

Introduction to R

Ryan Donovan

2024-02-20

Contents

1	Introduction	5
1.1	Who is this resource for?	5
1.2	Should I learn R?	5
1.3	What will I learn to do in R?	6
1.4	What will I not learn to do in R?	6
1.5	Where and when will the workshops take place?	7
1.6	Are there any prerequisites for taking this course?	8
1.7	Do I need to bring a laptop to the class?	8
2	Getting Started with R and RStudio	9
2.1	What is R?	9
2.2	Create a Posit Cloud Account.	10
2.3	Downloading R on to your Computer	12
2.4	Install and Open R Studio	16
2.5	Creating an R Project	17
2.6	Writing our first R Code	23
2.7	Console vs Source Script	23
2.8	Let's write some statistical code	24
2.9	Summary	30
2.10	Glossary	30

3	R Programming (Part I)	33
3.1	Activity 1: Set up your Working Directory	33
3.2	Using the Console	33
3.3	Data Types	37
3.4	Basic Data types in R	37
3.5	Variables	41
3.6	Data Structures	45
3.7	Summary	61
3.8	Glossary	61
3.9	Variable Name Table	62
4	R Programming (Part II)	65
4.1	Functions	65
4.2	The Factor Data Type	77
4.3	The List Data Structure	82
4.4	R Packages	89
4.5	Installing and Loading R Package	89
4.6	Importing and Exporting Data	94
4.7	Summary	101

Chapter 1

Introduction

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

1.1 Who is this resource for?

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

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

1.2 Should I learn R?

There are many reasons to learn R.

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

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

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

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

1.3 What will I learn to do in R?

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

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

The learning objectives of this course are:

- Learn how to import and create datasets in R.
- Learn and apply basic programming concepts such as data types, functions, and loops.
- Learn key techniques for data cleaning in R to enable statistical analysis.
- Learn how to create APA-standard graphs in R.
- Learn how to deal with errors or bugs with R code.
- Learn how to export data.

1.4 What will I not learn to do in R?

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

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

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

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

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

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

1.5 Where and when will the workshops take place?

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

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

- March 20th: Running Inferential Statistical Tests in R (Part I)
- March 27th: Running Inferential Statistical Tests in R (Part II)

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

1.6 Are there any prerequisites for taking this course?

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

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

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

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

Chapter 2

Getting Started with R and RStudio

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

2.1 What is R?

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

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

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

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

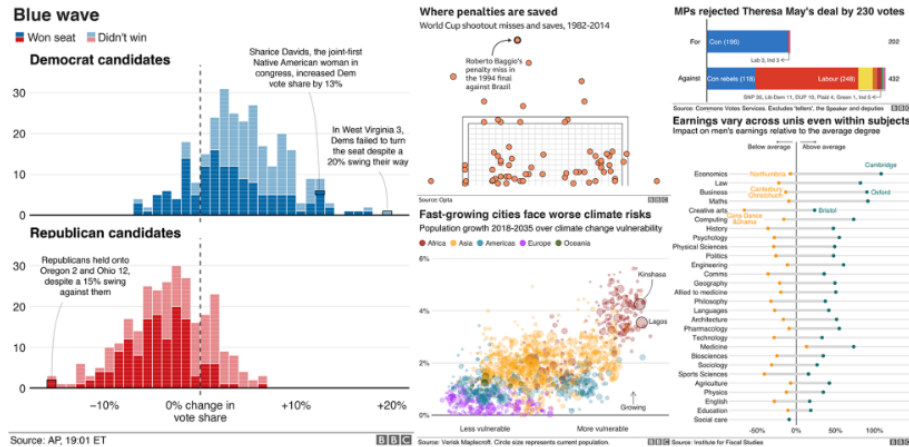


Figure 2.1: BBC graphs created in R.

But why should we learn R and not a different programming language? In contrast to other programming languages (Python, JavaScript, C), R was developed by statisticians. Consequently, R contains an extensive vocabulary to enable us to carry out sophisticated and precise statistical analysis. I have used R and Python to conduct statistical analysis, and anytime I wanted to use a less frequently used statistical test, there was significantly more support and information on how to conduct that analysis in R than in Python. For such reasons, R is typically used among statisticians, social scientists, data miners, and bioinformaticians - and will be used in this course¹.

2.2 Create a Posit Cloud Account.

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

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

¹There 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 your goal is to conduct data cleaning, analysis, visualization, and reporting, then R is an 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.

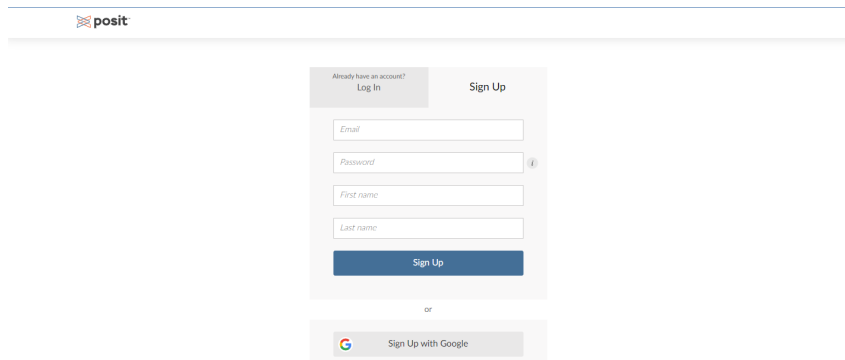
2.2. CREATE A POSIT CLOUD ACCOUNT.

11

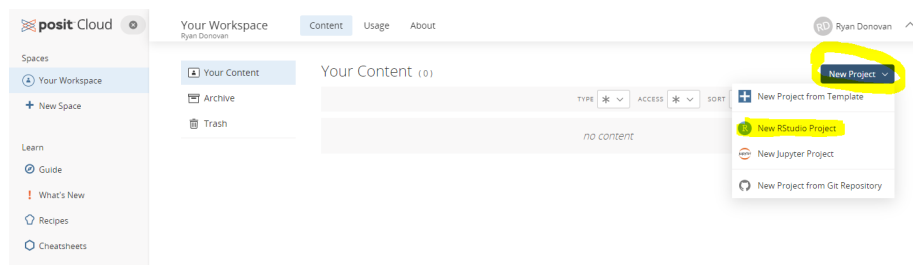
it with the free account), and I much rather you use your own computers in class. But it will be a handy back-up option in case any technical issues pop up. During class, I might not be able to solve that issue quickly and efficiently, so if it does occur, then you can sign in to Posit Cloud and keep following along with the session.

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

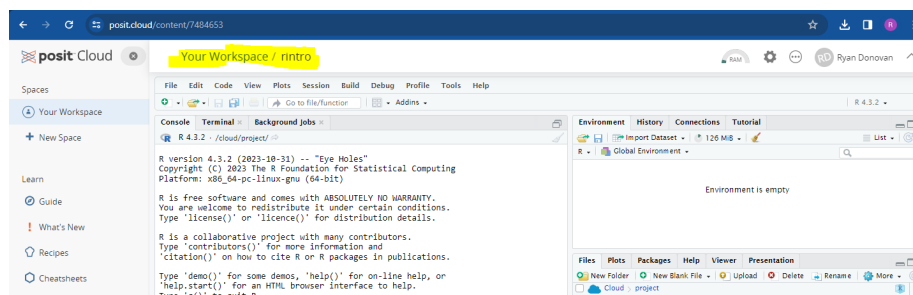
1. Go to their sign up page website and enter your details to create an account or Sign up with Google.

The image shows the Posit Cloud sign-up page. At the top left is the Posit logo. The main heading is "Sign Up". Below it, there are two options: "Already have an account? Log In" and "Sign Up". The "Sign Up" section contains four input fields: "Email", "Password", "First name", and "Last name". Below these fields is a blue "Sign Up" button. Underneath the button is the word "or" and a "Sign Up with Google" button with the Google logo.

2. Once you have created an account and are in Posit Cloud, click “New Project” From the drop-down menu click “New RStudio Project”. This should take a few seconds to set up (or “deploy”)



1. Once it is deployed, name your project at the top as *rintro*



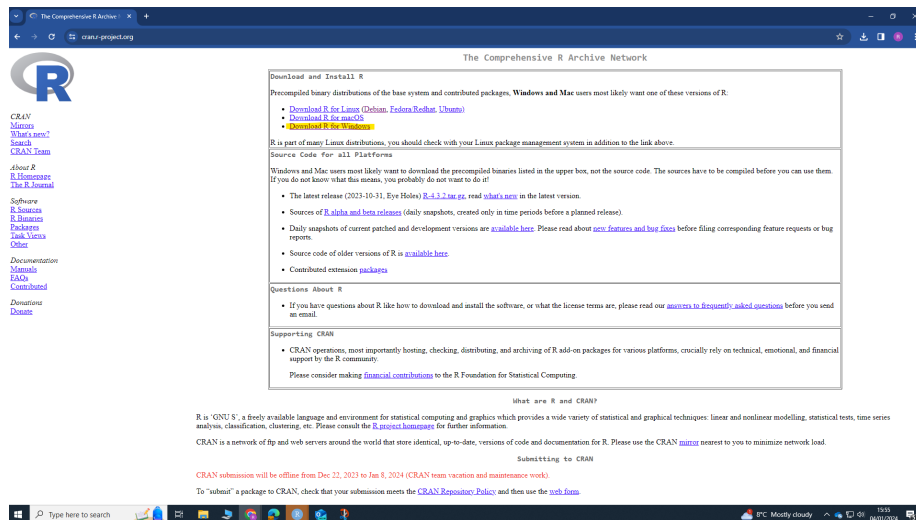
Don't worry about what anything on the screen means for now. We'll come back to that once we download RStudio on your computer. For now, you can sign out of Posit Cloud.

2.3 Downloading R on to your Computer

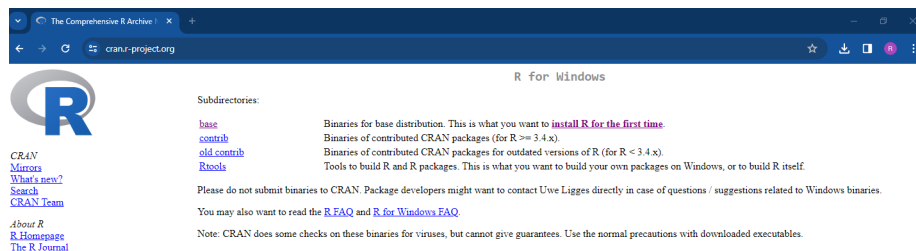
Please follow the following instructions to download R on either Windows or Mac.

2.3.1 Downloading R on Windows

1. Go to the website: <https://cran.r-project.org/>
2. Under the heading *Download and Install R*, click *Download R for Windows*



3. Click the hyperlink *base* or *install R for the first Time*



4. Click Download R-4.3.2 for Windows (depending on the date you accessed this, the version of R might have been updated. That's okay, you can download newer versions). Let the file download.

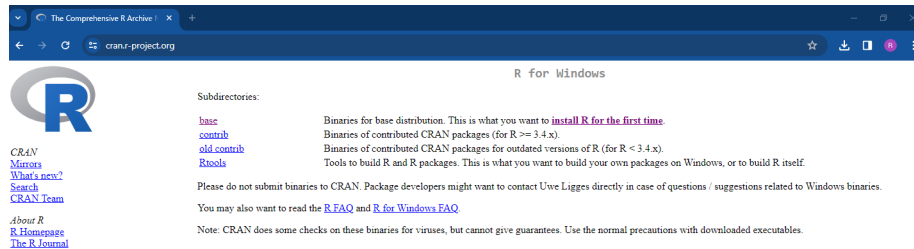


Figure 2.2: The R programming language is occasionally updated, so the specific version of R that you see might be different than mine. But that's okay!

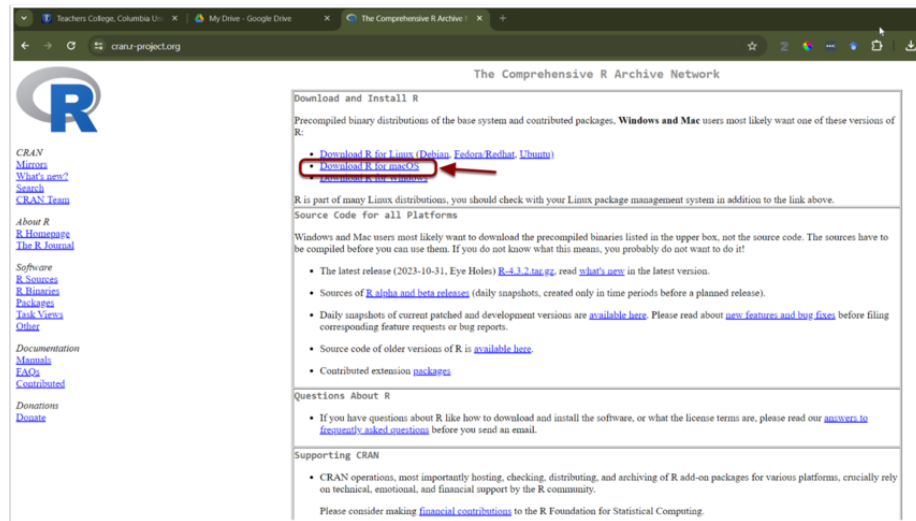
5. Once the file has been downloaded, open it, and click “Yes” if you are asked to allow this app to make changes to your device. Then choose English as your setup language. The file name should be something like “R-4.3.2.-win”. The numbers will differ depending on the specific version that was downloaded.
6. Agree to the terms and conditions and select a place to install R. It is perfectly fine to go with the default option.

2.3.2 Downloading R on Mac

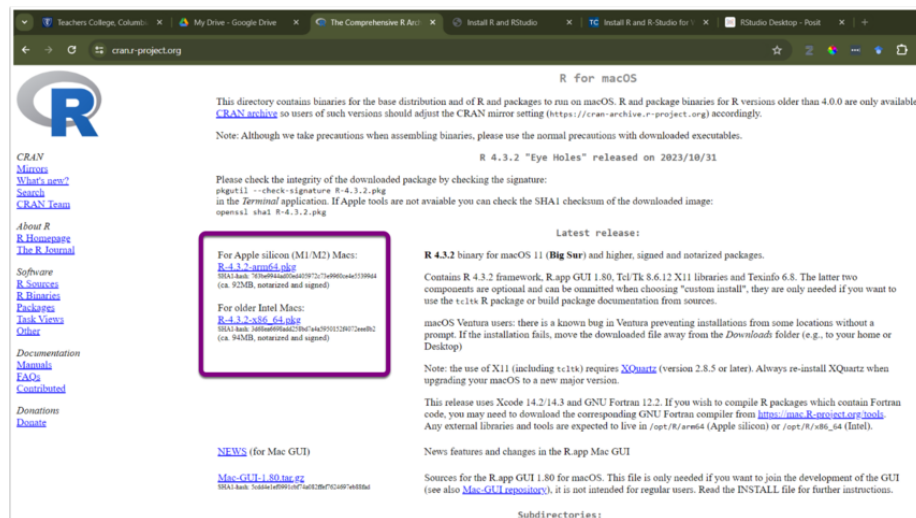
The instructions are largely the same for Mac.

1. Go to the website: <https://cran.r-project.org/>
2. Click Download R for (Mac) OS X.

14 CHAPTER 2. GETTING STARTED WITH R AND RSTUDIO



1. Check the Latest release: section for the appropriate version and follow the directions for download. If you are unsure about this, please ask me.



1. Once the file download is complete, click to open the installer. Click Continue and proceed through the installer, I recommend going with all default options.
1. Once the R installer has finished, click Close.

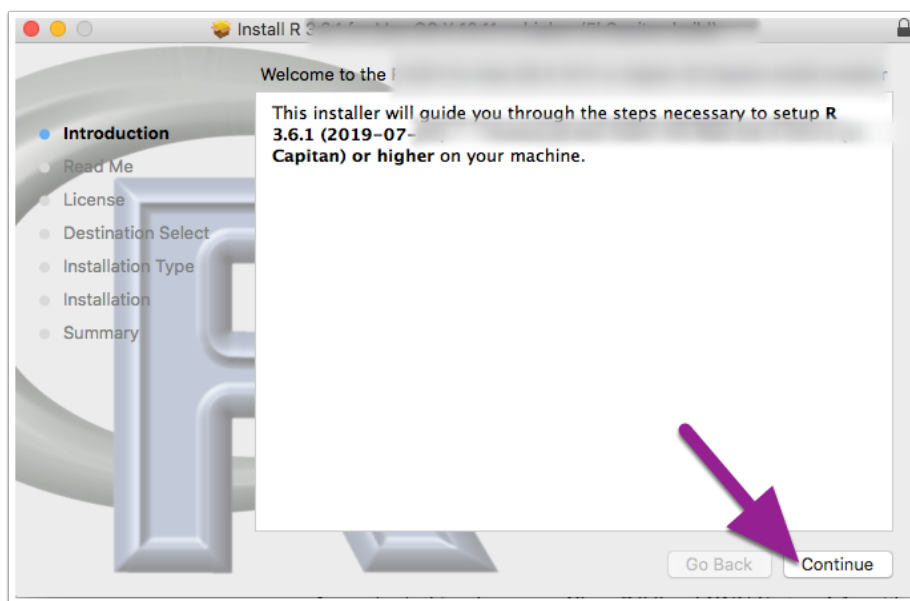
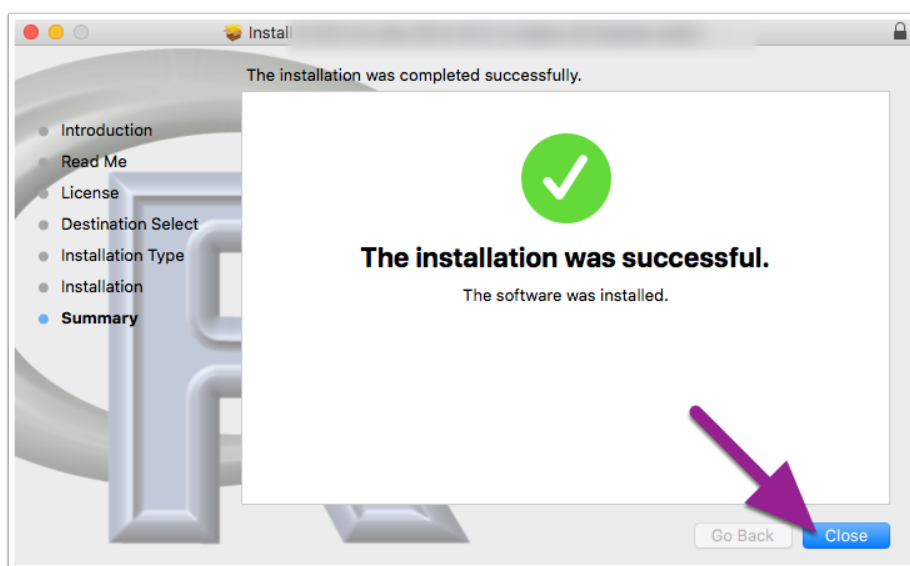


Figure 2.3: Depending on your version of Mac OS, this might look slightly different. But you should still be able to install it.



2.4 Install and Open R Studio

Once R is installed, we will install RStudio.

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

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

1: Install R

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

DOWNLOAD AND INSTALL R

2: Install RStudio

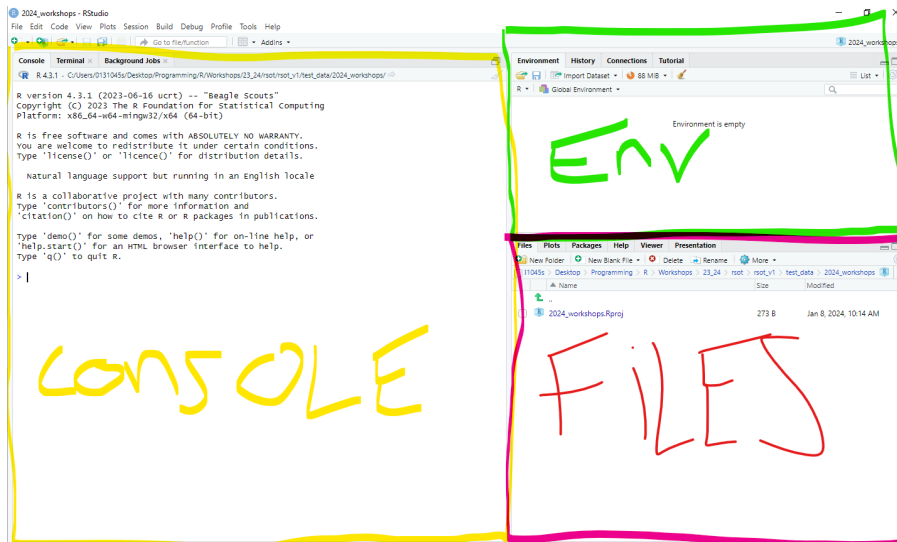
DOWNLOAD RSTUDIO DESKTOP FOR WINDOWS

Size: 215.66 MB | [SHA-256: 93C7F307](#) | Version: 2023.12.0+369
| Released: 2023-12-20

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

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

When you first open RStudio, you will see three panes or “windows” in RStudio: “Console” (left), “Environment” (top right), and “Files” (bottom right).



2.5 Creating an R Project

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

Activity

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

1. Click “File” in the top left hand corner of RStudio-> then click new “New Project”
2. The “New Project Wizard” screen will pop up. Click “New Directory” -> “New Project”
3. In the “Create New Project” screen, there are four options we are going to change.

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

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

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

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

- You can save it anywhere you like (e.g., your Desktop). Just ensure it’s in a place you can easily find and where it won’t be moved (e.g., if you save folders to your desktop but tend to relocate them later, avoid saving it on your desktop).
- My recommendation is to create a folder called “R_Programming” on your desktop and save your project inside this folder.
- Regardless of where you save your project, copy the location and keep it in a place you can check later (e.g., in a text file).

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

- Tick this option.

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

- Tick this option.

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

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

2.5.1 Navigating RStudio

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

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

Button approach: Click the “File” tab in the top-left hand corner (not the File pane) -> Click “New File” -> “R Script”

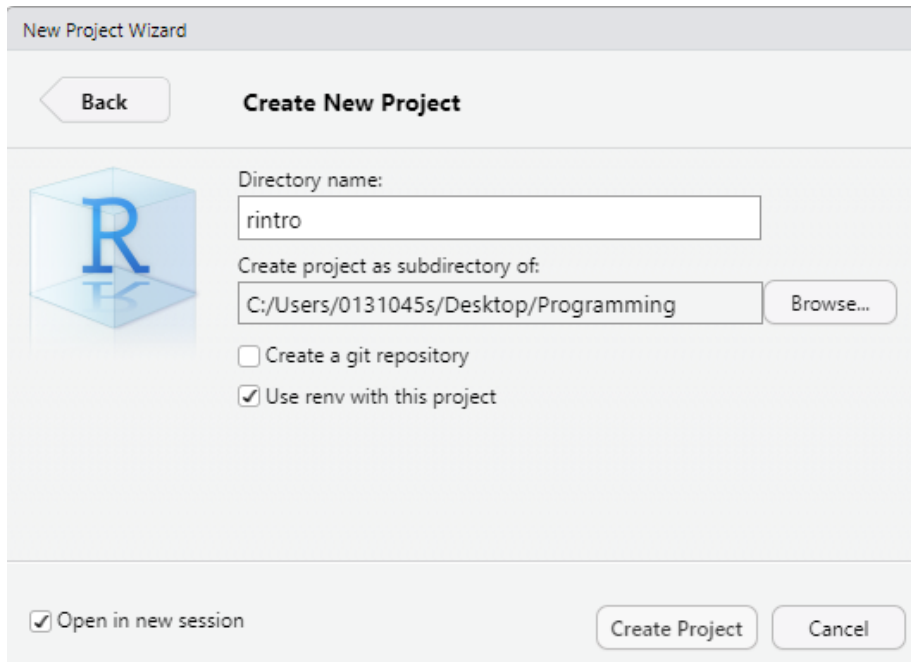
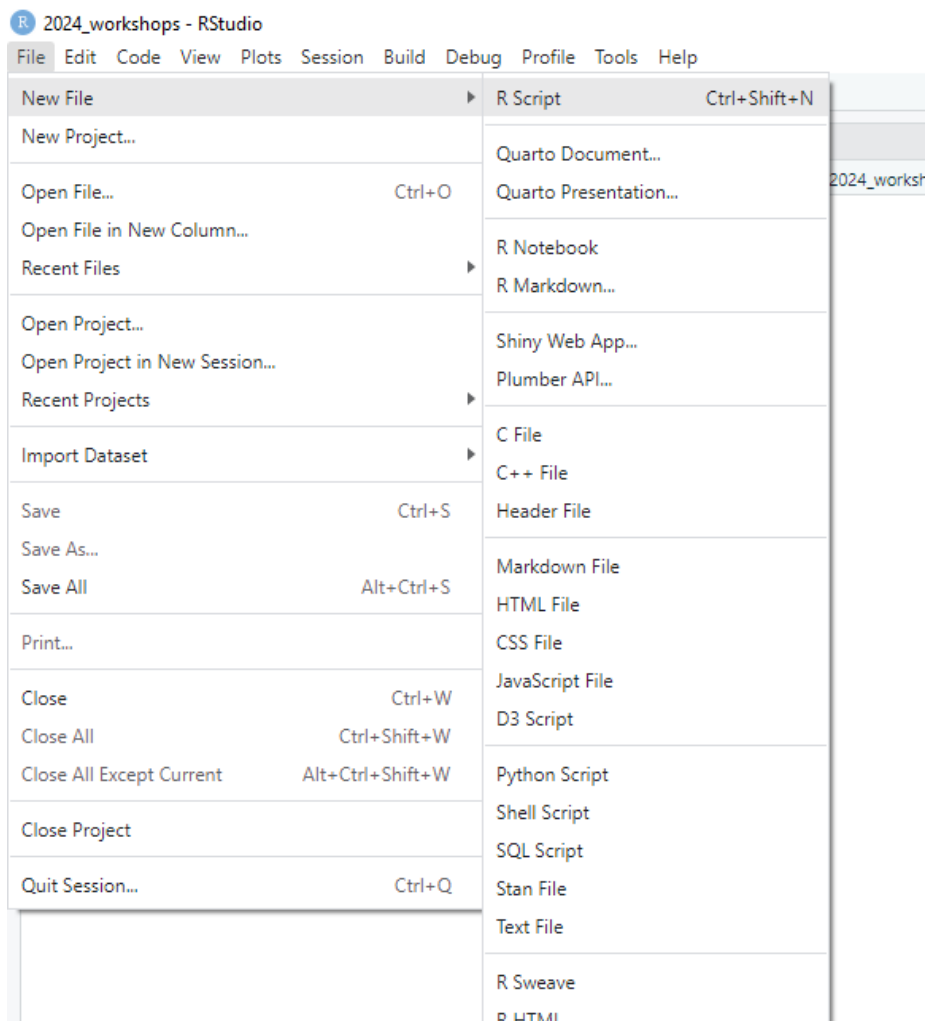


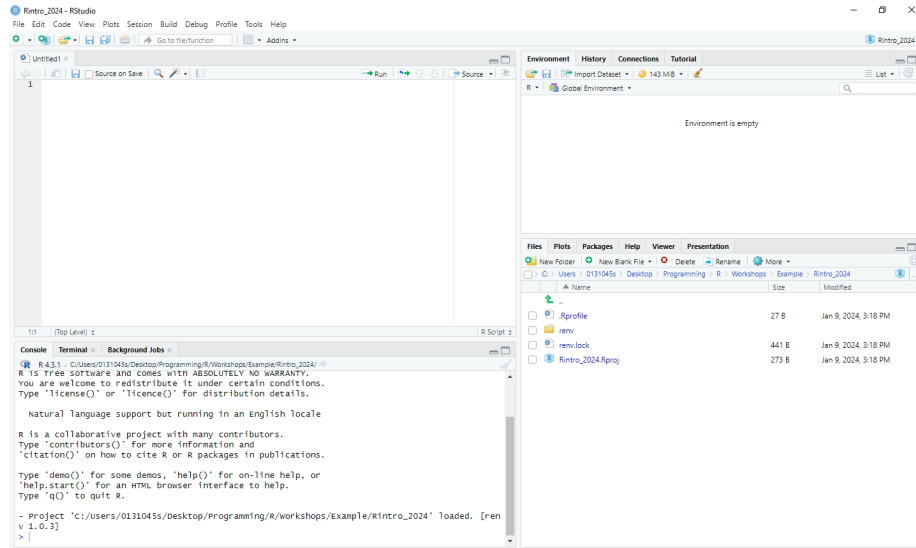
Figure 2.4: New Project Set Up



Button Shortcut: directly underneath the *File* tab, there is an icon of a white sheet with a green and white addition symbol. You can click that too.

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

Now you should see your four panes: Source, Console, Environment, and Files.



2.5.1.1 The RStudio Workspace

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

- The **Source Pane** is where you will write R scripts. R scripts enable you to write, save, and run R code in a structured format. For instance, you might have an R script titled “Descriptive,” containing the code for computing descriptive statistics on your data set. Similarly, you might have another R script titled “Regression” for performing regression analyses in R.
- The **Console Pane** is where you can write R code or enter commands into R. The console is also where you can find various outputs from your R scripts. For example, if you create a script for running a t-test in R, the results will appear in the Console Pane. Any error or warning messages related to your code will also be highlighted in the console. In short, the console is where R actually runs.
- The **Environment Pane** contains information about data sets and variables imported or created in R within a specific R project. The “History” tab shows a history of the R code executed during the project. This pane

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

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

All four panes will be used extensively during these workshops.

2.5.2 Checking our Working Directory

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

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

```
getwd()
```

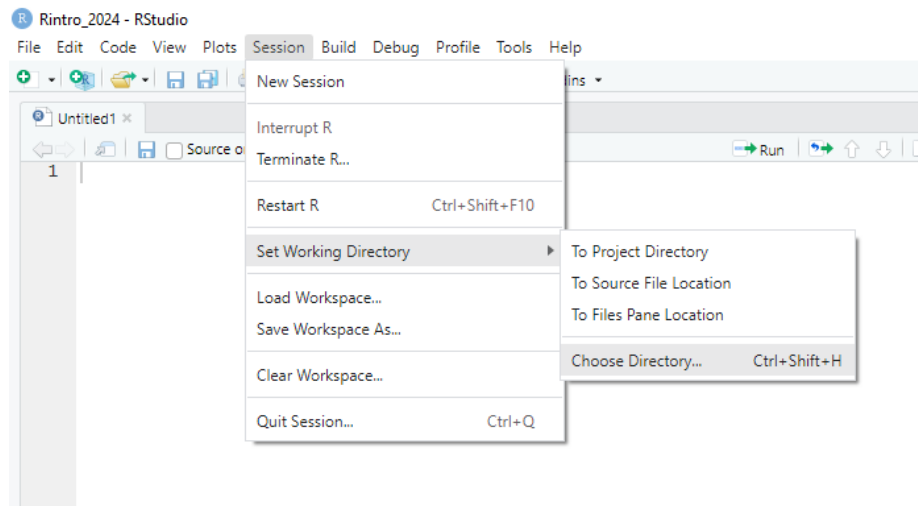
```
## [1] "/Users/ryandonovan 1/Desktop/Teaching/R/rintro"
```

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

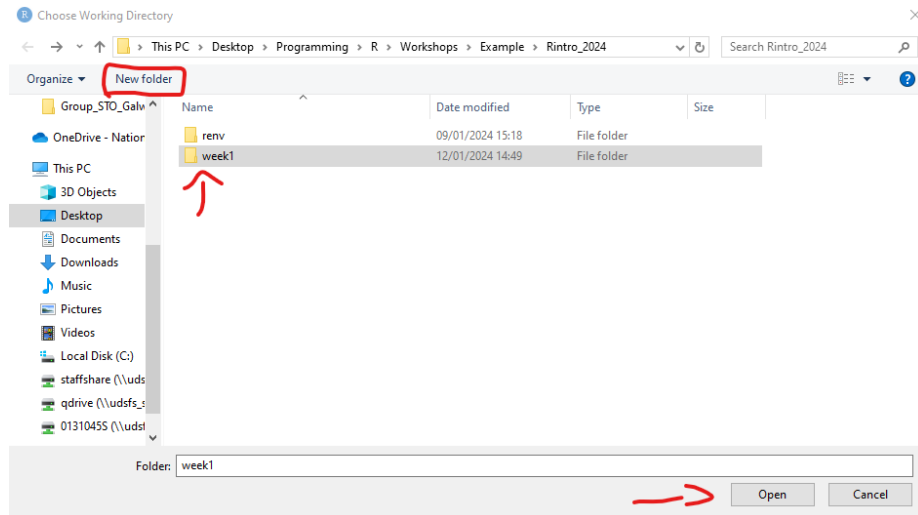
2.5.3 Setting up a new Working Directory

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

- Click “Session” on your RStudio toolbar -> Set Working Directory -> Choose Directory



- By default you should be in your R Project (e.g., *rintro*).
- Within this R Project, create a new folder and call it “week1”
- Click “week1” and then click Open



You should see something like the following in your console

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

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

2.6 Writing our first R Code

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

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

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

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

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

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

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

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

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

2.7 Console vs Source Script

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

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

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

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

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

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

2.8 Let's write some statistical code

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

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

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

```
print(sleep)
```

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



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

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

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

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

```
summary(sleep)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

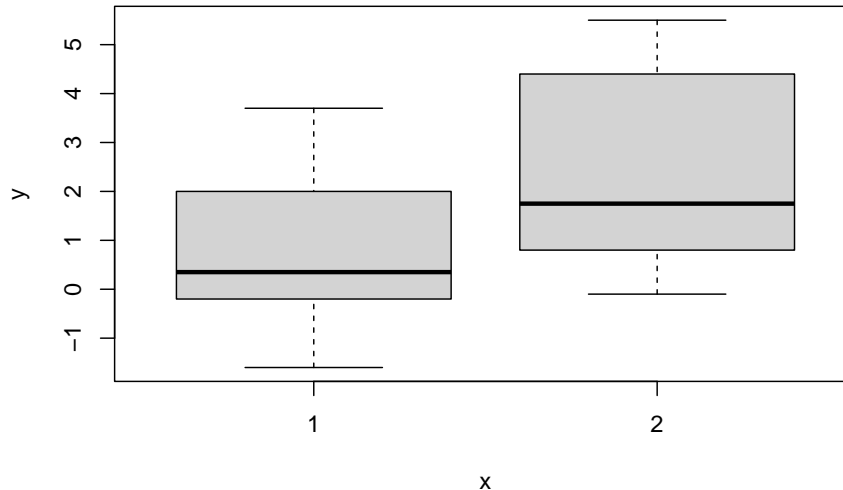


Figure 2.5: Generic Boxplot

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

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

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

```
#ylab = creates a title for the y-axis
```

```
#main = creates a title for the plot
```

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

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

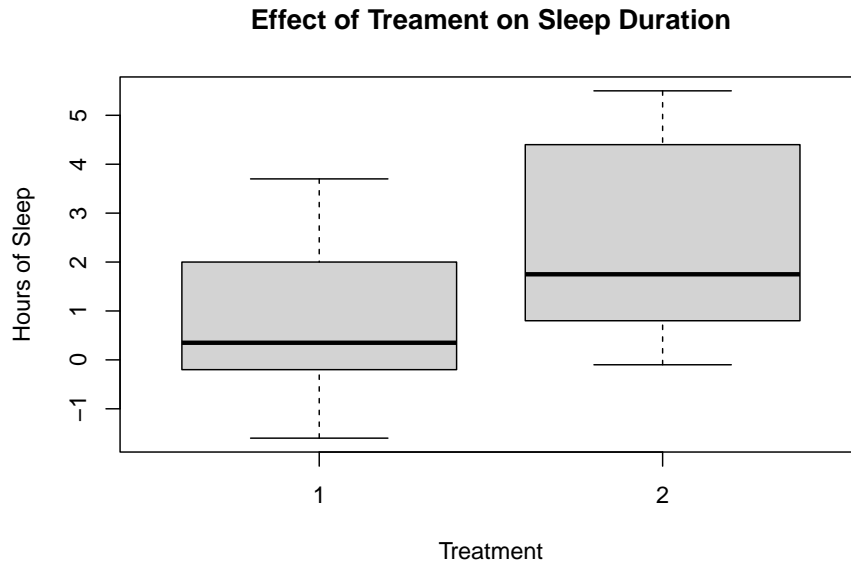


Figure 2.6: Generic Boxplot with appropriate labelling

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

```
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 it up the PDF and have a look at your graph².

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

2.8.1 Comments

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

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

There are various reasons for using comments.

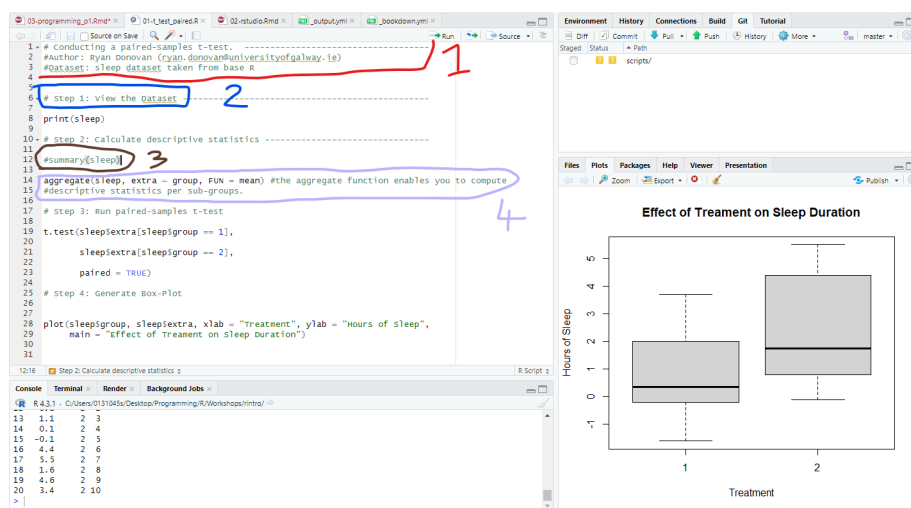


Figure 2.7: Four Examples of Comments Use

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

1. The first type of comment provides a quick introduction to the R script. It can be really useful here to provide clear information on what this script is trying to do (e.g., run a paired samples t-test), what data it is working on (the sleep dataset), and who wrote or developed this script. This makes it significantly easier for anyone who might be reviewing your work or trying to apply your code to their own work to understand what is going on.
2. The second type of comment structures the format of the script by providing headings or steps. Again, this just makes it easier to understand what is going on.
3. The third type of comment is placed before the `summary` function. This means that the code `summary(sleep)` will not be executed in R. Why would we do this? If you remember last week, we wanted to compute the mean per each of our two treatment groups, which the `summary` function does not enable

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

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

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

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

2.9 Summary

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

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

2.10 Glossary

This glossary defines key terms introduced in Chapter 2.

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

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

Chapter 3

R Programming (Part I)

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

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

3.1 Activity 1: Set up your Working Directory

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

Using those instructions, create a folder called “Week2” in the `rintro` project folder and set it as your working directory. Use the `'getwd()'` to check that it has been set as your working directory. Your output should be something like this:

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

3.2 Using the Console

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

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

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

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

3.2.1 Typing Commands in the Console

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

```
> 10 + 20
```

```
[1] 30
```

```
> 20 / 10
```

```
[1] 2
```

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

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

```
[1] 84
```

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

```
[1] 12
```

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

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

¹Including the “`>`” is a pain when formatting this book, so I won’t include “`>`” in examples of code from this point forward.

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

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

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

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

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

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

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

3.2.2 Console Syntax (Aka “I’m Ron Burgundy?”)

3.2.2.1 R Console and Typos

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

```
> 10 = 20

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

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

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

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

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

```
> 10 - 20  
[1] -10
```

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

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

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

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

3.2.2.2 R Console and Incomplete Commands

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

```
> (20 + 10  
+ )  
[1] 30
```

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

```
(20 + 10
```

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

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

3.2.3 Exercises

1. Practice performing basic calculations in R console. Calculate the following:
 1. 25 multiplied by 4
 2. 72 divided by 8 3
 3. 0 multiplied by 4, and then divided by 2
2. Imagine you want to calculate the average/mean of the following 5 numbers 15, 22, 18, 30, and 25. Use the R console to find the average.
3. If I type the following code, then I get the + operator, how can I fix it?

```
> (60 / 100  
+  
+
```

3.3 Data Types

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

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

3.4 Basic Data types in R

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

3.4.1 Character

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

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

```
class("a")
```

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

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

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

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

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

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

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

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

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

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

3.4.2 Numeric (or Double)

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

```
class(33)
```

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

```
class(33.33)
```

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

```
class(-1)
```

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

3.4.3 Integer

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

```
class(33L)
```

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

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

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

```
class(0L)
```

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

You might wonder why R has a separate data type for integers when numeric/double data types can also represent integers. The very 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’s unlikely that you will need to use integers over numeric/doubles for your own research, but it’s good to be aware of just in case.

3.4.4 Logical (otherwise known as Boolean)

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

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

```
class(TRUE)
```

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

```
class(FALSE)
```

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

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

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

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

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

```
[1] 14.23
```

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

```
[1] 120
```

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

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


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

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

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

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

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

3.4.5 Exercises

1. Have a look at each of the following pieces of code and guess what data type it is. Check whether you are correct by using the `class()` function.
 1. "Hello World!"
 2. 43
 3. "42.34"
 4. FALSE
 5. 44.4
2. The following data types have been erroneously entered in R. Use the appropriate converting function to correct for it.
 1. Convert "42.34" from character to numeric.
 2. Convert "FALSE" from logical to character.
 3. Convert 2024 from numeric to string.
 4. Convert 1 from integer to logical (see what happens!). For bonus points, convert 0 from numerical to logical as well.

3.5 Variables

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

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

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

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

```
print(name)
```

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

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

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

```
print(name)
```

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

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

```
age <- 30L
```

```
height <- 175 #centimetre
```

```
live_in_hot_country <- FALSE
```

```
print(age)
```

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

```
print(height)
```

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

```
print(live_in_hot_country)
```

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

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

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

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

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

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

print(total_extra)
```

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

```
mean_extra <- total_extra/5

print(mean_extra)
```

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

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

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

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

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

- Variable names can only contain uppercase alphabetic characters A-Z, lowercase a-z, numeric characters 0-9, periods ., and underscores _.
- Variable names must begin with a letter or a period (e.g., **1st_name** or **_1stname** is incorrect, while **first_name** or **.firstname** is correct).

- Avoid using spaces in variable names (`my name` is not allowed; use either `my_name` or `my.name`).
- Variable names are case-sensitive (`my_name` is not the same as `My_name`).
- Variable names cannot include special words reserved by R (e.g., `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `TRUE`, `FALSE`). While you don't need to memorize this list, it's helpful to know if an error involving your variable name arises. With experience, you'll develop an intuition for valid names.

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

- Choose informative variable names that clearly describe the information they represent. Variable names should be self-explanatory, aiding in code comprehension. For example, use names like “income,” “grades,” or “height” instead of ambiguous names like “money,” “performance,” or “cm.”
- Opt for short variable names when possible. Concise names such as `dob` (date of birth) or `iq` (intelligence quotient) are better than lengthy alternatives like `date_of_birth` or `intelligence_quotient`. Shorter names reduce the chances of typos and make the code more manageable.
- However, prioritize clarity over brevity. A longer but descriptive variable name, like `total_exam_marks`, is preferable to a cryptic acronym like `tem`.
- Avoid starting variable names with a capital letter. While technically allowed, it's a standard convention in R to use lowercase letters for variable and function names. Starting a variable name with a capital letter may confuse other R users.
- Choose a consistent naming style and stick to it. There are three common styles for handling variables with multiple words:
 1. **snake_case**: Words are separated by underscores (e.g., `my_age`, `my_name`, `my_height`). This is the preferred style for this course as it aligns with other programming languages.
 2. **dot.notation**: Words are separated by periods (e.g., `my.age`, `my.name`, `my.height`).
 3. **camelCase**: Every word, except the first, is capitalized (e.g., `myAge`, `myName`, `myHeight`).

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

3.5.2 Exercises

1. Create a variable called `favourite_colour` and assign your favourite colour to this `favourite_colour`. What data type is this variable? Check it with `class()`.
2. Create two numeric variables, `num1` and `num2`, and assign them any two different numeric values.
3. Calculate the sum of `num1` and `num2` and store it in a new variable called `sum_result`.
4. Print the value of `sum_result`.
5. Create a variable named `height_cm` and assign it your height in centimeters (a numeric value).
6. Create another variable named `height_m` and assign it the height in meters by dividing `height_cm` by 100.
7. Print the value of `height_m`.

3.6 Data Structures

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

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

3.6.1 Vectors

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

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

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

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

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

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

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

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

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

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

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

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

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

```
instructor <- "Ryan/Brendan"
is.vector(instructor)
```

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

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

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

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

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

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

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

```
class(rintro_marks)

[1] "numeric"

class(rintro_grades)

[1] "character"
```

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

3.6.1.1 Working with Vectors

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

Numeric and Integer Vectors

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

```
mean(rintro_marks)
```

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

```
median(rintro_marks)
```

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

```
sd(rintro_marks)
```

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

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

```
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.

```
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.

```
summary(rintro_names)
```

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

```
summary(rintro_satisfied)
```

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

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


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

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

Figure 3.1: Indexing for Numeric Vector

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

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

Figure 3.2: Indexing for Character Vector

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

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

Figure 3.3: Indexing for Logical Vector

3.6.1.2 Vector Indexing and Subsetting

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

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

```
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.

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 `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 actually changed the original vector. They just outputted a result to the R console. If I wanted to actually save their results, then I would need to assign them to a variable label.

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

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

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

```
print(marks_top)
```

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

3.6.1.3 Vectors - making it a little less abstract.

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

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

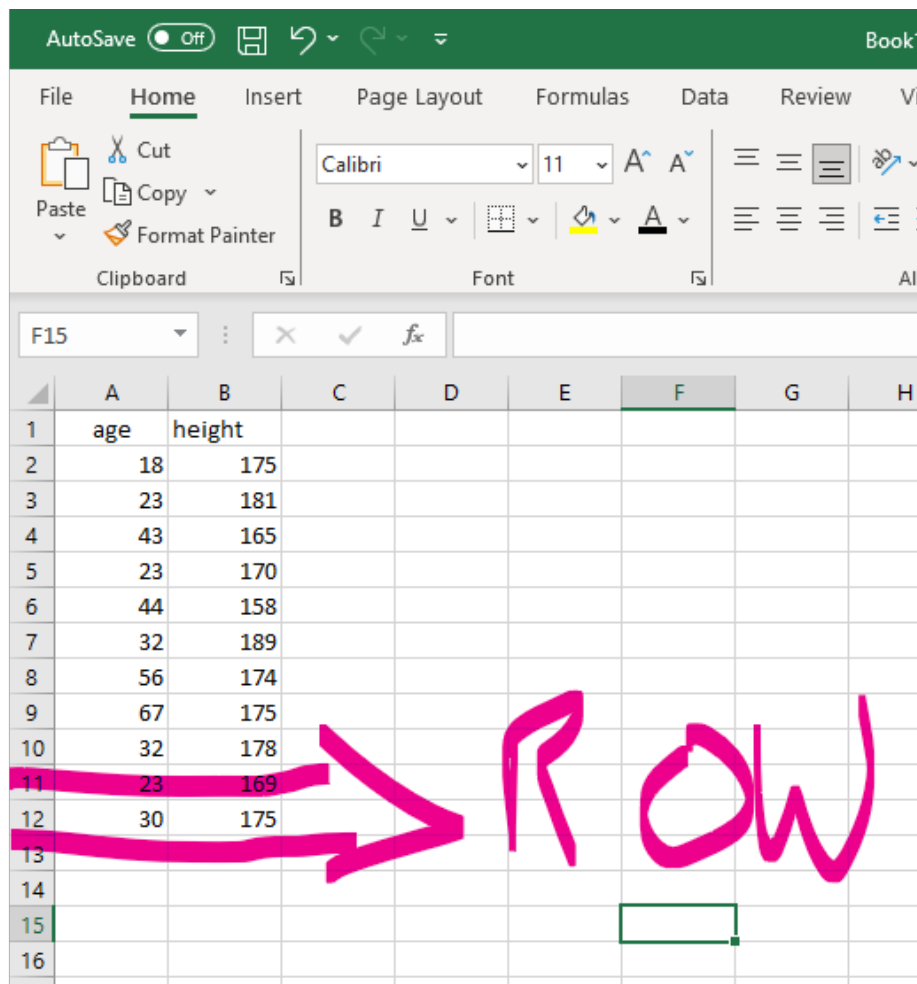
Creating that vector is the equivalent of creating a column in Excel:

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

	A	B	C	D	E	F
1	age	height				
2	18	175				
3	23	181				
4	43	165				
5	23	170				
6	44	158				
7	32	189				
8	56	174				
9	67	175				
10	32	178				
11	23	169				
12						
13						
14						
15						
16						
17						
18						
19						
20						

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

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



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

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

3.6.2 Data frames

A data frame is a rectangular data structure that is composed of rows and columns. A data frame in R is like a virtual table or a spreadsheet in Excel:

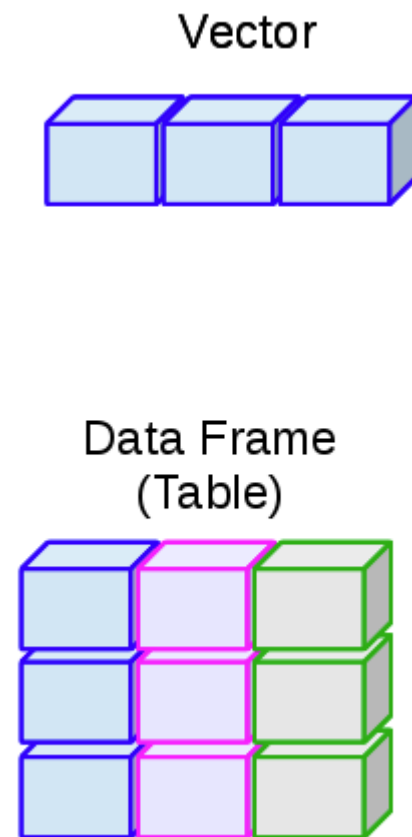


Figure 3.4: The relationship between data frames and vectors. The different colours in the data frame indicate they are composed of independent vectors

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

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

my_df
```

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

3.6.2.1 Selecting Data from a Data Frame

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

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

We select columns based on their names via two tools:

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

```
my_df$Name
```

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

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

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

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

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

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

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

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

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

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

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

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

```
my_df[1, ]
```

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

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

To access the third row:

```
my_df[3, ]
```

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

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

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

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

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

```
my_df[2:4, ]

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

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

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

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

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

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

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

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

my_df2

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

3.6.2.2 Adding Data to your Data Frame

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
my_df
```

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

3.6.3 Exercises

1. Create one vector of each data type:
 1. Create a character vector called `friends` with the name of 3 of your friends.
 2. Create an integer vector called `years` that describes the amount of years you have been friends (if it's less than 1 year, put 1).
 3. Create a numeric vector called `extra` with their extraversion scores (out of 5).
 4. Create a logical vector called `galway` that describes whether they live (TRUE) or don't live (FALSE) in Galway.
 5. Once you have created each vector, check whether it is the correct data type using the `class()` function.
2. Index the 2th, 4th, and 6th element for each of the following vectors.

```
vect1 <- c("Not this", "This", "Not This", "This", "Not This", "This")
vect2 <- c(0, 1, 0, 1, 0, 1)
vect3 <- c("FALSE", "TRUE", "FALSE", "TRUE", "FALSE")
```

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

4. Write code that adds a column to the `my_df` data frame called `Nationality`. The values for the column should be "Irish", "American", "English", "Irish".
5. Check whether that `Nationality` column has been successfully added by using the `$` notation. The output should look like this.

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

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

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

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

3.7 Summary

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

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

3.8 Glossary

This glossary defines key terms introduced in Chapter 3.

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

3.9 Variable Name Table

Rule	Type	Incorrect Example	Correct Example
Variable names can only contain uppercase alphabetic characters A-Z, lowercase a-z, numeric characters 0-9, periods ., and underscores _.	Strict	1st_name	first_name
Variable names must begin with a letter or a period.	Strict	_1stname	.firstname
Avoid using spaces in variable names.	Strict	my name	my_name
Variable names are case-sensitive.	Strict	my_name == my_Name	my_Name == my_Name
Variable names cannot include special words reserved by R.	Strict	print	to_print
Choose informative variable names that clearly describe the information they represent.	Recommended	money	income
Opt for short variable names when possible.	Recommended	date_of_birth	dob
Prioritize clarity over brevity.	Recommended	ten	total_exam_marks
Avoid starting variable names with a capital letter.	Recommended	firstName	firstName
Choose a consistent naming style and stick to it.	Recommended	myName, last_Name	my_name, last_name or myName and lastName

Chapter 4

R Programming (Part II)

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

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

- Understanding the logic of functions, including how and why they are created.
- Creating the “factor” data type and the list data structure.
- Importing a data set into R using both code and button-click interfaces.
- Be capable of spicing up your RStudio by installing and loading packages.
- Understand how to use the help function effectively and how to search for help online.

4.1 Functions

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

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

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

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

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

4.1.1 The Syntax for Creating a Function

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

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

The syntax for creating a function looks like this:

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

What's going on here?

1. First, we created a name, **my_function**, and used the assignment operator `<-` to tell R we are going to be storing some piece of information to that name.
2. Then, we wrote **function()** to tell R that we are going to be creating a function and storing it to our name **my_function**. Inside **function()**, we specified an **argument**, which is just a fancy word for **input**.
3. Inside the curly brackets **{}**, we write the code for **my_function**. This code comprises instructions (i.e., operations) that R will execute on our

argument (i.e., input). We could have 1 instruction here, or we could have several hundred lines of instructions. That depends on the complexity of the function we are creating.

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

4.1.2 Creating a Simple Function (1-Argument)

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

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

```
cup_to_grams <- function(cups) {  
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.3 Creating a Multi-Argument Function

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

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

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

In this function:

- `x` is the value for which we want to calculate the z-score.
- `mean_val` is the mean score of the variable within our dataset.
- `sd_val` is the standard deviation of the variable within our dataset.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#dataframe$newColumnName <- #new_vector
```

```
iq_df$iq_z_scores <- iq_z_scores
```

```
#now let's check our data frame
```

```
head(iq_df)
```

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

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

4.1.4 Functions inside Functions

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

```
iq_df$total_depression_beck <- c(32, 36, 34, 46,
                                30, 53, 40, 15) #adds the mean_depression beck vector to our data
head(iq_df) #check to see if it was added correctly.
```

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

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

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

```

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

# Return the z-score
return(z_score)
}

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

```

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

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

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

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

```

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

  z_score <- (x - mean_val) / sd_val

  # Return the z-score
  return(z_score)
}

calculate_z_score(x = iq_df$total_depression_beck)

```

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

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

4.1.5 Returning Multiple Objects from a Function

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

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

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

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

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

  z_score <- (x - mean_val) / sd_val

  results <- list(z_score, mean_val, sd_val)

  # Return the z-score
  return(results)
}

calculate_mean_sd_z(iq_df$total_depression_beck)
```

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

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

```

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

  z_score <- (x - mean_val) / sd_val

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

  # Return the z-score
  return(results)
}

calculate_mean_sd_z(iq_df$total_depression_beck)

```

```

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

```

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

```

scores <- calculate_mean_sd_z(iq_df$total_depression_beck)

scores$mean

```

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

```
scores$sd
```

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

```
scores$z
```

```

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

```

4.1.6 Some Important Features about Functions

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

4.1.6.1 Global vs Local Variables

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

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

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

local_global()

## [1] 10

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

## [1] 20
```

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

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

4.1.6.2 Default Arguments

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

4.1.6.3 Syntax for Default Arguments

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

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

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

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

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

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

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

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

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

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

Exercises

1. Create a function called **fahrenheit_to_celsius** that converts a temperature from Fahrenheit to Celsius.
 1. Include in the definition of the function the argument **fahrenheit**
 2. Create a variable inside the function called **celsius**
 3. The formula for calculating it is: $\text{fahrenheit} - 32 / 1.8$
 4. Return the **celsius** variable

2. Create a function called `calculate_discount` that calculates the discounted price of an item.
 1. The function should take two arguments: `price` and `discount_percent`.
 2. Create a variable called `discount_price` inside your function.
 3. Assign the following formula to that variable: $(\text{price} * \text{discount_percent}) * 100$
 4. Create a variable called `final price`, using the formula:
 5. Return the final price
3. Add the z-scores for the Beck scale back into the `iq_df` data frame.
4. Look up the `t.test` function - what is the default value for the 'paired' argument?

4.2 The Factor Data Type

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

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

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

4.2.1 Creating a Factor

Consider a hypothetical experimental study investigating the effect of caffeine on sleep duration. Participants are randomly assigned to three experimental conditions: **No Caffeine**, **Low Caffeine** (200 mg), and **High Caffeine** (400 mg). Our data frame might resemble the following, where 1 = **No Caffeine**, 2 = **Low Caffeine**, and 3 = **High Caffeine**:

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

head(caffeine_df)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

That's better!

4.2.2 Sort Character Data

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

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

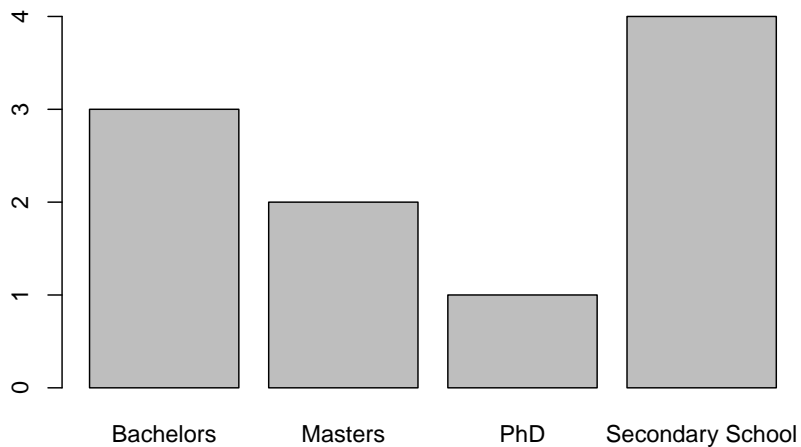
Using the **table()** function, we can count the number of participants per category:

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

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

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

```
barplot(count)
```



Now that gives us the information we need, but it's not ordered in an intuitive manner. Ideally, we would want the order of the bar plots to match the hierarchical order of the data, so that it would be: "Secondary School", "Bachelors",

“Masters”, and then “PhD”. However, unless you specify the order of your levels, R will specify their order alphabetically.

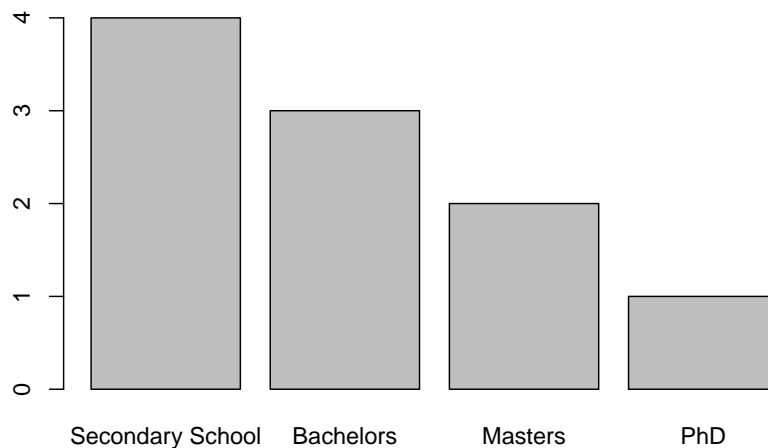
There is an argument called `levels` in the `factor` function that we can use to rectify this. In the `levels` argument, we specify the order of the levels.

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

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

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

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



As you can see, our data looks much cleaner and more intuitive.

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

4.3 The List Data Structure

The two data structures we have discussed so far, vectors and data frame, are excellent ways to store data. But each data structure has its limitations. The limitations of both the vector and data frame is that they only enable us to store one object of data at a time.

What do I mean by one object of data? Well, when we create a vector, we are only storing information on one single vector in R. Now I can store multiple vectors in the one place in R by combining them into a data frame. However, I am only able to do that if each vector has the same number of data points. Based on what I have taught you so far, there is no way to store 2 independent and separate vectors in the same place in R.

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

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

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

4.3.1 Understanding Lists

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

4.3.2 Creating Lists

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

```
participant1 <- list(  
  ID = 1,  
  age = 25,  
  gender = "Male",
```

```

    test_scores = c(80, 75, 90),
    questionnaire_responses = c("Agree", "Disagree", "Neutral", "Agree")
  )

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

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

```

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

Let's print out `participant1` list and break down the output.

```
print(participant1)
```

```

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

```

When we print out the list, what R does in the console is print out the name of each object and then print out each element in that object. One thing that you

might notice is the return of the `$` which we use to access elements from vectors or columns from data frames. Luckily, we can also use the `$` symbol to access objects and their elements from a list. So I wanted to extract `test_scores` from a list, I could type the following code:

```
participant1$test_scores
```

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

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

```
participant1[4]
```

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

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

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

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

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

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

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

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

print(df_v)
```

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

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

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

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

4.3.3 Lists within Lists (Indexing)

What happens if we combine lists together? Well we can do that by putting lists inside lists. But if we do that, the output is pretty ugly.

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

```
## [[1]]
## [[1]]$ID
## [1] 1
##
## [[1]]$age
```

```

## [1] 25
##
## [[1]]$gender
## [1] "Male"
##
## [[1]]$test_scores
## [1] 80 75 90
##
## [[1]]$questionnaire_responses
## [1] "Agree"      "Disagree" "Neutral"   "Agree"
##
##
## [[2]]
## [[2]]$ID
## [1] 2
##
## [[2]]$age
## [1] 30
##
## [[2]]$gender
## [1] "Female"
##
## [[2]]$test_scores
## [1] 85 70 88
##
## [[2]]$questionnaire_responses
## [1] "Neutral"      "Agree"      "Disagree"
## [4] "Strongly Disagree"
##
##
## [[3]]
## [[3]]$ID
## [1] 3
##
## [[3]]$age
## [1] 28
##
## [[3]]$gender
## [1] "Non-binary"
##
## [[3]]$test_scores
## [1] 78 82 85
##
## [[3]]$questionnaire_responses
## [1] "Disagree" "Neutral"  "Agree"    "Neutral"

```

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

Let's break down the first part of this output

```
[[1]]
```

So we know from indexing vectors that `vector_name[1]` would print out the first element of a vector. The same thing is happening here. If we had only 1 list, we could access the first thing from that list using `list[1]`

But when we are dealing with multiple lists, we need a convention to extract each list. That's why we have the double `[[]]` notation.

The output `'[[1]]'` basically says "This is the information from the first first list". So anytime you see `"[[]]"`, the number inside the central bracket indicates the list we are dealing with. So the code output

```
[[1]]$ID
[[1]]$age
[[1]]$test_scores
[[1]]$questionnaire_responses
```

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

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

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

```
## $p1
## $p1$ID
## [1] 1
##
## $p1$age
```

```

## [1] 25
##
## $p1$gender
## [1] "Male"
##
## $p1$test_scores
## [1] 80 75 90
##
## $p1$questionnaire_responses
## [1] "Agree"      "Disagree" "Neutral"   "Agree"
##
##
## $p2
## $p2$ID
## [1] 2
##
## $p2$age
## [1] 30
##
## $p2$gender
## [1] "Female"
##
## $p2$test_scores
## [1] 85 70 88
##
## $p2$questionnaire_responses
## [1] "Neutral"      "Agree"      "Disagree"
## [4] "Strongly Disagree"
##
##
## $p3
## $p3$ID
## [1] 3
##
## $p3$age
## [1] 28
##
## $p3$gender
## [1] "Non-binary"
##
## $p3$test_scores
## [1] 78 82 85
##
## $p3$questionnaire_responses
## [1] "Disagree" "Neutral"  "Agree"    "Neutral"

```


4.3.4 Summary

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

4.4 R Packages

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

4.4.1 Understanding R Packages

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

4.5 Installing and Loading R Package

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

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

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

First, I will show you two ways to install packages in R. Then I will show you two ways to load packages in R.

4.5.1 Installation

We are going to install three packages called **Haven**, **praise** and **fortunes**. The **praise** package provide users with, well, praise. And the **fortunes** package will

spit statistic and programming quotes at you. Neither of them are particularly useful, other than demonstrating the process of loading packages.

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

4.5.1.1 Installation using RStudio Interface

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

- Name (the name of the R package)
- Description (describes what the package does)
- Version (the version of the package currently installed on your computer)

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

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

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

4.5.1.2 Installation using Commands

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

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

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

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

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

trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-x86_64/contrib/4.3/fortunes_1.5-4.tgz'
Content type 'application/x-gzip' length 208808 bytes (203 KB)
=====
downloaded 203 KB

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

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

4.5.2 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.

4.5.2.1 Loading using RStudio Interface

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

Scroll down to find the package **praise** and load it in by ticking the box.

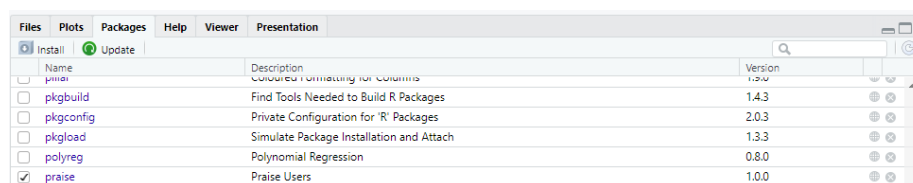


Figure 4.1: Loading Packages through RStudio Interface

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

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

4.5.3 Loading using the R Console Command

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

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

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

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

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

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

4.5.4 Testing our New Functions

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

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

##
## hist(rnorm(1e6), col = colors()[grep("grey", colors())], nclass = 108, main =
```

```
## "R's 108 Shades of Grey")  
##      -- Fritz Scholz (proving R has more than 50 Shades of Grey)  
##      private communication (March 2015)
```

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

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

4.5.5 Error Loading Packages

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

```
library(madeuppackage)
```

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

4.5.6 Package Conventions if Using Code

There is an important rule of thumb if you are writing the code to install and load packages in R.

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

Secondly, you should type the command “install.packages(“packages”)” in the console rather than the script. Because if someone downloads your script and accidentally runs it, it will automatically install the packages on their computer, which we want to avoid, as it might interfere with other aspects of their operating system (this is really rare, but it’s better to be cautious)

Thirdly, if you do have the command “install.packages()” in the script, make sure it is commented out. That way if they accidentally run it, R won’t run that command.

4.5.7 Summary

There you have it. You have successfully installed and loaded in your first packages in R. Are they particularly useful packages? No! For the rest of this course, we will be loading in packages more regularly, and loading in packages that will make our lives significantly easier than if they were not around. In

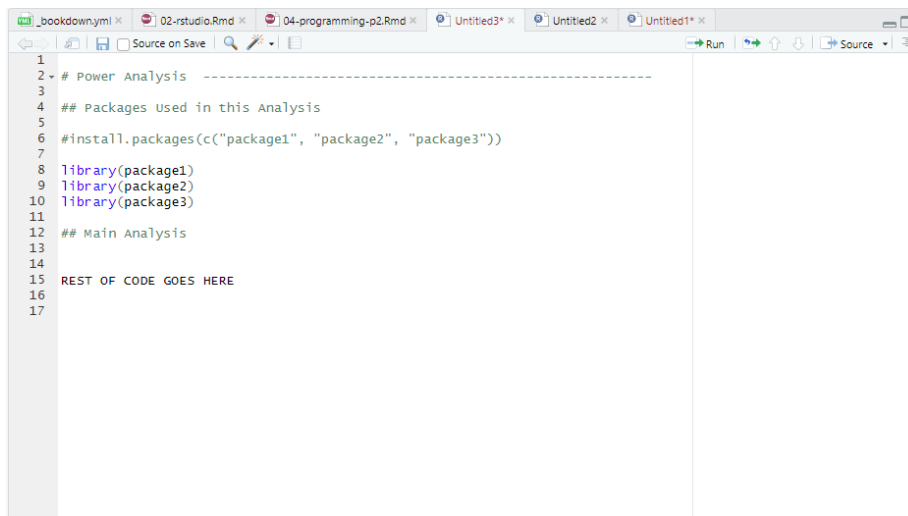


Figure 4.2: Conventions for Installing and Loading Packages in R Script

fact, we will load a package in the next section that will enable us to import SPSS data.

4.6 Importing and Exporting Data

While it's great to know how to create data frames and lists in R, realistically most of the data you will be analyzing in R will be with imported data. Similarly it is important to know how to export your data after you have cleaned it in R.

We'll explore how to import and export data using the graphical user interface (GUI) provided by RStudio in this section. The GUI offers a convenient and intuitive way to handle data files without writing any code. Let's dive into the process of importing and exporting data using the RStudio GUI.

First you are going to need to download two files: `psycho.csv` and `burnout.sav`. Both files are available in the `rintro` Teams channel. If you are accessing this resource and are not a member of the Teams channel, please contact me (ryan.donovan@universityofgalway.ie).

4.6.1 Importing CSV files.

Comma-Separated Values (CSV) files are a common format used for storing tabular data. As the name suggests, data in CSV files is organized in rows and

columns, with each row representing a single record and each column representing a different attribute or variable. In this regard, CSV files are very similar to excel file.

CSV files are plain text files, which means they can be easily created, edited, and viewed using a simple text editor. This simplicity and universality make CSV files a popular choice for data exchange between different 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 the locale settings, other delimiters such as semicolons (;) or tabs () 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 easily imported into statistical software for analysis, making them a versatile and widely adopted format for data storage and sharing.

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

1. Open RStudio.
2. Navigate to the Environment pane (top right hand corner) and make sure the Environment tab is clicked.
3. Click on the “Import Dataset” button.
4. Select “From Text (base)”.
5. Browse your folder and select the “`psycho.csv`” to import.
6. Click “Open”.
7. A pop-up window should now open. There is a lot of different information here, so let’s unpack it.
 1. The Name text box enables you to specify the variable name that will be assigned to the data frame. Leave it as `psycho`.
 2. Underneath the text box, there are several different options for influencing how the CSV file will be read. Most of the time, you can leave this with the default settings.
 3. The Input File box shows you a preview of the CSV file that is being imported, which is useful just to make sure it is the correct file.
 4. The Data Frame box shows you a preview of how the data frame will look like once the CSV is imported into R. Everything looks fine here, so we are good to go.

```
library(knitr)

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

Import Dataset

Name:

Input File: Participant_ID,Treatment,Neuroticism
 1,Placebo,39.39524353
 2,Placebo,42.69822511
 3,Placebo,60.58708314
 4,Placebo,45.70508391
 5,Placebo,46.29287735
 6,Placebo,62.15064987
 7,Placebo,49.60916206
 8,Placebo,32.34938765
 9,Placebo,38.13147148
 10,Placebo,40.5433803
 11,Placebo,57.24081797
 12,Placebo,48.59813827
 13,Placebo,49.00771451

Encoding:

Heading: ☒ Yes ☐ No

Row names:

Separator:

Decimal:

Quote:

Comment:

na.strings:

☐ Strings as factors

Data Frame

Participant_ID	Treatment	Neuroticism
1	Placebo	39.39524
2	Placebo	42.69823
3	Placebo	60.58708
4	Placebo	45.70508
5	Placebo	46.29288
6	Placebo	62.15065
7	Placebo	49.60916
8	Placebo	32.34939
9	Placebo	38.13147
10	Placebo	40.54338
11	Placebo	57.24082
12	Placebo	48.59814
13	Placebo	49.00771

Figure 4.3: Pop-up window for importing CSV files

8. Click Import. A tab in the source pane called “psycho” should now open showing the data frame. Additionally, you should also see the data frame in the Environment pane. You should see under value that it has “60 obs. of 3 variables” which means there are 60 rows of data and 3 columns of data.
9. Once you have imported your data in R, it is always good practice to briefly check that it has been imported successfully and you have imported the correct data set. Two ways you can do this is by calling the `head()` and `summary()` functions on the data frame.

```
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
```

```
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 are the same as mine then that means you have correctly imported the data.

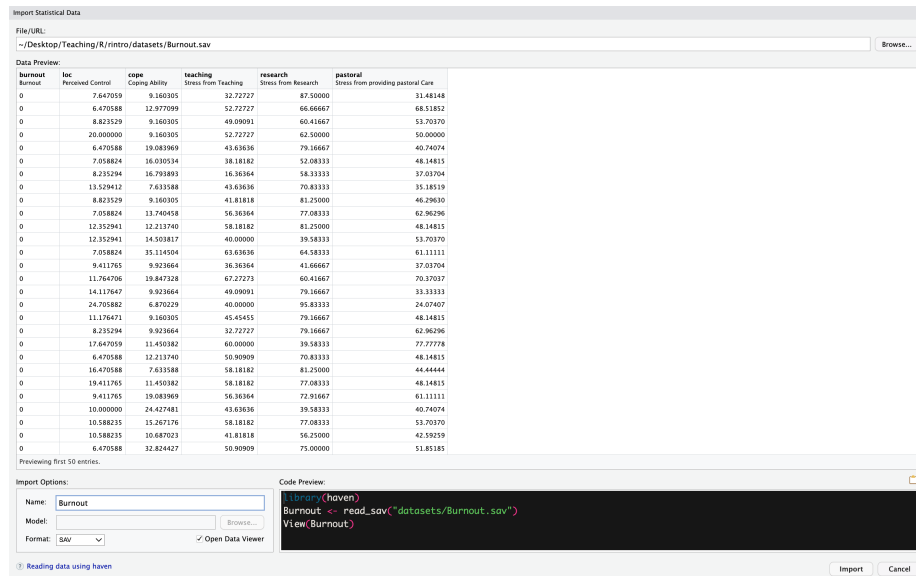
4.6.2 Importing SPSS (.sav files).

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

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

1. Open RStudio.
2. Navigate to the Environment pane (top right-hand corner) and make sure the Environment tab is clicked.
3. Click on the “Import Dataset” button.
4. Select “From SPSS”.
5. A pop-up window will open.

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



6. Browse your folder and select the **burnout.sav** file to import.
7. Under Import Options, you can set the name of the data frame variable in R for the SPSS data set. Leave it as **burnout.sav**.
8. The Code Preview shows you the written code commands that you could type out to import the data frame instead of using the RStudio interface.
9. To import the dataframe, click “Import”.

Click Import. A tab in the source pane called “burnout” should now open showing the data frame. Additionally, you should also see the data frame in the Environment pane. You should see under value that it has “467 obs. of 6 variables,” indicating there are 467 rows of data and 6 columns of data.

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

```
head(Burnout)
```

```
## # A tibble: 6 x 6
##   burnout      loc cope teaching research pastoral
##   <dbl> <dbl> <dbl>   <dbl>   <dbl>   <dbl>
## 1 0 [Not Burnt Out]  7.65  9.16    32.7    87.5    31.5
## 2 0 [Not Burnt Out]  6.47 13.0    52.7    66.7    68.5
## 3 0 [Not Burnt Out]  8.82  9.16    49.1    60.4    53.7
## 4 0 [Not Burnt Out] 20     9.16    52.7    62.5    50
```

```
## 5 0 [Not Burnt Out] 6.47 19.1      43.6      79.2      40.7
## 6 0 [Not Burnt Out] 7.06 16.0      38.2      52.1      48.1
```

```
summary(Burnout)
```

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

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

4.6.3 Exporting Datasets in R

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

4.6.3.1 Exporting to CSV Files

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

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

The argument `file` will create the name of the file and enable you to change the location of the file. The way this is currently written, it will save your file to your working directory. If you need a reminder on how to set and check your working directory click [here](#). Make sure it is set to the location you want your file to go.

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

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

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

4.6.3.2 Exporting to SPSS Files

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

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

Once you have done that, the syntax for exporting data frames to SPSS is:

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

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

- `datafile`: This argument specifies the file path where the data will be saved in the desired format (e.g., SPSS data file). The `datafile` argument is used to store the actual data values from the R dataset.
- `codefile`: This is where the instructions on how to use your data are saved. When you specify a file path for the `codefile` argument, R will create a file containing commands or instructions that another program (like SPSS) can use to understand and import your data correctly. It's like writing down step-by-step directions for SPSS to follow.

So, in simple terms, the `datafile` is where your data is saved, while the `codefile` contains the instructions for using that data in another program. They work together to make sure your data can be easily used in different software programs.

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

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

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

4.7 Summary

Well done, you've gotten through Programming Part I and II. We covered a lot of useful (but let's be honest, not exactly riveting) concepts in programming in R. We've learnt how R categorises data, stores in data structures, learn how to convert data types and create those structures. Additionally, we learnt how to create variables and functions, install and load in packages to spice up R, and how to import and export data.

This sets us up nicely to start actually data processing next week when we will look at ways to clean our data.