Introduction to R

Ryan Donovan

2024-01-29

Contents

Chapter 1

Introduction

This set 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 fine-tuned 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 will put you at a disadvantage.

R enables you to import, clean, analyze, 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 workshop textbook 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 analyze 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 visualizations, 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 programs (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 program.

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: Basic Programming (Part I)
- Feb 21st: Basic 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 actually is R?

R is a statistical programming language that enables us to directly ask our computer to carry out tasks. Typically, when we use our computers, we do not speak to it directly; instead we interact with "translators" (i.e., applications like SPSS), via button-click interfaces, to speak to our computer on our behalf. Such interfaces record and translate our instructions to our computers, who 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 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 middle man.

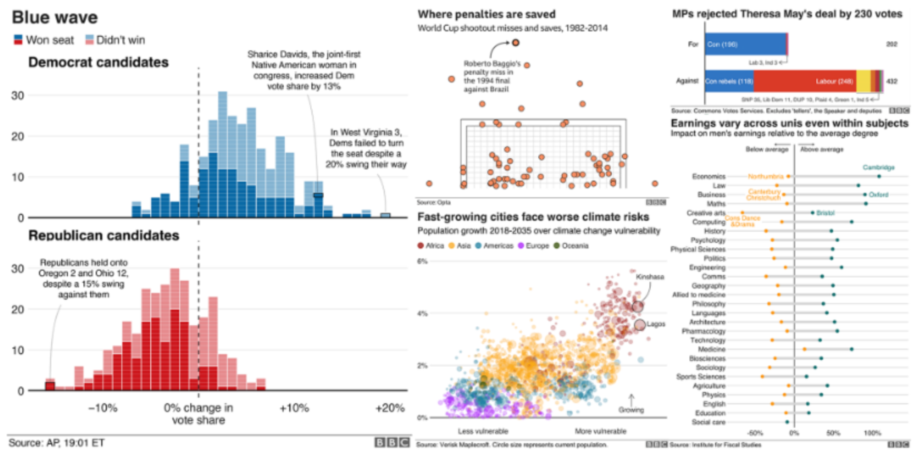


Figure 2.1: BBC graphs created in R.

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

2.2 Downloading R

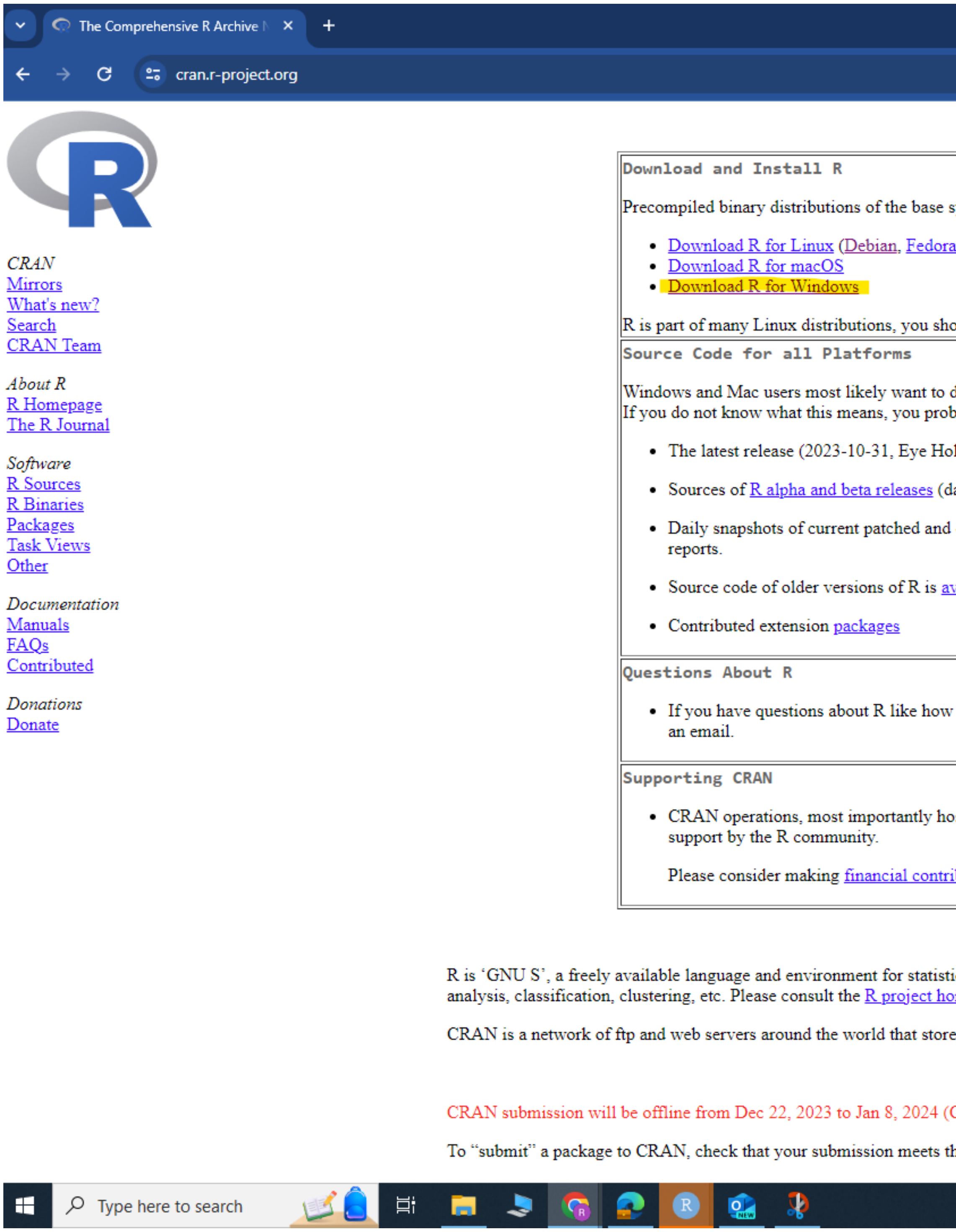
Please follow the following instructions to download R on either Windows or Mac.

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

But if you are goal is conduct data cleaning, analysis, visualization, and reporting, then R is a excellent choice. The good thing is that once you achieve a certain level of competency in one programming language, you will find it significantly easier to pick up a following one.

2.2.1 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*



CRAN
Mirrors
What's new?
Search
CRAN Team

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Task Views
Other

Documentation
Manuals
FAQs
Contributed

Donations
Donate

Download and Install R

Precompiled binary distributions of the base s

- Download R for Linux (Debian, Fedora
- Download R for macOS
- Download R for Windows

R is part of many Linux distributions, you sho

Source Code for all Platforms

Windows and Mac users most likely want to d
If you do not know what this means, you prob

- The latest release (2023-10-31, Eye Hol
- Sources of R alpha and beta releases (da
- Daily snapshots of current patched and
  reports.
- Source code of older versions of R is av
- Contributed extension packages

Questions About R

- If you have questions about R like how
  an email.

Supporting CRAN

- CRAN operations, most importantly ho
  support by the R community.

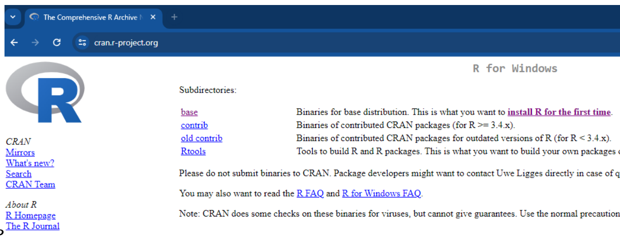
  Please consider making financial contri

R is ‘GNU S’, a freely available language and environment for statisti
analysis, classification, clustering, etc. Please consult the R project ho

CRAN is a network of ftp and web servers around the world that store

CRAN submission will be offline from Dec 22, 2023 to Jan 8, 2024 (C

To “submit” a package to CRAN, check that your submission meets th


The Comprehensive R Archive

cran.r-project.org

R for Windows

CRAN
Mirrors
What's new?
Search
CRAN Team

About R
R Homepage
The R Journal

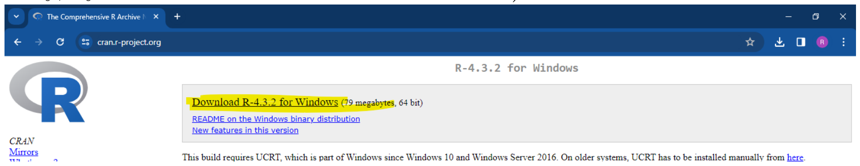
Subdirectories:

base	Binaries for base distribution. This is what you want to install R for the first time.
contrib	Binaries of contributed CRAN packages (for R >= 3.4.x).
old contrib	Binaries of contributed CRAN packages for outdated versions of R (for R < 3.4.x).
Rtools	Tools to build R and R packages. This is what you want to build your own packages

Please do not submit binaries to CRAN. Package developers might want to contact Uwe Ligges directly in case of q

You may also want to read the R FAQ and R for Windows FAQ.

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precaution

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

The Comprehensive R Archive

cran.r-project.org

R-4.3.2 for Windows

CRAN
Mirrors

Download R-4.3.2 for Windows (79 megabytes, 64 bit)
README on the Windows binary distribution
New features in this version

This build requires UCRT, which is part of Windows since Windows 10 and Windows Server 2016. On older systems, UCRT has to be installed manually from here.
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.

2.2.2 Downloading R on Mac

The instructions are largely the same for Windows. Please see this guide for more information https://teacherscollege.screenstepslive.com/a/1135059-install-r-and-r-studio-for-mac

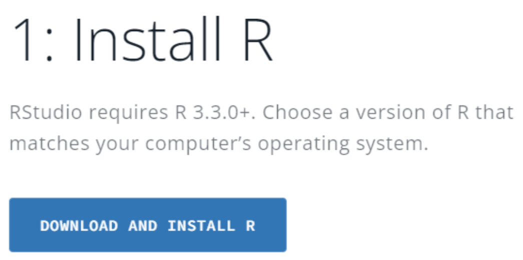
2.3 Install and Open R Studio

Once R is installed, we will install RStudio.

RStudio is a front-end program that makes it it much more user-friendly to use R without sacrificing our ability to code in R. R Studio will enable us to write and save R code, generate plots, manage our files, and do other useful things. RStudio relationship to R is similar to the relationship between a basic text editor and Microsoft Word. We could write a paper in a text editor, but it is much quicker and more efficient to use 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



1: Install R

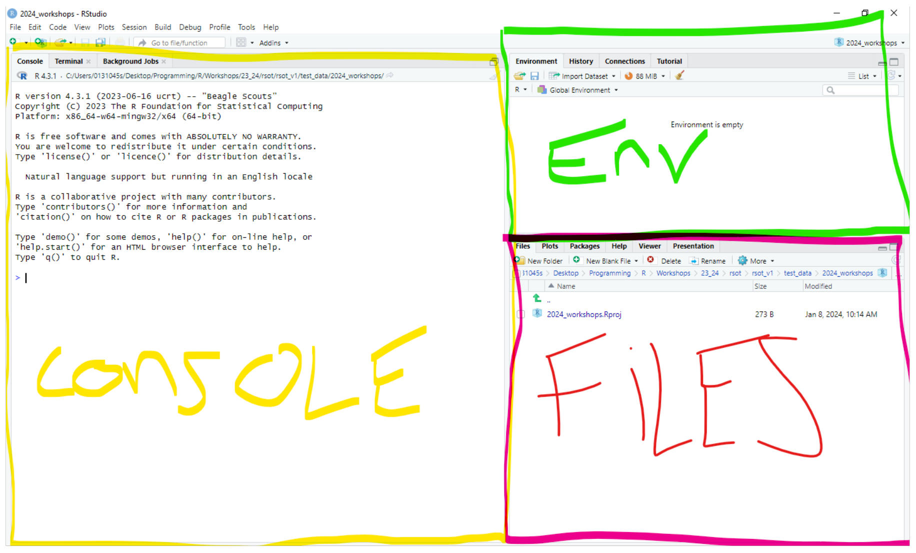
RStudio requires R 3.3.0+. Choose a version of R that matches your computer's operating system.

Download RStudio Desktop button.

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 going with all default options here.

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

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



```
R version 4.3.1 (2023-06-16 ucrt) -- "Beagle Scouts"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

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

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

> |
```

2.4 Creating an R Project

The first thing we will do in RStudio is create a *R Project*. R Projects are environments that will 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. Any time you open that project, you are telling R to work within this particular 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.

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 - ***introR_2024***

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

- You can save this anywhere else you like (e.g., your Desktop). Just make sure to save somewhere you can find and somewhere that will not change location (e.g., if you save folders to your desktop, but then tend to move them elsewhere once it gets cluttered, then do not save it to your desktop).
- My recommendation would be to create a folder called "R_Programming" on your desktop. And then save your project in this folder.
- Regardless of where you save your project, copy the location and paste it somewhere you can check (e.g., into a text file)

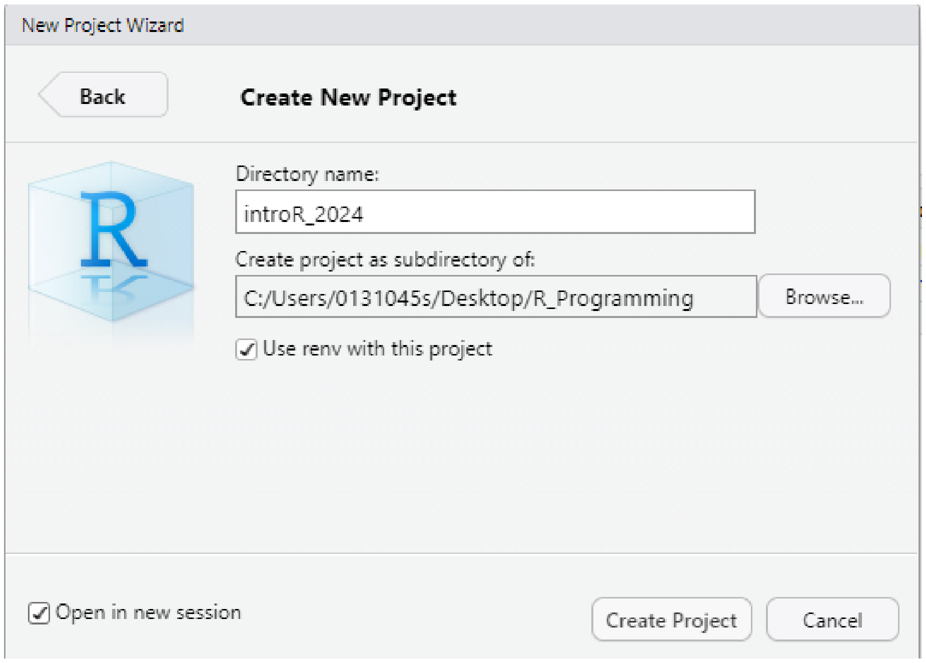
Option 3: The "Use renv with this project" option enables you to create a virtual environment for this project that will be separate to other R projects. Don't worry for now about what that means, it will be explained later on.

- Tick this option.

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

- Tick this option.

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 introR_2024.



Directory name: introR_2024

Create project as subdirectory of: C:/Users/0131045s/Desktop/R_Programming

☑ Use renv with this project

☑ Open in new session

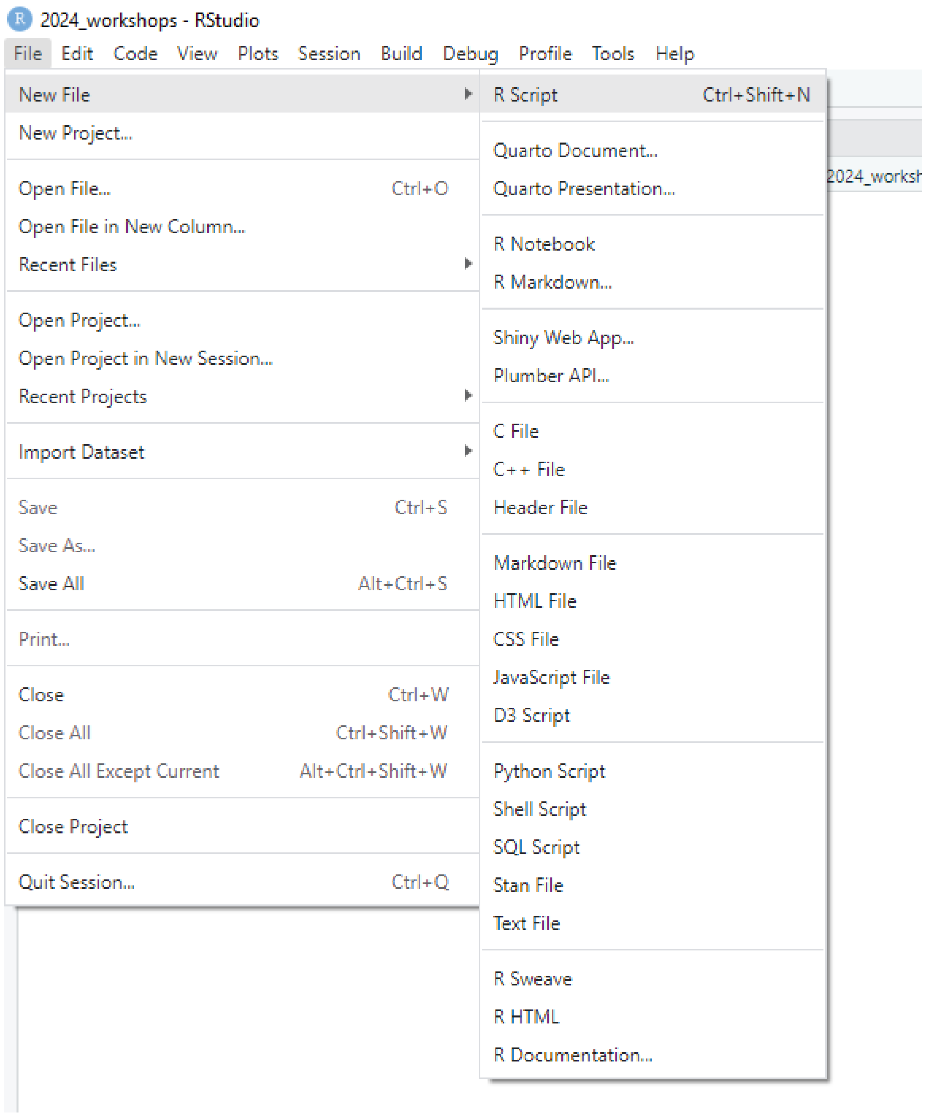
Figure 2.2: New Project Set Up

2.4.1 Navigating RStudio

In our new project, introR_2024, 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.

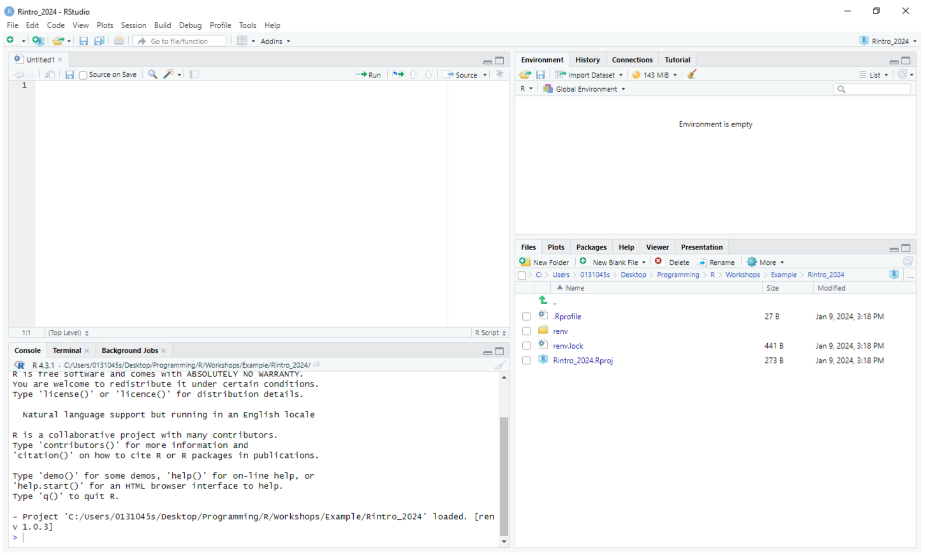
Button approach: Click the "File" tab in the top-left hand corner (not the File pane) -> Click "New File" -> "R Script"



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.


Rintro_2024 - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Rintro_2024
Untitled1
Source on Save
Run
Source
1:1
(Top Level)
R Script
Environment History Connections Tutorial
Import Dataset
143 MiB
List
R
Global Environment
Environment is empty
Files Plots Packages Help Viewer Presentation
New Folder
New Blank File
Delete
Rename
More
C: > Users > 0131045s > Desktop > Programming > R > Workshops > Example > Rintro_2024
Console Terminal Background Jobs
R 4.3.1 · C:/Users/0131045s/Desktop/Programming/R/Workshops/Example/Rintro_2024/

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

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

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

- Project 'C:/Users/0131045s/Desktop/Programming/R/Workshops/Example/Rintro_2024' loaded. [ren
v 1.0.3]
> |
```

Name	Size	Modified
..		
.Rprofile	27 B	Jan 9, 2024, 3:18 PM
renv		
renv.lock	441 B	Jan 9, 2024, 3:18 PM
Rintro_2024.Rproj	273 B	Jan 9, 2024, 3:18 PM

2.4.1.1 The RStudio Workspace

Now that we have each pane opened, let's briefly describe what each pane is for.

- 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 example, you might have an R script titled "Descriptive" which contains the code you need to compute descriptive statistics on your data set. Similarly, you might have another R script titled "Regression" that contains the code for computing your 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 several outputs from your R scripts. For example, if you create a script for running a t-test in R, then the results can be found in the Console Pane. You will also find any error or warning messages about any code that you run (e.g., if you make a mistake in your R code) highlighted in the console. In short, the console is where R is actually running.

- The ***Environment Pane*** is where you will find information on any data sets and variables that you import or create in R within a bespoke R project. The "History" tab will contain a history of any R code that you run during the project. This pane is really useful for getting a bird-eye's view of a project (which can be really useful if you are returning to a project after a long period of time or you are looking at someone's else code).

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

Each pane will be used extensively during these workshops.

2.4.2 Checking our Working Directory

Everytime you open up a project or file in RStudio, it is good practice to check the working directory. The working directory is the environment on our computer that R is currently operating in.

You want the working directory to be in the same location as your R project. That way any files you import into RStudio or any files you export (datasets, results, graphs) can easily be found in your R project folder. A lot of problems can be avoided in R by making sure that you check the working directory. To check the working directory, type the following into the console pane

```
getwd()
```

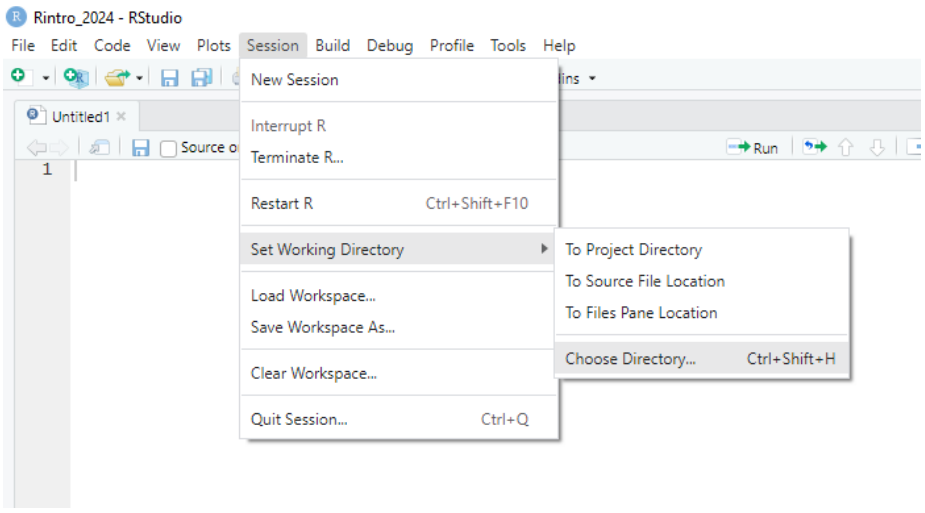
```
## [1] "C:/Users/0131045s/Desktop/Programming/R/Workshops/rintro"
```

What you get in return is the current working directory R is working in. Your working directory will not be the same as mine, that's perfectly normal. Just check to make sure that is in the same location you specified when you created your project (**Option 2**).

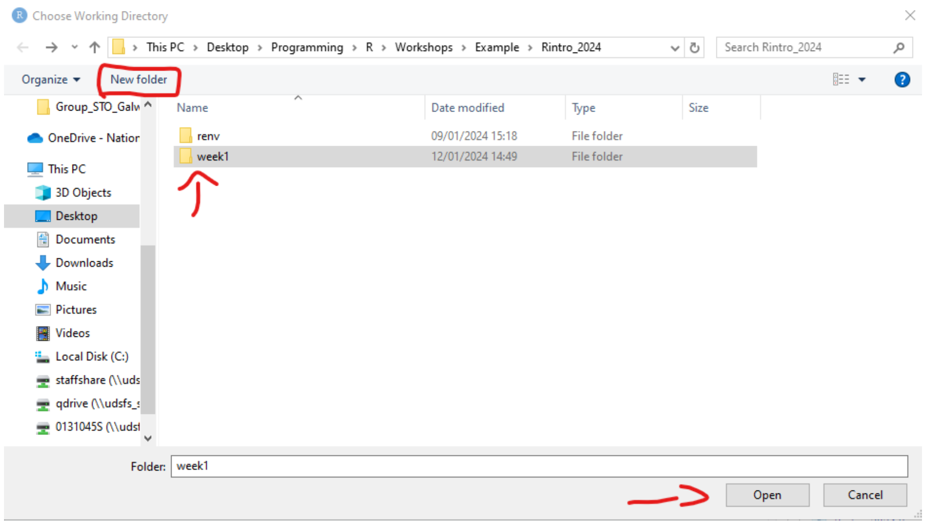
2.4.3 Setting up a new Working Directory

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_2024).
- 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/Example/Rintro_2024/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.

2.5 Writing our first R Code

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

Type in the 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 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.

2.6 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 do something, and it responds right away. The console The console is fantastic 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.

2.7 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 do not understand any of the following code. 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 are 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.

```r
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 difference variables (columns): extra (hours of extra sleep each participant had), group (which treatment they were taken), and ID (their participant ID).

Now let's calculate some descriptive statistics. One we can do this is by using the summary() function. This function takes in an an object (e.g., like a data set) and summaries 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 were +1.5, and that there was 10 participants within both the control and experimental condition.

But it's not perfect. Firstly, we don't need summary descriptive on the participant ID. Secondly, it only tells us the mean the entire sample, whereas we are more interested in the mean score per each treatment group. We can get this information by using the `aggregate()` function, which enables us to split our data into subset and then compute summary statistics per group. Remember to press run after you 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 a difference between treatment1 and treatment2. Participants slept an extra 2.33 hours on average when taking treatment 2, whereas they only slept .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 (assuming all parametric assumptions are met, of course...).

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

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

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

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

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

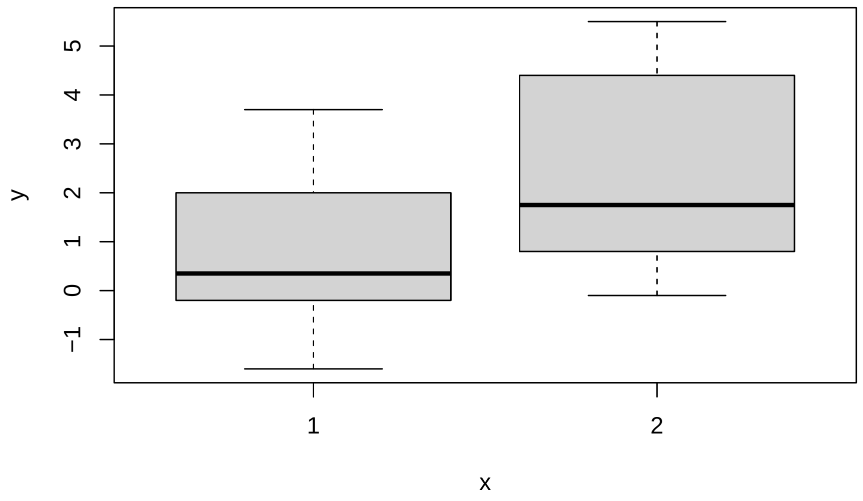


Figure 2.3: Generic Boxplot

The plot() function is an example of a generic function, which means it's a function that will try to adapt to our code. In this case, the plot() function looks at the variables we wants 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 axis, and by adding a title to it.

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

#ylab = creates a title for the y-axis

#main = creates a title for the plot



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

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

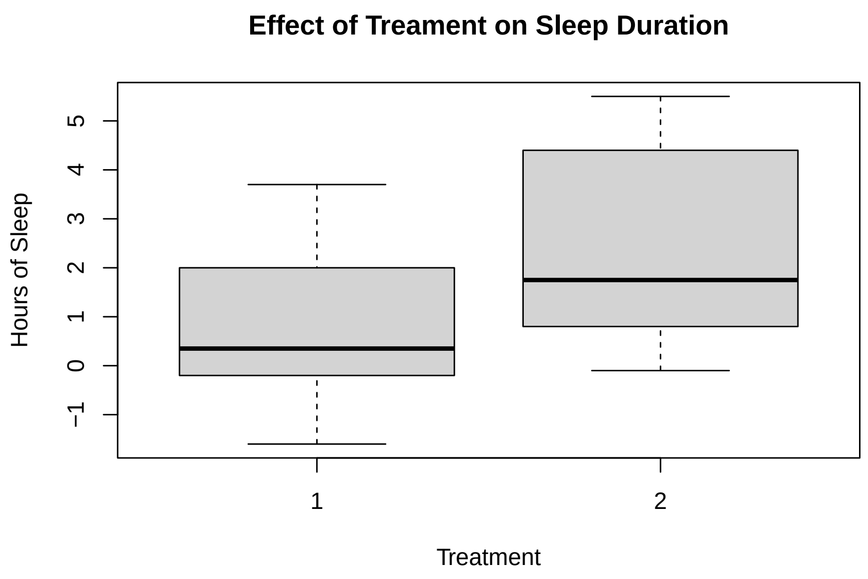


Figure 2.4: Generic Boxplot with appropriate labelling

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 graph[2].

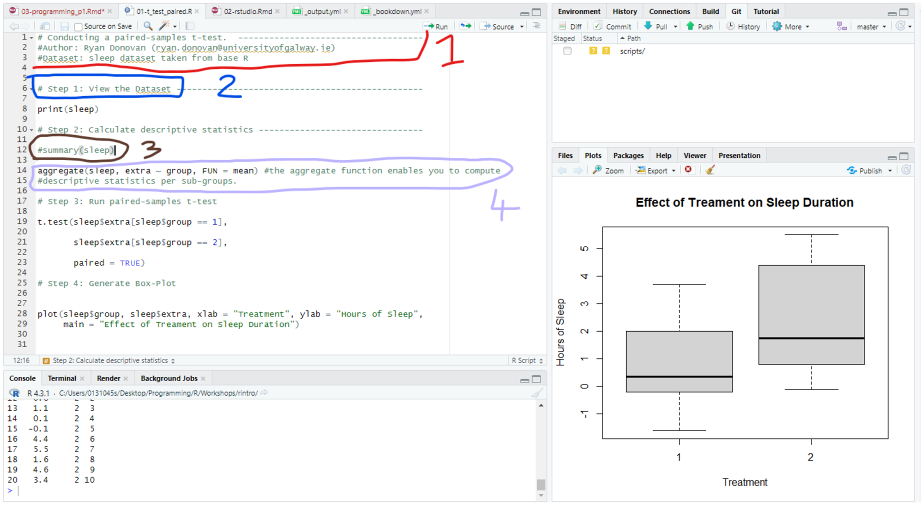
[2]This is a fairly generic type of graph offered by base R. During the course we will looking at ways we can create "sexier" and more APA friendly type of graphs. But for one line of code, it's not bad!

2.7.1 Comments

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 `#` symbol. This symbol tells R to ignore whatever comes directly ***afterwards***.

There are various reasons for using comments.



```r
# Conducting a paired-samples t-test. -----------------------------------
#Author: Ryan Donovan (ryan.donovan@universityofgalway.ie)
#Dataset: sleep dataset taken from base R


# Step 1: View the Dataset ----------------------------------------------

print(sleep)

# Step 2: Calculate descriptive statistics ------------------------------

#summary(sleep)

aggregate(sleep, extra ~ group, FUN = mean) #the aggregate function enables you to compute
#descriptive statistics per sub-groups.

# Step 3: Run paired-samples t-test

t.test(sleep$extra[sleep$group == 1],

       sleep$extra[sleep$group == 2],

       paired = TRUE)

# Step 4: Generate Box-Plot


plot(sleep$group, sleep$extra, xlab = "Treatment", ylab = "Hours of Sleep",
     main = "Effect of Treament on Sleep Duration")
```

```
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
> 
```

Figure 2.5: 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 a clear information on what this script is trying to do (e.g., run a paired samples t-test), what data it is working on (the sleep dataset), and who wrote or developed this script. This makes it significantly easier for anyone who might be reviewing your work or trying to apply your code to their own work to understand what is going on.

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

3. The third type of comment is placed before the summary. This means that 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 anytime 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 now well known.

Comments are extremely useful for orientating yourself to code. My advice would be to comment as much as your code as you. Anyone who has coded will have experience 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 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!

2.8 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.

Chapter 3

Basic 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 utilizing variables.
- Grasping data structures and how to construct them.
- Loading R packages and utilizing their functions.

3.0.1 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")
```

3.1 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 into our food.

3.1.1 Typing Commands in the Console

We can command the R console to compute calculations. Remember not to type the operator ´>´ if you are following along in RStudio, this just indicates that R is ready to execute a new command.[1]

```
> 10 + 20

[1] 30
```

```
> 20 / 10

[1] 2
```

If you are performing calculations in R, it's important to know that it follows the usual arithmetic convention of order of operations (remember BIMDAS - Bracets, Indices, Multiplication, Division, Addition, and Subtraction?).

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

[1] 84

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

[1] 12
```

Now you'll have noticed that the output of every line of code we entered starts with a `[1]` before our actual result. What does this mean?

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

This is a way for R to label and organize its responses. Imagine you have a conversation with R, and every time you ask it something, it gives you an answer. The square brackets with a number, like `[1]`, are like labels on each response, telling you which answer corresponds to which question. This is R ***indexing*** its answer.

In each of the above examples, we asked R a question that only has only 1 answer, which is why the output is always `[1]`. Look what happens when I ask 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 [17], which tells us that the 17th answer corresponds to 5.5. The reason that R does this is that R only prints out the index in the console each time it needs to make a new row. This helps us orient ourselves to answers, without adding a lot more data (e.g., if there was an index for every single answer that would double the number of data we were looking at).

We will come back to indexing later in this session. It is an extremely useful concept in R.

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

3.1.2.1 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 will not try and understand what you actually meant. For example, see what happens if you write the following code:

```
> 10 = 20
```

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

R thinks you making the claim that 10 equals 20. Since this is not true, R panics and refuses to run 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 mistake is not that costly because R will tell us immediately that something is wrong and stop us from doing anything.

However, there are silent type of mistakes that are harder to solve. Let's image that you typed in `-` instead of `+`.

```
> 10 - 20

[1] -10
```

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

In short calculations like this, it's clear what you typed wrong. But if you run a long-block of connected code and made a typo like this somewhere, the result you get can be significantly different from the intended result and it might be difficult to spot.

The main way to check for this is to always view the output of your code. If it looks significantly different from what you expected, then this type of silent error might be causing it.

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 be easily fixed by tiny changes.

3.1.2.2 R Console and Incomplete Commands

I have 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. If nothing is missing, then this indicates that your code might not be correctly formatted. Overall, the moral of this section can be summarized as: proofread your code!

Okay, that's a lot about using the console. Let's move on to other programming concepts.

3.2 Data Types

Our overall goal for this course is to give you the ability to import your data into R, select a subset of the data most of interest for a given analysis, carry out an analysis to summarize these data and create visualizations of the data. But it's important to consider ***What is Data and how is it stored in R?***

Data comes in many forms: Numbers (Integers and decimal values) or alphabetical (characters or lines of text). R has developed a system for classifying this range of data into different data types.

3.3 Basic Data types in R

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

3.3.1 Character

A character is anything wrapped inside quotation marks. It is often referred to as a *string*.

Strings can be anything inside 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 in quotation marks are also recognized as a character type in R.

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

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

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

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

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

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

3.3.2 Numeric (or Double)

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

```r
class(33)
```

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

```r
class(33.33)
```

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

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

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

3.3.3 Integer

An integer is any real whole number with no decimal points. We tell R to specify something as an integer by adding a capital "L" at the end.

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

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

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

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

```r
class(0L)
```

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

It might seem weird that R has a separate data type for integers when the numeric/double character type contains integers. Why bother with the separate data type?

The reason is that integers store less space in your computers memory than the numeric or double data type. There's less information in "33" compared to "33.00". So if you have a very large dataset (in the millions), and you know for certain the data will only be integer, using the integer data type will save you a lot of storage space.

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

3.3.4 Logical (otherwise know as Boolean)

The Logical Data type has two potential values: `TRUE` and `FALSE`.

In programming, we often need to deal with conditions and make decisions based on whether certain conditions are true or false. Did the student pass the exam? Is this p-value below `.05`?

The Logical data type in R enables us to represent and work with these truth values.

```r
class(TRUE)
```

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

```r
class(FALSE)
```

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

One thing to note is that it case sensitive, so typing in any of the following will produce errors.

```r
class(True)

Error: object 'True' not found

class(False)

Error: object 'False' not found

class(true)
```

```
Error: object 'true' not found

class(false)

Error: object 'false' not found
```

The main reason that there are different types of data in programming is that some commands can only be used on particular data types. For instance mathematical operations (+, -, x and /) are only meaningful for numbers.

```
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 important to know when troubleshooting errors in R. It's not uncommon that when you download a data set online that a column that should be numeric is actually saved as a character (e.g., the person entering the data might have a mistake in the excel file).

If you wanted to perform a statistical operation on that column (e.g., mean), you would first have to convert it to the numeric data type. You can do this by using the following function

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

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

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

```
as.character() #will convert data type to character

as.integer() #will convert to integer

as.logical() #will convert to logical
```

3.4 Variables

The code we have been using so far has been single-use code. Once we have typed out code, there is nothing else we can do but look at their output. But programming languages enable us to store information to objects called ***variables***.

Variables are labels for pieces of information. If we want to use that information or recall later on, instead of running the same code that produced the information, we can just refer to its label. Let's say I have a character object that refers to my name. I can save that character object to a variable.

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

To create a variable, we first type the name of the variable (`name`). We use the assignment operator to tell R that we will be assigning (i.e., storing) information to`name`. This information is the string "Ryan". Once we run this code, then whenever R sees the variable`name` it will replace it with Ryan.

```
print(name)
```

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

Now many of you who have seen my email will think *"Hold on a second, isn't your first name Brendan? You fraud!"*. Now before you grab your pitchforks, yes, you are technically correct. Thankfully, we are able to reassign our variable labels to new information.

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

print(name)
```

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

All data types can be stored as information to variables.

```
age <- 30L

height <- 175 #centimetre

live_in_hot_country <- FALSE

print(age)
```

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

```r
print(height)
```

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

```r
print(live_in_hot_country)
```

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

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

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

We can use variables compute calculations with their information. Let's say I have several variables relating to my scores on five items measuring Extraversion (which I will label with the variable names `extra1` to `extra5`). I can use the variable names to calculate my total Extraversion score.

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

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

print(total_extra)
```

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

```r
mean_extra <-  total_extra/5

print(mean_extra)
```

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

Variables are an extremely powerful tool in programming that enable us to create code that work across a variety of situations.

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

There are hard and soft rules for naming variables that you should know.

Hard Rules (Follow these, otherwise R won't create the variable for you)

Variable names can only include upper case alphabetic characters A-Z, lowercase a-z, numeric characters 0-9, as well period `.` and underscores `_`.

Variables names must start with either a letter or a period (1st_name or _1stname is wrong. first_name or .firstname is right)

Do not use spaces in variable names (`my name` is not okay. 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 that R reserves (e.g., if, else, repeat, while, function, for, in, TRUE, FALSE). You don't need to memories this, but it is worth remembering this if an error with your variable name comes up. After some time, you'll gain a strong intuition for what is a valid name and what is not.

Soft Rules (you should follow these, otherwise your scripts and code will be messy)

Choose informative variable names that clearly describe the information they represent. Someone should be able to look at your variable name and be able to reasonably guess what type of information it is storing. Variable names like "income", "grades", "height" are clear, whereas variables names like "money", "performance", or "cm" are ambiguous (e.g., what money - money received, spent, owed? performance on what? what was being measured in cm, height, width?). And for the love of God don't name your variables "variable1", "variable2", "variable3"!

Choose short variable names whenever possible. Concise names like `dob` or `iq` are better than `date_of_birth` or `intelligence_quotient`. It's much easier to work with shorter names as it avoids unnecessary and tedious typing thereby reducing the chance for making typos.

But you should choose a long variable name that is clear over a short variable name that is unclear. A long variable name like `total_exam_marks` is significantly better than a cryptic acronym like `tem`.

Do not start your variables with a capital letter.[2] A standard convention in the R programming language is to use lowercase letters when naming variables or

[2]This rule is the most "soft", but it does bear similarity with conventions in spoken languages. For example, German and English native-speakers tend to emphasize time at different points in a sentence. English speakers tend to specify the time at the end of the sentence "I will drive to Dublin on Friday", whereas German speakers tend to specify time near the beginning

functions. So you could start your variable names with a capital letter, but it would look and sound strange to other R users

Use a conventional naming style and be consistent with it. There are three conventional styles for handling variables that contain multiple words.

1. The first style separates each word with an underscore (e.g., my__age, my__name, my_height). This is called `Snake_case`.
2. The second style separates each word with a period . (e.g., my.age, my.name, my.height).This is called `dot.notation`.
3. The third style capitalizes every additional word after** the first word (e.g., myAge, myName, myHeight). This is called `camelCase`.

I recommend that you use the `Snake_case` during this course, because it will keep your code consistent with my code. Additionally, this case is consistent with other programming languages, whereas `dot.notation` is not.

Outside of this course, feel free to pick whichever style you prefer. Just be consistent with it.

3.5 Data Structures

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

This also holds true in R, which has several different types of ***data structures*** that organise and group together variables. Each data structure has specific rules and methods for creating or interacting with them. I'll briefly mention each data structure first, before we focus on two main data structures we'll use in this course: `vectors` and `data frames`.

3.5.1 Vectors

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

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

of the sentence "I will on Friday drive to Dublin" (Ich werde am Freitag nach Dublin fahren.). Of course, you could say it the German way in English, or the English way in German, but it would sound a bit unnatural.

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


print(rintro_names)
```

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

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

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

And I can create a numeric vector with their performance on the module.

```r
rintro_marks <- c(87, 91, 87, 90, 88, 89)

print(rintro_marks)
```

```
## [1] 87 91 87 90 88 89
```

And I can create a logical vectors that describes whether or not they like R.

```r
rintro_satisfied <- c(TRUE, FALSE, T, F, F, T) #you can use T or F as shortcuts

print(rintro_satisfied)
```

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

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

```r
instructor <- "Ryan/Brendan"

is.vector(instructor)
```

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

Activity: Try create a vector with only integers. Call it "int_vector". Check whether you have successfully created this vector with the code `class(int_vector)`

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

```r
rintro_marks <- c(87, "A1", 87, 90, 88, 89)


print(rintro_marks)
```

```
## [1] "87" "A1" "87" "90" "88" "89"
```

R has converted every element within the `rintro_marks` 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 and convert every element to one data type. This is a strict rule in R - a vector can only be created if ever single element (i.e., thing) inside that vector is of the same data type.

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

```r
rintro_marks <- c(87, 91, 87, 90, 88, 89) #original numeric vector


class(rintro_marks)
```

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

```r
rintro_marks <- c(87, "A1", 87, 90, 88, 89)


class(rintro_marks)
```

```
## [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 reads this column, it sees the text, and then R converts the entire column into characters.

3.5.1.1 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.

```
rintro_marks <- c(87, 91, 87, 90, 88, 89) #original numeric vector

mean(rintro_marks)
```

```
## [1] 88.66667
```

```
median(rintro_marks)
```

```
## [1] 88.5
```

```
sd(rintro_marks)
```

```
## [1] 1.632993
```

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 highest scores
```

```
## [1] 87 87 88 89 90 91
```

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] 91 90 89 88 87 87
```

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.

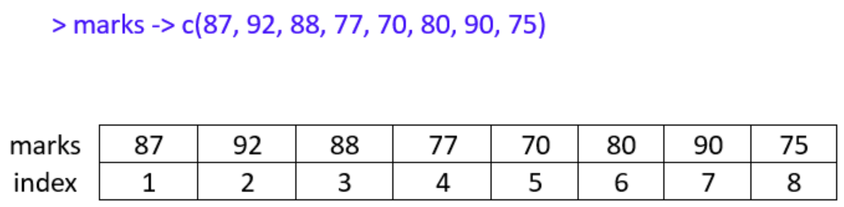
```
summary(rintro_names)
```

```
##    Length     Class      Mode 
##         6 character character
```

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

3.5.1.2 Vector Indexing and Subsetting

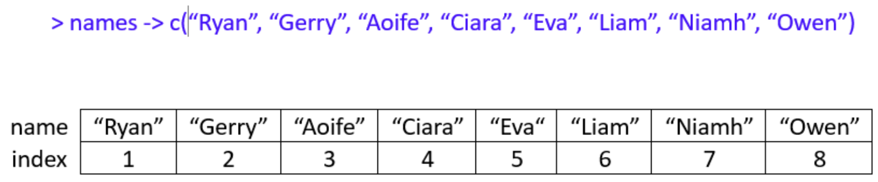
A vector in R is like a list of items. To be more specifc, vectors in R are actually *ordered* list of items. Each item in that list will have a position (known as its index). When you create that list (i.e., vector), the order in which you input the items (elements) determines its position (index). 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.



```
> 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

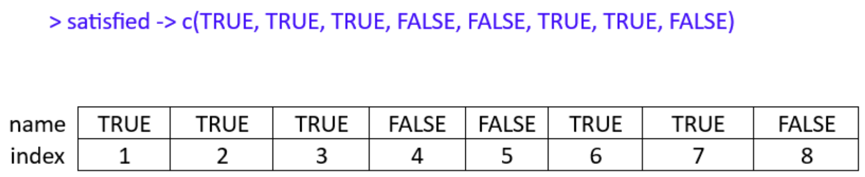
Figure 3.1: Indexing for Numeric Vector



```
> 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

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

```r
rintro_names[1]
```

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

Similarly, I could extract the 3rd element.

```r
rintro_marks[3]
```

```
## [1] 87
```

Or I could extract the last element.

```r
rintro_satisfied[8]
```

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

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 8th elements. I just need to use `c()` to tell R that I am subsetting several elements.

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

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

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

```
## [1] 91 90 NA
```

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

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

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

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

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

It's important to know however that when you perform an operation on a vector or you subset it, it does not actually change the original vector. For example, if you wanted to save the top three `rintro_marks`, the following code would not work:

```
sort(rintro_marks, decreasing = TRUE)


[1] 91 90 89 88 87 87

rintro_marks[c(1, 2, 3)]

[1] 87 91 87
print(rintro_marks)

[1] 87 91 87 90 88 89
```

You can see that neither the `sort()` function or 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] 91 90 89
```

Over to you! How could we extract and save the bottom 3 results from the `rintro_marks`‘ vector? Give it a go.

Bonus Points: Calculate the mean of both the top 3 marks and bottom 3 marks.

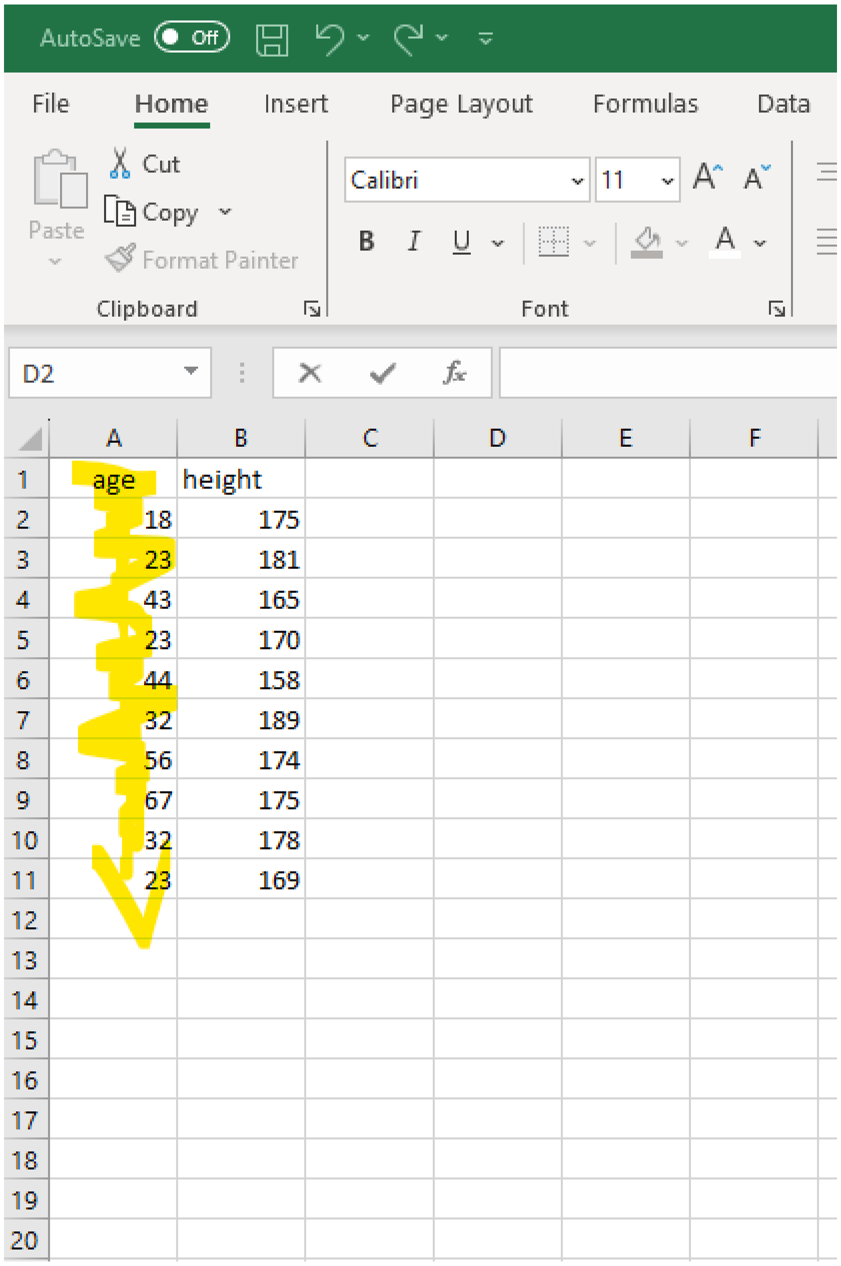
3.5.1.3 Vectors - making it a little less abstract.

You might find the discussion of vectors, elements, and operations very abstract. I certainly did when I was 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 it more 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" will all the ages of my participants. That column is a vector of data with the variable label "age".

Creating that vector is the equivalent of creating a column in excel.

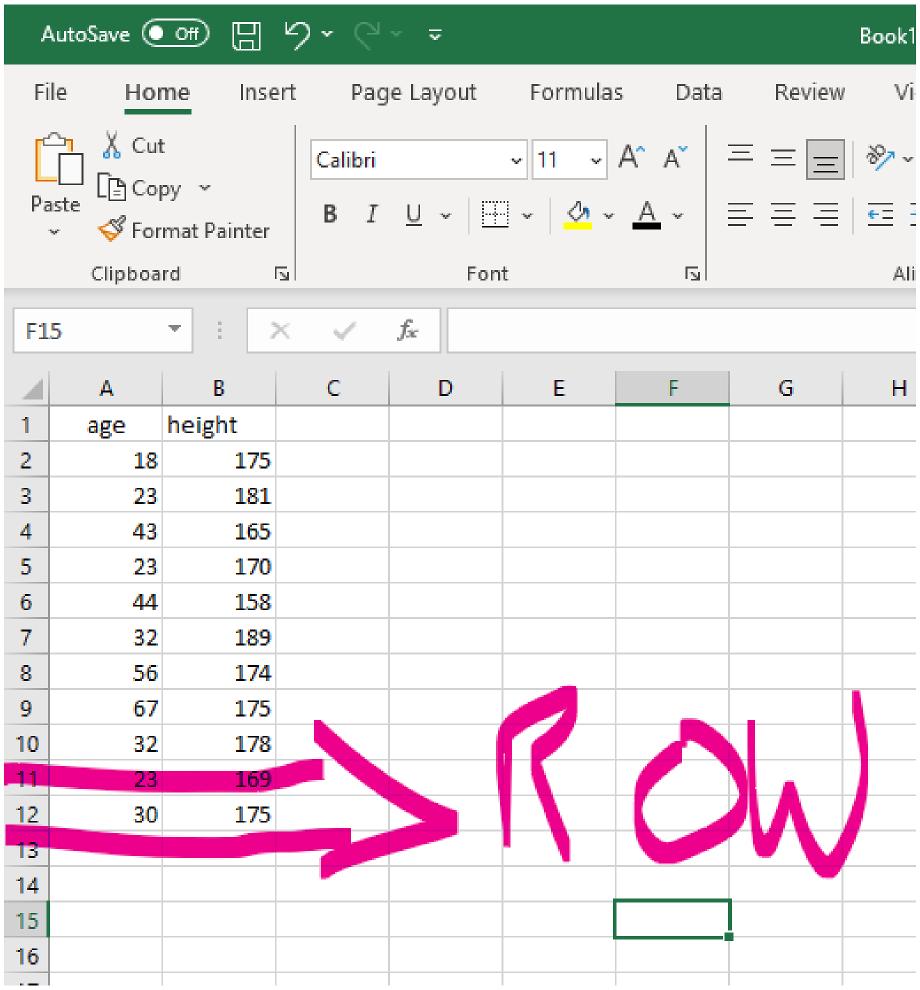
```r
age <- c(18, 23, 43, 23, 44,32, 56, 67, 32, 23)
```



	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 create a new row (line) of data. In R this is the equivalent of doing this:

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



	A	B
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	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 we combine different vectors (columns and rows) together? We create a `data frame`.

3.5.2 Data frames

Data frames 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.

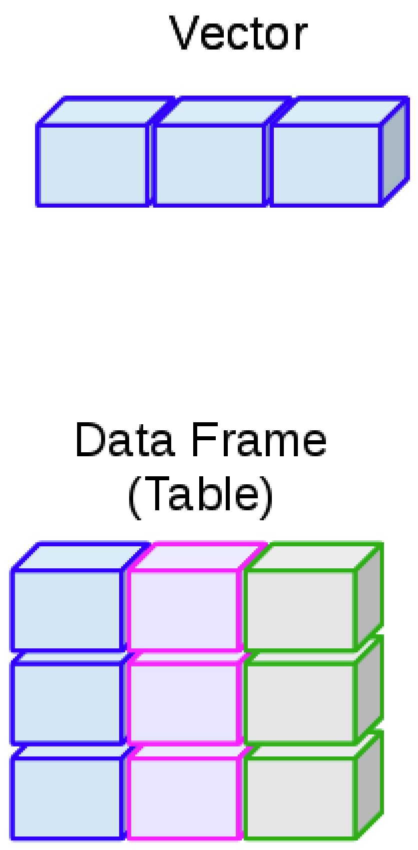


Figure 3.4: The relationship between data frames and vectors. The different colours in the data frame indicate they are composed of independent vectors

Data frames are an excellent way to store and manage data in R because it can store different types 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
```

3.5.2.1 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.

3.5.2.1.1 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 individual select columns? We can do so through a variety of tools:

The **$** Notation: You can use a dollar sign ($) followed by the column to select a particular 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 similar approach to accessing elements from a vector. Inside the brackets you can specify both the row and columns that you want to extract

In a data frame, every column has a name, similar to how each column in an excel spreadsheet has a header. These columns names enable you to access and manipulate specific columns or variables within your data frame.

You can also use square brackets to access columns. Inside the brackets, you specify the row indices (or leave it empty for all rows) and the column name in quotes. For example, to access the "Age" column of my_df:

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

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

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
```

3.5.2.1.2 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. Here are some examples:

To get the first row of your data frame (my_df):

```
my_df[1, ]
```

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

To access the third row:

```
my_df[3, ]
```

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

If you want multiple rows, you can specify a range of row indices:

```
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. You can combine row and column indexing to access specific elements within the data frame as needed.

3.5.2.1.3 Selecting Rows and Columns Following on from these examples, we can select both rows and columns using `[]`.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
```

3.5.2.2 Adding Data to your Data Frame

3.5.2.2.1 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 straight forward

```
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", "Male", "Non-binary")

#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       Male
## 3 Charlie  22 75.33 Non-binary
```

Easy pease.

3.5.2.2.2 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 creating 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
```