Select "R Script."
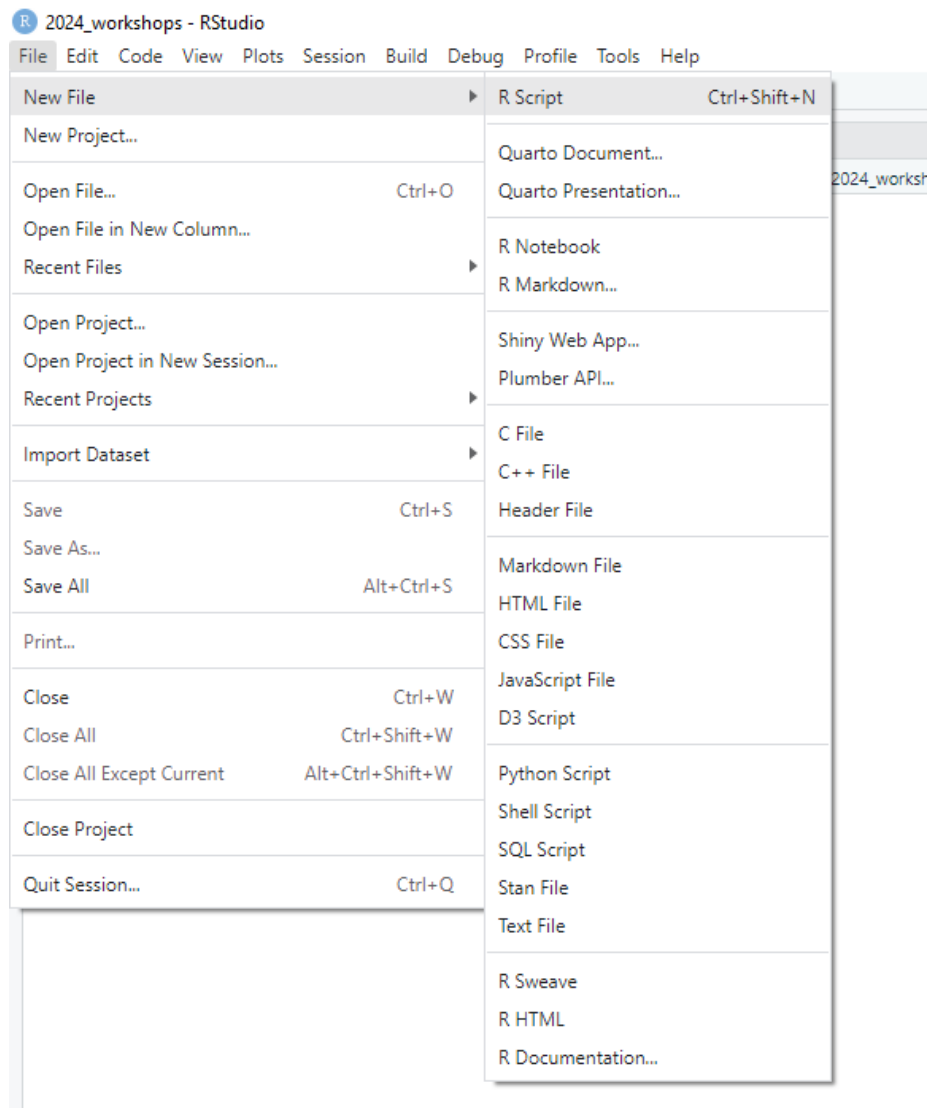
- **Button Shortcut**: Directly underneath the *File* tab, there is an icon of a white sheet with a green addition symbol. You can click that too.

- **Keyboard Shortcut:** Press "Ctrl" + "Shift" + "N" on Windows, or "Cmd" + "Shift" + "N" on Mac.

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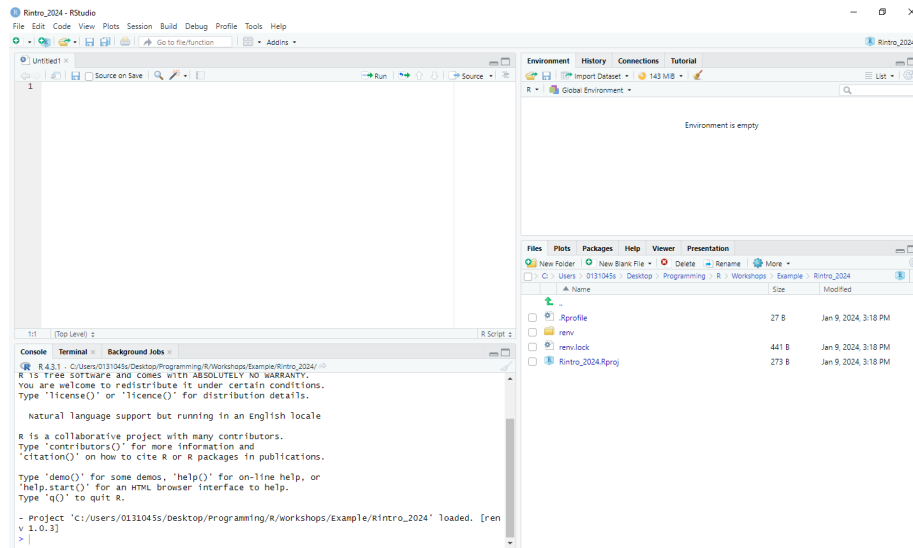
2.6.1.1 The RStudio Workspace

Let's briefly describe the purpose of each pane:

- **Source Pane**: Where you 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 code for computing descriptive statistics on your dataset. Similarly, you might have another R script titled "Regression" for performing regression analyses.

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

- **Environment Pane**: Displays information about datasets and variables imported or created in R within a specific project. The "History" tab shows a history of R code executed during the project. This pane is helpful for reviewing your work or returning to a project after some time.

- **Files Pane**: Includes project files (Files tab), outputs of plots you create (Plots tab), a list of downloaded packages (Packages tab), and help documentation about R functions (Help tab).

We will use all four panes extensively during these classes.

2.6.2   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:

```
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**).

2.6.3   Setting up a new Working Directory

In our R Project, we are going to create a folder for Week 1 of the workshop. Anything we create in R will then be saved into this folder.

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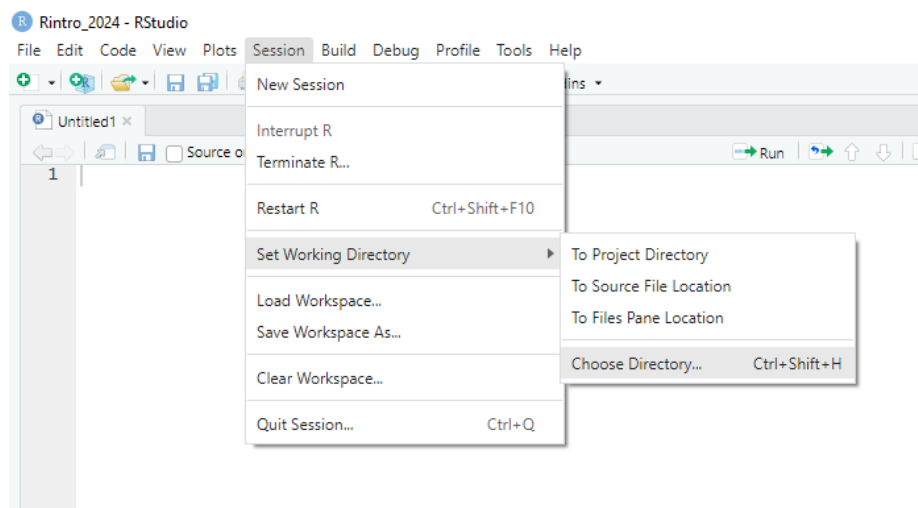
- By default, you should be in your R Project (e.g., ***rintro***).

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

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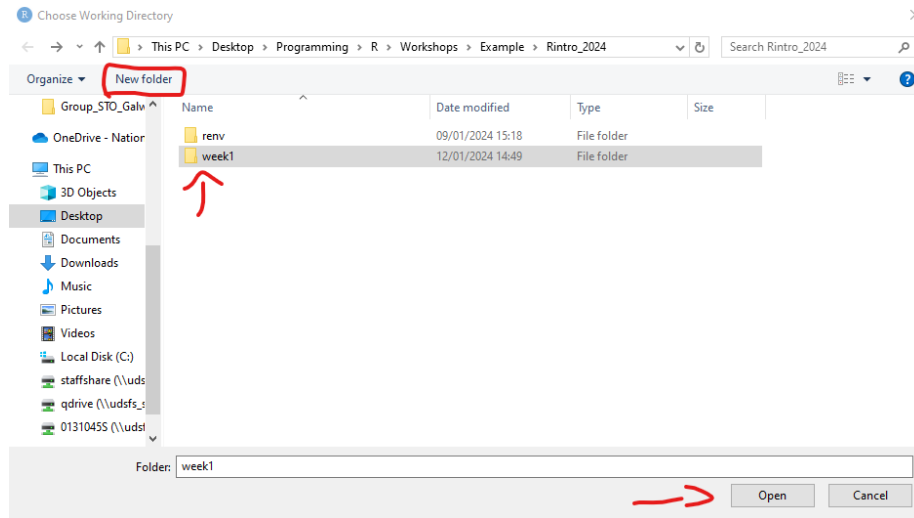
You should see something like the following in your console

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

Check whether this location is where you want to store your files for this course. If it is, you're all set. If not, let me know during class.

2.6.4   Changing some default settings

Like most applications, RStudio comes with default settings, some of which can be annoying if you use R frequently. One such setting restores your data and commands from your last session when you reopen RStudio. While this sounds helpful, it can cause issues if you're working on multiple projects or sharing code with others. Let's change this setting:

1. In the toolbar at the top of RStudio, click **Tools → Global Options**.

2. Under the **General** tab, look for the "Save workspace to .RData on exit" option. Change it to "Never." Untick the box as well.
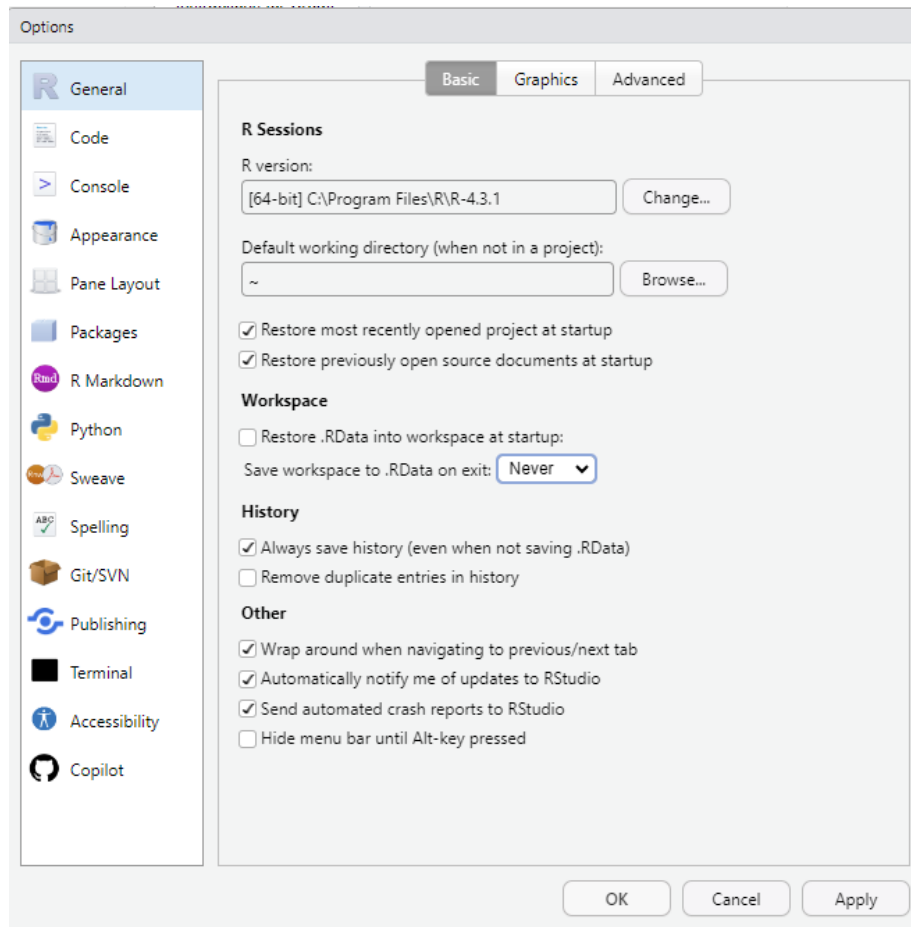
2.7   Writing our first R Code

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 > prompt) and press Enter. Feel free to modify the second line of code to include your own name:

```
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 actions. Here, R identifies the `print()` function, looks inside the parentheses to see what we want to print, and then displays the specified text. Pretty straightforward.

Functions are a core programming concept, and there is a lot more going on under the hood than I have described so far. We'll revist functions repeatedly and filling you in with more information. But in essence, functions are verbs that enable us to tell our computer what actions to perform.

2.8   Console vs Source Script

You might have noticed that I asked you to write code in the console rather than the source pane. Let's discuss the differences:

- **Console**: This is like having a live chat with R. You type commands, and R executes them immediately. The console is great for experimentation and quick feedback but not ideal for saving or organising your work.

- **Source Script**: This is where you write and save your code in a structured format. Scripts allow you to keep a record of your work, organise it into sections, and rerun it later. Think of the source pane as a document for your final code.

From now on, we'll write most of our code in R scripts. If I want you to use the console, I'll let you know.

2.9   Let's write some statistical code

Now that we've talked a lot about R and RStudio, let's write some code that will:

- Take a dataset

- Calculate descriptive statistics

- Generate a graph

- Save the results

Don't worry if you don't understand all the code provided below. Just follow along and type it yourself into the R script we opened earlier. If it's not open, click "File" → "New File" → "R Script." Save the script as **"01-sleep-descriptives."**

When you download R, it comes with several built-in functions (e.g., `print()`) and datasets. One of these datasets is called **sleep**, which we'll use here. To learn more about the **sleep** dataset, type **?sleep** into the console. You'll find more information in the "Help" tab in the Files pane.

First, let's take a look at the **sleep** dataset by writing the following code in your R script. To run scripts in R, highlight the code and click the "Run" button (with the green arrow) in the top right corner of the script pane.

```
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 displays the **sleep** dataset in the console. There are other functions to explore datasets, such as `head()`, `tail()`, `View()`, and `str()`. Try

these functions with the **sleep** dataset by typing them into the console to see their outputs.

From **print(sleep)**, we can see there are 20 observations (rows) with three variables (columns):

- **extra**: The extra hours of sleep participants had

- **group**: The treatment group they were assigned to

- **ID**: Their participant ID

2.9.1 Calculating Descriptive Statistics

Let's calculate some descriptive statistics using the **summary()** function. This function takes an object (e.g., a dataset) and summarizes its data. Write the following code in your 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
```

The **summary()** function provides descriptive statistics for each variable. For instance, it shows the mean change in hours of sleep (+1.5) and that there were 10 participants in each group.

However, this isn't quite what we need. For instance, we don't need descriptives for participant ID, and we want the mean scores split by treatment group. To get this, we can use the **aggregate()** function, which splits data into subsets and computes summary statistics for each subset. Add this to your script and run it:

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


#Here is what the code means
```

```
# code: data = sleep meaning: Go to the sleep data set

# code: extra ~ group meaning: Take the variable "extra" and split it into subsets bas

# code: FUN = mean meaning: Apply the mean() function (FUN) on each subset
```

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

Running this code shows the mean extra sleep for each treatment group. Participants in **group 2** slept an extra 2.33 hours on average, while participants in **group 1** slept only 0.75 hours more. It seems treatment 2 is more effective.

2.9.2   Creating a Visualisation

It's always a good step to create visualisations with your data to get a better picture of what is going on. Let's visualise our data with the `plot()` function.

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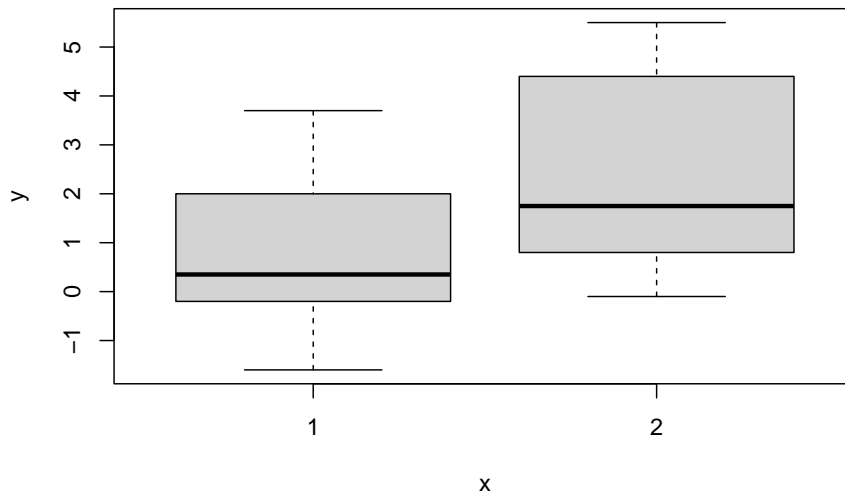

The `plot()` function automatically determines the most appropriate plot—in this case, a boxplot. While this plot is functional, we can make it more informative by adding a title and labels for the axes:

```
#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
```

Effect of Treament on Sleep Duration
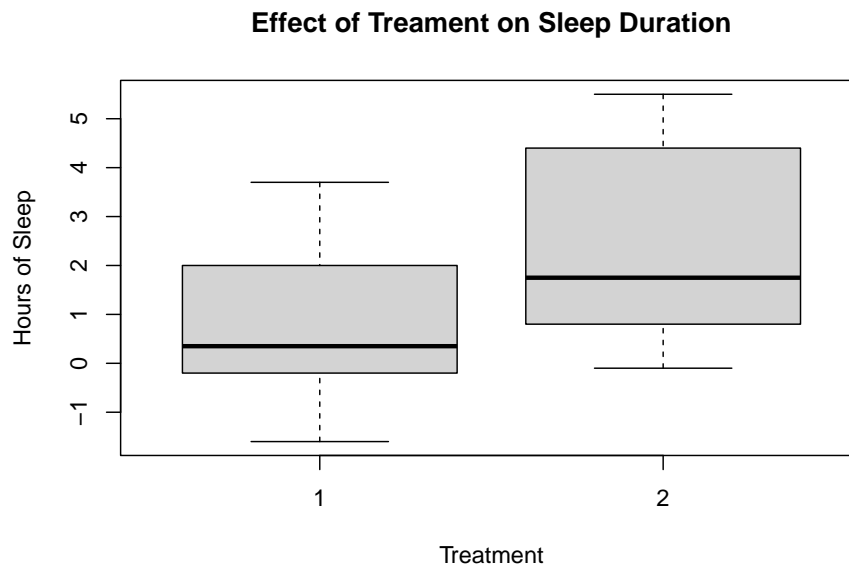
Figure 2.5: Generic Boxplot with appropriate labelling

This plot is more descriptive and also suggests a difference between the two groups. But is this difference meaningful? Later in the course, we'll learn how to evaluate whether differences like these are statistically significant using inferential statistical tests.

2.9.3 Saving the Plot

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 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.

```
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 = "Ef


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 up the PDF and have a look at your graph[2].

2.9.4   Comments

You might have noticed that I wrote several lines of text with a **#** before them. These are known as comments. Comments are pieces of text that R ignores— they are not executed as part of the code. They are fundamental to writing clear and understandable code.

We create comments using the **#** symbol. This tells R to ignore everything that comes **after** the **#** on the same line.

Comments serve a variety of purposes:

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

1. **Provide an Introduction**: It can be really useful here to provide clear information on what this script is trying to do, 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. **Structure Your Script**: 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. **Disable Code Temporarily**: 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 we wanted to temporarily

---

[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!
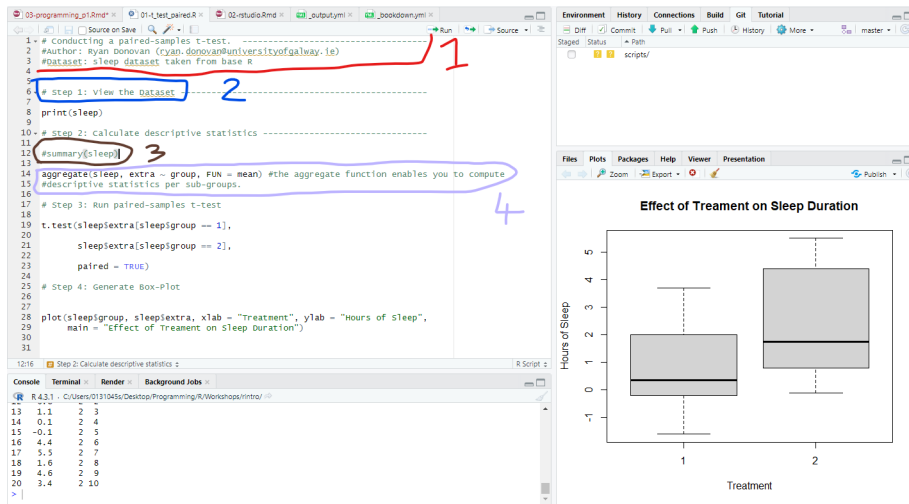
Figure 2.6: Four Examples of Comments Use

disable a piece of code but not delete it, because we think might be useful later. This is useful if you want to skip certain steps or keep old code for reference.

4. **Explain Specific Lines of Code**: 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.

2.9.5   Why Comments Are Important

Writing comments is like leaving notes for your future self (or for others working with your code). Imagine this scenario: you spend weeks creating a detailed R script to clean a messy dataset and run complex analyses. Months later, a reviewer or your supervisor asks for changes. When you reopen your script, you realise you've forgotten what each piece of code does! Without comments, you may spend hours trying to understand your own work.

By including comments, you save time and frustration. Additionally, comments help you solidify your understanding by requiring you to explain your code as you write it.

2.9.6   Best Practices for Comments

- Comment frequently: It's better to over-comment than to leave yourself guessing later.

- Keep comments concise but clear: Avoid overly verbose comments that are harder to read.

- Update comments if you change your code: Outdated comments can be misleading.

- Use comments to break your script into logical sections.

2.10   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 that will help you use R effectively, learn how to import data in R, and learn how to run descriptive statistics.

2.11   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.
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.

Term	Definition
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

Programming Fundamentals in R (Part I)

In this session, we are going to introduce fundamental programming concepts in R. In particular, we will learn important information about the syntax and rules of R, best practices on creating variables, the different ways that R stores, handles, and structures data, and how we can create and access that data.

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   How to read this chapter

If you are reading this chapter. I recommend that you type out every piece of code that I show on the screen, even the code with errors. The reason for this is that it will increase your comfortably with using R and RStudio and writing code. You can then test your understanding in the activities.

3.2   Activities

There are several activities associated with this chapter. You can find them by clicking this link.

3.3 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 create 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.

3.3.1 Typing Commands in the Console

We can type commands in the console to get R to perform calculations. Just a note, if you are typing these commands into the console, there is 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
```

R follows the BEMDAS convention when performing calculations (BEDMAS - Bracets, Exponents, Division, Multiplication, Addition, and Subtraction). So if you are using R for this purpose, just be mindful of this if the result looks different from what you expected.

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

[1] 84

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

[1] 12
```

---

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

You may have noticed that the output after each of line of code has `[1]` before the actual result. What does this mean?

This is how R labels and organises its response. 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.

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.

```
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. 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.

But why does it only show the `[1]` and `[16]`th index? 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.

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

3.3.2.1 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:

```
> 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 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 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 is clear what you have 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.

3.3.2.2   R Console and Incomplete Commands

I have been talking a lot of smack about 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

+
```

In this case, you just need to type finish the `)` next to the `+` symbol.

```
> (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 or press Esc
+
+
+
+ )

[1] 30
```

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.

3.4   Data Types

Our overarching goal for this course is to enable you to import your data into R, prepare it for analysis, conduct descriptive and statistical analysis, and create nice data visualisations.

Each of these steps becomes significantly easier to perform if we understand ***What is data and how is it stored in R?***

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

3.5   Basic Data types in R

R has 4 basic data types that are used 99% of the time. We will focus on these following data types:

3.5.1   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"
```

3.5.2   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"
```

```r
class(33.33)
```

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

```r
class(-1)
```

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

3.5.3 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.

```r
class(33L)
```

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

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

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

```r
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 technical 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 is 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.

3.5.4   Logical (otherwise know as Boolean)

The Logical data type has two possible values: **TRUE** or **FALSE**. In programming, we frequently need to 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[2]?

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
```

3.5.5   Data Types - So What?

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
```

---

[2]If you do not know what a p-value is, do not worry. We will introduce this concept later and discuss it extensively.

```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. This can be problematic if we need to perform statistical operations (e.g., calculating the mean) on that column. Luckily, there are ways to convert data types from one type to another.

3.5.6 Data Type Transformation

Following on from our previous example, we can convert a data type into numeric 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
```

3.6 Variables

Until now, the code we have 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 (in this case, **name**), use the assignment operator (**<-**) to inform R that we're storing information in **name**, and finally, provide the data that we will be stored in that variable (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 are technically correct. Fortunately, we can reassign our variable labels to new information.

```r
name <- "Brendan" #please do not call me this

print(name)
```

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

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

```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",
```

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

We can also 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, enabling us to create code that works across various situations.

3.6.1 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`. A rule of thumb is that if an acronym would make sense to anyone seeing the data, then use it. Otherwise, you a more descriptive variable name.

- 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.

3.7   Data Structures

So far, we have talked about the different types of data that we encounter in the world and how R classifies them. We have also discussed how we can store this type of data in variables for later use. 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.

Collections of variables like datasets are known as **data structures.** Data structures provide a framework for organising and grouping variables together. In R, there are several different types of data structures, with each structure having specified rules for how to create, change, or interact with them. For the final part of this session, we are going to introduce the two main data structures used in this course: `vectors` and `data frames`.

3.7.1 Vectors

The most basic and 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"
```

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

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

And I can create a numeric vector with their marks (which were randomly generated!)[3]

```r
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!):

---

[3]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.

```r
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:

```r
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 marks vector, look what happens when we try to add in a character grade to it.

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


print(rintro_grades)
```

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

So what happened here? Well, R has converted every element within the `rintro_grades` vector into a character. The reason for this is that R sees that we are trying to create a vector, but sees that there are different data types (numeric and character) within that vector. Since a vector can only have elements with the same data type, it will try to convert each element to one data type. Since it is easier to convert numerics into character (all it has to do it put quotation marks around each number) then characters into a vector (how would R know what number to convert A1 into?), it converts every element in `r_intro_grades` into a character.

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.

3.7.1.1 Working with Vectors

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

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
```

```
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.

3.7.1.2   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:

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

name	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

```
rintro_names[1]
```

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

Similarly, I could extract the 3rd element.

```
rintro_marks[3]
```

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

Or I could extract the last element.

```
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.

```
> 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

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:

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

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

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

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

```
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:

```
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 the variable **rintro_marks.**

```
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 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.

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)
```
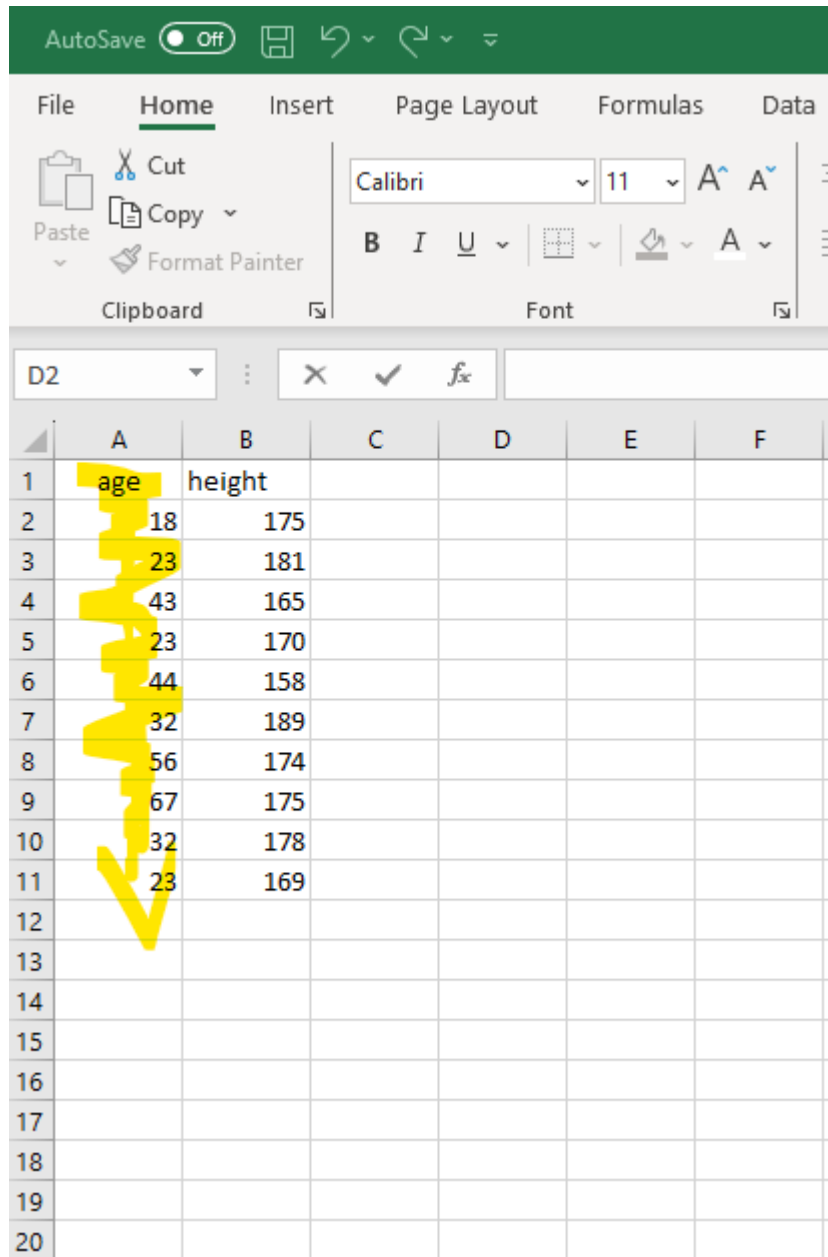
```
## [1] 92 88 86
```

3.7.1.3 Vectors - making it a little less abstract.

You might find the discussion of vectors, elements, and operations very abstract - I did when I was learning R. While the list analogy is helpful, it only works for so long before it becomes problematic, mainly because there's another data structure called **lists**. This confused me.

What helped me understand vectors was realising that a vector is simply a "line of data." Imagine we're running a study and collecting data on participants' age. When we open the Excel file, there will be a column called "age" with all the ages of our participants. That column is like a vector in R, containing a single line of data, where every value must be of the same type. For example, a column of ages in Excel becomes this vector in R:

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

Similarly, rows are lines of data going horizontally. Imagine, I collect data from another participant (p11). I could represent the data from that individual participant like this in R:

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

So whenever you think of a vector, just remember that it refers to a line of data, like a column or a row.
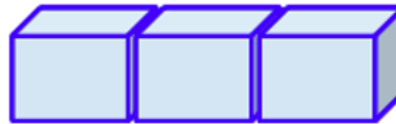
Figure 3.4: Vectors can visually conceptualised as a column or row of data.

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

3.7.2 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 spreadsheet in Excel or a table in a word document:

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:

```
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
```
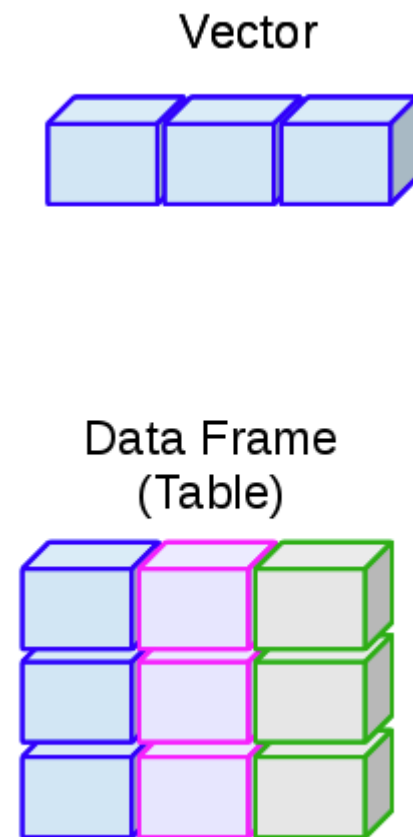
Figure 3.5: The relationship between data frames and vectors. The different colours in the data frame indicate they are composed of independent vectors

```
)

my_df
```

```
##      name age score
## 1   Alice  25 95.65
## 2     Bob  30 88.12
## 3 Charlie  22 75.33
```

Take a moment and think about what is going on inside `data.frame`. We have three variables `name`, `age`, `score`. Each of these variables correspond to a different type of vector (character, integer, and numeric). Or in other words, each of these variables correspond to different lines of data. We use the `data.frame` function to combine these vectors together into a table.

3.7.2.1 Selecting Data from a Data Frame

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

3.7.2.1.1 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 methods:

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:

```
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:

```r
my_df[, "age"]
```

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

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

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

```r
my_df[, c("age", "score")]
```

```
##   age score
## 1  25 95.65
## 2  30 88.12
## 3  22 75.33
```

3.7.2.1.2   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:

```r
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:

```r
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:

```r
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:

```r
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.

3.7.2.1.3 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:

```r
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:

```r
my_df2 <- my_df[c(1,3), c("age", "score")]

my_df2
```

```
##   age score
## 1  25 95.65
## 3  22 75.33
```

3.7.2.2   Adding Data to your Data Frame

3.7.2.2.1   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
```

3.7.2.2.2   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. And this new data frame will contain the new row(s) we want to add.

```
new_row <- data.frame(name = "John", age = 30, score = 77.34, gender = "Male")
```

Once we have this new data frame 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:

```
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
```

There is one important thing to note when adding rows. There must be the
same amount of columns as in the original data frame. Otherwise you will get
an error. See what happens when I try to add the following new row to our
data frame without adding the score column.

```
new_row2 <- data.frame(name = "Eric", age = 34, gender = "Non-binary")

my_df <- rbind(my_df, new_row2)
```

```
## Error in rbind(deparse.level, ...): numbers of columns of arguments do not match
```

```
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
```

The names in this error message refers to the names of the columns (in this
example, name, age, scores, and gender). Since our new_row2 is missing a
column name that is in my_df, we cannot add this row to the column.

But what if we don't have a score for Eric? Is there no way to add his result
to our data frame? There is. All we need to do is add the column **score** in
our **new_row2**, but give it the value of **NA**. The term **NA** basically means Not
Available - as in we don't know the value for this variable.

```r
new_row2 <- data.frame(name = "Eric", age = 34, score = NA, gender = "Non-binary")

my_df <- rbind(my_df, new_row2)

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
## 5     Eric  34    NA Non-binary
```

NA values can be quite common in real life datasets - sometimes data goes missing! But we'll come back to the concept of NA later on in this course and we will learn a variety of ways to deal with them.

3.8 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 remembered 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.

3.9 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.

Term	Definition
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.

3.10   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

Programming Fundamentals in R (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.

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

- Importing and exporting datasets using R.

4.1   How to read this chapter

If you are reading this chapter. I recommend that you type out every piece of code that I show on the screen, even the code with errors. The reason for this is that it will increase your comfortably with using R and RStudio and writing code. You can then test your understanding in the activities.

4.2   Activities

There are several activities associated with this chapter. You can find them by clicking this link.

4.3   Functions

In the previous sessions, we used several functions, like `print()`, `head()`, `View()`, `mean()`, `sd()`, `summary()`, `aggregate()`, `plot()`, `pdf()`, `class()`, and `c()`. Each of these functions performs a specific task: they take input (e.g., data or a variable), process it, and return an output. In a way, you can think of functions as action words—like verbs—that tell R to do something.

But what exactly *are* functions? Where do they come from?

Functions are basically a set of pre-written instructions that perform a specific task. Every function we'll use in this course has been written by someone else (or a group of people) and shared within the R community.

You might think getting good at R means memorising loads of functions and knowing exactly what each one does. Sure, that helps. But the real secret to improving at programming is understanding *how* functions work and how to create your own. Once you get that, you won't just use existing functions more effectively—you'll also be able to write your own custom functions when you need them.

Let's learn how to create our own function to get a better feel for how they work.

4.3.1   The Syntax for Creating a Function

Creating a function is a bit like creating a variable.

When we make a variable, we give it a name (e.g., `my_name`) and assign (`<-`) some information to it (e.g., `"Ryan"`). Then, we can use that variable wherever we need it, like this:

```r
my_name <- "Ryan"

print(my_name)
```

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

When we create a function, we give it a name (e.g., `my_function`) and assign a set of instructions to it. Once we've done that, we can call the function whenever we need to, instead of writing out the same code over and over again.

Here's the syntax for creating a function:

```r
my_function <- function(input) {
  instruction1
```

```
  instruction2
  ...
  instruction3
  return(output) # Return the result
}
```

Let's break this down:

- **Naming the Function** – We give our function a name (e.g., `my_function`) and use `<-` to assign something to it. Just like with variables, this tells R that the name will now store something.

- **Telling R It's a Function** – We write `function()` to let R know we're creating a function, not just storing data. Inside the parentheses `()`, we can include input information (also called an *argument*). Arguments are just placeholders for values that the function will use later.

- **Writing the Instructions** – Inside the curly brackets `{}`, we write the actual steps the function should follow. Some functions only need one step, while others might have multiple.

- **Returning the Output** – To make the function give us a result, we use `return()`. Whatever we include inside `return()` will be the final output that R displays.

This might still seem a bit abstract, so let's go through a concrete example.

4.3.2 Creating a Simple Function (1-Argument)

4.3.2.1 Defining the Function

You may be surprised to hear that I'm not much of a chef. One big reason is that recipes confuse me with measurements like "1 cup," "10 ounces," or preheating an oven to "1000 degrees Fahrenheit." Wouldn't it be great if we could automate these conversions?

That's where functions come in handy!

Let's create a simple function that converts cups to grams. We'll call it `cups_to_grams`. When naming functions, it's best to pick a name that clearly describes what the function does—it makes your code much easier to read and understand later.

Here's how we start:

```
cups_to_grams <- function(cups) {

}
```

What's Happening Here?

- Inside `function()`, we've written `cups`, which is the input or *argument.* Think of it as a placeholder for the value we want to use later.

- Right now, our function doesn't do anything because we haven't given it any instructions yet. Let's fix that.

4.3.2.2 Writing the Instructions

According to the metric system, 1 cup is approximately 250 grams. So, our function needs to multiply the number of cups by 250.

Let's add this instruction:

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

We've created a variable called `grams` inside the function. It stores the result of `cups * 250`.

But there's still one problem—if we run this function, nothing will actually be displayed on the screen. That's because we haven't told R to return the result yet.

4.3.2.3 Returning the Output

Here's the final version of our function:

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

The `return()` function tells R to keep the value of `grams` and print it out in the console.

Now, let's see if it works!

4.3.3 Calling the Function

Calling a function is just like using any of the built-in functions we've used before. We type the function's name and provide an input inside the parentheses.

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

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

This tells R: "Take **1 cup**, multiply it by 250, and give me the result." The output should be **250 grams**.

4.3.3.1 Using the Function with Different Inputs

You can call the function multiple times with different values:

```
cups_to_grams(cups = 4)     # 4 cups = 1000 grams
```

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

```
cups_to_grams(cups = 2)     # 2 cups = 500 grams
```

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

```
cups_to_grams(cups = 1.5)   # 1.5 cups = 375 grams
```

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

```
cups_to_grams(cups = 5L)    # 5 cups = 1250 grams
```

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

4.3.3.2 Defining the Argument Outside the Function

Instead of passing the value directly inside the function, you can also define it separately and then use it:

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

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

4.3.3.3   Key Takeaways

- This is an example of a **1-argument function** because it only takes in one input (**cups**).

- The placeholder (**cups**) makes the function reusable. It doesn't assume what the input would be ahead of time, so you can use it with any value.

One of the most powerful features of functions is that it can take multiple input values, or multiple arguments. In the next section, we'll learn how to create functions with multiple arguments.

4.3.4   Creating a Multi-Argument Function

A **multi-argument function** is a function that takes *more than one input* to perform its task. These inputs are called **arguments**, and they allow the function to work with different pieces of information.

For example, imagine you're making pancakes. You need flour, sugar, baking powder, and butter to bake them. Each of these ingredients is like an argument — the recipe (function) needs all of them to do its job properly.

The process for creating a one-argument function is practically identical to creating a multi-argument function. The only difference is that we need to enter into more than one input when we define the function.

```r
my_function <- function(input1, input2, input...3) {
  instruction1
  instruction2
  ...
  instruction3
  return(output) # Return the result
}
```

Let's create a function called `average_score()` that calculates the average score on a psychological test when given the total score and the number of test items.

```r
average_score <- function(total_score, num_items) {
  # Calculate the average
  average <- total_score / num_items

  # Return the average
  return(average)
}
```

You can see we have provided two inputs `total_score` and `num_items` inside `function()`. This tells R that it should expect multiple inputs when this function is called. Again, we have not defined what the values of `total_score` or `num_items` is ahead of time, because we want our function to be reusable and adaptable.

Let's give our function a test drive with multiple different possible input values.

```
average_score(total_score = 30, num_items = 10)
```

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

```
average_score(total_score = 40, num_items = 5)
```

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

```
average_score(total_score = 78, num_items = 17)
```

```
## [1] 4.588235
```

The function carries out the same set instructions regardless of what values we insert for `total_score` and `num_items`.

4.3.4.1 What Happens if I Don't Name the Inputs?

You don't always need to name the arguments when calling the function *as long as* you provide them in the correct order. For example:

```
average_score(65, 5) #what happens if I do not write out the input variable names?
```

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

This works because R remembers the order of the inputs (`total_score` first, then `num_items`).

However, what if you mix up the order of the inputs?

```
average_score(5, 65)
```

```
## [1] 0.07692308
```

The function still runs, but the result is wrong. Why? Because R doesn't have common sense — it doesn't know you meant to input the `total_score` as 65 and the `num_items` as 5. It blindly follows the order you gave.

To avoid this kind of mistake, it's a good idea to **specify the names of your input variables** when calling a function. This way, you make your code clearer and less prone to errors. It's also great practice when sharing your code with others — naming the inputs makes your code more readable, especially if someone hasn't seen the function before.

4.3.5 Some Important Features about Functions

Before we finish up on functions, there are some important features about functions that you should know. Namely, the difference between Global and Local Variables, the ability to set and override Default Arguments or inputs, and the ability to search for help on functions in R (and why it's not always the best idea).

4.3.5.1 Global vs Local Variables

When working with R, it's important to understand that variables you define *inside* a function are treated differently from those you define *outside* a function. Here's the key idea:

- Variables defined **inside a function** are called *local variables*. They only exist while the function is running and can't be seen or used anywhere else.

- Variables defined **outside a function** are called *global variables*. These can be used anywhere in your script, including inside functions.

Again this is all abstract and boring, so let's use an example.

I'm going to create a simple function called `my_favourite_number_generator()`. This function will always tell you my favourite number (spoiler: it's 7). It's not the most useful function in the world, but it'll help us understand local and global variables.

```r
my_favourite_number_generator <- function() {
  my_favourite_number <- 7  # This is a local variable
  return(my_favourite_number)  # The function returns 7
}
```

Here's what's happening:

- I define the variable `my_favourite_number` *inside* the function. This makes it a local variable — it only exists while the function is running.

- When I call the function, it will return the value of the local variable (7).

Let's try it out:

```
my_favourite_number_generator()
```

```
## [1] 7
```

It prints 7, just as expected.

Now, let's see what happens when I define a global variable called `my_favourite_number` *outside* the function, and then call the function again:

```
my_favourite_number <- 10
my_favourite_number_generator()
```

```
## [1] 7
```

Even though we defined `my_favourite_number` globally as 10, the function still returns 7. Why? Because the function uses its *local* version of `my_favourite_number` — it doesn't care about the global variable.

Finally, let's check what happens if we print the global variable after calling the function:

```
my_favourite_number <- 10 #global variable

my_favourite_number_generator()
```

```
## [1] 7
```

```
print(my_favourite_number)
```

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

The function only works with its *local* variable (7), but the global variable (10) remains unchanged outside the function.

Key Takeaway

- **Local variables** exist only inside the function and disappear once the function finishes running.

- **Global variables** exist outside the function and can be used anywhere in your script.

If you keep these ideas in mind, you'll avoid common mistakes and better understand how functions work in R!

4.3.5.2   Default Arguments

Default arguments in R are like having a "backup plan" for your functions. You can assign a **default value** to an argument when you define the function. This means that if you don't provide a specific value for that argument when you call the function, R will automatically use the default.

Think of it like ordering coffee at a café. If you don't specify the type of milk you want, they might use regular milk as the default. But if you do specify oat milk, they'll make it with oat milk instead. Default arguments in R work the same way — you can stick with the default or override it with your own value.

We've actually already seen a default argument in action! Remember the `sort()` function? By default, it sorts numbers in **ascending order** (from smallest to largest). But if you add the argument `decreasing = TRUE`, it sorts the numbers in **descending order** (from largest to smallest).

Here's how it works:

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

# Using the default (ascending order)
sort(rintro_marks)  # Output: 65, 69, 71, 77, 80, 86, 88, 92
```

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

```r
# Overriding the default to sort in descending order
sort(rintro_marks, decreasing = TRUE)  # Output: 92, 88, 86, 80, 77, 71, 69, 65
```

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

In this example:

- The default value of `decreasing` is `FALSE`. Unless we change it, it will sort from smallest to largest.

- By specifying `decreasing = TRUE`, we override the default and sort from largest to smallest.

Now, let's write our own function that uses a default argument. Imagine I want a function called `greet()` that says hello to someone. If I don't specify who to greet, the function will say "Hello, World!" by default.

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

Here's what happens when we call the function:

Using the default value:

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

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

- Since we didn't specify a name, the function used the default value of `"World"`.

- **Overriding the default value:**

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

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

By specifying `name = "Ryan"`, we told the function to use `"Ryan"` instead of the default `"World"`.

4.3.6   How to Search for Help on Functions in R

When learning R, you'll often come across functions that you don't fully understand or need more details about. Thankfully, R has built-in tools to help you learn how functions work and how to use them properly.

1. **Using the ? Operator**

The easiest way to find help for a specific function is to type a `?` followed by the name of the function. This will bring up the function's help page, which includes:

- A description of what the function does.

- A list of arguments the function takes.

- Examples of how to use the function.

```
?mean
```

2. **Using `help()`**

Another way to access a function's help page is by using the `help()` function. This works the same way as the `?` operator.

```
help(mean)
```

3. **Searching for Keywords with ??**

If you don't know the exact name of a function but have a general idea of what you're looking for, you can use `??` followed by a keyword. This searches through R's documentation for functions or topics related to that keyword.

Example:

If you're looking for functions related to "regression":

```
??regression
```

This will show you a list of related functions and packages.

4. **Using the Help Tab in RStudio**

If you're using RStudio, the **Help** tab (in the bottom-right panel) is a great resource. You can:

- Search for a function or topic in the search bar.

- Browse R documentation for built-in functions and packages.

- View the help page for any function you're working with.

5. **Online Resources**

If the built-in documentation isn't enough, you can find additional help online:

- **R Documentation website**: https://rdocumentation.org

- **Stack Overflow**: A forum where R users ask and answer questions.

- **CRAN Vignettes**: Many R packages include detailed tutorials or "vignettes" that explain how to use them. You can search for these on CRAN (https://cran.r-project.org)).

4.3.6.1   When R's Help Pages Are Difficult to Understand

While R's built-in help pages are useful, they can sometimes feel overwhelming, especially for beginners. The language might be too technical, or the examples might not clearly show what you need. If this happens, don't worry — it's very common (I still come across help pages that I don't understand), and there are plenty of other ways to find help.

1. **Googling for Help**

When the help page doesn't make sense, a quick Google search can often provide more beginner-friendly explanations or practical examples. Here's how to effectively search for help online:

- Include the function name and the word "R" in your search query.

  For example:

```
"mean function R example"
```

2. **YouTube and Tutorials**

Sometimes, written documentation isn't enough. Watching a tutorial on YouTube can be incredibly helpful for visual learners. Search for:

- "[function name] in R tutorial"

- "How to use [function name] in R"

3. **Ask Us**

We are here to help!

4.3.7   PSA: Don't Feel Frustrated

It's important for students to know that *nobody memorises everything in R*. Even experienced programmers regularly Google or use forums to find help. We do it all the time. The goal is not to memorise everything but to know how to find the resources you need.

4.4   R Packages

Throughout this course, we've been exploring the tools that come with **base R**, which is the version of R you get right out of the box. Base R provides many useful functions for tasks like data analysis and visualisation. However, one of the biggest advantages of R is that you can expand its capabilities significantly by using **packages**.

Think of packages like apps on your phone. When you first set up your phone, it comes with essential apps like a web browser and maps. But you'd missing out on a lot of the potential for your smartphone if you only used the basic applications. To make more use of your phone, you might download additional apps like Spotify or Netflix.

Similarly, R packages allow you to "install" extra features for R, created by the R community, that can make your programming life much easier.

4.4.1   What Are R Packages?

R packages are **add-ons** that extend the functionality of R. They often contain:

- **Functions**: Special tools to make tasks easier (e.g., functions to visualise data or run statistical tests).

- **Datasets**: Pre-loaded data that you can use for practice.

- **Documentation**: Guides on how to use the package.

R packages are created by people in the R community and shared for free. They're often designed to solve specific problems, such as making a statistical test easier to run or creating beautiful graphs. By using R packages, you benefit from the hard work and expertise of others — in a sense, you're standing on the shoulders of R giants.

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[1].

----

[1]There are a couple of important exceptions to this rule.

Firslty, If you are creating R projects on your own personal computers using the `renv` option (which we did), this basically partitions your R project into its own little self-contained space on your computer where it will operate. So when you download and install R packages in that project, they will only exist in that location on your computer (i.e. your project). You won't need to install packages on that project more than once. However, if you set up a new

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.

We do something similar when we download apps on our phone. Once you download Spotify, you don't need to install it again and again every time you want to use it. However, if you do want to use it, you will need to open (i.e. load) the application.

4.4.2.1 Installation

We are going to install three packages - pwr, jmv, and praise. We will be using jmv and pwr packages later on in the course, but we will install them for now. The praise package provides users with, well, praise. And the pwr package will spit out statistic and programming quotes at you. It's not particularly useful, other than demonstrating the process of loading packages.

There are two main ways to install packages in R:

Using the RStudio Interface:

1. Look for the **Packages** tab in the bottom-right pane of RStudio.

2. Click the "Install" button above the list of installed packages.

3. In the pop-up window, type the name of the package (e.g., `pwr`) and make sure "Install dependencies" is checked.

4. Click "Install." You'll see messages in the console that indicate the installation process.

```
> install.packages("pwr")
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
```

---

project, and you need the same packages, then you will need to install those packages on that new project.

This is equivalent to having separate user accounts on your desktop. If I install something on my account, that doesn't mean it will be installed on your account.

Secondly, the same holds true for Posit Cloud. In Posit Cloud, every new R project you make will be self-contained, so any packages you install in one project will not transfer over to other projects.

Don't worry about the "scary" red text — it's normal and means the package is being installed correctly.

Using R Commands:

You can also install packages directly by typing the `install.packages()` command in the console. For example:

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

The important thing here is that whatever goes inside the parentheses is inside quotation marks.

You can even install multiple packages at once:

```
install.packages(c("jmv", "praise"))
```

```
trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-x86_64/contrib/4.3/praise_1.0.(
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/pwr_1.5-4.t
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 'pwr' 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

Once a package is installed, you need to **load it** into your R session using the `library()` function. You'll need to do this every time you start RStudio and

want to use the package.

Using the RStudio Interface:

1. Go to the **Packages** tab.

2. Scroll down to find the package you want (e.g., `praise`) and tick the checkbox next to its name. This loads the package.
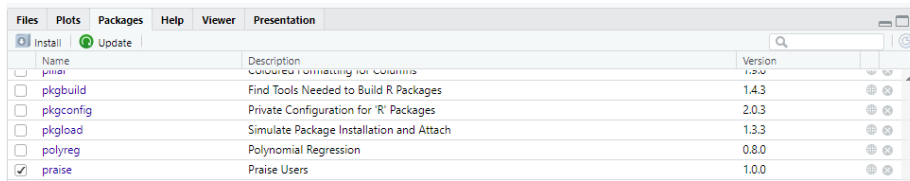

Figure 4.1: Loading Packages through RStudio Interface

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 `pwr` and `jmv` packages, you can type each of these into your script and run each one.

```
library(pwr)
library(jmv)
```

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

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


#This code will work
library(package1)
```

```r
library(package2)
library(package3)


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

4.4.4   Testing a Package

Let's test the `praise` package, which generates random compliments (not groundbreaking, but fun to use!). After loading the package, run:

```
## Warning: package 'praise' was built under R version 4.3.3
```

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

```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
```

Every time you run `praise()`, you'll get a different compliment. Try it out!

4.4.5   Troubleshooting Common Issues

1. **Error: "There is no package called. . . "**

This means the package hasn't been installed. Use `install.packages("package_name")` to install it.

2. **Error: "Package was built under a different version of R"**

This is a warning that your version of R might be out of data. Usually it can be ignored, but if you are unable to use the package, then you will need to download the latest version of R (using the same steps we used to download it in Chapter 2!).

3. **Dependencies Missing**:

If you see a message about missing dependencies, reinstall the package with "Install dependencies" checked, or run:

```r
install.packages("package_name", dependencies = TRUE)
```

4. **Conflicting Function Names Between Packages**:

If two loaded packages have functions with the same name, R will use the version from the package loaded most recently. You will typically see a message that looks like this in your R console

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag
```

Here's what this means:

R is informing you that the functions filter and lag from the stats package (which comes with base R) are now "masked" by the versions from the dplyr package.

This simply means that if you call filter() or lag() without specifying the package, R will use the version from dplyr because it was loaded most recently.

How to Handle This:

If you see this message, it's not an error — it's just a warning that there's a conflict. To explicitly use a function from a specific package, you can use the :: operator

```r
stats::filter()   # Use the filter function from stats
dplyr::filter()   # Use the filter function from dplyr
```

4.4.6   Best Practices for Installing and Loading Packages

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

1. **Load Packages at the Top of Your Script:** Place all `library()` calls at the very top of your R script. This helps others see which packages are required for your code.

2. **Avoid Running `install.packages()` in a Script:** Use `install.packages()` in the console, not in your script. If someone downloads your script and accidentally runs it, it will automatically install the packages on their computer. Generally it's better for people to make a decision themselves on whether they want to install anything on their computer. By not writing the `install.packages()` command in our script, we give them the power.

3. **Comment Out Install Commands:** If you include `install.packages()` in your script for documentation purposes, make sure it's commented out:
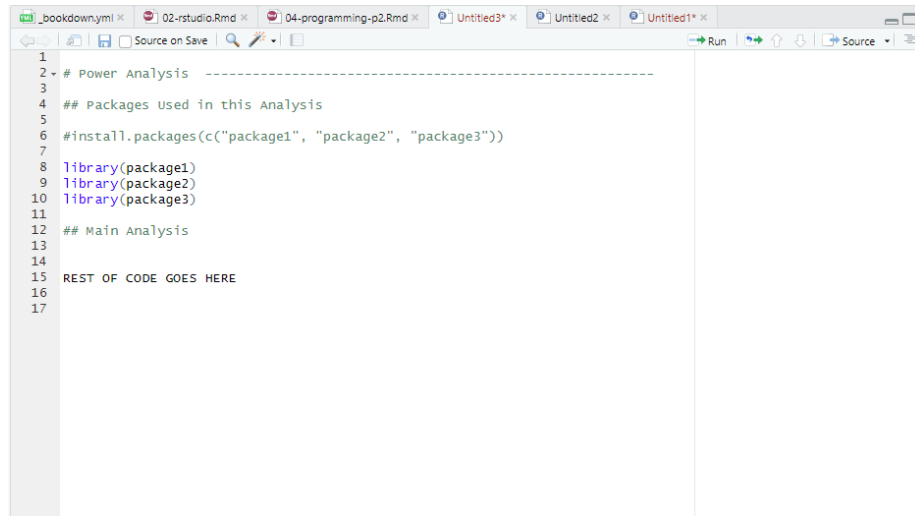

Figure 4.2: Conventions for Installing and Loading Packages in R Script

4.4.7   Summary

There you have it. You have successfully installed and loaded your first packages in R. The praise package is not exactly groundbreaking, but we will be coming back to the jmv and pwr packages later on.

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 analysed your data in R, you'll often need to export it for further use or sharing.

First, you'll need to download two files: `psycho.csv`. Both files are available on Canvas.

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. Make sure you have download (or uploaded) the "psycho.csv" file to your `week3` folder

2. In your script, write and run the following line of code:

```r
getwd()
```

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

```r
psycho_df <- read.csv("datasets/psycho.csv")
```

3. Once you have run that line of code, you can have a look at the data frame by using the `head()` and `summary()` functions.

```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 Exporting Datasets in R

After analysing 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.

4.5.2.1 Exporting to CSV Files

To export a dataset to a CSV file, we 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 export a copy of our psycho dataframe:

```
write.csv(psycho, file = "psycho_copy.csv")
```

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

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 categories 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.

For the next few weeks, we will focus on using R to run descriptive and inferential statistical analysis.

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.

Chapter 5

Descriptive Statistics and T-Tests in R

In this weeks workshop, we are going to learn how to perform descriptive statistics and conduct both independent and paired-samples t-tests (which you covered in today's lecture).
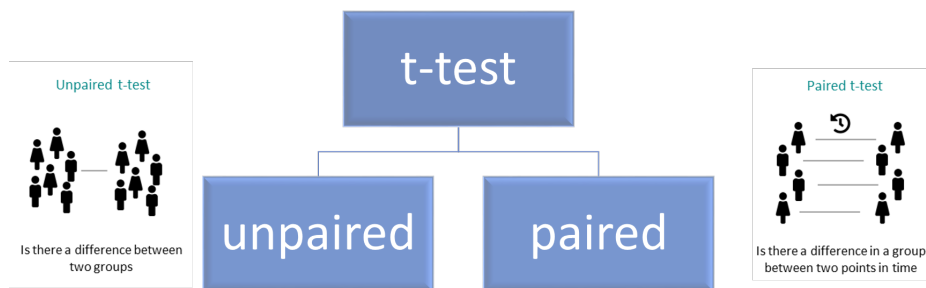

Figure 5.1: A graphical image illustrating the two main types of t-test. The first shows two groups of people with the header "unpaired-t-test" and a caption underneath reading "Is there a difference between two groups". The second shows the same group of people with a clock between them, the caption "paired t-test" and written underneath is "Is there a difference in a group between two points in time".

Additionally, we will learn how to check parametric assumptions in R. By the end of this session, you will be able to:

- Use the jmv package to run descriptive statistics and check parametric assumptions.

- Conduct an independent samples t-test in R.

- Conduct a paired-samples t-test in R.

- Conduct an apriori power analysis in R for t-tests.

5.1   How to read this chapter

This chapter aims to supplement your in-lecture learning about t-tests, to recap when and why you might use them, and to build on this knowledge to show how to conduct t-tests in R.

5.2   Activities

As in previous sessions there are several activities associated with this chapter. You can find them here or on Canvas under the **Week 4 module**.

5.3   Between-Groups Comparisons

For our between group comparisons we will be using the **wellbeing.csv** data, which we have saved as **df_wellbeing**

This data was collected from an experimental study investigating the effects of **Caffeine** (Low Caffeine, and High Caffeine) on various outcome variables, including experiences of pain, fatigue, depression, and overall wellbeing. Additionally, participants' age and gender were recorded.

After loading the datasets, it's always good practice to inspect it before doing any analyses. You can use the **head()** function to get an overview of the wellbeing dataset:

5.4   Descriptive Statistics

Descriptive statistics (such as the mean and standard deviation) give us important additional information to the results of our statistical tests. As such it is important to calculate them, and to include them in your write-up.

5.4.1 Writing up the results of an independent samples t-test

The results of a t-test are typically written up as such:

A Welch independent samples t-test was perfomed and **a / no** significant difference on performance was found between group A (M= **Group A mean value**, SD= **Group A standard deviation**) and group B (M= **Group B mean value**, SD= **Group B standard deviation**) on the task t(**df value**)= **t statistic value**, p=**exact p value to two or three decimal places**), with a large effect (d= **Cohen's d to two decimal places**, 95% CI [**lower bound - higher bound**]).

In the associated activities with this chapter (specifically Activity 4) you will learn how to use the descriptives function from the jmv package to get the descriptive statistics you need to fill in the blanks above.

5.5 Statistically comparing between 2 groups: Independent samples t-test (unpaired t-test)

Let's imagine we're interested in investigating the effects of caffeine consumption on levels of self-reported health. Specifically, we want to determine whether people in the high caffeine condition scored significantly differently from those in the low caffeine condition.

In this case:

Our **independent variable** is caffeine consumption group (low caffeine vs high caffeine)

Our **dependent variable** is health

We could specify our hypothesis as such:

H1: We predict there will be statistically significant difference in reported health levels between people in the high versus low caffeine conditions.

H0 (Null hypothesis): There will not be a statistically significant difference in reported health levels between people in the high versus low caffeine conditions.

Note that as we do not specify which group will have higher/lower health levels this is a **nondirectional hypothesis**

As we have two independent groups we want to compare, this would be best addressed via an **independent samples t-test**. Before we can do this, there are a couple of preliminary steps we need to take. First, we need to check the parametric assumptions required for an independent samples t-test.
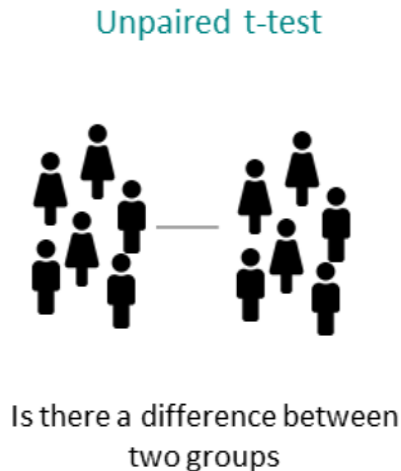
Figure 5.2: A graphical image which shows two groups of people with the header "unpaired-t-test" and a caption underneath reading "Is there a difference between two groups".

Checking our parametric assumptions

There are several key assumptions for conducting an independent samples t-test. We don't need R to check the first three assumptions (a-c). A quick visual inspection of the dataset and knowledge of the design will tell us whether these are met, and in this case, they are.:

a. The dependent variable should be measured on a continuous scale.

b. The independent variable should consist of two categorical, independent groups.

c. The groups should be independent of each other.

d. There should be no significant outliers. We can check this assumption via a boxplot, which will flag any outliers by placing numbers next to them.

e. The dependent variable should be approximately normally distributed for each group. We can check this using a Shapiro-Wilk test. When interpreting Shapiro-Wilk A **significant p-value (p < 0.05)** means that the *assumption has been violated*, and you cannot run an independent samples t-test. A **not significant p-value (p > 0.05)** means that the *assumption has been met*, and you can proceed.

f. The dependent variable should exhibit homogeneity of variance. We can check this using Levene's Test using the function *levenetest* from the *car* package. The syntax for this is below:

```
leveneTest(DependentVariable ~ IndependentVariable, data = ourdataset)
```

When interpreting Levenes test A **significant p-value (p < 0.05)** means that the *assumption has been violated*, and you cannot run an independent samples t-test. A **not significant p-value (p > 0.05)** means that the *assumption has been met*, and you can proceed.

Activity 5 in the associated Activities for this week will show in detail how to check assumptions d-f. Once again we will be using the **descriptives** function for this.

For future reference you could use code like that below to check your assumptions (except for homogeneity of variance) **and** get your descriptive statistics

```
descriptives(dataframe_name, # our dataset
             vars = "DV", # our DV
             splitBy = "IV", # our IV
             ci = TRUE, # this outputs confidence intervals to include in your descriptives
             box = TRUE, # this outputs boxplots, so you can check for outliers
             sw = TRUE) # this conducts a shapiro-wilk test to check if our dependent variable is
```

If all of the assumptions are met you can proceed to running the independent samples t-test.

We use the **t.test function** to perform the t-test. The syntax is:

```
t.test(DV ~ IV,
       paired = FALSE,
       alternative = c("two.sided", "less", "greater"),
       data = ourdataset)
```

In this function:

DV ~ IV specifies the **dependent variable (DV)** and **independent variable (IV)**.

paired = FALSE indicates that we are conducting an independent samples t-test. If we were comparing related groups (e.g., pre-test vs. post-test), we would set paired = TRUE.

The **alternative** argument specifies the hypothesis test type:

"two.sided" tests for any difference between groups.

"less" tests whether the first group has a lower mean than the second.

"greater" tests whether the first group has a higher mean than the second.

As we have a **non-directional hypothesis** we will run a *two-sided* t-test on our df_wellbeing dataset.

Remember that in Psychology we set our alpha level as 0.05. That means that in interpreting the results of your t-test.

If your p value is **less than** 0.05 (aka p < 0.05) then you have a **statistically significant** effect.

If your value is **greater than** or equal to 0.05 (aka p > 0.05) then you do not have a **statistically significant** effect

5.6   Effect sizes!

The final thing to calculate before writing up the results of our t-test is the effect size. If you recall for t-tests we use Cohen's d as our effect size of interest (helpful visualisation here to recap). Cohen's d has the following rules of thumb for interpretation:

d = 0.2 is a small effect

d = 0.5 is a medium effect

d = 0.8 is a large effect

We can use the below code to calculate the effect size for the independent samples t-test which we ran this week. We need to do a little data wrangling to run this for our paired-samples t-test, which is fine, but we'll come back to it in later weeks.

```
library(effectsize) #that calculates the effect size - note we may need to install thi
cohens_d(health ~ condition, # DV ~ IV
            paired = FALSE, #paired = FALSE for independent samples, but would be tr
            data = df_wellbeing) # our data
```

Writing up the results:

After following the steps outlined above/in the activities you should be able to fill in the below blanks to write up the results of the independent samples t-test.

A Welch independent samples t-test was perfomed and **a / no** significant difference on performance was found between group A (M= **Group A mean value**, SD= **Group A standard deviation**) and group B (M= **Group B mean value**, SD= **Group B standard deviation**) on the task t(**df value**)= **t statistic value**, p=**exact p value to two or three decimal places**), with a large effect (d= **Cohen's d to two decimal places**, 95% CI [**lower bound - higher bound**]).

Figure 5.3: A picture that says I love effect sizes.

5.7   Within-Subjects Comparisons

For our within-subjects comparisons we will be using the **reading.csv** data, which we have saved as df_reading

This data was collected from a reading intervention study investigating the effects of a literacy intervention, comparing the same participants before and after the intervention. As such we will be comparing participants reading at: **Baseline**, and **Time2**.

A lot of the steps for conducting within-subjects comparisons are very similar to between-groups comparisons. So we can refer to the above sections for help if we get unsure.

5.8   Descriptive Statistics for a paired-samples t-test

A paired-samples t-test was perfomed and **a / no** significant difference on **dependent variable** was found between **Baseline** (M= **Baseline mean value**, SD= **Baseline standard deviation**) and **Time 2** (M= **Time 2 mean value**, SD= **Time 2 standard deviation**) on the task t(**df value**)= **t statistic value**, p=**exact p value to two or three decimal places**), with a **large** effect (d= **Cohen's d to two decimal places**, 95% CI [**lower bound - higher bound**]).

Once again you can use the **descriptives** function to fill in the descriptive statistics needed above.

Let's imagine we're interested in investigating the effects of our intervention on levels of reading ability. Specifically, we want to determine whether children's reading ability was significantly better at Time 2 (after intervention) compared to at baseline (before the intervention).

In this case:

Our **independent variable** is time (Baseline vs Time2)

Our **dependent variable** is reading ability

We could specify our **hypothesis** as such:

H1: We predict there will be statistically higher reading ability in children at Time 2 (after intervention) than at Baseline.

H0 (Null hypothesis): There will not be statistically significant higher reading ability in children at Time 2 (after intervention) than at Baseline.

Note that as we do specify which condition will have higher reading ability this is a **directional hypothesis**

As all participants take part in both conditions (e.g. are tested at two time-points), this would be best addressed via a **paired samples t-test**. Before we can do this, there are a couple of preliminary steps we need to take. First, we need to check the parametric assumptions required for a paired samples t-test.
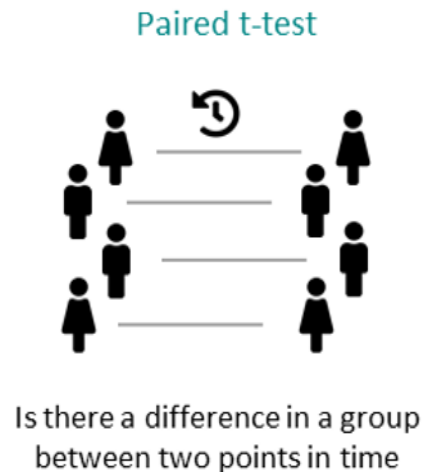

Figure 5.4: A graphical image which shows the same group of people with a clock between them, the caption "paired t-test" and written underneath is "Is there a difference in a group between two points in time".

Checking our parametric assumptions

a. Our dependent variable should be measured on a continuous scale

b. The observations are independent of one another

c. There should be no significant outliers

d. Our dependent variable should be normally distributed

Here, it's only really the outliers and the normal distribution that needs to be evaluated using functions in R, both of which we can assess using the **descriptives** function. The below code will output the descriptive statistics you need, **and** check your assumptions.

```
descriptives(df_reading, # your dataframe name here
             vars = c("Baseline", "Time2"), # your two conditions / timepoints
             ci = TRUE, # this outputs confidence intervals to include in your descriptives
             box = TRUE, # this outputs boxplots, so you can check for outliers
             sw = TRUE) # this conducts a shapiro-wilk test to check if our dependent variable is
```

We use the t.test function to perform the t-test. Here the syntax is:

```
t.test(ourDataset$Condition1, ourDataset$Condition2, # here we specify our two conditi
        paired = TRUE, # here we specify that it is a paired samples t-test
        alternative = c("two.sided", "less", "greater")) #here you specify whether the
```

Once you have run the t-test you should have the information to replace the information in bold with the correct information.

A paired-samples t-test was perfomed and **a / no** significant difference on **dependent variable** was found between **Baseline** (M= **Baseline mean value**, SD= **Baseline standard deviation**) and **Time 2** (M= **Time 2 mean value**, SD= **Time 2 standard deviation**) on the task t(**df value**)= **t statistic value**, p=**exact p value to two or three decimal places**), with a **large** effect (d= **Cohen's d to two decimal places**, 95% CI [**lower bound** - **higher bound**]).

5.9   Power analyses

Now you may recall in your Week 3 lecture Ciara being very enthusiastic about power analyses, and the importance of conducting one before you collect any data (called an a priori or prospective power analysis). There are also power analyses you can conduct after data collection, but there are issues with them, and generally best practice is to do one beforehand (A useful paper if you're interested in learning more).

Here we are going to learn about how to conduct a power analysis for both an independent samples and paired-samples t-test.

As you may recall there are some key pieces of information we need for a power analysis

- Alpha level (typically 0.05 in Psychology and the social sciences)

- The minimum effect size of interest

- Our desired power

- If our test is one or two-tailed (i.e. do we have a directional or nondirectional hypothesis)

Reminder that this interactive visualisation can be helpful in understanding how these things interact.

5.10 Power analysis for an independent samples t-test

The syntax for conducting an apriori statistical power analysis for an independent samples t-test is the following:

```
# Conduct power analysis for an independent samples t-test
pwr.t.test(d = 0.5,          # Your Expected effect size
           sig.level = 0.05, # Significance level
           power = 0.80,     # Desired power level
           type = "two.sample",  # Indicates an independent t-test
           alternative = "two.sided")  # Indicates a two-tailed test, #can be changed to "one.sid
```

5.11 Power analysis for a paired-samples t-test

If we want to run a paired samples-test, then we can change the type from "two.sample" to "one.sample":

```
pwr.t.test(d = 0.5,          # Your Expected effect size
           sig.level = 0.05, # Significance level
           power = 0.80,     # Desired power level
           type = "paired",  # Indicates an independent t-test
           alternative = "two.sided")  # Indicates a two-tailed test, #can be changed to "one.sid
```