

A Simple Cookbook for Wrangling and Visualization

Bernard Liew

2021-10-11

Contents

1	Preface	7
1.1	Why is R so great?	7
1.2	Why R is like a relationship...	8
1.3	3 things students get wrong	8
1.4	R resources	11
2	Getting Started	15
2.1	Installing Base-R and RStudio	15
2.2	The four RStudio Windows	17
2.3	Reading and writing Code	21
2.4	Debugging	24
2.5	Learning check	26
3	The Basics	29
	Download and load packages	29
3.1	The command-line (Console)	29
3.2	Writing R scripts in an editor	31
3.3	A brief style guide: Commenting and spacing	33
3.4	Objects and functions	35
4	Navigating the Software	43
	Introduction	43
	Download and load packages	43
4.1	Getting and Setting the Working Directory	43

4.2	Creating a new Rstudio project	45
4.3	Installing Packages	45
4.4	Learning check	50
5	Input and Output	53
	Download and load packages	53
5.1	Dealing with “Cannot Open File” in Windows	53
5.2	Reading in Excel “.xlsx” data	54
5.3	Writing a Data Frame to Excel	55
5.4	Learning check	55
6	Data manipulation	57
	Download and load packages	57
	Import data	57
6.1	Tidy data	61
6.2	Renaming variables	64
6.3	Selecting rows and columns	65
6.4	Convert characters to numeric	67
6.5	Convert characters to date and time	69
6.6	Split Numeric Variable into Categories	71
6.7	Gathering	72
6.8	Spreading	75
6.9	Rename values of a variable	75
6.10	Creating factors	76
6.11	Making a new variable	78
6.12	Filtering	79
6.13	Global summary	84
6.14	Group-by summary	85
6.15	Merge two tables together	87
6.16	Learning check	88

CONTENTS	5
7 Bar Graphs	91
7.1 Making a Basic Bar Graph	95
7.2 Anatomy of a Graph	97
7.3 Grouping Bars Together	100
7.4 Using Colors in a Bar Graph	103
7.5 Changing Axes titles in a Bar Graph	104
7.6 Changing Legend titles in a Bar Graph	104
7.7 Changing font size uniformly across the Bar Graph	107
7.8 Outputting to Bitmap (PNG/TIFF) Files	108
7.9 Learning check	110
8 Line Graphs	111
8.1 Making a Basic Line Graph	112
8.2 Making a Line Graph with Multiple Lines	113
8.3 Changing the Appearance of Lines	116
8.4 Using Themes to Change Overall Appearance of Plot	119
8.5 Learning check	120
9 Your Assignment	123
9.1 Tasks to complete	124
9.2 Codes	125

Chapter 1

Preface

The purpose of this book is to help you **CLEAN** your data and produce **AMAZING** graphics **EASILY** but not fast at first.

What this book is

This book is intended to be a cookbook based approach. It has problems and it has solutions. Recipes are provided from which you can edit the necessary portions for your needs.

What this book is not

This book does not cover any one topic in extensive detail. If you are interested in conducting analyses or creating plots not covered in the book, I'm sure you'll find the answer with a quick Google search!

1.1 Why is R so great?

1. R is 100% free and as a result, has a huge support community. Unlike SPSS and even Excel, R is, and always will be completely free. This doesn't just help your wallet - it means that a huge community of R programmers who will constantly develop and distribute new R functionality and packages at a speed that leaves all those other packages in the dust! If you ever have a question about how to implement something in R, a quick Google search will lead you to your answer virtually every single time.
2. R is incredibly versatile. You can use R to do everything from calculating simple summary statistics, to performing complex simulations to creating gorgeous plots. If you can imagine an analytical task, you can almost certainly implement it in R.

3. Analyses conducted in R are transparent, easily shareable, and reproducible. If you ask an SPSS user how they conducted a specific analyses, they will either A) Not remember, B) Try (nervously) to construct an analysis procedure on the spot that makes sense - which may or may not correspond to what they actually did months or years ago, or C) Ask you what you are doing in their house. I used to primarily use SPSS, so I speak from experience on this. If you ask an R user (who uses good programming techniques!) how they conducted an analysis, they should always be able to show you the exact code they used. Of course, this doesn't mean that they used the appropriate analysis or interpreted it correctly, but with all the original code, any problems should be completely transparent!

1.2 Why R is like a relationship...

Yes, R is very much like a relationship. Like relationships, there are two major truths to R programming:

1. There is nothing more *frustrating* than when your code does *not* work
2. There is nothing more *satisfying* than when your code *does* work!

Anything worth doing, from losing weight to getting a degree, takes time. Learning R is no different. Especially if this is your first experience programming, you are going to experience a *lot* of headaches when you get started. You will run into error after error and pound your fists against the table screaming: "WHY ISN'T MY CODE WORKING?!?!? There must be something wrong with this stupid software!!!" You will spend hours trying to find a bug in your code, only to find that - frustratingly enough, you had had an extra space or missed a comma somewhere. You'll then wonder why you ever decided to learn R when (::sigh::) SPSS was so "nice and easy."

Trust me, as you gain more programming experience, you'll experience fewer and fewer bugs (though they'll never go away completely). Once you get over the initial barriers, you'll find yourself conducting analyses much, much faster than you ever did before.

1.3 3 things students get wrong

Based on teaching the previous cohort, there are 3 common errors students make, which will result in errors, leaving many scratching their head, feeling frustrated. Be aware of these errors, and do not fall into the trap.

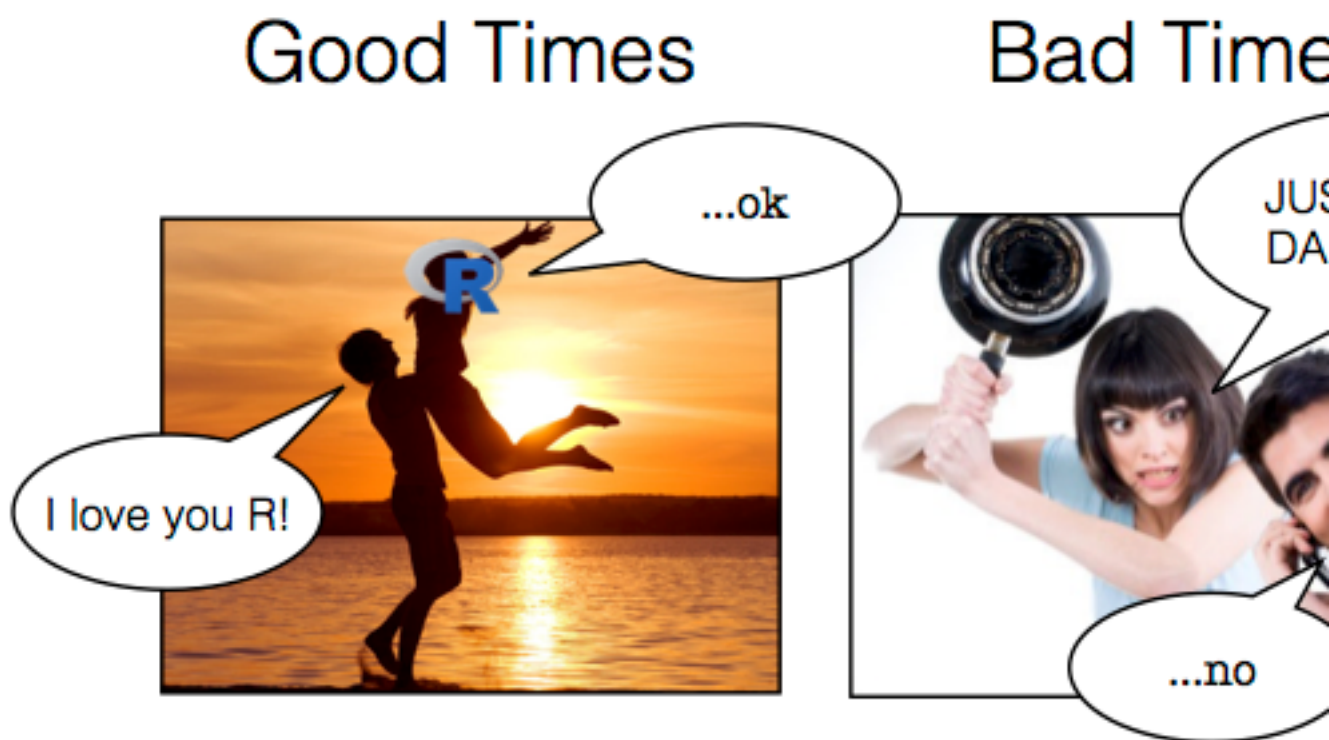


Figure 1.1: R will become both your best friend and your worst nightmare. The bad times will make the good times oh so much sweeter.

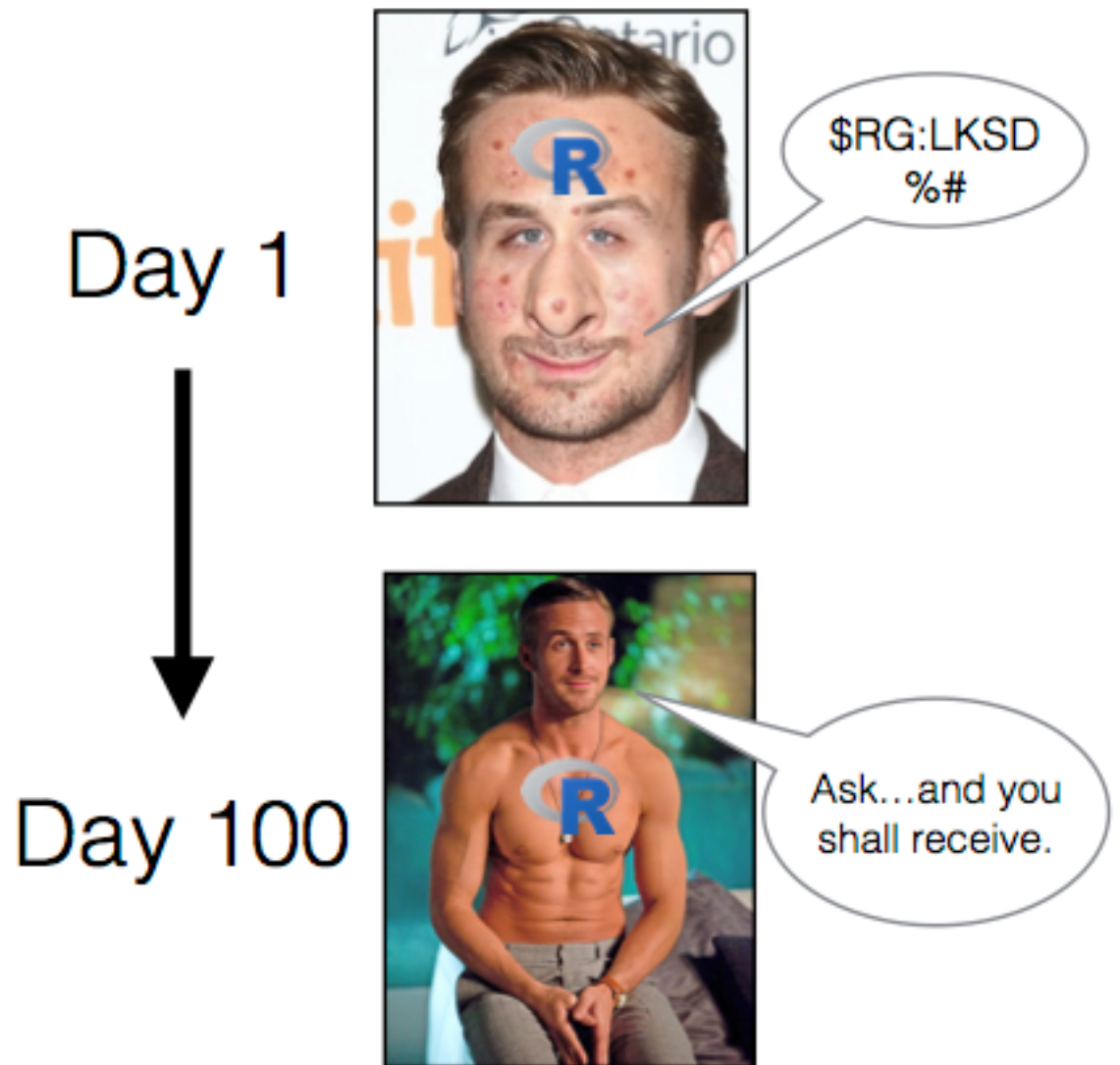


Figure 1.2: When you first meet R, it will look so fugly that you'll wonder if this is all some kind of sick joke. But trust me, once you learn how to talk to it, and clean it up a bit, all your friends will be crazy jealous.

1.3.1 Wrong method of launching software

This is the biggest source of frustration. Once you have created a project folder for your work, ONLY launch the software by clicking on the blue cube symbol as seen in Figure 1.3. In chapter 4, I will explain why you should launch the software in this manner.



Figure 1.3: This is the symbol to press to launch the software in your assignment.

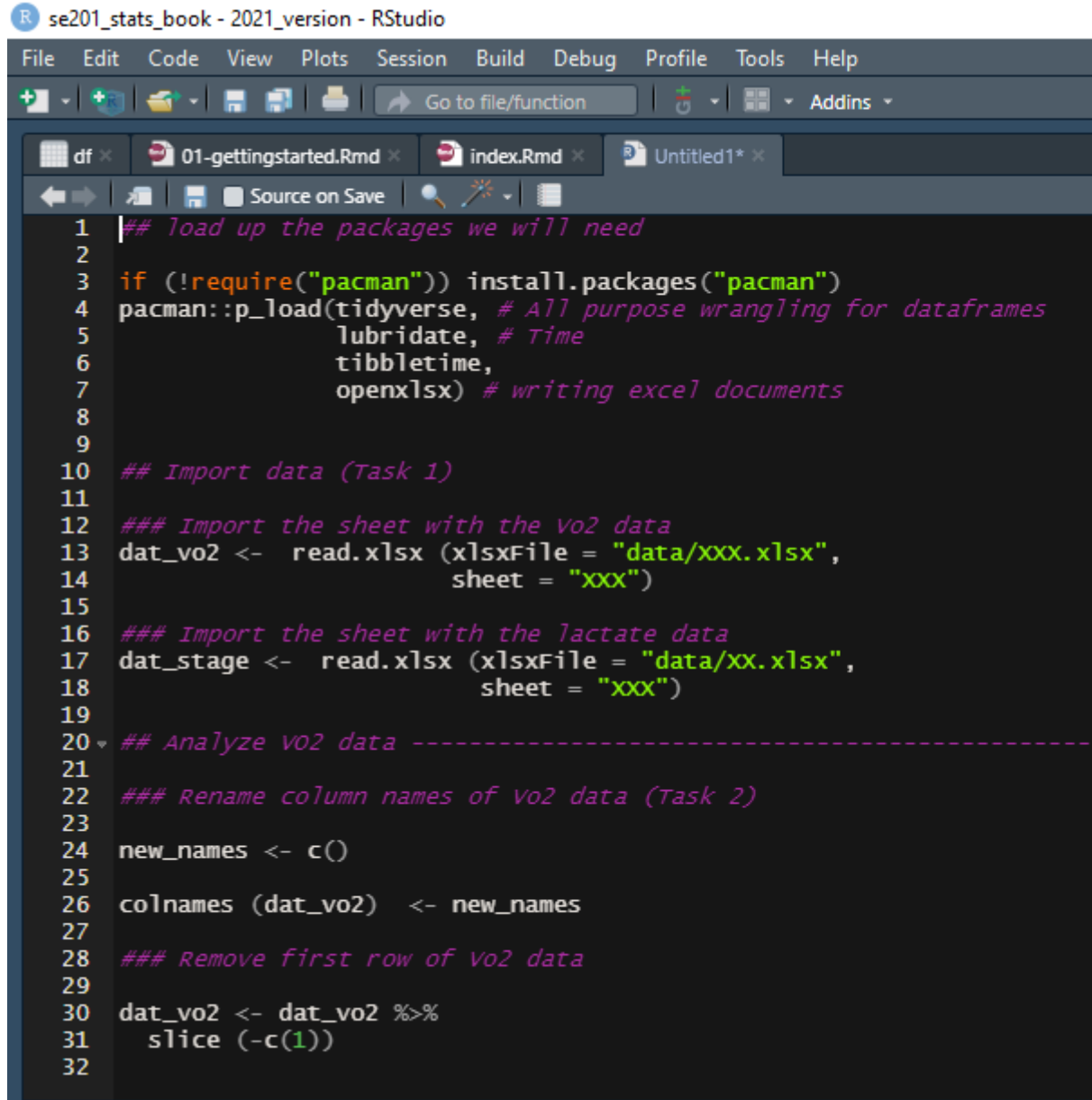
1.3.2 Not running code from START

Looking at the figure 1.4 below, say you ran the code from line 3 to line 20, when you close the software - maybe leaving the work partially done for the day, the next time you launch the software, it has no recollection of where you stopped. You need to run the software from the first line of the code to where you stopped, and then continue with your work.

1.3.3 Typographical errors

More than 50% of the student queries in the past, centered around the issue of typographical errors. Given that humans can (and do) communicate without perfect grammar, punctuation, and spelling, you WILL naturally assume (wrongly) that computers can ignore mistakes as we can. Computers cannot distinguish what is or is not supposed to be there. For example, if you create a table and call it `mytable`, but then tried to do some calculations on `my table`, and error will appear, such as `Error: unexpected symbol in "my table"`. This error means the computer cannot find `my table`, only because it does not EXIST.

1.4 R resources



```

1  ## load up the packages we will need
2
3  if (!require("pacman")) install.packages("pacman")
4  pacman::p_load(tidyverse, # All purpose wrangling for dataframes
5                  lubridate, # Time
6                  tibbletime,
7                  openxlsx) # writing excel documents
8
9
10 ## Import data (Task 1)
11
12 ### Import the sheet with the vo2 data
13 dat_vo2 <- read.xlsx (xlsxFile = "data/xxx.xlsx",
14                      sheet = "xxx")
15
16 ### Import the sheet with the lactate data
17 dat_stage <- read.xlsx (xlsxFile = "data/xx.xlsx",
18                        sheet = "xxx")
19
20 ## Analyze VO2 data -----
21
22 ### Rename column names of vo2 data (Task 2)
23
24 new_names <- c()
25
26 colnames (dat_vo2) <- new_names
27
28 ### Remove first row of vo2 data
29
30 dat_vo2 <- dat_vo2 %>%
31   slice (-c(1))
32

```

Figure 1.4: If you restart your analysis, the software does not have a memory of where you stopped previously .

1.4.1 R books

There are many, many excellent (non-pirate) books on R, some of which are available online for free. Here are some that I highly recommend:

Book	Description
R for Data Science by Garrett Golemund and Hadley Wickham	The best book to learn the latest tools for elegantly doing data science.
R Graphics Cookbook by Winston Chang	Is indispensable for creating graphics.
R Cookbook by James (JD) Long and Paul Teetor	Is a useful bag of tips and tricks to get started with R .

Chapter 2

Getting Started

2.1 Installing Base-R and RStudio

To use R, we'll need to download 2 software: **Base-R**, and **RStudio**. Base-R is the basic software which contains the R programming language. RStudio is a software that makes programming easier. In everyday parlance, R is the engine and RStudio is the car's frame. Just like you can transfer an engine to different car frames, you can use R using other platforms. But I will use RStudio. Of course, they are totally free and open source.

2.1.1 Check for version updates

R and RStudio have been around for several years – however, they are *constantly* being updated with new features and bug-fixes. At the time that I am writing this sentence (“2021-10-11”), the latest version of Base-R is 4.1.1, and the latest version of RStudio is 1.4.1717.

To install Base-R, click on one of the following links and follow the instructions.

Operating System	Link
Windows	http://cran.r-project.org/bin/windows/base/
Mac	http://cran.r-project.org/bin/macosx/

Once you've installed base-R on your computer, try opening it. When you do you should see a screen like the one in Figure 2.2 (this is the Windows version). As you can see, base R is very much bare-bones software. It's kind of the equivalent of a simple text editor that comes with your computer.



Figure 2.1: R logo

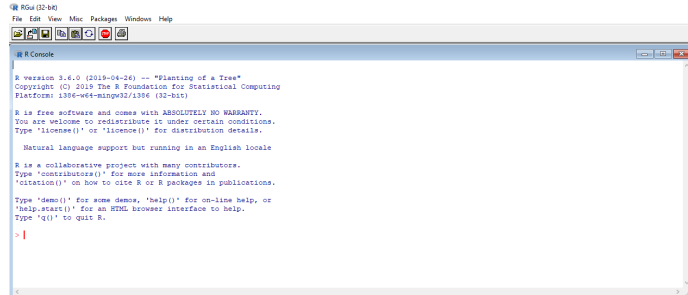


Figure 2.2: Here is how the base R application looks. While you can use the base R application alone, most people I know use RStudio – software that helps you to write and use R code more efficiently!



Figure 2.3: RStudio logo

While you can do pretty much everything you want within base-R, you'll find that most people these days do their R programming in an application called RStudio. RStudio is a graphical user interface (GUI)-like interface for R that makes programming in R a bit easier. In fact, once you've installed RStudio, you'll likely never need to open the base R application again.

To download and install RStudio (around 40mb), click on the link below:
<https://rstudio.com/products/rstudio/download/#download/>

2.2 The four RStudio Windows

Let's go ahead and boot up RStudio and see how she looks! When you open RStudio, you'll see the following four windows (also called panes) shown in in Figure 2.4. However, your windows might be in a different order than those in Figure 2.4. Ignore for a fact that my screen shows black and yours shows up as white. This is like the wall paper on your computer, it is what makes my eyes at ease.

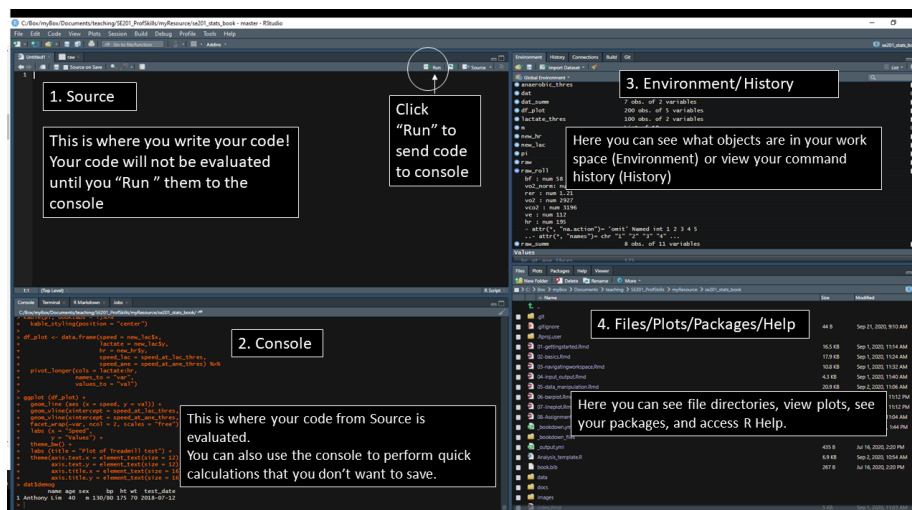


Figure 2.4: The four panes of RStudio.

Now, let's see what each window does in detail.

2.2.1 Source - Your notepad for code

The source pane is where you create and edit "R Scripts" - your collections of code. Don't worry, R scripts are just text files with the ".R" extension. When you

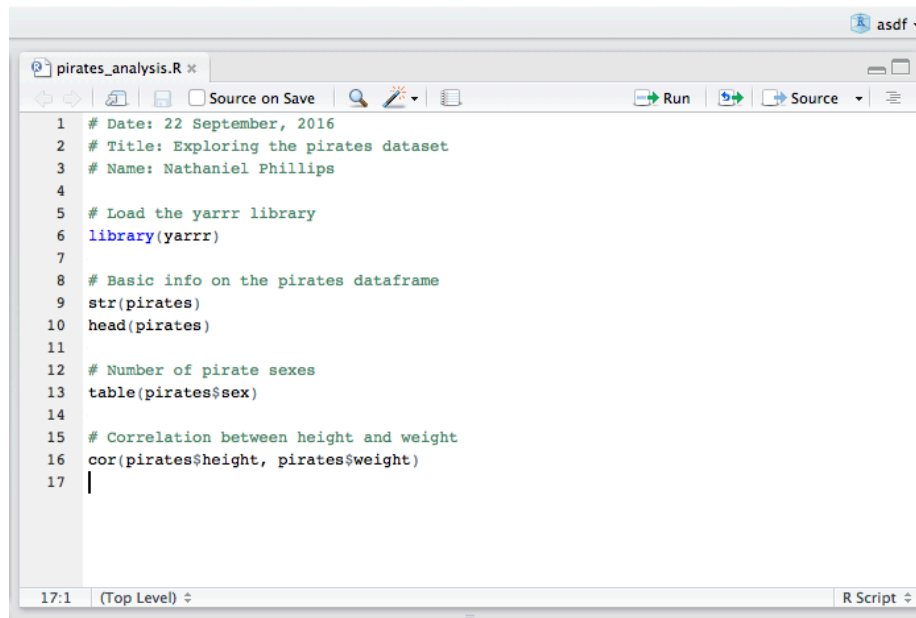


Figure 2.5: The Source contains all of your individual R scripts. The code won't be evaluated until you send it to the Console.

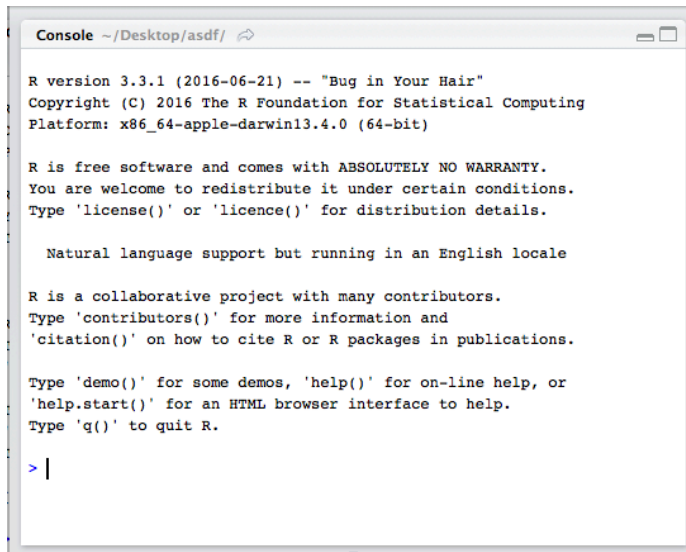
open RStudio, it will automatically start a new Untitled script. Before you start typing in an untitled R script, you should always save the file under a new file name (like, "StatsAnal.R"). That way, if something on your computer crashes while you're working, R will have your code waiting for you when you re-open RStudio.

You'll notice that when you're typing code in a script in the Source panel, R won't actually evaluate the code as you type. To have R actually evaluate your code, you need to first 'send' the code to the Console (we'll talk about this in the next section).

There are many ways to send your code from the Source to the console. The slowest way is to copy and paste. A faster way is to highlight the code you wish to evaluate and clicking on the "Run" button on the top right of the Source. Alternatively, you can use the hot-key "Command + Return" on Mac, or "Control + Enter" on PC to send all highlighted code to the console.

2.2.2 Console: R's Heart

The console is the heart of R. Here is where R actually evaluates code. At the beginning of the console you'll see the character `>`. This is a prompt that tells

A screenshot of the R console window. The title bar reads "Console ~/Desktop/asdf/". The text inside the console is the standard R startup message, including the version (3.3.1), copyright (2016), platform (x86_64-apple-darwin13.4.0), and various help prompts like 'license()', 'demo()', and 'q()'. The prompt ">|" is visible at the bottom.

```
Console ~/Desktop/asdf/

R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (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.

> |
```

Figure 2.6: The console the calculation heart of R. All of your code will (eventually) go through here.

you that R is ready for new code. You can type code directly into the console after the `>` prompt and get an immediate response. For example, if you type `1+1` into the console and press enter, you'll see that R immediately gives an output of 2.

```
1+1
```

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

Try calculating `1+1` by typing the code directly into the console - then press Enter. You should see the result `[1] 2`. Don't worry about the `[1]` for now, we'll get to that later. For now, we're happy if we just see the 2. Then, type the same code into the Source, and then send the code to the Console by highlighting the code and clicking the "Run" button on the top right hand corner of the Source window. Alternatively, you can use the hot-key "Command + Return" on Mac or "Control + Enter" on Windows.

Tip: Try to write most of your code in a document in the Source. Only type directly into the Console to do quick calculations like you are using a calculator.

So as you can see, you can execute code either by running it from the Source or by typing it directly into the Console. However, 99% most of the time, you should be using the Source rather than the Console. The reason for this is straightforward: If you type code into the console, it won't be saved (though

you can look back on your command History). And if you make a mistake in typing code into the console, you'd have to re-type everything all over again. Instead, it's better to write all your code in the Source. When you are ready to execute some code, you can then send "Run" it to the console.

2.2.3 Environment / History

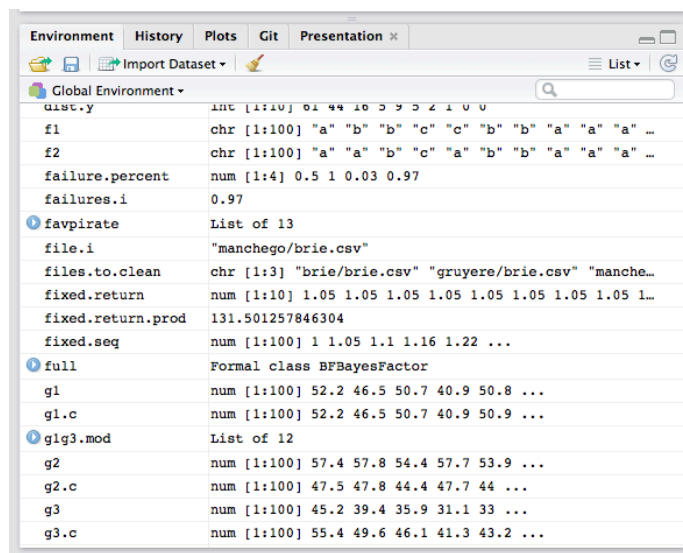


Figure 2.7: The environment panel shows you all the objects you have defined in your current workspace.

The Environment tab of this panel shows you the names of all the data objects (like vectors, matrices, and dataframes) that you've defined in your current R session. You can also see information like the number of observations and rows in data objects. The tab also has a few clickable actions like "Import Dataset" which will open a graphical user interface (GUI) for importing data into R.

The History tab of this panel simply shows you a history of all the code you've previously evaluated in the Console.

As you get more comfortable with R, you might find the Environment / History panel useful. But for now you can just ignore it. If you want to declutter your screen, you can even just minimize the window by clicking the minimize button on the top right of the panel.

2.2.4 Files / Plots / Packages / Help

The Files / Plots / Packages / Help panel shows you lots of helpful information. Let's go through each tab in detail:

1. Files - The files panel gives you access to the file directory on your hard drive. One nice feature of the "Files" panel is that you can use it to set your working directory. We'll talk about working directories in more detail soon.
2. Plots - The Plots panel (no big surprise), shows all your plots. There are buttons for opening the plot in a separate window and exporting the plot as a pdf or jpeg (though you can also do this with code using the `pdf()` or `jpeg()` functions.)

Let's see how plots are displayed in the Plots panel. Run the code on the right to display a histogram of the weights of chickens stored in the `ChickWeight` dataset. When you do, you should see a plot similar to the one in Figure 2.8 show up in the Plots panel.

```
hist(x = ChickWeight$weight,  
     main = "Chicken Weights",  
     xlab = "Weight",  
     col = "skyblue",  
     border = "white")
```

3. Packages - Shows a list of all the R packages installed on your harddrive and indicates whether or not they are currently loaded. Packages that are loaded in the current session are checked while those that are installed but not yet loaded are unchecked. We'll discuss packages in more detail in the next section.
4. Help - Help menu for R functions. You can either type the name of a function in the search window, or use the code `?function.name` to search for a function with the name `function.name`

```
?hist # How does the histogram function work?  
?t.test # What about a t-test?
```

2.3 Reading and writing Code

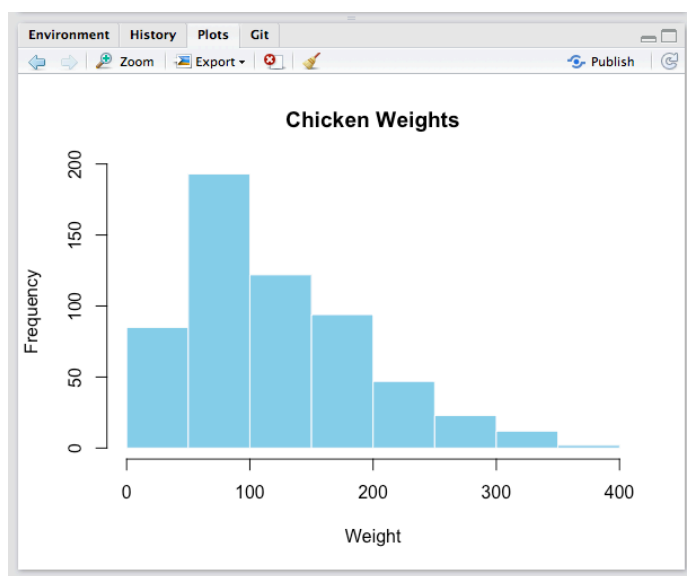


Figure 2.8: The plot panel contains all of your plots, like this histogram of the distribution of chicken weights.

2.3.1 Code Chunks

In this book, R code is (almost) always presented in a separate gray box like this one:

```
# A code chunk

# Define a vector a as the integers from 1 to 5
a <- 1:5

# Print a
a

## [1] 1 2 3 4 5

# What is the mean of a?
mean(a)

## [1] 3
```

This is called a *code chunk*. You should always be able to copy and paste code chunks directly into R. If you copy a chunk and it does not work for you, it is most

likely because the code refers to a package, function, or object that I defined in a previous chunk. If so, read back and look for a previous chunk that contains the missing definition.

2.3.2 Comments with

Lines that begin with `#` are comments. If you evaluate any code that starts with `#`, R will just ignore that line. In this book, comments will be either be literal comments that I write directly to explain code, or they will be *output* generated automatically from R. For example, in the code chunk below, you see lines starting with `##`. These are the output from the previous line(s) of code. When you run the code yourself, you should see the same output in your *console*.

```
# This is a comment I wrote
```

```
1 + 2
```

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

```
# The line above (## [1] 3) is the output from the previous code that has been 'commented out'
```

2.3.3 Element numbers in output [1]

The output you see will often start with one or more number(s) in brackets such as `[1]`. This is just a visual way of telling you where the numbers occur in the output. For example, in the code below, I will print a long vector containing the multiples of 2 from 0 to 100:

```
seq(from = 0, to = 100, by = 2)
```

```
## [1] 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36  
## [20] 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74  
## [39] 76 78 80 82 84 86 88 90 92 94 96 98 100
```

As you can see, the first line of the output starts with `## [1]`, and the next two lines start with `[18]` and `[35]`. This is just telling you that 0 is the `[1]`st element, 34 is the `[18]`th element, and 68 is the `[35]`th element. Sometimes this information will be helpful, but most of the time you can just ignore it.

2.4 Debugging

When you are programming, you will always, and I do mean always, make errors (also called bugs) in your code. You might misspell a function, include an extra comma, or some days...R just won't want to work with you (again, see section Why R is like a Relationship).

Debugging will always be a challenge. However, over time you'll learn which bugs are the most common and get faster and faster at finding and correcting them.

Here are the most common bugs you'll run into as you start your R journey.

2.4.1 R is not ready (>)

Another very common problem occurs when R does not seem to be responding to your code. That is, you might run some code like `mean(x)` expecting some output, but instead, nothing happens. This can be very frustrating because, rather than getting an error, just nothing happens at all. The most common reason for this is because R isn't *ready* for new code, instead, it is *waiting* for you to finish code you started earlier, but never properly finished.

Think about it this way, R can be in one of two states: it is either **Ready** (>) for new code, or it is **Waiting** (+) for you to finish old code. To see which state R is in, all you have to do is look at the symbol on the console. The > symbol means that R is Ready for new code – this is usually what you want to see. The + symbol means that R is Waiting for you to (properly) finish code you started before. If you see the + symbol, then no matter how much new code you write, R won't actually evaluate it until you finish the code you started before.

Thankfully there is an easy solution to this problem (See Figure 2.9): Just hit the escape key on your keyboard. This will cancel R's waiting state and make it Ready!

2.4.2 Misspelled object or function

If you spell an object or function incorrectly, you'll receive an error like `Error: could not find function` **OR** `Error: object 'x' not found`.

In the code below, I'll try to take the mean of a vector `data`, but I will misspell the function `mean()`

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

# Misspelled function: should be mean(x), not meeen(x)
meeen(data)
```

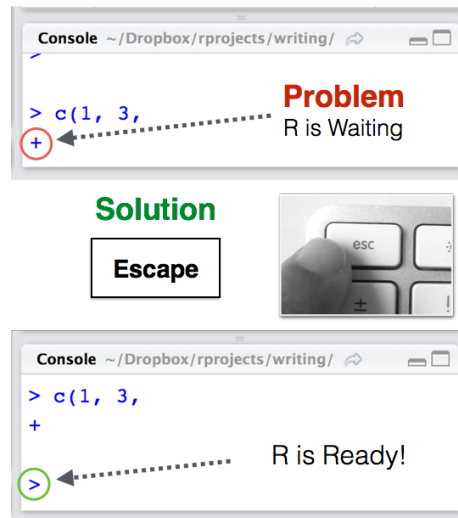



Figure 2.9: To turn R from a Waiting (+) state to a Ready (>) state, just hit Escape.

Error: could not find function “meeen”

Now, I’ll misspell the object `data` as `dta`:

```
# Misspelled object: should be data, not dta
mean(dta)
```

Error: object ‘dta’ not found

R is case-sensitive, so if you don’t use the correct capitalization you’ll receive an error. In the code below, I’ll use `Mean()` instead of the correct version `mean()`

```
# Capitalization is wrong: should be mean(), not Mean()
Mean(data)
```

Error: could not find function “Mean”

Here is the correct version where both the object `data` and function `mean()` are correctly spelled:

```
# Correct: both the object and function are correctly spelled
mean(data)
```

```
## [1] 2.2
```

2.4.3 Punctuation problems

Another common error is having bad coding “punctuation”. By that, I mean having an extra space, missing a comma, or using a comma (,) instead of a period (.). In the code below, I’ll try to create a vector using periods instead of commas:

```
# Wrong: Using periods (.) instead of commas (,)
mean(c(1. 4. 2))
```

Error: unexpected numeric constant in “mean(c(1. 4.”

Because I used periods instead of commas, I get the above error. Here is the correct version

```
# Correct
mean(c(1, 4, 2))
```

```
## [1] 2.333333
```

If you include an extra space in the middle of the name of an object or function, you’ll receive an error. In the code below, I’ll accidentally write `Chick Weight` instead of `ChickWeight`:

```
# Wrong: Extra space in the ChickWeight object name
head(Chick Weight)
```

Error: unexpected symbol in “head(Chick Weight”

Because I had an extra space in the object name, I get the above error. Here is the correction:

```
# Correct:
head(ChickWeight)
```

2.5 Learning check

1. Download Base-R software and RStudio software.
2. Open up RStudio software and type the following code below into the **console**. What does it give you?

```
1+10
```

3. Look at the code below. What will R return after the third line? Make a prediction, then test the code yourself.

```
a <- 10  
a + 10  
a
```


Chapter 3

The Basics

Download and load packages

Packages are like your iPhone apps. The iPhone comes with some basic functionality, e.g. weather-app. If you wanted more, you have to download. Subsequent chapters are going to start with this code chunk. This is only needed if you are running one chapter independent from others. Notice how I am using the package called `pacman`. This is a package manager, which loads any package you typed into it, and if it is not available, download it automatically from CRAN and load it.

```
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse, # All purpose wrangling for dataframes
               yarr)
```

In this chapter, we'll go over the basics of the R language and the RStudio programming environment.

3.1 The command-line (Console)

R code, on its own, is just text. You can write R code in a new script within R or RStudio, or in any text editor. However, just writing the code won't do the whole job – in order for your code to be executed (aka, interpreted) you need to send it to the Console.

In R, the command-line interpreter starts with the `>` symbol. This is called the **prompt**. Why is it called the prompt? Well, it's "prompting" you to feed it with some R code. The fastest way to have R evaluate code is to type your R code



Figure 3.1: Yep. R is really just a fancy calculator.

This is the **console**

```
Console ~/Dropbox/manuscripts/FFTrees_man/ ↵

R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (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.

[Workspace loaded from ~/Dropbox/manuscripts/FFTrees_man/]

> 1+1
[1] 2
> |
```

Here is the
command-line.
You can type here to
get an immediate
response

Figure 3.2: You can always type code directly into the command line to get an immediate response.

directly into the command-line interpreter. For example, if you type `1+1` into the interpreter and hit enter you'll see the following

```
1+1
```

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

As you can see, R returned the (thankfully correct) value of 2. You'll notice that the console also returns the text `[1]`. This is just telling you the index of the value next to it. Don't worry about this for now, it will make more sense later. As you can see, R can, thankfully, do basic calculations. In fact, at its heart, R is technically just a fancy calculator. But that's like saying Michael Jordan is *just* a fancy ball bouncer. It (and they), are much more than that.

3.2 Writing R scripts in an editor

There are certainly many cases where it makes sense to type code directly into the console. For example, to open a help menu for a new function with the `?` command, to take a quick look at a dataset with the `head()` function, or to do simple calculations like `1+1`, you should type directly into the console. However, the problem with writing all your code in the console is that nothing that you write will be saved. So if you make an error, or want to make a change to some earlier code, you have to type it all over again. Not very efficient. For this (and many more reasons), you'll should write any important code that you want to save as an R script. An R script is just a bunch of R code in a single file. You can write an R script in any text editor, but you should save it with the `.R` suffix to make it clear that it contains R code.} in an editor.

In RStudio, you'll write your R code in the *Source* window. To start writing a new R script in RStudio, **click File – New File – R Script**.

When you open a new script, you'll see a blank page waiting for you to write as much R code as you'd like. In Figure 3.3, I have a new script called `examplescript` with a few random calculations.

You can have several R scripts open in the source window in separate tabs (like I have above).

3.2.1 Send code from a source to the console

When you type code into an R script, you'll notice that, unlike typing code into the Console, nothing happens. In order for R to interpret the code, you need to send it from the Editor to the Console. There are a few ways to do this, but the most common way I use is:

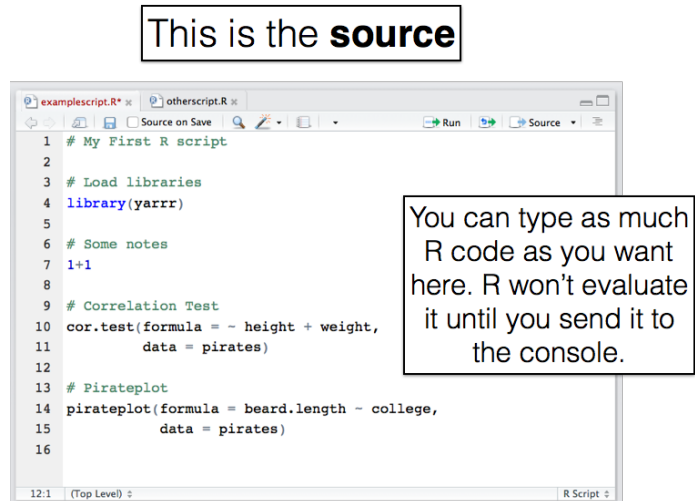


Figure 3.3: Here's how a new script looks in the editor window on RStudio. The code you type won't be executed until you send it to the console.

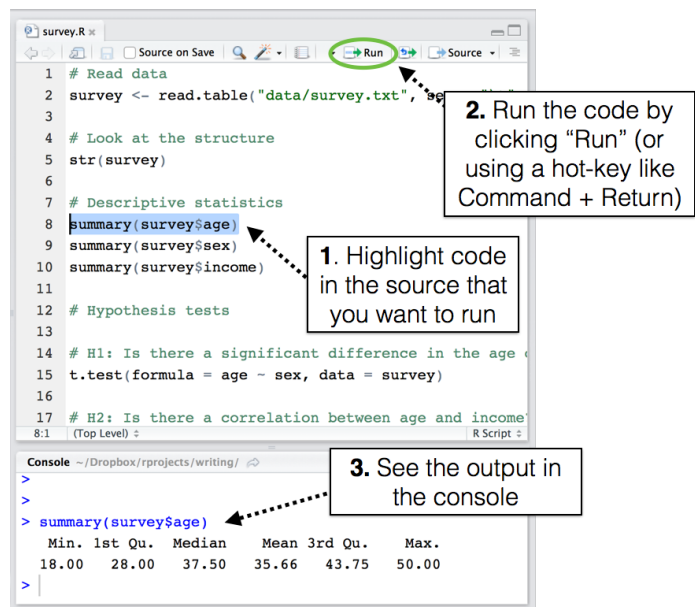


Figure 3.4: To evaluate code from the source, highlight it and run it.

1. Highlight the code you want to run (with your mouse or by holding Shift), then use the Alt+Enter shortcut.

3.3 A brief style guide: Commenting and spacing

Like all programming languages, R isn't just meant to be read by a computer, it's also meant to be read by other humans. For this reason, it's important that your code looks nice and is understandable to other people and your future self. To keep things brief, I won't provide a complete style guide – instead I'll focus on the two most critical aspects of good style: commenting and spacing.



Figure 3.5: As Stan discovered in season six of South Park, your future self is a lazy, possibly intoxicated moron. So do your future self a favor and make your code look nice. Also maybe go for a run once in a while.

3.3.1 Commenting code with the # (pound) sign

Comments are completely ignored by R and are just there for whomever is reading the code. You can use comments to explain what a certain line of code is doing, or just to visually separate meaningful chunks of code from each other. Comments in R are designated by a # (pound) sign. Whenever R encounters a # sign, it will ignore **all** the code after the # sign on that line. Additionally, in most coding editors (like RStudio) the editor will display comments in a separate color than standard R code to remind you that it's a comment:

Here is an example of a short script that is nicely commented. Try to make your scripts look like this!

```
# Author: Pirate Jack
# Title: My nicely commented R Script
# Date: None today :(
```

```
# Step 1: Load the yarr package
library(yarr)

# Step 2: See the column names in the movies dataset
names(movies)

# Step 3: Calculations

# What percent of movies are sequels?
mean(movies$sequel, na.rm = T)

# How much did Pirate's of the Caribbean: On Stranger Tides make?
movies$revenue.all[movies$name == 'Pirates of the Caribbean: On Stranger Tides']
```

I cannot stress enough how important it is to comment your code! Trust me, even if you don't plan on sharing your code with anyone else, keep in mind that your future self will be reading it in the future.

3.3.2 Spacing

How would you like to read a book if there were no spaces between words? I'm guessing you wouldn't. So every time you write code without proper spacing, remember this sentence.

Commenting isn't the only way to make your code legible. It's important to make appropriate use of spaces and line breaks. For example, I include spaces between arithmetic operators (like =, + and -) and after commas (which we'll get to later). For example, look at the following code:

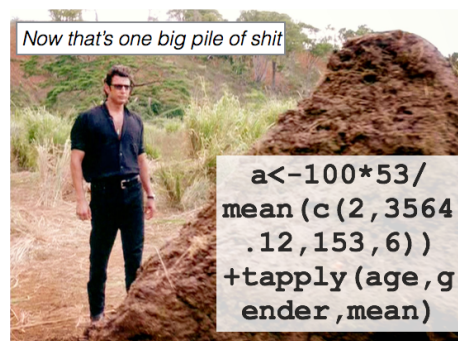


Figure 3.6: Don't make your code look like what a sick Triceratops with diarrhea left behind for Jeff Goldblum.

```
# Shitty looking code
a<-(100+3)-2
mean(c(a/100,642564624.34))
t.test(formula=revenue.all~sequel,data=movies)
plot(x=movies$budget,y=movies$dvd.usa,main="myplot")
```

That code looks like shit. Don't write code like above. It will make your eyes hurt. Now, let's use some liberal amounts of commenting and spacing to make it look less shitty.

```
# Some meaningless calculations. Not important

a <- (100 + 3) - 2
mean(c(a / 100, 642564624.34))

# t.test comparing revenue of sequels v non-sequels

t.test(formula = revenue.all ~ sequel,
       data = movies)

# A scatterplot of budget and dvd revenue.
# Hard to see a relationship

plot(x = movies$budget,
     y = movies$dvd.usa,
     main = "myplot")
```

See how much better that second chunk of code looks? Not only do the comments tell us the purpose behind the code, but there are spaces and line-breaks separating distinct elements.

3.4 Objects and functions

To understand how R works, you need to know that R revolves around two things: objects and functions. Almost everything in R is either an object or a function. In the following code chunk, I'll define a simple object called `tattoos` using a function `c()`:

```
# 1: Create a vector object called tattoos
tattoos <- c(4, 67, 23, 4, 10, 35)

# 2: Apply the mean() function to the tattoos object
mean(tattoos)
```

```
## [1] 23.83333
```

What is an object? An object is a thing – like a number, a dataset, a summary statistic like a mean or standard deviation, or a statistical test. Objects come in many different shapes and sizes in R. There are simple objects like the single digit `25` which represent single numbers, **vectors** (like our `tattoos` object above) which represent several numbers, more complex objects like **dataframes** which represent tables of data, and even more complex objects like **hypothesis tests** or **regression** which contain all sorts of statistical information.

What is a function? A function is a *procedure* that typically takes one or more objects as arguments (aka, inputs), does something with those objects, then returns a new object. For example, the `mean()` function we used above takes a vector object, like `tattoos`, of numeric data as an argument, calculates the arithmetic mean of those data, then returns a single number (a scalar) as a result. A great thing about R is that you can easily create your own functions that do whatever you want – but we will not get to that in the book. Thankfully, R has hundreds (thousands?) of built-in functions that perform most of the basic analysis tasks you can think of.

99% of the time you are using R, you will do the following: 1) Define objects. 2) Apply functions to those objects. 3) Repeat!. Seriously, that's about it. However, as you'll soon learn, the hard part is knowing how to define objects they way you want them, and knowing which function(s) will accomplish the task you want for your objects.

3.4.1 Numbers versus characters

For the most part, objects in R come in one of two flavors: **numeric** and **character**. It is very important to keep these two separate as certain functions, like `mean()`, and `max()` will only work for numeric objects, while functions like `grep()` and `strtrim()` only work for character objects.

A numeric object is just a number like `1`, `10` or `3.14`. You don't have to do anything special to create a numeric object, just type it like you were using a calculator.

```
# These are all numeric objects
1
10
3.14
```

A **character** object is a name like `"Madisen"`, `"Brian"`, or `"University of Konstanz"`. To specify a character object, you need to include quotation marks `"` around the text.

```
# These are all character objects
"Madisen"
"Brian"
"10"
```

If you try to perform a function or operation meant for a numeric object on a character object (and vice-versa), R will yell at you. For example, here's what happens when I try to take the mean of the two character objects "1" and "10":

```
# This will return an error because the arguments are not numeric!
mean(c("1", "10"))
```

Warning message: argument is not numeric or logical, returning NA

If I make sure that the arguments are numeric (by not including the quotation marks), I won't receive the error:

```
# This is ok!
mean(c(1, 10))
```

```
## [1] 5.5
```

3.4.2 Creating new objects with <-

By now you know that you can use R to do simple calculations. But to really take advantage of R, you need to know how to create and manipulate objects. All of the data, analyses, and even plots, you use and create are, or can be, saved as objects in R.

To create new objects in R, you need to do *object assignment*. Object assignment is our way of storing information, such as a number or a statistical test, into something we can easily refer to later. This is a pretty big deal. Object assignment allows us to store data objects under relevant names which we can then use to slice and dice specific data objects anytime we'd like to.

To do an assignment, we use the almighty <- operator called *assign*. To assign something to a new object (or to change an existing object), use the notation `object <- ...`, where `object` is the new (or updated) object, and `...` is whatever you want to store in `object`. Let's start by creating a very simple object called `a` and assigning the value of 100 to it:

Good object names strike a balance between being easy to type (i.e.; short names) and interpret. If you have several datasets, it's probably not a good idea to name them `a`, `b`, `c` because you'll forget which is which. However, using long names like `March2015Group1OnlyFemales` will give you carpal tunnel syndrome.

```
# Create a new object called a with a value of 100
a <- 100
```

Once you run this code, you'll notice that R doesn't tell you anything. However, as long as you didn't type something wrong, R should now have a new object called `a` which contains the number 100. If you want to see the value, you need to call the object by just executing its name. This will print the value of the object to the console:

```
# Print the object a
a
```

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

Now, R will print the value of `a` (in this case 100) to the console. If you try to evaluate an object that is not yet defined, R will return an error. For example, let's try to print the object `b` which we haven't yet defined:

```
b
```

Error: object 'b' not found

As you can see, R yelled at us because the object `b` hasn't been defined yet.

Once you've defined an object, you can combine it with other objects using basic arithmetic. Let's create objects `a` and `b` and play around with them.

```
a <- 1
b <- 100

# What is a + b?
a + b
```

```
## [1] 101
```

```
# Assign a + b to a new object (c)
c <- a + b

# What is c?
c
```

```
## [1] 101
```

3.4.2.1 To change an object, you must assign it again!

Normally I try to avoid excessive emphasis, but because this next sentence is so important, I have to just go for it. Here it goes...

To change an object, you *must* assign it again!

No matter what you do with an object, if you don't assign it again, it won't change. For example, let's say you have an object `z` with a value of 0. You'd like to add 1 to `z` in order to make it 1. To do this, you might want to just enter `z + 1` – but that won't do the job. Here's what happens if you **don't** assign it again:

```
z <- 0
z + 1

## [1] 1
```

Ok! Now let's see the value of `z`

```
z

## [1] 0
```

Damn! As you can see, the value of `z` is still 0! What went wrong? Oh yeah...

To change an object, you *must* assign it again!

The problem is that when we wrote `z + 1` on the second line, R thought we just wanted it to calculate and print the value of `z + 1`, without storing the result as a new `z` object. If we want to actually update the value of `z`, we need to reassign the result back to `z` as follows:

```
z <- 0
z <- z + 1 # Now I'm REALLY changing z
z

## [1] 1
```

Phew, `z` is now 1. Because we used assignment, `z` has been updated. About freaking time.

3.4.3 How to name objects

You can create object names using any combination of letters and a few special characters (like `.` and `_`). Here are some valid object names

```
# Valid object names
group.mean <- 10.21
my.age <- 32
FavoritePirate <- "Jack Sparrow"
sum.1.to.5 <- 1 + 2 + 3 + 4 + 5
```

All the object names above are perfectly valid. Now, let's look at some examples of *invalid* object names. These object names are all invalid because they either contain spaces, start with numbers, or have invalid characters:

```
# Invalid object names!
famale ages <- 50 # spaces
5experiment <- 50 # starts with a number
a! <- 50 # has an invalid character
```

If you try running the code above in R, you will receive a warning message starting with

Error: unexpected symbol

. Anytime you see this warning in R, it almost always means that you have a naming error of some kind.

3.4.3.1 R is case-sensitive!

Like English, R is case-sensitive – it R treats capital letters differently from lower-case letters. For example, the four following objects `Plunder`, `plunder` and `PLUNDER` are totally different objects in R:

```
# These are all different objects
Plunder <- 1
plunder <- 100
PLUNDER <- 5
```

I try to avoid using too many capital letters in object names because they require me to hold the shift key. This may sound silly, but you'd be surprised how much easier it is to type `mydata` than `MyData` 100 times.



Figure 3.7: Like a text message, you should probably watch your use of capitalization in R.

Chapter 4

Navigating the Software

Introduction

Both R and RStudio are big chunks of software, first and foremost. You will inevitably spend time doing what one does with any big piece of software: configuring it, customizing it, updating it, and fitting it into your computing environment. This chapter will help you perform those tasks. There is nothing here about numerics, statistics, or graphics. This is all about dealing with R and RStudio as software.

Download and load packages

```
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse) # All purpose wrangling for dataframes
```

4.1 Getting and Setting the Working Directory

Your computer is a maze of folders and files. Outside of R, when you want to open a specific file, you probably open up an explorer window that allows you to visually search through the folders on your computer. Or, maybe you select recent files, or type the name of the file in a search box to let your computer do the searching for you. While this system usually works for non-programming tasks, it is a no-go for R. Why? Well, the main problem is that all of these methods require you to *visually* scan your folders and move your mouse to



Figure 4.1: Your workspace – all the objects, functions, and delicious glue you’ve defined in your current session.

select folders and files that match what you are looking for. When you are programming in R, you need to specify *all* steps in your analyses in a way that can be easily replicated by others and your future self. This means you can’t just say: “Find this one file I emailed to myself a week ago” or “Look for a file that looks something like `experimentAversion3.txt`.” Instead, you need to be able to write R code that tells R *exactly* where to find critical files – either on your computer or on the web.

To make this job easier, R uses **working directories**. A working directory is where everything starts and ends. Your working directory is important because it is the default location for all file input and output—including reading and writing data files, opening and saving script files, and saving your workspace image. Many of you who previously worked with SPSS using the point-click interface, would often wonder 1) where did my saved SPSS file went? or 2) where did my exported image go to? This confusion arises because often you do not know what was the default working directory. Rather than relying on the default, we specify it explicitly so we know where to store our files, and where this software goes looking for files.

The easiest and recommended way to set your working directory is using RStudio projects. For every piece of work/assessment, you create a project. The project is a folder that can live anywhere on your computer - your desktop, downloads folder, documents folder, etc. In this folder, it contains everything from your files to be analyzed, codes, and exported files and images. Everything is self-contained, there is no confusion.

4.2 Creating a new Rstudio project

You want to create a new RStudio project to keep all your files related to a specific project. Click File → New Project as in Figure 4.2. **** I ALWAYS use this approach, please use it too****

This will open the New Project dialog box and allow you to choose which type of project you would like to create, as shown in Figure 4.3.

Projects are a powerful concept that's specific to RStudio. They help you by doing the following:

- Setting your working directory to the project directory.
- Preserving window state in RStudio so when you return to a project your windows are all as you left them. This includes opening any files you had open when you last saved your project.
- Preserving RStudio project settings.

To hold your project settings, RStudio creates a project file with an *.Rproj* extension in the project directory. If you open the project file in RStudio, it works like a shortcut for opening the project. In addition, RStudio creates a hidden directory named *.Rproj.user* to house temporary files related to your project.

Any time you're working on something nontrivial in R we recommend creating an RStudio project. Projects help you stay organized and make your project workflow easier.

4.3 Installing Packages

When you download and install R for the first time, you are installing the Base R software. Base R will contain most of the functions you'll use on a daily basis like `mean()` and `hist()`. However, only functions written by the original authors of the R language will appear here. If you want to access data and code written by other people, you'll need to install it as a *package*. An R package is simply a bunch of data, from functions, to help menus, to vignettes (examples), stored in one neat package.

A package is like a light bulb. In order to use it, you first need to order it to your house (i.e.; your computer) by *installing* it. Once you've installed a package, you never need to install it again. However, every time you want to actually use the package, you need to turn it on by *loading* it. Here's how to do it.

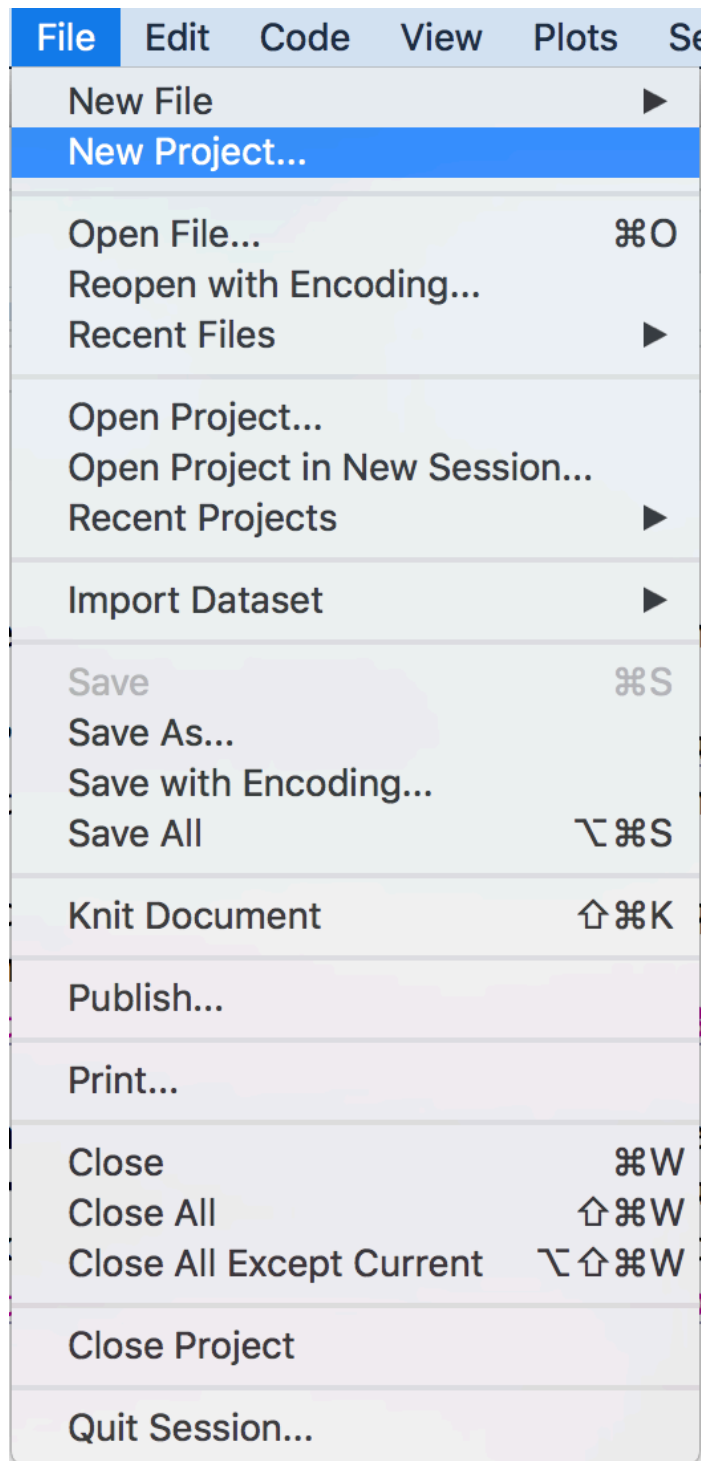


Figure 4.2: Selecting New Project

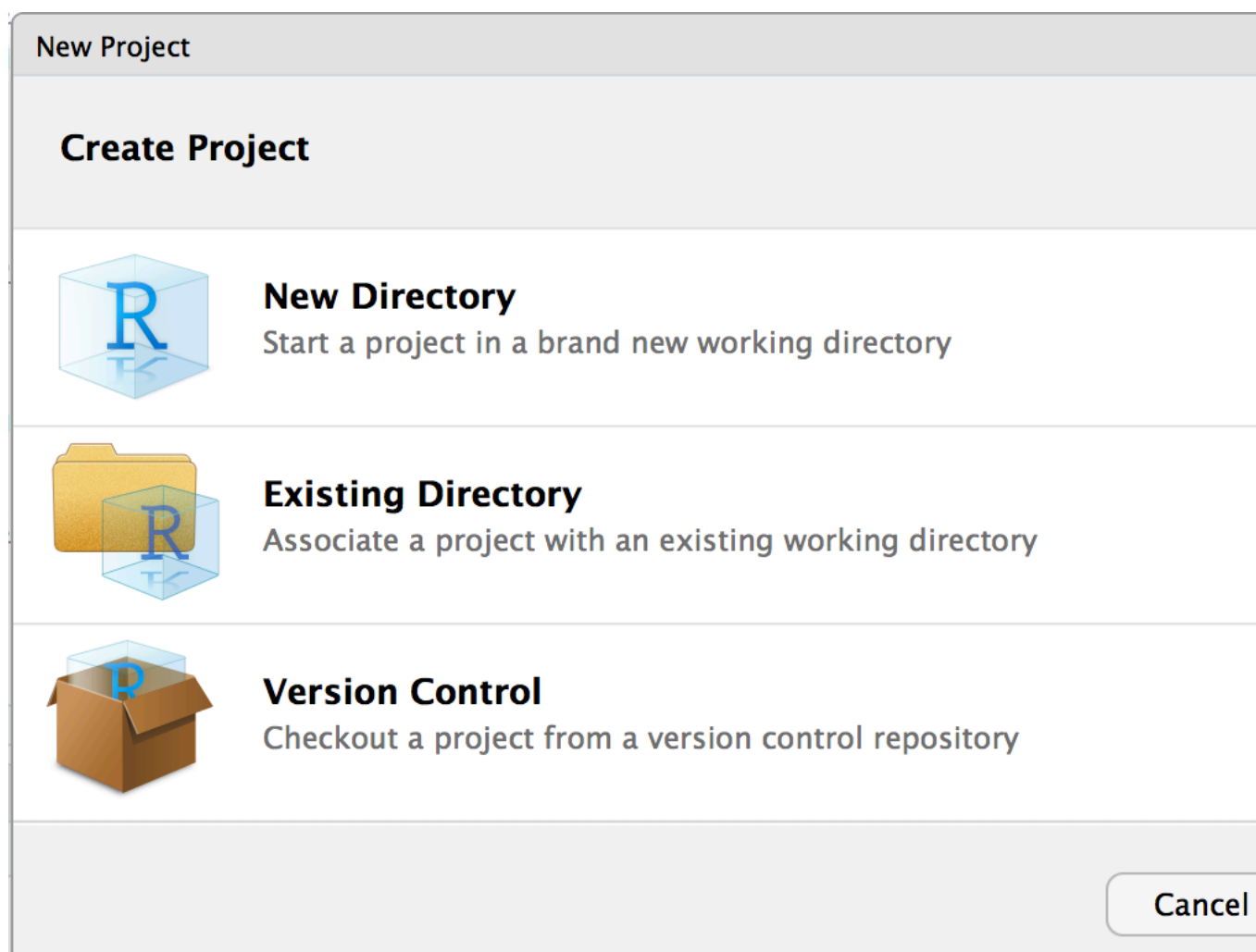


Figure 4.3: New Project dialog

Installing a package
`install.packages('my.package')`



Loading a package
`library('mypackage')`

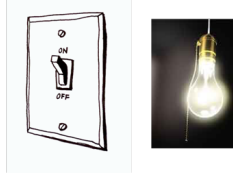


Figure 4.4: An R package is like a lightbulb. First you need to order it with `install.packages()`. Then, every time you want to use it, you need to turn it on with `library()`

4.3.1 Installing a new package

Installing a package simply means downloading the package code onto your personal computer. There are two main ways to install new packages. The first, and most common, method is to download them from the Comprehensive R Archive Network (CRAN). CRAN is the central repository for R packages. To install a new R package from CRAN, you can simply run the code `install.packages("name")`, where “name” is the name of the package. For example, to download the `tidyverse` package, which contains several functions we will use in this book, you should run the following:



Figure 4.5: CRAN (Comprehensive R Archive Network) is the main source of R packages


```
# Install the tidyverse package from CRAN
# You only need to install a package once!
install.packages("tidyverse")
```

When you run `install.packages("name")` R will download the package from CRAN. If everything works, you should see some information about where the package is being downloaded from, in addition to a progress bar.

```
> install.packages("circlize")
trying URL 'https://cran.rstudio.com/bin/macosx/mavericks/contrib/3.3/circlize_0.3.8.tgz'
Content type 'application/x-gzip' length 3856952 bytes (3.7 MB)
=====
downloaded 3.7 MB

The downloaded binary packages are in
/var/folders/gy/bsyxftvn37q93cm6vtiv56fr0000gp/T//RtmpOzg22R/downloaded_packages
>
```

Figure 4.6: When you install a new package, you'll see some random text like this you the download progress. You don't need to memorize this.

Like ordering a light bulb, once you've installed a package on your computer you never need to install it again (unless you want to try to install a new version of the package). However, every time you want to use it, you need to turn it on by loading it.

4.3.2 Loading a package

Once you've installed a package, it's on your computer. However, just because it's on your computer doesn't mean R is ready to use it. If you want to use something, like a function or dataset, from a package you *always* need to *load* the package in your R session first. Just like a light bulb, you need to turn it on to use it!

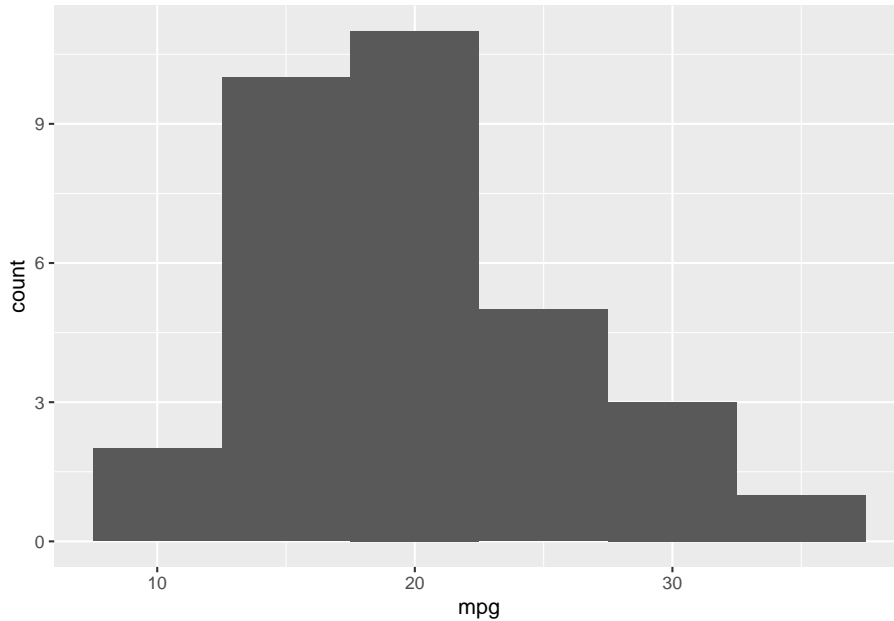
To load a package, you use the `library()` function. For example, now that we've installed the `tidyverse` package, we can load it with `library("tidyverse")`:

```
# Load the tidyverse package so I can use it!
# You have to load a package in every new R session!
library("tidyverse")
```

Now that you've loaded the `tidyverse` package, you can use any of its functions! Let us create a very simple histogram plot using a default dataset found within R. Don't worry about the specifics of the code below, you'll learn more about how all this works later. For now, just run the code and marvel at your plot in TWO LINES.

```
# Make a pirateplot using the pirateplot() function  
# from the yarr package!
```

```
ggplot(mtcars, aes(x=mpg)) +  
  geom_histogram(binwidth=5)
```



4.3.3 A simple approach to package

For novices, the `pacman` package can be used. All you need to do is to type in the name of the package in the function `pacman::p_load()`. In the example below, I want `pacman` to load the package `tidyverse` – notice how `"` are not used. If `tidyverse` is not found in your computer, `pacman` will download it first, then automatically load it. I will use this command from now on when loading packages.

```
if (!require("pacman")) install.packages("pacman")  
pacman::p_load(tidyverse) # All purpose wrangling for dataframes
```

4.4 Learning check

1. Create a folder called `se201` on your computer desktop.

2. Create a new project **inside** the folder `se201`, click File – New Project – New Directory – New Project – Browse, search for `se201` folder – under Directory name, type `practice`.
3. Create a new R script, click File – New File – R Script.
4. Save the new R script, click File – Save As. Save it inside the `se201/practice` folder. Use the file name `practice_script`. It will have the extension `.R` assigned to it automatically.
5. Enter the code below, run it, to install and load the packages.

```
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse, # All purpose wrangling for dataframes
               openxlsx, # writing excel documents
               lubridate, # date-time
               tibbletime) # moving average for vo2
```

6. Close RStudio, reopen RStudio via the `.Rproj` symbol Figure 1.3. In the Files tab on the bottom right, you should see the script you created `practice_script.R`. Click on it to open and you should see the code you typed. Run the codes you typed above again.
7. Download the solution to this learning check below.

Click to download the solution

Chapter 5

Input and Output

Download and load packages

```
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse, # All purpose wrangling for dataframes
               openxlsx) # writing excel documents
```

5.1 Dealing with “Cannot Open File” in Windows

You are running R on Windows, and you are using filenames such as `C:\data\sample.xlsx`. R says it cannot open the file, but you know the file does exist.

The backslashes in the filepath are causing trouble. You can solve this problem in one of two ways:

- Change the backslashes, `\`, to forward, `/`, slashes: `"C:/data/sample.txt"`.
- Double the backslashes: `"C:\\data\\sample.txt"`.

When you open a file in R, you give the filename as a character string. Problems arise when the name contains backslashes (`\`) because backslashes have a special meaning inside strings. You'll probably get something like this:

```
samp <- read.xlsx ("data\strength.xlsx")
```

```
## Error: '\s' is an unrecognized escape in character string starting ""data\s"
```

R escapes every character that follows a backslash and then removes the backslashes. That leaves a meaningless filepath, such as `C:Datasample-data.csv` in this example.

The simple solution is to use forward slashes instead of backslashes. R leaves the forward slashes alone, and Windows treats them just like backslashes. Problem solved:

```
samp <- read.xlsx ("data/strength.xlsx")
```

An alternative solution is to double the backslashes, since R replaces two consecutive backslashes with a single backslash:

```
samp <- read.xlsx ("data\\strength.xlsx")
```

5.2 Reading in Excel “.xlsx” data

If you have a .xlsx file that you want to read into R, use the `read.xlsx()` function in the `openxlsx` package.

Argument	Description
<code>xlsxFile</code>	The document's file path relative to the working directory unless specified otherwise. For example <code>xlsxFile = "SubjectData.xlsx"</code> looks for the text file directly in the working directory, while <code>xlsxFile = "data/SubjectData.xlsx"</code> will look for the file in an existing folder called <code>data</code> inside the working directory. If the file is outside of your working directory, you can also specify a full file path (<code>xlsxFile = "C:/Users/bl19622/Box/myBox/Documents/teaching/se747_ResearchMeth/sample_book/data/SubjectData.xlsx"</code>).
<code>sheet</code>	The name of the excel sheet or the numerical index. This is useful if you have many sheets in one Excel workbook. If this is not specified, the function automatically reads in the first sheet in the Excel workbook.

In my projects folder, I tend to have a habit of create a folder named `data`. This contains all my original data, that I do not want to touch!!! Let's test this function out by reading in an Excel file titled `strength.xlsx`. Since the file is located in a folder called `data` in my working directory, I'll use the file path `xlsxFile = "strength.xlsx"`, and since the sheet I have is named "data", I will use `sheet = "data"`:

```
strn <- read.xlsx (xlsxFile = "data/strength.xlsx",
                  sheet = "data")
```

5.3 Writing a Data Frame to Excel

You want to write an R data frame to an Excel file.

Argument	Description
<code>x</code>	The object you are trying to export and write into an Excel file, commonly a dataframe you modified. For example <code>x = strn</code> .
<code>sheetName</code>	If you want to name the sheet of the Excel workbook.
<code>file</code>	Specifying where you want to export the Excel sheet relative to the working directory, and how you want to name the sheet you exported.

```
write.xlsx(x = strn,
          sheetName = "strength",
          file = "data/STRENGTH_write.xlsx")
```

5.4 Learning check

You can download the 3 files for this learning check below:

[Click to download treadmill data](#)

[Click to download the individual FMS data](#)

[Click to download the group FMS data](#)

1. In your `se201/practice` folder, create another folder called `data`.
2. Place the files downloaded inside the `se201/practice/data` folder.
3. From your learning check in 4.4, open RStudio via the `.Rproj` symbol. In the Files tab on the bottom right, you should see the script you created `practice_script.R`. Click on it to open.
4. Import the Excel file `data/Athlete_1_treadmill.xlsx`, and the sheet named `raw` and assign it to an object `dat`. See Recipe 5.2. Remember to add some comments to remind yourself what this line of code is trying to do. **Save it.**

5. Import the Excel file `data/Athlete_1_treadmill.xlsx`, and the sheet named `stage` and assign it to an object `dat_stage`. See Recipe 5.2. Remember to add some comments to remind yourself what this line of code is trying to do. **Save it.**
6. Import the Excel file `data/Athlete_1_FMS.xlsx`, and the sheet named `Sheet1` and assign it to an object `dat_fms`. See Recipe 5.2. Remember to add some comments to remind yourself what this line of code is trying to do. **Save it.**
7. Import the Excel file `data/simFMS.xlsx`, and the sheet named `FMS` and assign it to an object `dat_fms_grp`. See Recipe 5.2. Remember to add some comments to remind yourself what this line of code is trying to do. **Save it.**
8. Download the solution to this learning check below.

[Click to download the solution](#)

Chapter 6

Data manipulation

Download and load packages

```
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse, # All purpose wrangling for dataframes
               openxlsx, # writing excel documents
               lubridate, # date-time
               tibbletime) # moving average for vo2
```

Import data

Let's load the 3 files from `data` folder into the workspace again. File one contains the FMS data, another the VO2 raw data from the treadmill test, and lastly the lactate data from the treadmill test.

```
dat_fms <- read.xlsx (xlsxFile = "data/fms_pt3.xlsx",
                     sheet = "Sheet1")

dat_fms_grp <- read.xlsx (xlsxFile = "data/simFMS.xlsx",
                         sheet = "FMS")

dat_vo2 <- read.xlsx (xlsxFile = "data/treadmill_pt4.xlsx",
                    sheet = "raw")

dat_vo2_stage <- read.xlsx (xlsxFile = "data/treadmill_pt4.xlsx",
                          sheet = "stage")
```

Let us see the data

6.0.0.1 dat_fms table

This is a table representing a single subject's FMS test scores

task	side	score
squat	c	3
hurdle	l	2
hurdle	r	2
lunge	l	3
lunge	r	3
leg_raise	l	2
leg_raise	r	1
shd_mob	l	1
shd_mob	r	2
rot_stab	l	2
rot_stab	r	2
push_up	c	1

6.0.0.2 dat_fms_grp

This is a table representing a team's total FMS test score for each task

id	squat	hurdle	lunge	shd_mob	leg_raise	push_up	rot_stab
athlete_a	2	2	2	3	3	3	2
athlete_b	2	2	3	2	1	3	2
athlete_c	2	2	3	2	1	1	2
athlete_d	2	2	3	2	3	2	1
athlete_e	2	2	2	3	1	1	2
athlete_f	2	2	2	2	2	2	2
athlete_g	2	2	3	3	3	1	2
athlete_h	3	2	3	2	2	3	2
athlete_i	3	2	2	2	2	1	2
athlete_j	1	2	3	2	2	3	2
athlete_k	3	2	2	2	2	3	2
athlete_l	2	2	2	3	3	2	2
athlete_m	1	2	2	3	2	3	2
athlete_n	2	2	2	2	1	3	2
athlete_o	3	2	3	2	3	3	2
athlete_p	2	3	2	2	3	1	2
athlete_q	3	2	3	3	2	3	2
athlete_r	2	2	3	2	1	2	2
athlete_s	2	2	2	3	2	1	2
athlete_t	2	2	3	2	1	3	2
athlete_u	2	2	3	2	2	3	2
athlete_v	3	2	2	2	3	3	2
athlete_w	3	2	3	3	2	3	2
athlete_x	2	2	2	2	1	1	2

6.0.0.3 dat_vo2

This is a table representing a single subject's raw VO2 results during incremental treadmill running

Time	BF	VO2/kg	RER	V'O2	V'CO2	V'E	HR
min	1/min	ml/ min/kg		ml/min	ml/min	L/min	bpm
00:05	37	21.4	0.93	1371	1272	46	119
00:10	58	2.5	0.84	159	133	12	120
00:15	30	23.5	0.74	1506	1110	35	121
00:20	34	22.4	0.86	1437	1229	43	123
00:25	23	15.6	0.73	1004	735	22	124
00:30	27	30.5	0.76	1960	1489	48	124
00:35	39	23.3	0.73	1499	1092	38	124
00:40	34	29.6	0.75	1902	1427	49	124
00:45	26	31.9	0.8	2048	1641	53	124
00:50	27	29.4	0.8	1891	1519	49	125
00:55	34	28.9	0.79	1856	1465	47	126
01:00	32	29.9	0.8	1919	1526	49	126
01:05	31	29.9	0.79	1922	1513	46	124
01:10	24	33	0.84	2118	1779	56	126
01:15	42	29.9	0.8	1921	1543	50	126
01:20	33	31.7	0.8	2035	1638	51	124
01:25	24	32.1	0.82	2060	1692	50	125
01:30	28	32.9	0.82	2112	1731	53	126
01:35	28	33.700000000000003	0.83	2164	1795	52	130
01:40	24	33.4	0.87	2142	1863	57	128
01:45	31	31.1	0.85	1997	1694	52	129
01:50	21	34.5	0.88	2214	1945	58	130
01:55	37	32.700000000000003	0.84	2098	1761	55	131
02:00	28	34.700000000000003	0.85	2230	1905	58	131
02:05	33	34.299999999999997	0.87	2201	1904	61	132
02:10	30	28.3	0.87	1814	1579	51	133
02:15	29	32.6	0.89	2094	1861	57	133
02:20	32	35.4	0.86	2275	1961	60	133
02:25	30	33.5	0.88	2149	1889	58	133
02:30	28	33.9	0.9	2176	1962	60	134
02:35	24	33.700000000000003	0.93	2166	2013	63	134
02:40	34	17.5	0.88	1124	993	34	135
02:45	25	36.299999999999997	0.79	2332	1844	48	136
02:50	33	38	0.84	2439	2053	65	137
02:55	34	33.9	0.87	2179	1896	61	138
03:00	34	32.4	0.9	2077	1868	60	139
03:05	46	32.200000000000003	0.86	2069	1773	61	140
03:10	26	25.3	1	1625	1623	63	141
03:15	27	17	1.0900000000000001	1094	1198	49	145
03:20	27	15.2	1.1499999999999999	977	1122	47	148
03:25	27	15.8	1.18	1017	1196	49	151
03:30	24	12.1	1.1299999999999999	779	879	36	154
03:35	51	18.3	0.96	1175	1130	45	156
03:40	40	35.799999999999997	0.82	2299	1892	57	149
03:45	57	29.3	0.77	1883	1442	50	138
03:50	33	33	0.88	2118	1869	62	131
03:55	35	26.9	0.94	1727	1620	55	130
04:00	46	27.7	0.81	1776	1444	49	132
04:05	35	35	0.84	2245	1888	63	132
04:10	43	29.7	0.84	1906	1595	54	133
04:15	27	30.5	0.84	1961	1643	52	137
04:20	23	31.3	0.8	2012	1612	48	139

6.0.0.4 dat_vo2_stage

This is a table representing a single subject's lactate values during incremental treadmill running

stage	speed	rpe	lactate
1	1	10	1.0
2	2	11	1.5
3	3	12	1.8
4	4	13	2.5
5	5	14	6.0
6	6	15	7.7
7	7	16	9.0
8	8	17	10.0

6.1 Tidy data

You can represent the same underlying data in multiple ways. The example below shows the same data organised in four different ways. This dataset is **not the data you loaded**, but rather came with the `tidyverse` package. Each dataset shows the same values of four variables *country*, *year*, *population*, and *cases*, but each dataset organises the values in a different way.

```
table1
```

```
## # A tibble: 6 x 4
##   country    year cases population
##   <chr>    <int> <int>    <int>
## 1 Afghanistan 1999   745  19987071
## 2 Afghanistan 2000  2666  20595360
## 3 Brazil      1999 37737  172006362
## 4 Brazil      2000 80488  174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

```
table2
```

```
## # A tibble: 12 x 4
##   country    year type      count
##   <chr>    <int> <chr>    <int>
## 1 Afghanistan 1999 cases         745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases         2666
```

```
## 4 Afghanistan 2000 population 20595360
## 5 Brazil      1999 cases      37737
## 6 Brazil      1999 population 172006362
## 7 Brazil      2000 cases      80488
## 8 Brazil      2000 population 174504898
## 9 China       1999 cases      212258
## 10 China      1999 population 1272915272
## 11 China      2000 cases      213766
## 12 China      2000 population 1280428583
```

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>      <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
```

```
# Spread across two tables
```

```
table4a # cases
```

```
## # A tibble: 3 x 3
##   country      `1999` `2000`
## * <chr>      <int> <int>
## 1 Afghanistan    745    2666
## 2 Brazil         37737  80488
## 3 China          212258 213766
```

```
table4b # population
```

```
## # A tibble: 3 x 3
##   country      `1999`      `2000`
## * <chr>      <int>      <int>
## 1 Afghanistan 19987071 20595360
## 2 Brazil      172006362 174504898
## 3 China       1272915272 1280428583
```

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset, will be much easier to work with.

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

Figure 6.1 shows the rules visually.

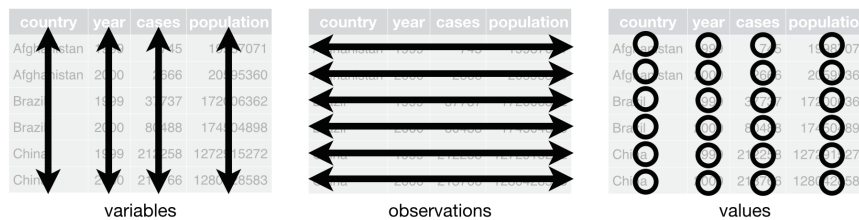


Figure 6.1: Following three rules makes a dataset tidy: variables are in columns, observations are in rows, and values are in cells.

These three rules are interrelated because it's impossible to only satisfy two of the three.

In this example, only `table1` is tidy. It's the only representation where each column is a variable.

Why ensure that your data is tidy? There are two main advantages:

1. There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.
2. There's a specific advantage to placing variables in columns because most of R functions work with data in the tidy format. That makes transforming tidy data feel particularly natural.

The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons:

1. Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.
2. Data is often organised to facilitate some use other than analysis. For example, data is often organised to make entry as easy as possible.

This means for most real analyses, you'll need to do some tidying. The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. The second step is to resolve one of two common problems:

1. One variable might be spread across multiple columns.
2. One observation might be scattered across multiple rows.

Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in *tidyr*: `pivot_wider()` in Recipe 6.8 and `pivot_longer()` in Recipe 6.7 below.

6.2 Renaming variables

6.2.1 Problem

The current column names in the `dat_vo2` dataset is ugly, and you like to abbreviate it. I dislike excessive usage of capitalisations. It is a lot of effort to type.

```
# This prints the current column names
```

```
colnames (dat_vo2)
```

```
## [1] "Time"    "BF"      "VO2/kg"  "RER"     "V'O2"    "V'CO2"   "V'E"     "HR"
```

6.2.2 Solution

Create a vector of 8 new names that you can give to the data. The order of new names must be identical to the order of the old names from left to right.

```
# New names. Notice the commas and be pedantic about spacing
```

```
new_names <- c("time", "bf", "vo2_norm", "rer", "vo2", "vco2", "ve", "hr")
```

```
# Give the new names to the old names
```

```
colnames (dat_vo2) <- new_names
```



```
# This prints the new column names
```

```
colnames (dat_vo2)
```

```
## [1] "time"      "bf"        "vo2_norm" "rer"       "vo2"       "vco2"      "ve"
## [8] "hr"
```

6.3 Selecting rows and columns

6.3.1 Problem

You want to remove a specific row or a range of rows, and remove a specific column or a range of columns. See also Recipe 6.12.

6.3.2 Solution

To keep or remove rows based on the row numbers, use `slice()`, `slice_head()`, or `slice_tail()`. In `slice()`, when you want to keep the row, add that row number. If you want to remove that row number, use a minus sign, `-`, before the number. If you want to keep or remove a range of numbers, add `start:end` inside `slice()`. In `slice_tail()` and `slice_head()`, you can keep the bottom, and top, *n* number of rows, respectively, by adding `n = 3` for example inside `slice_tail()` or `slice_head()`.

To keep based on the column names, use the `select()` function.

```
# This keeps only the second row
```

```
vo2_slice <- dat_vo2 %>%
  slice (2)
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr
00:05	37	21.4	0.93	1371	1272	46	119

```
# This removes the second row
```

```
vo2_slice <- dat_vo2 %>%
  slice (-2)
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr
min	1/min	ml/ min/kg		ml/min	ml/min	L/min	bpm
00:10	58	2.5	0.84	159	133	12	120
00:15	30	23.5	0.74	1506	1110	35	121
00:20	34	22.4	0.86	1437	1229	43	123
00:25	23	15.6	0.73	1004	735	22	124
00:30	27	30.5	0.76	1960	1489	48	124

```
# This removes the second and third rows
```

```
vo2_slice <- dat_vo2 %>%
  slice (- (2:3))
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr
min	1/min	ml/ min/kg		ml/min	ml/min	L/min	bpm
00:15	30	23.5	0.74	1506	1110	35	121
00:20	34	22.4	0.86	1437	1229	43	123
00:25	23	15.6	0.73	1004	735	22	124
00:30	27	30.5	0.76	1960	1489	48	124
00:35	39	23.3	0.73	1499	1092	38	124

```
# This keeps the top six rows
```

```
vo2_slice <- dat_vo2 %>%
  slice_head (n = 6)
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr
min	1/min	ml/ min/kg		ml/min	ml/min	L/min	bpm
00:05	37	21.4	0.93	1371	1272	46	119
00:10	58	2.5	0.84	159	133	12	120
00:15	30	23.5	0.74	1506	1110	35	121
00:20	34	22.4	0.86	1437	1229	43	123
00:25	23	15.6	0.73	1004	735	22	124

```
# This keeps the bottom six rows
```

```
vo2_slice <- dat_vo2 %>%
  slice_tail (n = 6)
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr
27:05	55	45.3125	1.0689655172413792	2900	3100	112	194
27:10	57	45.328125	1.0341261633919339	2901	3000	111	195
27:15	52	44.609375	1.063047285464098	2855	3035	111	194
27:20	59	46.5	1.0416666666666667	2976	3100	112	197
27:25	60	45.140625	1.0678435444790586	2889	3085	113	196
27:30	61	46	1.0506114130434783	2944	3093	114	195

```
# This selects the first, second, and fourth column, and discards the rest
```

```
vo2_slice <- dat_vo2 %>%
  dplyr::select (time, bf, rer)
```

time	bf	rer
min	1/min	
00:05	37	0.93
00:10	58	0.84
00:15	30	0.74
00:20	34	0.86
00:25	23	0.73

```
# This removes the first row, and replaces the old data in object dat_vo2

dat_vo2 <- dat_vo2 %>%
  slice (-(1))
```

6.4 Convert characters to numeric

6.4.1 Problem

The raw Vo2 treadmill dataset contains predominantly numbers, meaning it is numeric in nature. See also Section 3.4.1. But let us see the type of data that was imported. The `str()` function provides us with some summary information about the dataframe.

```
str(dat_vo2)

## 'data.frame':   330 obs. of  8 variables:
## $ time      : chr  " 00:05 " " 00:10 " " 00:15 " " 00:20 " ...
## $ bf        : chr  "37" "58" "30" "34" ...
## $ vo2_norm: chr  "21.4" "2.5" "23.5" "22.4" ...
## $ rer       : chr  "0.93" "0.84" "0.74" "0.86" ...
## $ vo2       : chr  "1371" "159" "1506" "1437" ...
## $ vco2      : chr  "1272" "133" "1110" "1229" ...
## $ ve        : chr  "46" "12" "35" "43" ...
## $ hr        : chr  "119" "120" "121" "123" ...
```

What you will see is something like this. For the variable `bf`, you will see `$ bf : chr "37" "58" "30" "34" chr` after the colon. This indicates that this variable is a character (`chr`). In R language, a character is anything from a letter, e.g. `a`, to a word, e.g. `word`, to even a phrase of sentence, e.g. `i hate biomechanics`. A character is always enclosed inside a `"`.

Why is R so “stupid”, that it cannot differentiate numbers from words!! Well it is not that R is stupid, but it is that in our original data, each column have both words and numbers (Figure 6.2). It is our fault that the data was untidy.

This was touched on in Section 6.1, about why this is bad data formatting. In brief, each column should be made up of one type of data. If it is numbers, make it all numbers; characters, all characters. If you have mix, the default is that R treats that column as all characters.

The reason why you should not leave numbers as characters is that you cannot do math on it. You cannot add `apples` and `orange` can you?

Time	BF	VO2/kg	RER	V'O2	V'CO2	V'E	HR
min	1/min	ml/ min/kg		ml/min	ml/min	L/min	bpm
00:05	37	21.4	0.93	1371	1272	46	119
00:10	58	2.5	0.84	159	133	12	120
00:15	30	23.5	0.74	1506	1110	35	121
00:20	34	22.4	0.86	1437	1229	43	123
00:25	23	15.6	0.73	1004	735	22	124
00:30	27	30.5	0.76	1960	1489	48	124
00:35	39	23.3	0.73	1499	1092	38	124
00:40	34	29.6	0.75	1902	1427	49	124
00:45	26	31.9	0.8	2048	1641	53	124
00:50	27	29.4	0.8	1891	1519	49	125
00:55	34	28.9	0.79	1856	1465	47	126

Figure 6.2: The type of data for each class as it was imported.

6.4.2 Solution

Use the `mutate()` function and the `as.numeric` function.

```
# Make all variables from bf to hr numeric, and replace the current data

dat_vo2 <- dat_vo2 %>%
  mutate (bf = as.numeric(bf),
          vo2_norm = as.numeric(vo2_norm),
          rer = as.numeric(rer),
          vo2 = as.numeric(vo2),
          vco2 = as.numeric(vco2),
          ve = as.numeric(ve),
          hr = as.numeric(hr))

# Recheck the type of each column
```

```
str(dat_vo2)
```

```
## 'data.frame': 330 obs. of 8 variables:
## $ time : chr " 00:05 " " 00:10 " " 00:15 " " 00:20 " ...
## $ bf : num 37 58 30 34 23 27 39 34 26 27 ...
## $ vo2_norm: num 21.4 2.5 23.5 22.4 15.6 30.5 23.3 29.6 31.9 29.4 ...
## $ rer : num 0.93 0.84 0.74 0.86 0.73 0.76 0.73 0.75 0.8 0.8 ...
## $ vo2 : num 1371 159 1506 1437 1004 ...
## $ vco2 : num 1272 133 1110 1229 735 ...
## $ ve : num 46 12 35 43 22 48 38 49 53 49 ...
## $ hr : num 119 120 121 123 124 124 124 124 124 125 ...
```

6.5 Convert characters to date and time

6.5.1 Problem

The raw `dat_vo2` `time` variable is a character with ugly white spaces. When you use the `str()` function on the `time` column using the `$` symbol, you can see that each value looks like `" 00:05 "`. It means that there is a blank before and after the `00:05`.

```
str(dat_vo2$time)
```

```
## chr [1:330] " 00:05 " " 00:10 " " 00:15 " " 00:20 " " 00:25 " ...
```

6.5.2 Solution

First, let me create a carbon copy of the data `dat_vo2`, so that I can modify the original data, and keep the carbon copy for explanation and illustration.

```
dat_vo2_copy <- dat_vo2
```

Use the `mutate()` function and the `as.numeric` function. You will learn the `mutate()` function more in Recipe 6.11.

```
# 4 in 1 processing
```

```
dat_vo2 <- dat_vo2 %>%
  mutate (time = time %>%
    str_squish() %>% # function strips any whitespaces)
```

```
ms() %>% # convert to minutes and seconds
as.period(unit = "sec") %>% # converts entirely to seconds
as.numeric () # strips the S symbol to make it a number
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr
5	37	21.4	0.93	1371	1272	46	119
10	58	2.5	0.84	159	133	12	120
15	30	23.5	0.74	1506	1110	35	121
20	34	22.4	0.86	1437	1229	43	123
25	23	15.6	0.73	1004	735	22	124
30	27	30.5	0.76	1960	1489	48	124

6.5.3 Discussion

Notice in the function above, I chained a series of steps together, using the pipe, `%>`, function. Alternatively, I could do it in separate steps, and we can take a look at how each step changed the appearance of the `time` variable. Let me illustrate each step using the carbon copy data `dat_vo2_copy`.

```
# First, removes whitespaces around the time variable
time_mutate <- dat_vo2_copy %>%
  mutate (time = time %>%
    str_squish())
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr
00:05	37	21.4	0.93	1371	1272	46	119
00:10	58	2.5	0.84	159	133	12	120
00:15	30	23.5	0.74	1506	1110	35	121
00:20	34	22.4	0.86	1437	1229	43	123
00:25	23	15.6	0.73	1004	735	22	124
00:30	27	30.5	0.76	1960	1489	48	124

```
# Second, convert to minutes and seconds

time_mutate <- dat_vo2_copy %>%
  mutate (time = time %>%
    str_squish() %>%
    ms())
```

	time	bf	vo2_norm	rer	vo2	vco2	ve	hr
325	27M 5S	55	45.31	1.07	2900	3100	112	194
326	27M 10S	57	45.33	1.03	2901	3000	111	195
327	27M 15S	52	44.61	1.06	2855	3035	111	194
328	27M 20S	59	46.50	1.04	2976	3100	112	197
329	27M 25S	60	45.14	1.07	2889	3085	113	196
330	27M 30S	61	46.00	1.05	2944	3093	114	195

```
# Third, converts entirely to seconds

time_mutate <- dat_vo2_copy %>%
  mutate (time = time %>%
            str_squish() %>%
            ms() %>%
            as.period(unit = "sec"))
```

	time	bf	vo2_norm	rer	vo2	vco2	ve	hr
325	1625S	55	45.31	1.07	2900	3100	112	194
326	1630S	57	45.33	1.03	2901	3000	111	195
327	1635S	52	44.61	1.06	2855	3035	111	194
328	1640S	59	46.50	1.04	2976	3100	112	197
329	1645S	60	45.14	1.07	2889	3085	113	196
330	1650S	61	46.00	1.05	2944	3093	114	195

```
# Fourth, strips the S symbol to make it a number, and assign it to object

time_mutate <- dat_vo2_copy %>%
  mutate (time = time %>%
            str_squish() %>%
            ms() %>%
            as.period(unit = "sec") %>%
            as.numeric ())
```

	time	bf	vo2_norm	rer	vo2	vco2	ve	hr
325	1625	55	45.31	1.07	2900	3100	112	194
326	1630	57	45.33	1.03	2901	3000	111	195
327	1635	52	44.61	1.06	2855	3035	111	194
328	1640	59	46.50	1.04	2976	3100	112	197
329	1645	60	45.14	1.07	2889	3085	113	196
330	1650	61	46.00	1.05	2944	3093	114	195

6.6 Split Numeric Variable into Categories

6.6.1 Problem

Remember, each stage in VO2 testing is 3:30min (210sec) long, and you collected data in 5sec intervals. You want to split the `time` variable of the `dat_vo2` dataset into chunks of 210 sec, and create a new variable called `stage`. You will learn the `mutate()` function more in Recipe 6.11. I will not explain the `cut_interval` function, other than to say that the `length` argument is used to specify the range of evenly spaced values to categorize over.

6.6.2 Solution

```
dat_vo2 <- dat_vo2 %>%
  mutate (stage = cut_interval(time, length = 210, labels = FALSE))
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr	stage
5	37	21.4	0.93	1371	1272	46	119	1
10	58	2.5	0.84	159	133	12	120	1
15	30	23.5	0.74	1506	1110	35	121	1
20	34	22.4	0.86	1437	1229	43	123	1
25	23	15.6	0.73	1004	735	22	124	1
30	27	30.5	0.76	1960	1489	48	124	1

6.6.3 Discussion

Why chunks of 210 sec? Vo2 data analysis, requires you to find the average values of the last 30s of each treadmill testing stage. Based on your Autumn School lessons, recall that each stage is 3 min 30 sec long or 210 sec. When doing data analysis, there is an amount of pre-planning on how to get the desired end product. There is no magic pill, the more analysis you do, the more short cuts you know.

6.7 Gathering

6.7.1 Problem

You want to make your data longer. In the data `dat_fms_grp` (go to the start of this chapter to see this data's original shape), you want to merge the 7 columns indicating the 7 FMS tasks into a single column. For this, you can use the `pivot_longer()` function.

6.7.2 Solution

```
dat_long <- dat_fms_grp %>% # original data
  pivot_longer(cols = -id,
               names_to = "task",
               values_to = "score")
```

id	task	score
athlete_a	squat	2
athlete_a	hurdle	2
athlete_a	lunge	2
athlete_a	shd_mob	3
athlete_a	leg_raise	3
athlete_a	push_up	3
athlete_a	rot_stab	2
athlete_b	squat	2
athlete_b	hurdle	2
athlete_b	lunge	3
athlete_b	shd_mob	2
athlete_b	leg_raise	1
athlete_b	push_up	3
athlete_b	rot_stab	2
athlete_c	squat	2
athlete_c	hurdle	2
athlete_c	lunge	3
athlete_c	shd_mob	2
athlete_c	leg_raise	1
athlete_c	push_up	1
athlete_c	rot_stab	2
athlete_d	squat	2
athlete_d	hurdle	2
athlete_d	lunge	3
athlete_d	shd_mob	2
athlete_d	leg_raise	3
athlete_d	push_up	2
athlete_d	rot_stab	1
athlete_e	squat	2
athlete_e	hurdle	2
athlete_e	lunge	2
athlete_e	shd_mob	3
athlete_e	leg_raise	1
athlete_e	push_up	1
athlete_e	rot_stab	2
athlete_f	squat	2
athlete_f	hurdle	2
athlete_f	lunge	2
athlete_f	shd_mob	2
athlete_f	leg_raise	2
athlete_f	push_up	2
athlete_f	rot_stab	2
athlete_g	squat	2
athlete_g	hurdle	2
athlete_g	lunge	3
athlete_g	shd_mob	3
athlete_g	leg_raise	3
athlete_g	push_up	1
athlete_g	rot_stab	2
athlete_h	squat	3
athlete_h	hurdle	2
athlete_h	lunge	3
athlete_h	shd_mob	2

6.8 Spreading

6.8.1 Problem

You want to make your data wider, in this instance let us look at the `dat_fms` dataset. You want to have one column indicating the left FMS score, and one indicating the right FMS score. The caveat is that because some tasks in the FMS do not have left and right, we need to remove the tasks in the FMS without a left and right. In this case, we will use the `filter()` function, which will be discussed in Recipe 6.12.

```
dat_fms_sub <- dat_fms %>%  
  filter (side != "c")
```

6.8.2 Solution

```
dat_wide <- dat_fms_sub %>% # original data  
  pivot_wider(names_from = "side",  
              values_from = "score")
```

task	l	r
hurdle	2	2
lunge	3	3
leg_raise	2	1
shd_mob	1	2
rot_stab	2	2

6.9 Rename values of a variable

6.9.1 Problem

You want to rename some values, either because it is too long, too short, or for other reasons. In this instance, in the `dat_fms` dataset, you want to call `l`, left, and `r`, right.

6.9.2 Solution

Use the `mutate()` function and the `recode()` function. You will learn the `mutate()` function more in Recipe 6.11.

```
dat_fms <- dat_fms %>%
  mutate (side = recode (side, # the variable name
    "l" = "left", # old label = new label
    "r" = "right")) # old label = new label
```

task	side	score
squat	c	3
hurdle	left	2
hurdle	right	2
lunge	left	3
lunge	right	3
leg_raise	left	2
leg_raise	right	1
shd_mob	left	1
shd_mob	right	2
rot_stab	left	2
rot_stab	right	2
push_up	c	1

6.10 Creating factors

6.10.1 Problem

You want to create an order in the values of a variable - like small, medium, large.

6.10.2 Solution

```
# Order of values I desire

new_lvls <- c( "push_up", "squat", "rot_stab", "hurdle", "lunge", "leg_raise", "shd_mob")

dat_fms_relvl <- dat_fms %>%
  mutate (task = factor (task, levels = new_lvls))
```

6.10.3 Discussion

Why do you need to convert categorical variables to factors? For visualization, the simple reason is that it allows you to control the order in which items appear

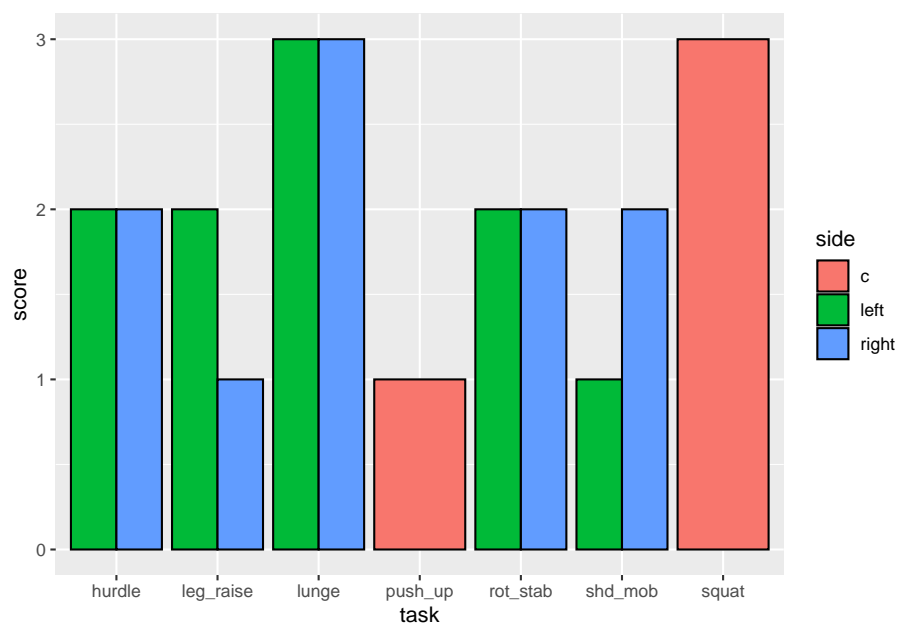
first to last. Of course, it has important statistical reasons, of which we do not discuss presently.

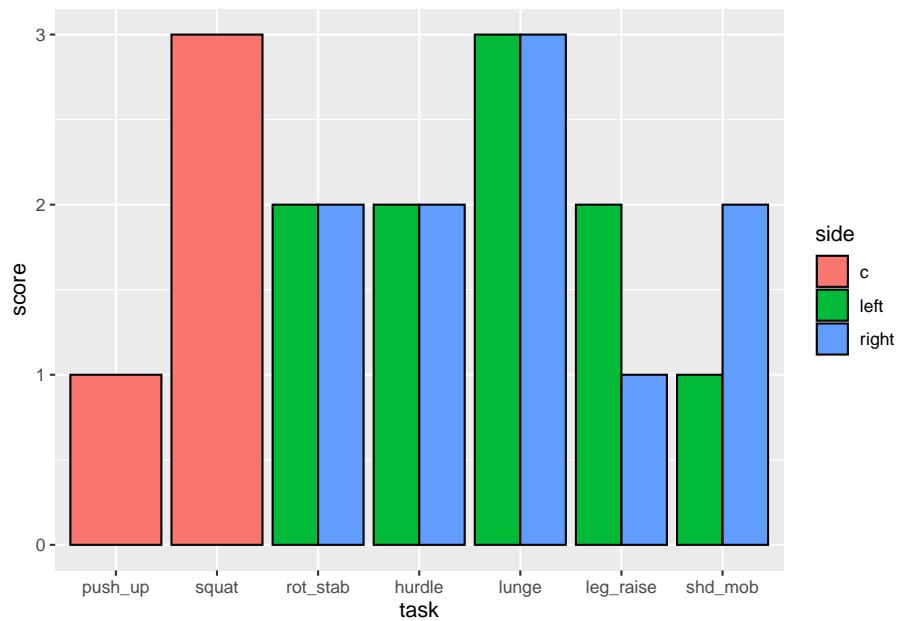
```
# Without factor

ggplot(dat_fms) +
  geom_col(aes(x = task, y = score, fill = side), position = "dodge", color = "black")

# With factor new levels

ggplot(dat_fms_relvl) +
  geom_col(aes(x = task, y = score, fill = side), position = "dodge", color = "black")
```





6.11 Making a new variable

6.11.1 Problem

You want to create a new column which involves some math between columns

6.11.2 Solution

Example, you want to create a column called `ratio` in the `dat_vo2` dataset, by dividing `vo2` by `vco2`.

```
dat_vo2 <- dat_vo2 %>%
  mutate (ratio = vo2/vco2)
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr	stage	ratio
5	37	21.4	0.93	1371	1272	46	119	1	1.08
10	58	2.5	0.84	159	133	12	120	1	1.20
15	30	23.5	0.74	1506	1110	35	121	1	1.36
20	34	22.4	0.86	1437	1229	43	123	1	1.17
25	23	15.6	0.73	1004	735	22	124	1	1.37
30	27	30.5	0.76	1960	1489	48	124	1	1.32

6.11.3 Discussion

There are many math operations you can do including subtraction (-), multiplication (*), addition (+), exponentiation/power to the power of 2 (^2). Let us go crazy and try some random creation of new variables.

```
dat_vo2_crazy <- dat_vo2 %>%
  mutate (vo2_power = vo2^3, # vo2 powered to 3
          vco2_mod = vco2/3, # vco2 divided by 3
          junk = (vo2+ vco2)/ (hr + bf)) # divided the sum of vo2 and vco2, and sum of hr and bf
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr	stage	ratio	vo2_power	vco2_mod	junk
5	37	21.4	0.93	1371	1272	46	119	1	1.08	2576987811	424.00	16.94
10	58	2.5	0.84	159	133	12	120	1	1.20	4019679	44.33	1.64
15	30	23.5	0.74	1506	1110	35	121	1	1.36	3415662216	370.00	17.32
20	34	22.4	0.86	1437	1229	43	123	1	1.17	2967360453	409.67	16.98
25	23	15.6	0.73	1004	735	22	124	1	1.37	1012048064	245.00	11.83
30	27	30.5	0.76	1960	1489	48	124	1	1.32	7529536000	496.33	22.84

You can even create a new variable, which reflects the present row number (`row_id`). This row number reflects the number of 5 sec VO2 windows. If there is 10 sec of data collected, there will be two row numbers, Really quite useful for VO2 analysis and cleaning. Here, inside the `mutate()` function, I will be using the function `row_number()` to get the respective number for each row.

```
dat_vo2 <- dat_vo2 %>%
  mutate (row_id = row_number())
```

time	bf	vo2_norm	rer	vo2	vco2	ve	hr	stage	ratio	row_id
5	37	21.4	0.93	1371	1272	46	119	1	1.08	1
10	58	2.5	0.84	159	133	12	120	1	1.20	2
15	30	23.5	0.74	1506	1110	35	121	1	1.36	3
20	34	22.4	0.86	1437	1229	43	123	1	1.17	4
25	23	15.6	0.73	1004	735	22	124	1	1.37	5
30	27	30.5	0.76	1960	1489	48	124	1	1.32	6

6.12 Filtering

Filtering is removing rows you do not want and keeping rows you want based on some condition(s). In Recipe 6.3, you learnt the `slice()` function. That is for really simple filtering. The `filter()` function you will learn here gives you more flexibility. Filtering deals with keeping or throwing out **rows** of data. Keeping or throwing out columns of data requires the `select()` function, which you learnt in Recipe 6.3.

6.12.1 Keep rows you want based on condition

6.12.1.1 Problem

In the `dat_fms` dataset, you want to keep rows that have `side == "left"` (i.e. only rows where the `side` value equals `left`). Note the `==`, double equal sign, instead of the usual `=`. In this short book, I will not go at length to explain why R is so difficult, only that I ask you to obey the rules of the language.

6.12.1.2 Solution

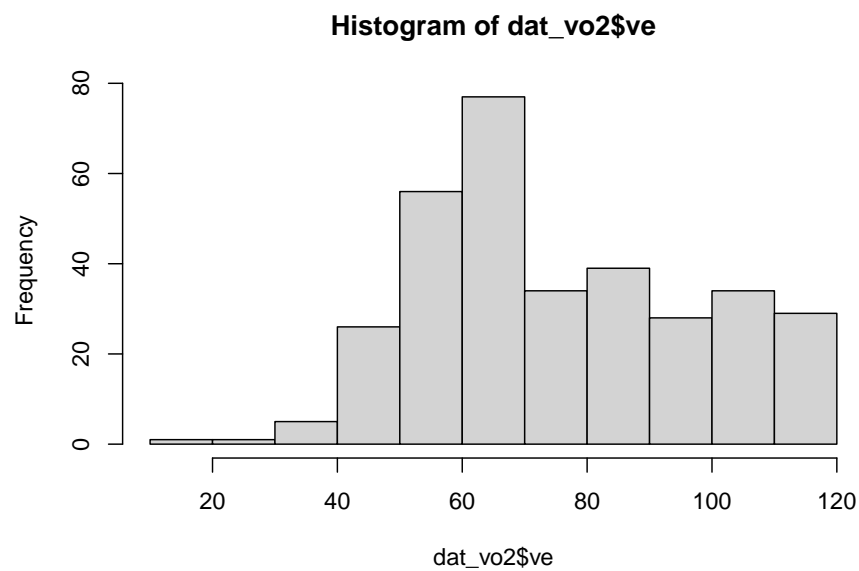
```
fms_left <- dat_fms %>%  
  filter (side == "left")
```

task	side	score
hurdle	left	2
lunge	left	3
leg_raise	left	2
shd_mob	left	1
rot_stab	left	2

6.12.2 Keep rows based on a numerical range

You want to keep the data when a variable is within a certain window range. Let us use the `dat_vo2` dataset. Let us see the range of values of the variable `ve`.

```
hist (dat_vo2$ve)
```

6.12.2.1 Problem

Say I want to keep rows where 1) `ve` is less than 80, 2) `ve` more than 40, and 3) `ve` is between 40 to 80.

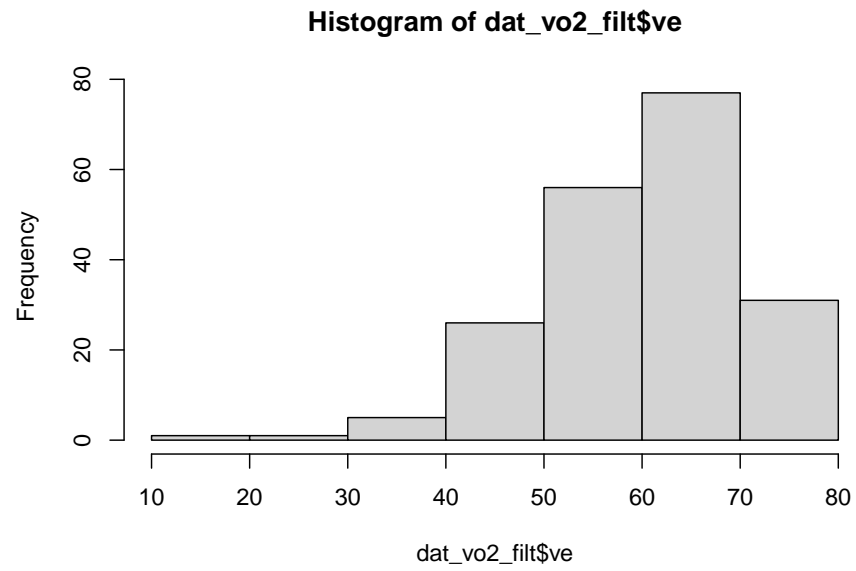
6.12.2.2 Solution

Notice for each graph, how the minimum and maximum values have been clipped off.

```
# Keep ve less than 80

dat_vo2_filt <- dat_vo2 %>%
  filter (ve < 80)

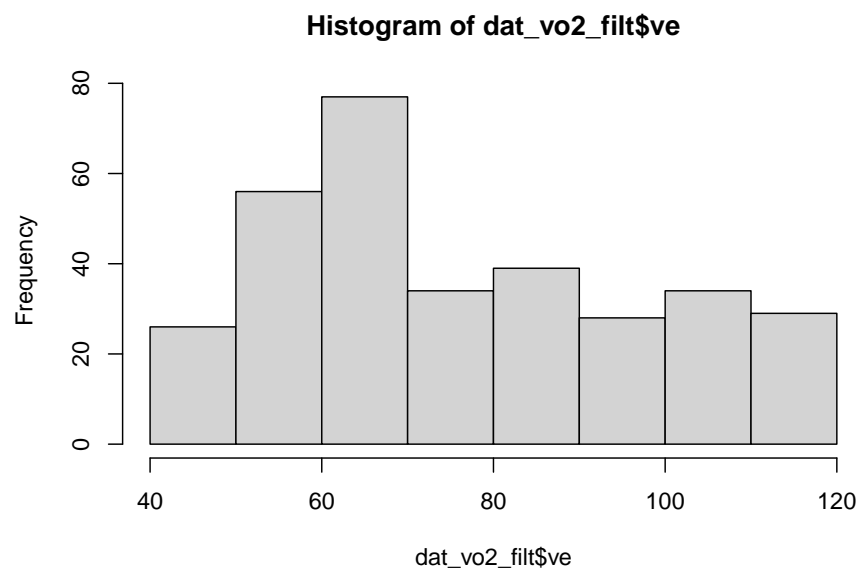
hist (dat_vo2_filt$ve)
```



```
# Keep ve lmore than 40

dat_vo2_filt <- dat_vo2 %>%
  filter (ve > 40)

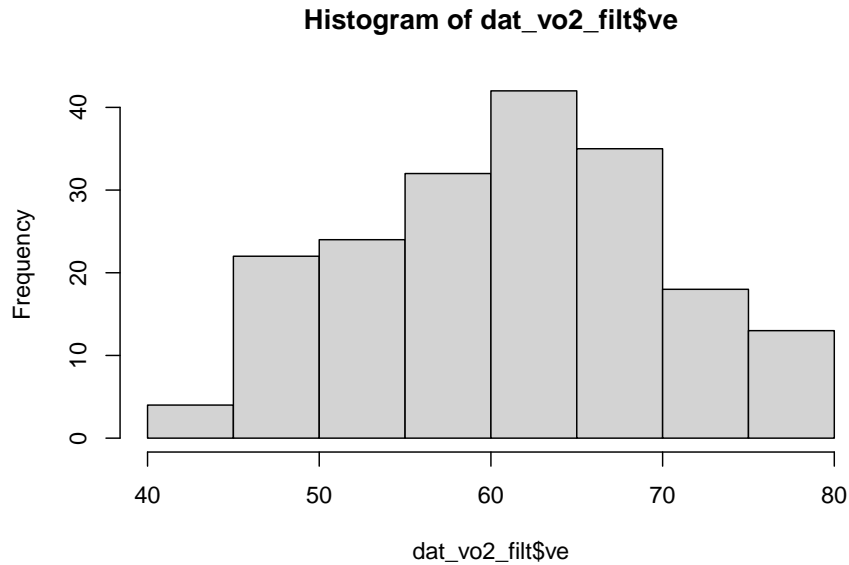
hist (dat_vo2_filt$ve)
```



```
# Keep ve between 40 to 80

dat_vo2_filt <- dat_vo2 %>%
  filter (ve > 40 & ve < 80)

hist (dat_vo2_filt$ve)
```



6.12.3 Discussion

`filter()` works really when you want to throw out or keep rows of data based on some ranges or criteria of the variables you have. When you want to keep rows of data based on the row number, use `slice()`. See also Recipe 6.3.

6.13 Global summary

6.13.1 Problem

You want to find the total FMS score across all sides and tasks

6.13.2 Solution

In the `dat_fms` dataset, we want to use the `summarize()` function.

```
dat_fms %>%  
  summarize (total_score = sum (score))
```

```
## total_score
## 1          24
```

I dare you to find an easier way to calculate such scores in one line of code. That is one reason why I use a programming language like R. It is fast!!!

6.13.3 Discussion

We can also go berserk by trying out different summary functions, like `mean()`, `median()`, `min()` (minimum value), `max()` (maximum value). The function `n()` is a really useful one to count the number of rows. If your rows indicate the number of subjects, `n()` essentially counts the number of subjects.

```
fms_summ <- dat_fms %>%
  summarize (total_score = sum (score),
            mean_score = mean (score),
            median_score = median (score),
            min_score = min (score),
            max_score = max (score),
            count = n())
```

total_score	mean_score	median_score	min_score	max_score	count
24	2	2	1	3	12

6.14 Group-by summary

6.14.1 Problem

You want to perform the same summary function for each chunk of group. For example, the FMS is typically scored by taking the lower of the two values of left and right for each task. In `side`, we have three values, `c`, `left` and `right`. The minimum of `c` is the same value itself.

6.14.2 Solution

In the `dat_fms_grp` dataset, we want to use the `group_by()` and `summarize()` function. The code below reads as: collapse all tasks into a single column, for each task and each score, count the number of rows (subjects) who has that score.

```
fms_grp_summ <- dat_fms_grp %>%
  pivot_longer(cols = -id,
               names_to = "task",
               values_to = "score") %>%
  group_by(task, score) %>%
  summarize (count = n())

## `summarise()` regrouping output by 'task' (override with `.groups` argument)
```

task	score	count
hurdle	2	23
hurdle	3	1
leg_raise	1	7
leg_raise	2	10
leg_raise	3	7
lunge	2	12
lunge	3	12
push_up	1	7
push_up	2	4
push_up	3	13
rot_stab	1	1
rot_stab	2	23
shd_mob	2	16
shd_mob	3	8
squat	1	2
squat	2	15
squat	3	7

6.14.3 Discussion

Why do you need to learn to create summaries? It is the basis for generating plots easily. Let us look at the `dat_vo2` dataset.

Remember in your lessons, the last 30 sec of each 3 min 30 sec **complete** stage is discarded. That means that I want to throw away the data interval between 3:00 to 3:30 min for every stage.

Thereafter, for the **complete** stage, the last 30 sec of each 3 min interval is then averaged for further analysis. This means keeping data between 2:30 to 3:00 min of each stage. That corresponds to the last six rows of each stage (i.e. rows 31 to 36 of each stage). Try to think why i selected these row numbers!

For the last stage, there may be the chance that it is **incomplete**. Based on Kelly's lesson, you will need to extract the last 30 sec (i.e. last six rows) of this last incomplete stage.

Table 6.1: Stage by stage VO2 analysis

stage	bf	vo2_norm	rer	vo2	vco2	ve	hr
1	30.67	31.97	0.87	2052.83	1777.83	55.17	136.50
2	34.67	34.60	0.88	2222.17	1957.50	59.33	147.00
3	34.67	37.42	0.91	2401.83	2191.33	66.67	161.33
4	38.00	37.83	0.95	2429.00	2312.50	71.17	172.17
5	44.00	41.67	1.00	2675.00	2686.00	85.67	178.50
6	45.67	43.85	1.06	2813.67	2981.17	99.83	185.83
7	54.50	45.50	1.09	2920.17	3195.83	110.33	192.33
8	57.33	45.48	1.05	2910.83	3068.83	112.17	195.17

I then want to average all my variables over these 6 rows per stage.

I should end up with a dataframe that has 8 rows given 8 stages, one for each stage in this example.

```
dat_vo2_summ <- dat_vo2 %>%
  group_by(stage) %>% # for each group
  mutate (row_id = row_number()) %>%
  filter (row_id < 37) %>%
  slice_tail (n = 6) %>%
  summarise (bf = mean (bf),
             vo2_norm = mean (vo2_norm),
             rer = mean (rer),
             vo2 = mean (vo2),
             vco2 = mean (vco2),
             ve = mean (ve),
             hr = mean (hr))

## `summarise()` ungrouping output (override with `.groups` argument)
```

6.15 Merge two tables together

6.15.1 Problem

You want to combine two tables together, ensuring that each row is linked appropriately. For example, you want to combine your `dat_vo2_summ` summary data with the `dat_vo2_stage` table which contains the RPE and blood lactate values.

6.15.2 Solution

Use the `inner_join` function, with the `by` key as the common identifier that must be in both dataframes.

```
dat_vo2_comb <- dat_vo2_summ %>%
  inner_join(dat_vo2_stage, by = "stage")
```

stage	bf	vo2_norm	rer	vo2	vco2	ve	hr	speed	rpe	lactate
1	30.67	31.97	0.87	2052.83	1777.83	55.17	136.50	1	10	1.0
2	34.67	34.60	0.88	2222.17	1957.50	59.33	147.00	2	11	1.5
3	34.67	37.42	0.91	2401.83	2191.33	66.67	161.33	3	12	1.8
4	38.00	37.83	0.95	2429.00	2312.50	71.17	172.17	4	13	2.5
5	44.00	41.67	1.00	2675.00	2686.00	85.67	178.50	5	14	6.0
6	45.67	43.85	1.06	2813.67	2981.17	99.83	185.83	6	15	7.7
7	54.50	45.50	1.09	2920.17	3195.83	110.33	192.33	7	16	9.0
8	57.33	45.48	1.05	2910.83	3068.83	112.17	195.17	8	17	10.0

6.15.3 Discussion

It is common sense, that if we want to join two tables, there must be some rules. First, if we want to join two tables side by side left-right, the number of rows must match. Second, there must be a way the computer knows how to join, much like a key and a keyhole. The key is the column to join by, in this example, `stage`. In the `dat_vo2_summ` there is the `stage` variable with identical number of levels 1-8, and the same variable appears in `dat_vo2_stage`. You don't have to worry if after merging, the data is sorted correctly, if you follow these rules.

6.16 Learning check

1. From your learning check in 5.4, open up your `practice_script.R`. You should already have the codes to import the `data/Athlete_1_treadmill.xlsx` file, and label the data called `dat`.
2. Rename the columns of `dat` to be `("time", "bf", "vo2_norm", "rer", "vo2", "vco2", "ve", "hr")`, respectively. See Recipe 6.2.
3. Remove the first row of the `dat` data, as it is useless. See Recipe 6.3.
4. Convert the variable `time` of `dat` into numeric seconds, and all other variables to numeric. See Recipe 6.5 and 6.4, respectively.
5. Create a variable called `stage`, where each stage represents 3:30 min worth of VO2 data. See Recipe 6.6.

6. For each stage, create a variable called `row_id`, which essentially represents the number of 5 sec windows in each stage. See Recipe 6.11 and 6.14.
7. For each stage, remove the data between 3:00 to 3:30min. See Recipe 6.14 and 6.12.2.
8. For each stage, keep the last six rows of data. See Recipe 6.14 and 6.3.
9. For each stage, calculate for all variables for the mean of the last six rows of data. See Recipe 6.14 and 6.13.
10. Save this cleaned data set into the `data/` folder called `Athlete_1_treadmill_clean.xlsx`. Go into this folder physically, and see this new excel file is there and open it. See Recipe 5.3
11. Download the solution to this learning check below.

Click to download the solution

```
## Warning: package 'igraph' was built under R version 4.0.3
```

```
## Warning: package 'MASS' was built under R version 4.0.3
```


Chapter 7

Bar Graphs

There are so many types of figures that you can create in R (Figure 7.1). In the next two chapters, I will only delve into two types of figures that I believe is most useful for your case. With this basic skills, you have the foundation to delve deeper into this software if you desire.

Bar graphs are perhaps the most commonly used kind of data visualization. They're typically used to display numeric values (on the y-axis), for different categories (on the x-axis). For example, a bar graph would be good for showing the prices of four different kinds of items. A bar graph generally wouldn't be as good for showing prices over time, where time is a continuous variable – though it can be done.

There's an important distinction you should be aware of when making bar graphs: sometimes the bar heights represent **counts** of cases in the data set, and sometimes they represent **values** in the data set. Keep this distinction in mind – it can be a source of confusion since they have very different relationships to the data, but the same term is used for both of them. In this chapter I'll discuss always use bar graphs with **values**.

Let us prepare for this chapter by importing two FMS tests data. The Excel sheet is called "fms_pt3.xlsx" and "simFMS.xlsx".

```
# Define factor levels
fct_lvls <- c("squat", "push_up", "hurdle", "lunge", "leg_raise", "rot_stab", "shd_mob")

# Import data
dat <- read.xlsx (xlsxFile = "data/fms_pt3.xlsx",
                 sheet = "Sheet1")

dat_grp <- read.xlsx (xlsxFile = "data/simFMS.xlsx",
                    sheet = "FMS")
```

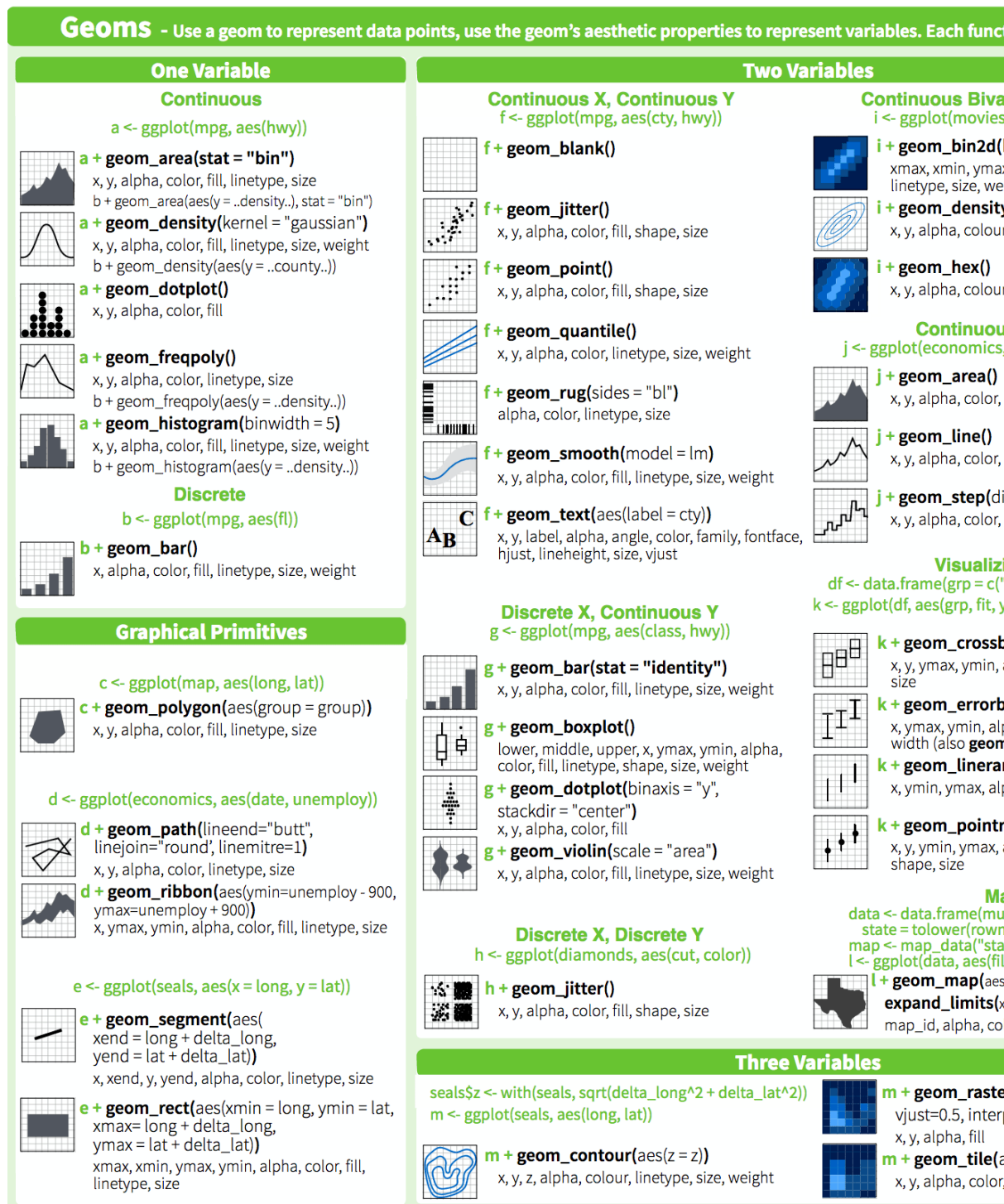


Figure 7.1: The plotting capabilities are endless.

```

# Tidy the data
dat <- dat %>%
  mutate (task = factor (task, levels = fct_lvls))

dat_grp <- dat_grp %>%
  pivot_longer(cols = -id,
               names_to = "task",
               values_to = "score") %>%
  group_by(task, score) %>%
  mutate (task = factor (task, levels = fct_lvls),
          score = factor (score, levels = c("0", "1", "2", "3")))

```

task	side	score
squat	c	3
hurdle	l	2
hurdle	r	2
lunge	l	3
lunge	r	3
leg_raise	l	2
leg_raise	r	1
shd_mob	l	1
shd_mob	r	2
rot_stab	l	2
rot_stab	r	2
push_up	c	1

id	task	score
athlete_a	squat	2
athlete_a	hurdle	2
athlete_a	lunge	2
athlete_a	shd_mob	3
athlete_a	leg_raise	3
athlete_a	push_up	3
athlete_a	rot_stab	2
athlete_b	squat	2
athlete_b	hurdle	2
athlete_b	lunge	3
athlete_b	shd_mob	2
athlete_b	leg_raise	1
athlete_b	push_up	3
athlete_b	rot_stab	2
athlete_c	squat	2
athlete_c	hurdle	2
athlete_c	lunge	3
athlete_c	shd_mob	2
athlete_c	leg_raise	1
athlete_c	push_up	1
athlete_c	rot_stab	2
athlete_d	squat	2
athlete_d	hurdle	2
athlete_d	lunge	3
athlete_d	shd_mob	2
athlete_d	leg_raise	3
athlete_d	push_up	2
athlete_d	rot_stab	1
athlete_e	squat	2
athlete_e	hurdle	2
athlete_e	lunge	2
athlete_e	shd_mob	3
athlete_e	leg_raise	1
athlete_e	push_up	1
athlete_e	rot_stab	2
athlete_f	squat	2
athlete_f	hurdle	2
athlete_f	lunge	2
athlete_f	shd_mob	2
athlete_f	leg_raise	2
athlete_f	push_up	2
athlete_f	rot_stab	2
athlete_g	squat	2
athlete_g	hurdle	2
athlete_g	lunge	3
athlete_g	shd_mob	3
athlete_g	leg_raise	3
athlete_g	push_up	1
athlete_g	rot_stab	2
athlete_h	squat	3
athlete_h	hurdle	2
athlete_h	lunge	3
athlete_h	shd_mob	2

Let us also create two more datasets where 1) we take the lower of the two scores for tasks which are assessed bilaterally and 2) count the number of subjects who attain a specific score for each task. See Recipe 6.14.

```
dat_summ <- dat %>%
  group_by(task) %>%
  summarize (total = min (score)) %>%
  ungroup ()
#> `summarise()` ungrouping output (override with `.groups` argument)

dat_grp_summ <- dat_grp %>%
  group_by(task, score) %>%
  summarize (count = n())
#> `summarise()` regrouping output by 'task' (override with `.groups` argument)
```

task	total
squat	3
push_up	1
hurdle	2
lunge	3
leg_raise	1
rot_stab	2
shd_mob	1

7.1 Making a Basic Bar Graph

7.1.1 Problem

You have a data frame where one column represents the x position of each bar, and another column represents the vertical y height of each bar.

7.1.2 Solution

Use `ggplot()` with `geom_col()` and specify what variables you want on the x- and y-axes (Figure 7.2):

```
ggplot(dat_summ) +
  geom_col(aes(x = task, y = total))
```

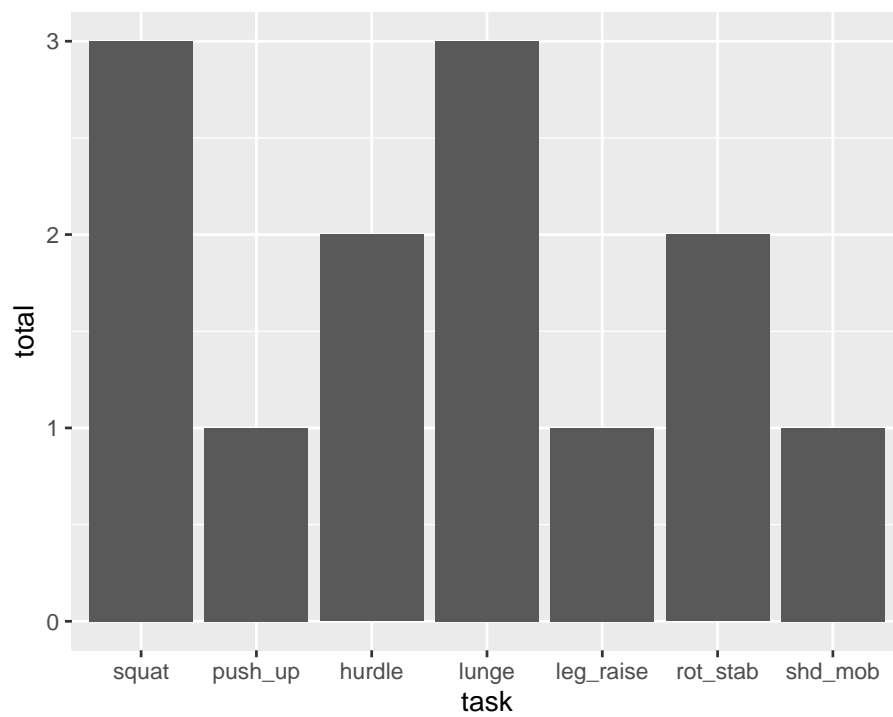


Figure 7.2: Bar graph of values with a discrete x-axis

7.1.3 Discussion

By default, bar graphs use a dark grey for the bars. To use a color fill, use `fill`. Also, by default, there is no outline around the fill. To add an outline, use `colour`. For Figure 7.3, we use a light blue fill and a black outline:

```
ggplot(dat_summ) +  
  geom_col(aes(x = task, y = total), fill = "lightblue", colour = "black")
```

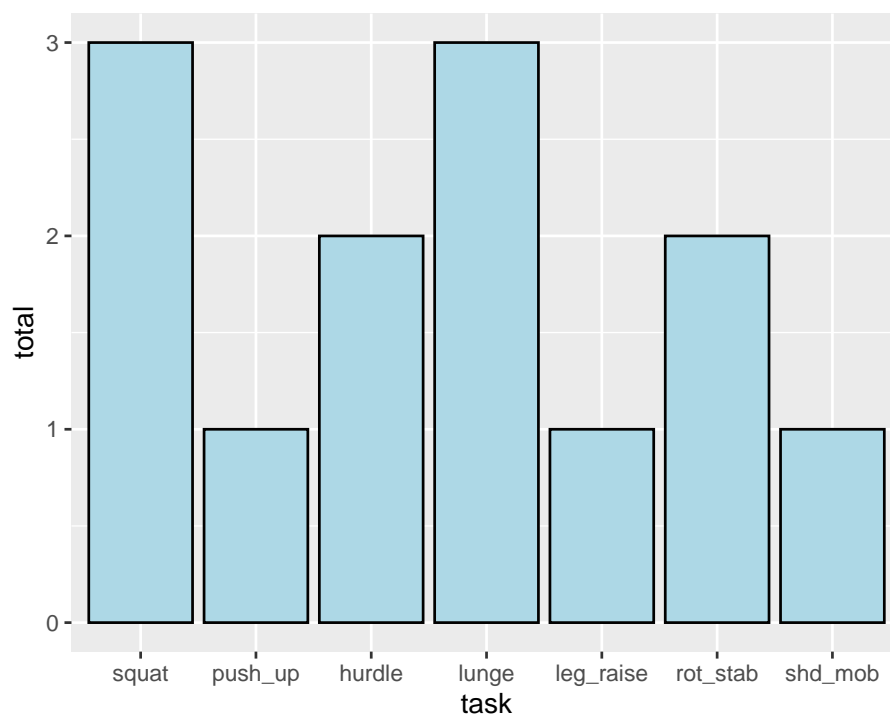


Figure 7.3: A single fill and outline color for all bars

Note In `ggplot2`, which is the package used for plotting, the default is to use the British spelling, `colour`, instead of the American spelling, `color`. Internally, American spellings are remapped to the British ones, so if you use the American spelling it will still work.

7.2 Anatomy of a Graph

There is a lot of things that is going on behind the scene in this simple code of `ggplot(dat_summ) + geom_col(aes(x = task, y = total))`. Let us delve a little into

it, to understand the grammar of any graph, not just the bar graph we created. Remember, for any software, this grammar or anatomy towards a graph will be similar.

7.2.1 Plot Background

To start building the plot, we first specify the data frame that contains the relevant data. Here we are ‘sending the `dat_summ` data set into the `ggplot` function’:

```
# render background  
ggplot(data = dat_summ)
```

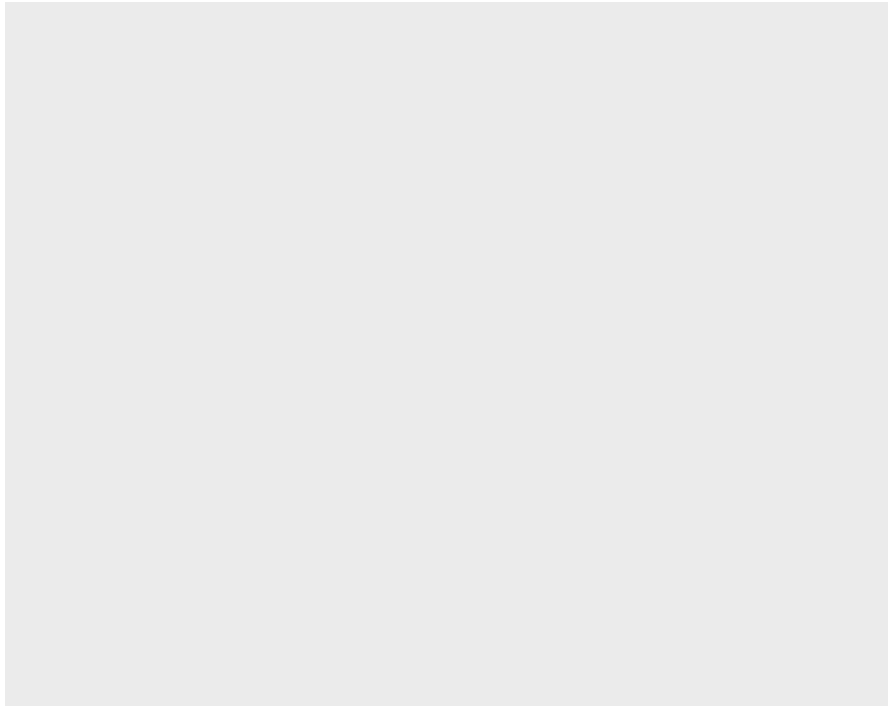


Figure 7.4: An empty plot area waiting to be filled

Running this command will produce an empty grey canvas. This is because we not yet specified what variables are to be plotted.

7.2.2 Aesthetics `aes()`

We can call in different columns of data from `dat_summ` based on their column names. Column names are given as 'aesthetic' elements to the `ggplot` function, and are wrapped in the `aes()` function.

Because we want a bar plot, each bar will have an x and a y coordinate. We want the x axis to represent `task` (`x = task`), and the y axis to represent the `total` FMS score (`y = total`).

See how the x- and y-axis titles, labels, and tick-marks become populated? But still nothing plotted, and that is because you have not tell it what shapes to plot.

```
ggplot(data = dat_summ, aes(x = task, y = total) )
```

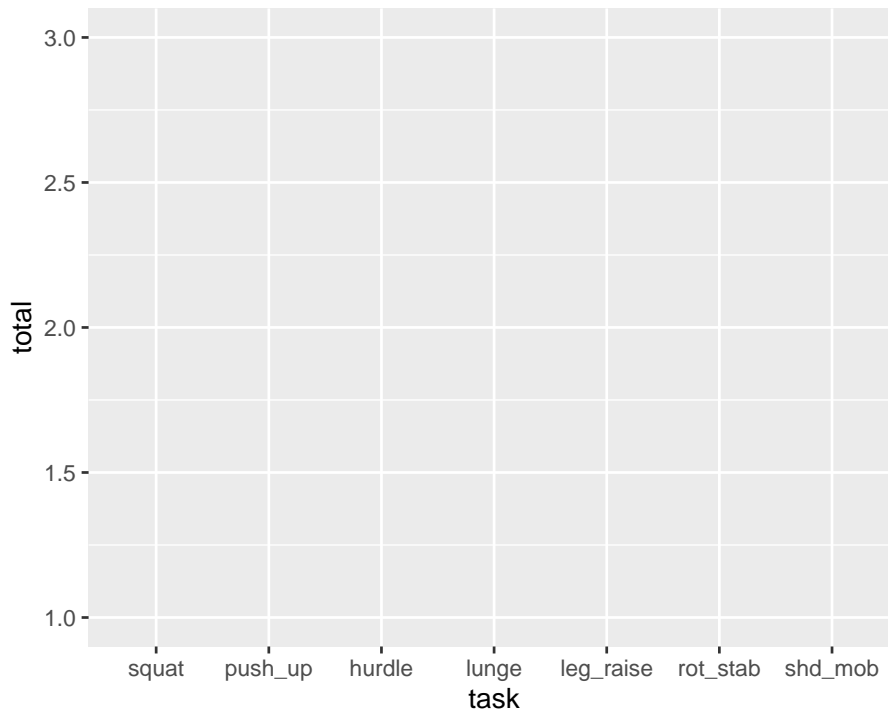


Figure 7.5: Setting the aesthetics

7.2.3 Geometric representations `geom()`

Now we tell the computer what shapes to plot. Given we want a bar plot, we need to specify that the geometric representation (i.e. shape) of the data will be

in the bar form, using `geom_col()`.

Here we are adding a layer (hence the + sign) of points to the plot.

```
ggplot(dat_summ, aes(x = task, y = total)) +  
  geom_col()
```

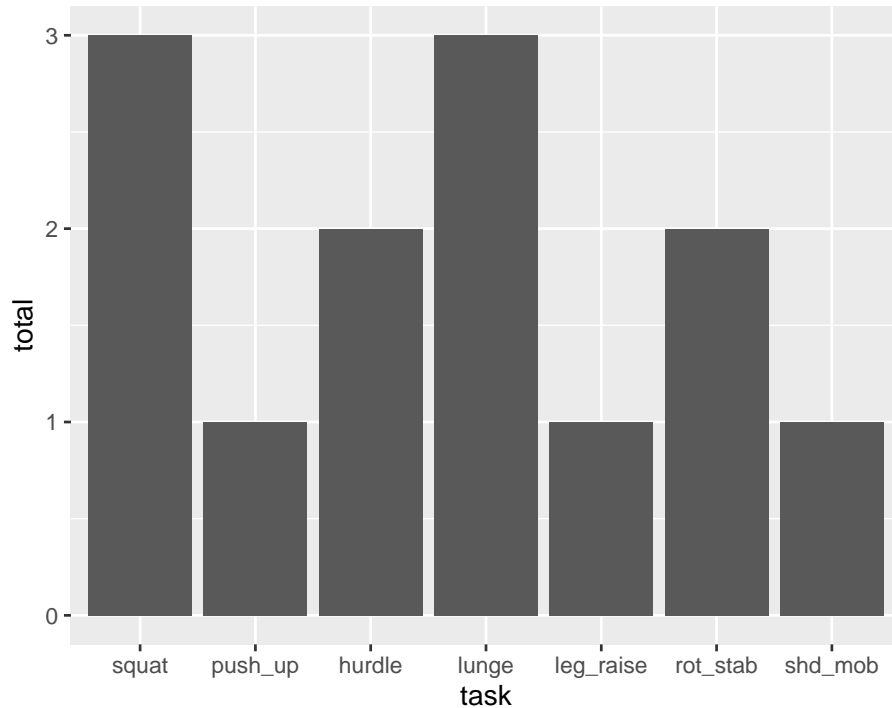


Figure 7.6: Setting the geometric representation

Notice the code difference in Recipe 7.2.3 and Recipe 7.1. I put the `aes()` inside `geom_col()` in Recipe 7.1, but inside `ggplot()` in Recipe 7.2.3. Putting the `aes()` inside `ggplot()` means the aesthetic mapping will trickle down to however many layers of plots you want to overlay your figure with. I will not expand further on this to keep this book simple.

7.3 Grouping Bars Together

7.3.1 Problem

You want to group bars together by a second variable.

7.3.2 Solution

Map a variable to fill, and use `geom_col(position = "dodge")`.

In this example we'll use the `dat` data set, in which we have an FMS score one for each `side`.

We'll map `task` to the x position and map `side` to the fill color (Figure 7.7):

```
ggplot(dat) +  
  geom_col(aes(x = task, y = score, fill = side), position = "dodge")
```

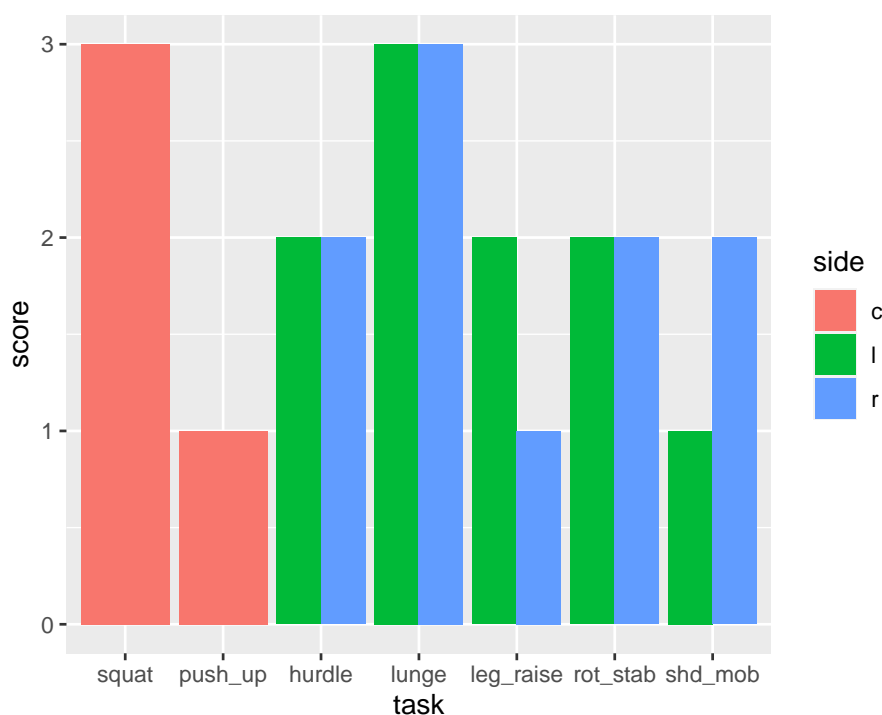


Figure 7.7: Graph with grouped bars

Let's try this example on another dataset `dat_grp_summ`, in which we have the number of subjects who attain a specific FMS `score` for each `task`.

We'll map `task` to the x position, `count` to the y position, and map `score` to the fill color (Figure 7.8):

```
ggplot(dat_grp_summ) +  
  geom_col(aes(x = task, y = count, fill = score), position = "dodge") +
```

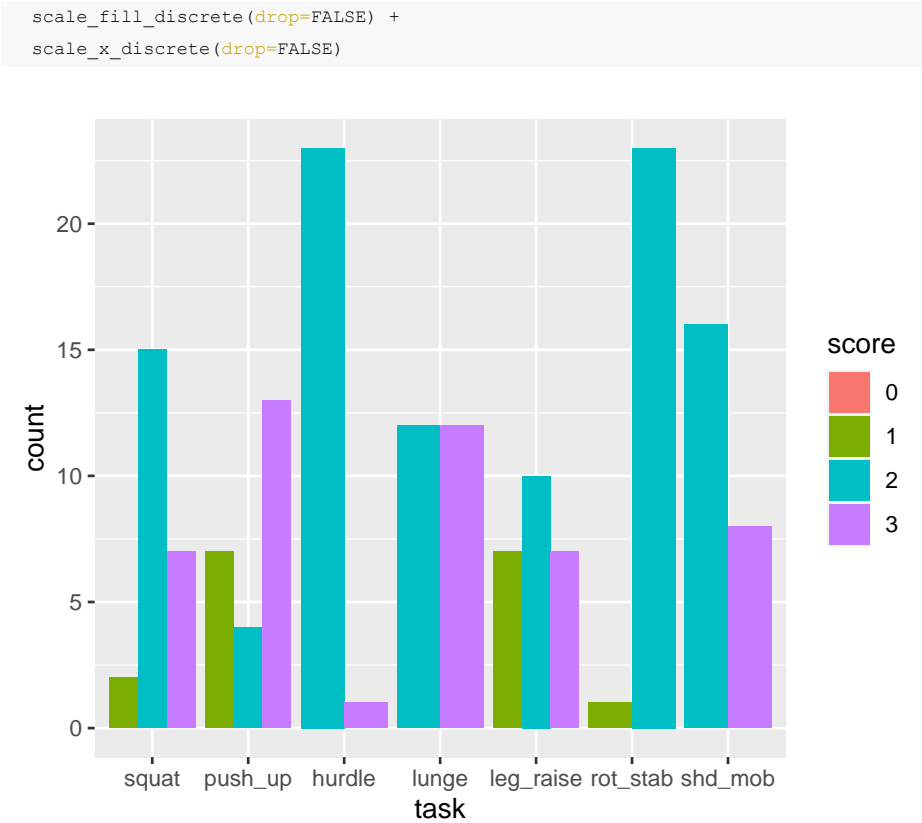


Figure 7.8: Graph with grouped bars

7.3.3 Discussion

The most basic bar graphs have one categorical variable on the x-axis and one continuous variable on the y-axis. Sometimes you'll want to use another categorical variable to divide up the data, in addition to the variable on the x-axis. You can produce a grouped bar plot by mapping that variable to fill, which represents the fill color of the bars. You must also use `position = "dodge"`, which tells the bars to "dodge" each other horizontally; if you don't, you'll end up with a stacked bar plot. Try remove this argument `position = "dodge"`, and see what happens!

As with variables mapped to the x-axis of a bar graph, variables that are mapped to the fill color of bars must be categorical rather than continuous variables.

Other aesthetics, such as `colour` (the color of the outlines of the bars), can also be used for grouping variables, but `fill` is probably what you'll want to use.

7.4 Using Colors in a Bar Graph

7.4.1 Problem

You want to use different colors for the bars in your graph. The default colors aren't the most appealing, so you may want to set them using `scale_fill_manual()`. We'll set the outline color of the bars to black, with `colour="black"` (Figure 7.9).

7.4.2 Solution

Map the appropriate variable to the fill aesthetic (Figure 7.9).

```
ggplot(dat) +  
  geom_col(aes(x = task, y = score, fill = side), position = "dodge", color = "black") +  
  scale_fill_manual(values = c("red", "blue", "green"))
```

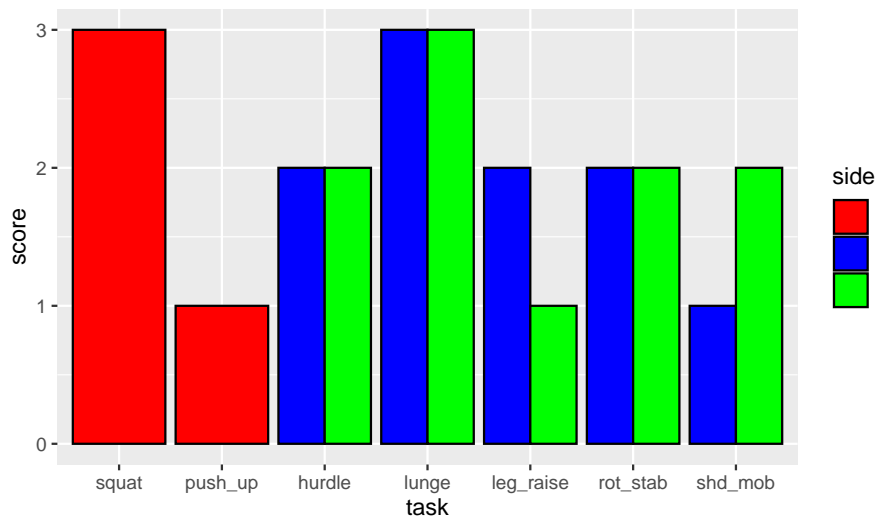


Figure 7.9: Graph with different colors, black outlines, and sorted by percentage change

7.4.3 Discussion

In the variable `side`, there are three values - `c`, `l`, `r`. How does R know if `red` is for what value, and ditto for other colors. Well, if you did not specify the

levels, it goes in alphabetical order. So "red" is for `c`, and "green" is for `r`. See Recipe 6.10 for how to change the order of levels in a factor. There are plethora of color names that is available in R and that you can select to be used in `scale_fill_manual` (Figure 7.10).

7.5 Changing Axes titles in a Bar Graph

7.5.1 Problem

You want to use a different name to label each axis. Some may simply want to use the same names with capitalizations, or totally different names, especially if abbreviations are used in your spreadsheet. For this we will be using the `labs()` function.

7.5.2 Solution

Map the appropriate variable to the fill aesthetic (Figure 7.11).

```
ggplot(dat) +  
  geom_col(aes(x = task, y = score, fill = side), position = "dodge", color = "black") +  
  scale_fill_manual(values = c("red", "blue", "green")) +  
  labs (x = "Tasks",  
        y = "FMS Score")
```

7.6 Changing Legend titles in a Bar Graph

7.6.1 Problem

You want to use a different name for the legend title. Some may simply want to use the same names with capitalizations, or totally different names, especially if abbreviations are used in your spreadsheet. For this we will be using the `labs()` function, and within it the `fill` argument. In this example, the visual component that separates different sides was the fill color, that is why we changed the name of the fill component.

7.6.2 Solution

Map the appropriate variable to the fill aesthetic (Figure 7.12).

#1

white	bisque2	burlywood4	coral4	darkgreen
aliceblue	bisque3	cadetblue	cornflowerblue	darkgrey
antiquewhite	bisque4	cadetblue1	cornsilk	darkkhaki
antiquewhite1	black	cadetblue2	cornsilk1	darkmagenta
antiquewhite2	blanchedalmond	cadetblue3	cornsilk2	darkolivegreen
antiquewhite3	blue	cadetblue4	cornsilk3	darkolivegreen1
antiquewhite4	blue1	chartreuse	cornsilk4	darkolivegreen2
aquamarine	blue2	chartreuse1	cyan	darkolivegreen3
aquamarine1	blue3	chartreuse2	cyan1	darkolivegreen4
aquamarine2	blue4	chartreuse3	cyan2	darkorange
aquamarine3	blueviolet	chartreuse4	cyan3	darkorange1
aquamarine4	brown	chocolate	cyan4	darkorange2
azure	brown1	chocolate1	darkblue	darkorange3
azure1	brown2	chocolate2	darkcyan	darkorange4
azure2	brown3	chocolate3	darkgoldenrod	darkorchid
azure3	brown4	chocolate4	darkgoldenrod1	darkorchid1
azure4	burlywood	coral	darkgoldenrod2	darkorchid2
beige	burlywood1	coral1	darkgoldenrod3	darkorchid3
bisque	burlywood2	coral2	darkgoldenrod4	darkorchid4
bisque1	burlywood3	coral3	darkgray	darkred

Figure 7.10: Names of many colors available in R.

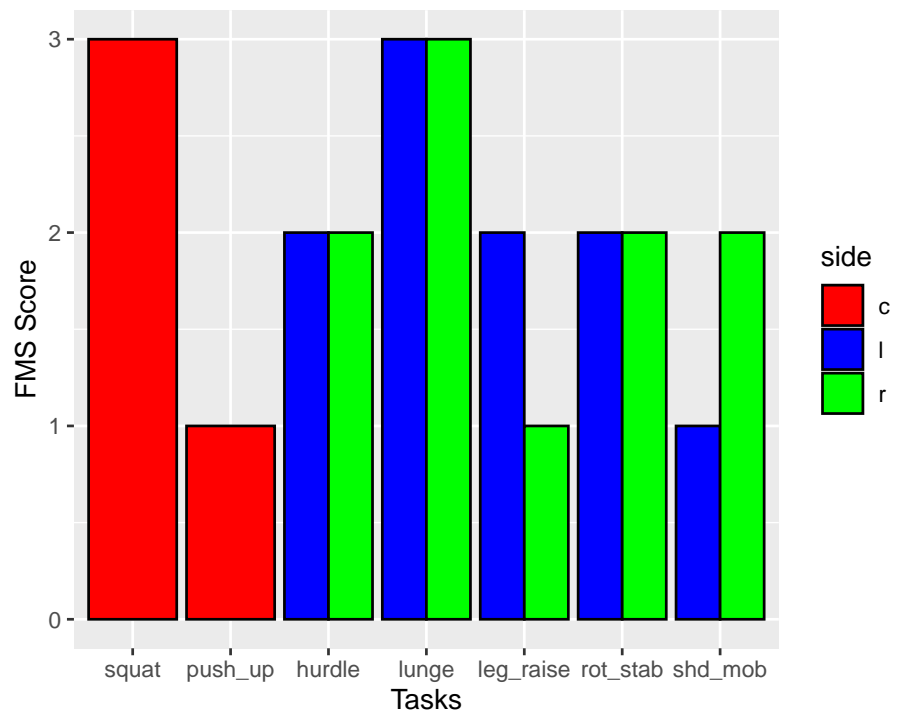


Figure 7.11: Graph different axes titles

7.7. CHANGING FONT SIZE UNIFORMLY ACROSS THE BAR GRAPH 107

```
ggplot(dat) +  
  geom_col(aes(x = task, y = score, fill = side), position = "dodge", color = "black") +  
  scale_fill_manual(values = c("red", "blue", "green")) +  
  labs (x = "Tasks",  
        y = "FMS Score",  
        fill = "Side")
```

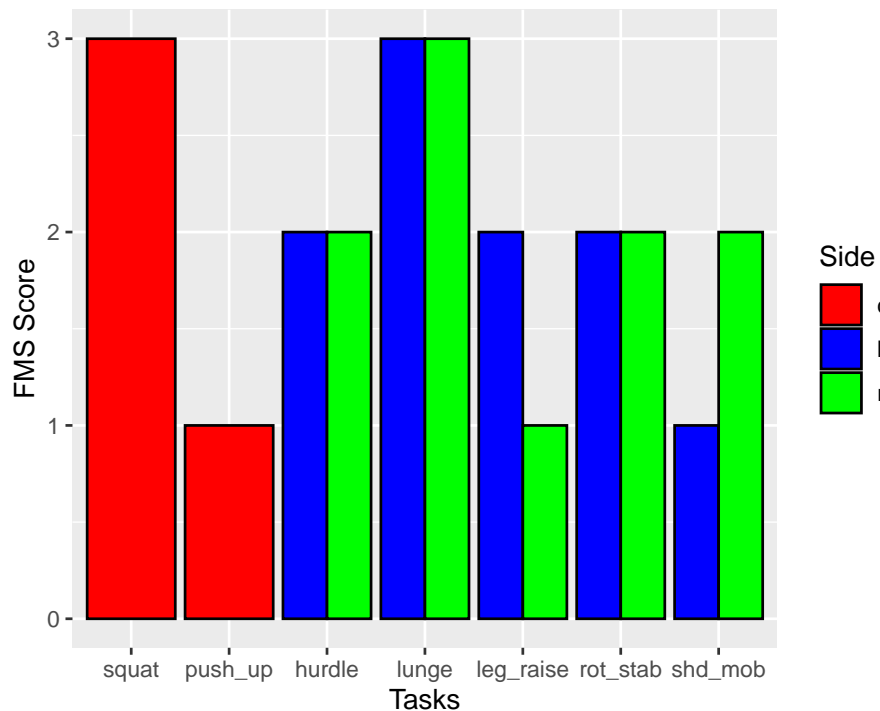


Figure 7.12: Graph different legend title

7.7 Changing font size uniformly across the Bar Graph

7.7.1 Problem

You want to magnify the font size for the axes titles, axes labels, legend title, and legend labels. In this case you can use the `theme(text = element_text(size=))` function. For advance users which is not covered in this book, you can actually custom the fontsize of each and every component to be different.

7.7.2 Solution

Map the appropriate variable to the fill aesthetic (Figure 7.13).

```
ggplot(dat) +
  geom_col(aes(x = task, y = score, fill = side), position = "dodge", color = "black") +
  scale_fill_manual(values = c("red", "blue", "green")) +
  labs(x = "Tasks",
       y = "FMS Score",
       fill = "Side") +
  theme(text = element_text(size = 16))
```

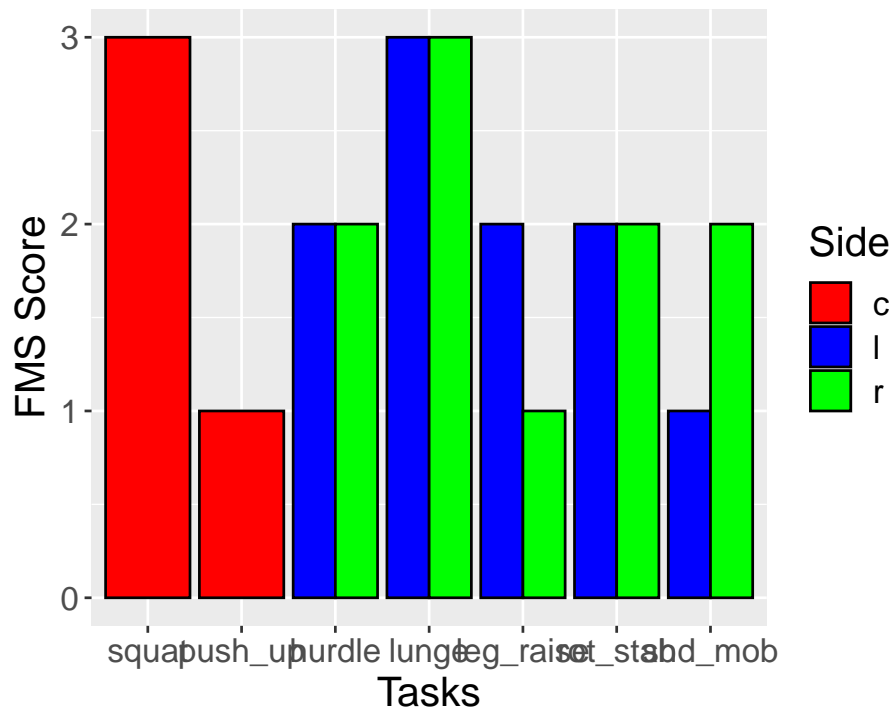


Figure 7.13: Graph with font size = 16

7.8 Outputting to Bitmap (PNG/TIFF) Files

7.8.1 Problem

You want to create a bitmap of your plot, writing to a PNG file.

7.8.2 Solution

We will be using `ggsave()`. First we need to assign the `gplot` we created with `ggplot()` to an object, which we can name anything. Here we call the object simply `f`. There are several important arguments you need. `filename` is the name of the file and extension you want your image to be called. Here we will use `filename = "my_plot.png"`. `plot` is the specific figure you want to save. Here we will use `plot = f`. `width` and `height` allows you to specify how big your image is. `unit` is whether your `width` and `height` are defined in centimeters, "cm", or inches, "in". Here I will use `units = "cm"`, and a 8 cm by 4 cm `width` and `height`, respectively. Lastly, the `dpi` argument specifies the resolution of the image. Here we use `dpi = 300`. The file is saved to the working directory of the session.

```
f <- ggplot(dat) +
  geom_col(aes(x = task, y = score, fill = side), position = "dodge", color = "black") +
  scale_fill_manual(values = c("red", "blue", "green")) +
  labs (x = "Tasks",
        y = "FMS Score",
        fill = "Side") +
  theme(text = element_text(size= 16))

# Default dimensions are in inches, but you can specify the unit
ggsave(filename = "myplot.png",
        plot = f, # the name of the image object you created above.
        width = 8,
        height = 8,
        unit = "cm",
        dpi = 300)
```

7.8.3 Discussion

For high-quality print output, use at least 300 ppi. Figure 7.14 shows portions of the same plot at different resolutions.

R supports other bitmap formats, like BMP, TIFF, and JPEG, but there's really not much reason to use them instead of PNG.

The exact appearance of the resulting bitmaps varies from platform to platform. Unlike R's PDF output device, which renders consistently across platforms, the bitmap output devices may render the same plot differently on Windows, Linux, and Mac OS X. There can even be variation within each of these operating systems.

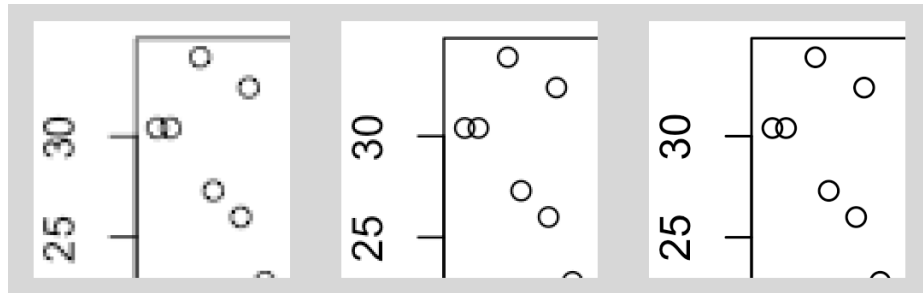


Figure 7.14: From left to right: PNG output at 72, 150, and 300 ppi (actual size)

7.9 Learning check

1. Open up your `practice_script.R`, updated from the learning check in 6.16.
2. Import the Excel file `data/Athlete_1_FMS.xlsx`, and assign it to an object `dat_fms`. See Recipe 5.2.
3. Create another dataset, `dat_summ` where we take the lower of the two scores for tasks which are assessed bilaterally. See Recipe 6.14.
4. Using the data `dat_summ`, create a barplot of `task` as the x axis, and the `total FMS score` as the y axis. See Recipe 7.1.
5. Make the line `colour "red"`, and `fill` the barplot with the colour `"blue"`. See 7.3.
6. Remember to save your file.
7. Download the solution to this learning check below.

[Click to download the solution](#)

Chapter 8

Line Graphs

Line graphs are typically used for visualizing how one continuous variable, on the y-axis, changes in relation to another continuous variable, on the x-axis. Often the x variable represents time, but it may also represent some other continuous quantity, for example, the amount of a drug administered to experimental subjects.

As with bar graphs, there are exceptions. Line graphs can also be used with a discrete variable on the x-axis. This is appropriate when the variable is ordered (e.g., “small”, “medium”, “large”), but not when the variable is unordered (e.g., “cow”, “goose”, “pig”). Most of the examples in this chapter use a continuous x variable, but we’ll see one example where the variable is converted to a factor and thus treated as a discrete variable.

Let us prepare for this chapter by first importing a Vo2 treadmill test data that we have cleaned, and placed in the data folder. See Recipe 6.14. The Excel sheet is called “treadmill_pt4_clean.xlsx”.

```
dat <- read.xlsx (xlsxFile = "data/treadmill_pt4_clean.xlsx",
                  sheet = "Sheet1")

dat <- dat %>%
  mutate (stage = cut_interval(time, length = 210, labels = FALSE)) %>%
  group_by(stage) %>%
  mutate (row_id = row_number()) %>%
  filter (row_id < 37) %>%
  slice_tail (n = 6) %>%
  summarise_at (vars(bf:hr), mean)
```

8.1 Making a Basic Line Graph

8.1.1 Problem

You want to make a basic line graph.

8.1.2 Solution

Use `ggplot()` with `geom_line()`, and specify which variables you mapped to x and y (Figure 8.1):

```
ggplot(dat) +  
  geom_line(aes(x = stage, y = vo2))
```

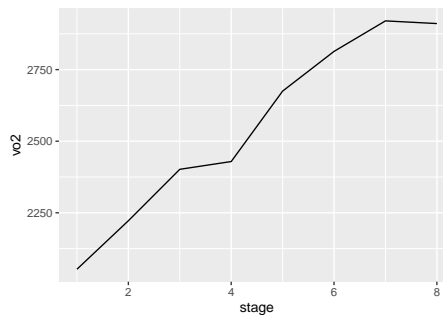


Figure 8.1: Basic line graph

8.1.3 Discussion

In this sample data set, the x variable, `time`, is in one column and the y variable, `vo2`, is in another:

stage	bf	vo2_norm	rer	vo2	vco2	ve	hr
1	30.67	31.97	0.87	2052.83	1777.83	55.17	136.50
2	34.67	34.60	0.88	2222.17	1957.50	59.33	147.00
3	34.67	37.42	0.91	2401.83	2191.33	66.67	161.33
4	38.00	37.83	0.95	2429.00	2312.50	71.17	172.17
5	44.00	41.67	1.00	2675.00	2686.00	85.67	178.50
6	45.67	43.85	1.06	2813.67	2981.17	99.83	185.83
7	54.50	45.50	1.09	2920.17	3195.83	110.33	192.33
8	57.33	45.48	1.05	2910.83	3068.83	112.17	195.17

With `ggplot2`, the default `y` range of a line graph is just enough to include the `y` values in the data. For some kinds of data, it's better to have the `y` range start from zero. You can use `ylim()` to set the range, or you can use `expand_limits()` to expand the range to include a value. This will set the range from zero to the maximum value of the demand column in `BOD` (Figure 8.2):

```
# These have the same result
ggplot(dat) +
  geom_line(aes(x = stage, y = vo2)) +
  ylim(0, max(dat$vo2))

ggplot(dat) +
  geom_line(aes(x = stage, y = vo2)) +
  expand_limits(y = 0)
```

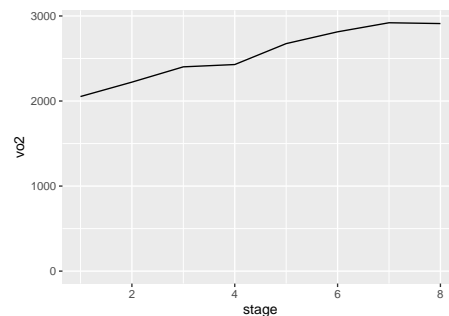


Figure 8.2: Line graph with manually set `y` range

8.2 Making a Line Graph with Multiple Lines

8.2.1 Problem

You want to make a line graph with more than one line.

8.2.2 Solution

In addition to the variables mapped to the `x`- and `y`-axes, map another (discrete) variable to colour or `linetype`, as shown in Figure 8.3:

```
# Bring all variables into one column

dat_long <- dat %>%
  dplyr::select (stage, vo2, vco2) %>%
  pivot_longer(cols = c(vo2:vco2),
               names_to = "var",
               values_to = "val") %>%
  mutate (var = factor (var))

# Map supp to colour
ggplot(dat_long) +
  geom_line(aes(x = stage, y = val, colour = var))

# Map supp to linetype
ggplot(dat_long) +
  geom_line(aes(x = stage, y = val, linetype = var))
```

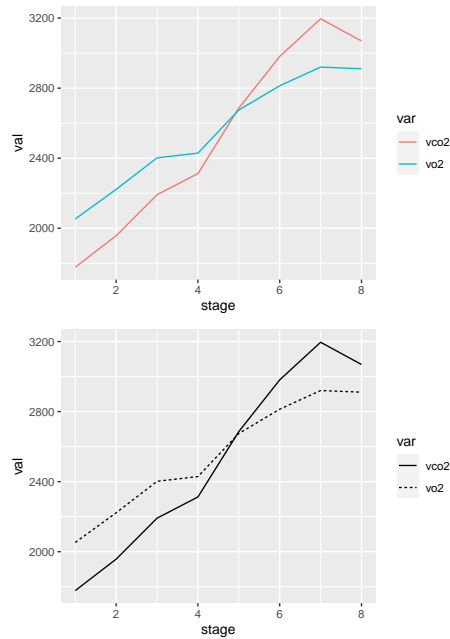


Figure 8.3: A variable mapped to colour (left); A variable mapped to linetype (right)

8.2.3 Discussion

The `dat_long` data has three columns, including the factor `var`, which we mapped to `colour` and `linetype`:

stage	var	val
1	vo2	2052.83
1	vco2	1777.83
2	vo2	2222.17
2	vco2	1957.50
3	vo2	2401.83
3	vco2	2191.33
4	vo2	2429.00
4	vco2	2312.50
5	vo2	2675.00
5	vco2	2686.00
6	vo2	2813.67
6	vco2	2981.17
7	vo2	2920.17
7	vco2	3195.83
8	vo2	2910.83
8	vco2	3068.83

Note

If the x variable is a factor, you must also tell ggplot to group by that same variable, as described below.

Line graphs can be used with a continuous or categorical variable on the x-axis. Sometimes the variable mapped to the x-axis is *conceived* of as being categorical, even when it's stored as a number. In the example here, there are eight values of stage: 1, 2, 3, 4, 5, 6, 7, 8. You may want to treat these as categories rather than values on a continuous scale. To do this, convert `stage` to a factor (Figure 8.4):

```
ggplot(dat_long) +  
  geom_line(aes(x = factor (stage), y = val, colour = var, group = var))
```

To convert a variable to a factor , see also Recipe 6.10.

Notice the use of `group = var`. Without this statement, ggplot won't know how to group the data together to draw the lines, and it will not plot anything:

```
ggplot(dat_long) +  
  geom_line(aes(x = factor (stage), y = val, colour = var))
```

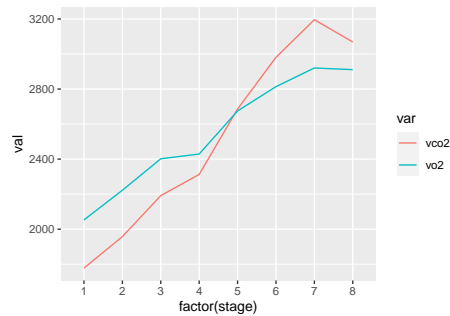
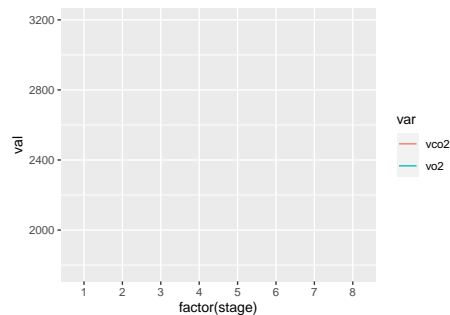


Figure 8.4: Line graph with continuous x variable converted to a factor

```
#> geom_path: Each group consists of only one observation. Do you need to adjust  
#> the group aesthetic?
```



8.3 Changing the Appearance of Lines

8.3.1 Problem

You want to change the appearance of the lines in a line graph.

8.3.2 Solution

The type of line (solid, dashed, dotted, etc.) is set with `linetype`, the thickness (in mm) with `size`, and the color of the line with `colour` (or `color`).

These properties can be set (as shown in Figure 8.5) by passing them values in the call to `geom_line()`:

```
ggplot(dat) +  
  geom_line(aes(x = stage, y = vo2),  
            linetype = "dashed", size = 1, colour = "blue")
```

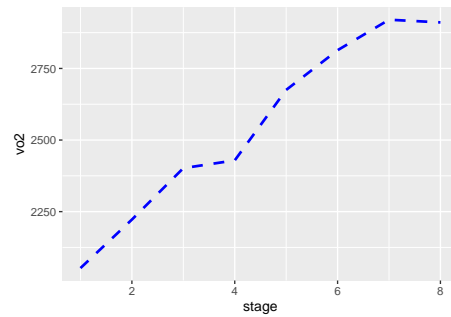


Figure 8.5: Line graph with custom linetype, size, and colour

If there is more than one line, setting the aesthetic properties will affect all of the lines. On the other hand, *mapping* variables to the properties, as we saw in Recipe 8.2, will result in each line looking different. The default colors aren't the most appealing, so you may want to use a different palette, as shown in Figure 8.6, by using `scale_colour_brewer()` or `scale_colour_manual()`:

```
ggplot(dat_long) +  
  geom_line(aes(x = stage, y = val, colour = var)) +  
  scale_colour_brewer(palette = "Set1")
```

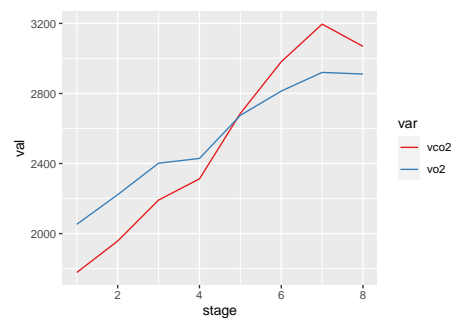


Figure 8.6: Using a palette from RColorBrewer

8.3.3 Discussion

To set a single constant color for all the lines, specify `colour` outside of `aes()`. The same works for `size`, `linetype`, and `point shape` (Figure 8.7). You may also have to specify the grouping variable:

```
# If both lines have the same properties, you need to specify a variable to
# use for grouping

ggplot(dat_long) +
  geom_line(aes(x = stage, y = val, group = var),
            colour = "darkgreen", size = 1.5)

# Since supp is mapped to colour, it will automatically be used for grouping
ggplot(dat_long) +
  geom_line(aes(x = stage, y = val, colour = var), linetype = "dashed") +
  geom_point(aes(x = stage, y = val, colour = var), shape = 22, size = 3, fill = "white")
```

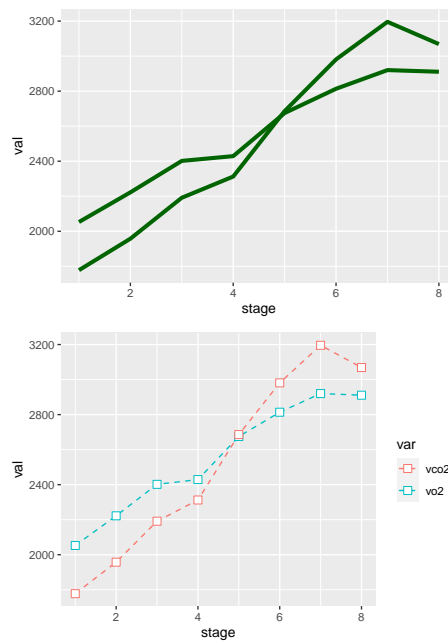


Figure 8.7: Line graph with constant size and color (left); With `supp` mapped to `colour`, and with points added (right)

The range of different linetypes that is available can be seen in (Figure 8.8)

8.4. USING THEMES TO CHANGE OVERALL APPEARANCE OF PLOT 119

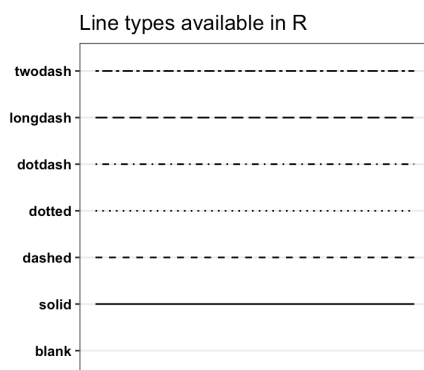


Figure 8.8: The different types of linetype you can use.

8.4 Using Themes to Change Overall Appearance of Plot

8.4.1 Problem

You want to use premade themes to control the overall plot appearance.

8.4.2 Solution

There are many premade themes that are already included in `ggplot2`. The default `ggplot2` theme is `theme_grey()`, but the examples below also showcase `theme_bw()`, `theme_minimal()`, and `theme_classic()`.

To use a premade theme, add `theme_bw()` or another theme to your plot (Figure 8.9):

```
# Create the base plot
hw_plot <- ggplot(dat) +
  geom_line(aes(x = stage, y = vo2))

# Grey theme (the default)
hw_plot +
  theme_grey()

# Black-and-white theme
```

```
hw_plot +  
  theme_bw()  
  
# Minimal theme without background annotations  
  
hw_plot +  
  theme_minimal()  
  
# Classic theme, with axis lines but no gridlines  
  
hw_plot +  
  theme_classic()
```

8.5 Learning check

1. From your learning check in 7.9, open up your `practice_script.R`.
2. Import the Excel file `data/Athlete_1_treadmill.xlsx`, and the sheet `raw`, and assign it to an object `dat`. See Recipe 5.2. Also import the the sheet `stage` from the same file, and assign it to an object `dat_stage`.
3. Rename the columns of `dat` to be `("time", "bf", "vo2_norm", "rer", "vo2", "vco2", "ve", "hr")`, respectively. See Recipe 6.2.
4. Remove the first row of the `dat` data, as it is useless. See Recipe 6.3.
5. Convert the variable `time` of `dat` into numeric seconds, and all other variables to numeric. See Recipe 6.5 and 6.4, respectively.
6. Create a variable called `stage`, where each stage represents 3:30 min worth of VO2 data. See Recipe 6.6.
7. For each stage, create a variable called `row_id`, which essentially represents the number of 5 sec windows in each stage. See Recipe 6.11 and 6.14.
8. For each stage, remove the data between 3 to 3:30min. See Recipe 6.14 and 6.12.2.
9. For each stage, keep the last six rows of data. See Recipe 6.14 and 6.3.
10. For each stage, calculate for all variables for the mean of the last six rows of data. See Recipe 6.14 and 6.13. Assign this table to an object called `dat_summ`.

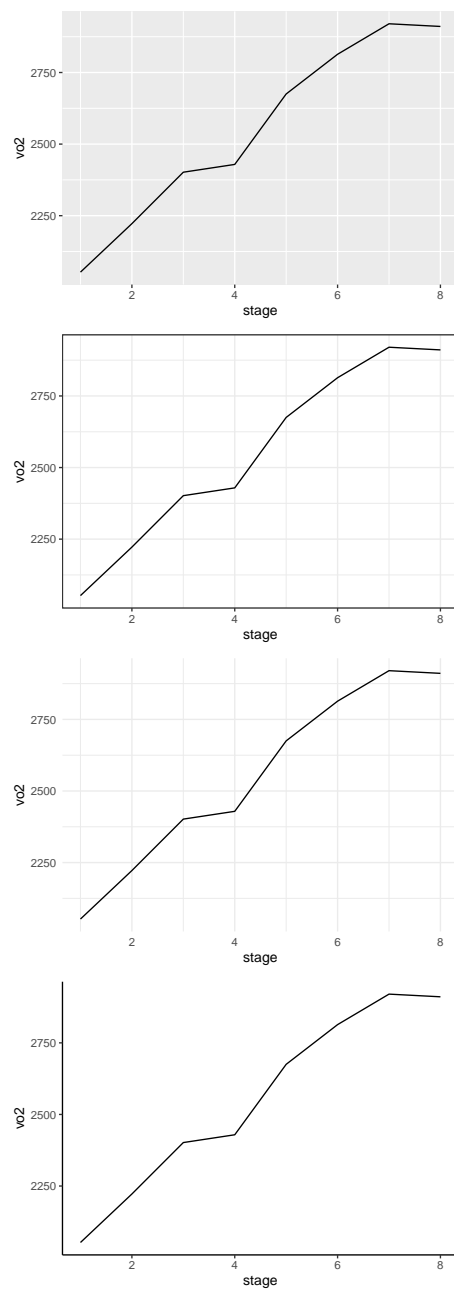


Figure 8.9: Scatter plot with `theme_grey()` (the default, top left); with `theme_bw()` (top right); with `theme_minimal()` (bottom left); with `theme_classic()` (bottom right)

11. Merge the two tables `dat_summ` and `dat_stage`, by the common variable called `stage`. See Recipe 6.15. Assign this table to an object called `df_plot`.
12. Using `df_plot`, make a line plot with `speed` as the x axis, and `vo2` as the y axis, and assign it to an object called `f`. See Recipe 8.1.
13. Save image `f` to a file called `myfig.png`. See Recipe 7.8.
14. You must be wondering, why can't I see the figure. What must you do to print the picture? See Recipe 2.3.
15. Download the solution to this learning check below.

[Click to download the solution](#)

Chapter 9

Your Assignment

This assignment is designed to create a graphs to visualize the results of the FMS and Incremental Treadmill Test analyses. There are two things to note:

1. With the graphs created, export it and paste into your word assignment document.
2. Submit the code together with your assignments onto FASER.

The assignment is **EASY!!!** I am only asking you to copy and paste relevant recipes which you have learned, and edit relevant values in the function's arguments. For example, when you see the value `xxx`, I am asking you to replace it with the relevant value. I am NOT going to ask you to create codes from scratch. I am not cruel.

To do your assignment, follow these steps:

1. In your desktop's `se201` folder, created in learning check 4.2.
2. Create a project called `assignment` inside the folder. See Recipe 4.2.
3. Download the R script in this link `assignment_analysis.R`. Save this script into the `se201/assignment` folder. When you open it, you should see the codes below.

Click to download the script for assignment

4. Inside `se201/assignment` folder, create a `data` folder. Put all your raw Excel data into the `data` folder. **All raw Excel data for this assignment will be on SE201 Moodle>Assessment Information.**

9.1 Tasks to complete

1: Import data. Replace `xxx` with your Excel file's name and appropriate sheet name. See Recipe 5.2.

2: Group FMS summary. Find the number of athletes who scored each level in each FMS tasks. See Recipe 6.14.

3: Group FMS barplot. Plot a bar graph of `task` as the x axis, and `count` as the y axis, `fill` colour set to the different FMS `tasks`. Give the graph a succinct title. Make the axis texts and titles to be font `size` 12. Replace `xxx` with the appropriate values. See Recipe 7.3. Save the plot - See Recipe 7.8.

4: Individual FMS barplot. Plot a bar graph of `task` as the x axis, and `score` as the y axis, `fill` colour set to the different tested `side`. Give the graph a succinct title. Make the axis texts and titles to be font `size` 12. Replace `xxx` with the appropriate values. See Recipe 7.3. Save the plot - See Recipe 7.8.

5: Rename columns. Rename all columns to lower cases, without white spaces. See Recipe 6.2.

6: Make characters to numeric. Replace `xxx` with the appropriate function to convert all columns apart from time from character to numbers. See Recipe 6.4.

7: Find the average 30 sec value (between 2:30 to 3:00min) per stage for all variables. Replace `xxx` with the appropriate values. See Recipe 6.14.

8: Combine two tables. Merge the average data from *Task 4* with the object called `dat_stage` along the common header. Replace `xxx` with the appropriate values. See Recipe 6.15.

9: Make a lactate graph. Plot a line graph of `speed` as the x axis, and `lactate` as the y axis, with the line colour `blue`. Give the graph a succinct title. Make the axis texts and titles to be font `size` 12. Replace `xxx` with the appropriate values. See Recipes 8.1, 8.3. Save the plot - See Recipe 7.8.

10: Make a heart rate graph. Plot a line graph of `speed` as the x axis, and `hr` as the y axis, with the line colour `blue`. Give the graph a succinct title. Make the axis texts and titles to be font `size` 12. Replace `xxx` with the appropriate values. See Recipes 8.1, 8.3. Save the plot - See Recipe 7.8.

11: Export the `dat_vo2_comb` dataframe to an excel table. This result can be used to fill in the “stage by stage results” in your assignment under incremental treadmill test. Replace `xxx` with the name of the dataframe you are wanting to export. See where the data is being exported into. See Recipe 5.3.

9.2 Codes

This is the codes you will see in the script downloaded.

```
## -----
##
##
## Author: Put your name
##
## Date Created: Put the date
##
##
## -----
##
## Notes:
##
##
## -----

## -----

## load up the packages we will need

if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse, # All purpose wrangling for dataframes
               lubridate, # Time
               tibbletime,
               openxlsx) # writing excel documents

## Custom function to get interection between two lines
## To get Lactate and Anaerobic Threshold values
## Ignore the complexity, highlight between Start and End, Run -----
## Start -----
curve_intersect <- function (curve1, curve2, empirical = TRUE, domain = NULL)
{
  if (!empirical & missing(domain)) {
    stop("'domain' must be provided with non-empirical curves")
  }
  if (!empirical & (length(domain) != 2 | !is.numeric(domain))) {
    stop("'domain' must be a two-value numeric vector, like c(0, 10)")
  }
  if (empirical) {
    curve1_f <- approxfun(curve1$x, curve1$y, rule = 2)
    curve2_f <- approxfun(curve2$x, curve2$y, rule = 2)
    point_x <- uniroot(function(x) curve1_f(x) - curve2_f(x),
```

```

        c(min(curvel$x), max(curvel$x)))$root
    point_y <- curve2_f(point_x)
  }
  else {
    point_x <- uniroot(function(x) curvel(x) - curve2(x),
                      domain)$root
    point_y <- curve2(point_x)
  }
  return(list(x = point_x, y = point_y))
}

## End -----

## Import data (Task 1)

### Import the sheet with the Vo2 data
dat_vo2 <- read.xlsx (xlsxFile = "data/XXX.xlsx",
                     sheet = "XXX")

### Import the sheet with the lactate data
dat_stage <- read.xlsx (xlsxFile = "data/XXX.xlsx",
                       sheet = "XXX")

### Import group FMS data
dat_fms_grp <- read.xlsx (xlsxFile = "data/XXX.xlsx",
                         sheet = "XXX")

### Import individual FMS data
dat_fms_indv <- read.xlsx (xlsxFile = "data/XXX.xlsx",
                          sheet = "XXX")

## Analyze FMS data -----

##### Group FMS #####

### Number of athletes scoring a level in FMS (Task 2)

dat_fms_grp <- dat_fms_grp %>% # original data
  pivot_longer(cols = -id,
               names_to = "task",
               values_to = "score") %>%
  group_by(XXX, XXX) %>%
  summarize (count = n()) %>%
  mutate (count = factor (count),
          score = factor (score, levels = c("0", "1", "2", "3")))

```

```

### Plot group FMS (Task 3)

plot_fms_grp <- ggplot (XXX) +
  geom_col(aes(x = XXX, y = XXX, fill = XXX), position = "dodge") +
  scale_fill_discrete(drop=FALSE) +
  scale_x_discrete(drop=FALSE)

ggsave(filename = "grp_fms.png",
  plot = plot_fms_grp , # the name of the image object you created above.
  width = 8,
  height = 8,
  unit = "cm",
  dpi = 300)

##### Individual FMS #####

### Plot group FMS (Task 4)

plot_fms_indv <- ggplot(XXX) +
  geom_col(aes(x = XXX, y = XXX, fill = XXX), position = "dodge")

ggsave(filename = "ind_fms.png",
  plot = plot_fms_indv, # the name of the image object you created above.
  width = 8,
  height = 8,
  unit = "cm",
  dpi = 300)

## Analyze VO2 data -----

### Rename column names of Vo2 data (Task 5)

new_names <- c()

colnames (dat_vo2) <- new_names

### Remove first row of Vo2 data

dat_vo2 <- dat_vo2 %>%
  slice (-c(1))

### Convert column type of Vo2 data (Task 6)

dat_vo2 <- dat_vo2 %>%

```

```

mutate (bf = XXX (bf),
        vo2_norm = XXX(vo2_norm),
        rer = XXX(rer),
        vo2 = XXX(vo2),
        vco2 = XXX(vco2),
        ve = XXX(ve),
        hr = XXX(hr)) %>%
# Convert time to seconds
mutate (time = time %>%
        str_squish() %>%
        ms() %>%
        as.period(unit = "sec") %>%
        as.numeric ())

### Create a "stage" variable in Vo2 data

dat_vo2 <- dat_vo2 %>%
  mutate (stage = cut_interval(time, length = 210, labels = FALSE))

### Calculate average 30s data per stage of Vo2 data (Task 7)

dat_vo2_summ <- dat_vo2 %>%
  group_by(XXX) %>% # for each group
  mutate (row_id = row_number()) %>%
  filter (row_id < XXX) %>% # throw away all data between 3 to 3:30 min
  slice_tail (n = XXX) %>% # keep last 30 sec per stage
  summarise (bf = mean (bf),
             vo2_norm = mean (vo2_norm),
             rer = mean (rer),
             vo2 = mean (vo2),
             vco2 = mean (vco2),
             ve = mean (ve),
             hr = mean (hr))

### Combine Vo2 staged data with lactate data (Task 8)

dat_vo2_comb <- dat_vo2_summ %>%
  inner_join(dat_stage, by = "XXX")

### Plot

#### Lactate (Task 9)

f <- ggplot (dat_vo2_comb) +

```



```

geom_line (aes (x = XXX, y = XXX), colour = "XXX", size = 1.5) +
labs (x = "Speed (km/h)",
      y = "Lactate (mmol)") +
theme_bw() +
labs (title = "XXX") +
theme(axis.text.x = element_text(size = XXX),
      axis.text.y = element_text(size = XXX),
      axis.title.x = element_text(size = XXX),
      axis.title.y = element_text(size = XXX))

ggsave(filename = "lactate.png",
       plot = f, # the name of the image object you created above.
       width = 8,
       height = 8,
       unit = "cm",
       dpi = 300)

#### Heart rate (Task 10)

f <- ggplot (dat_vo2_comb) +
  geom_line (aes (x = XXX, y = XXX), colour = "XXX", size = 1.5) +
  labs (x = "Speed (km/h)",
        y = "Heart Rate (bpm)") +
  theme_bw() +
  labs (title = "XXX") +
  theme(axis.text.x = element_text(size = XXX),
        axis.text.y = element_text(size = XXX),
        axis.title.x = element_text(size = XXX),
        axis.title.y = element_text(size = XXX))

ggsave(filename = "heartrate.png",
       plot = f, # the name of the image object you created above.
       width = 8,
       height = 8,
       unit = "cm",
       dpi = 300)

#### Export table (Task 11)

write.xlsx(x = XXX,
          sheetName = "vo2",
          file = "data/vo2_table.xlsx")

## Bonus Codes to help you -----
### Click on everything below and run

```

```

### Get VO2 max , VO2 max relative

rolling_mean6 <- rollify(mean, window = 6)

raw_roll <- dat_vo2 %>%
  arrange (desc (time)) %>%
  mutate_at (vars(bf:hr), rolling_mean6) %>%
  na.omit() %>%
  summarise_at(vars(bf:hr), max, na.rm = TRUE) %>%
  mutate (vo2 = vo2/1000)

cat ("The relative peak O2 uptake is:", raw_roll$vo2_norm, "(ml/kg/min)")
cat ("The absolute peak O2 uptake is:", raw_roll$vo2, "(L/min)")

### Get max aerobic speed

#### Get maximal stage completed number
max_stage <- max (dat_vo2$stage)

#### Find number of rows in final stage
dat_vo2_last <- dat_vo2 %>%
  filter (stage == max_stage)

#### Calculate based on rows the proportion of stage completed
last_stage_prop <- nrow (dat_vo2_last)/42

#### Get the speed increment, which should be even across all stage
increment <- mean (diff(dat_vo2_comb$speed, lag = 1))

#### Get the maximal aerobic speed
ans <- last_stage_prop * increment
max_aerobic_speed <- ans + max (dat_vo2_comb$speed)

cat ("The maximal aerobic speed:", max_aerobic_speed, "(Km/h)")

### Get Lactate and Anaerobic Threshold

m <- loess (lactate ~ speed, data = dat_vo2_comb)
min.speed <- ceiling (min (dat_vo2_comb$speed))
max.speed <- floor (max (dat_vo2_comb$speed))
n_points <- 100

new_lac <- data.frame (speed = seq (min.speed, max.speed, length.out = n_points))

```

```

new_lac$y <- predict (m, newdata = new_lac)
colnames(new_lac)[1] <- "x"

#### Threshold values

lactate_thres <- data.frame (x = seq (min.speed, max.speed, length.out = n_points),
                             y = 2)

anaerobic_thres <- data.frame (x = seq (min.speed, max.speed, length.out = n_points),
                              y = 4)

#### Speed at threshold

speed_at_lac_thres <- curve_intersect(new_lac, lactate_thres)$x
speed_at_ane_thres <- curve_intersect(new_lac, anaerobic_thres)$x

### Get HR at thresholds

m <- loess (hr ~ speed, data = dat_vo2_comb)

new_hr <- data.frame (speed = seq (min.speed, max.speed, length.out = n_points))
new_hr$y <- predict (m, newdata = new_hr)
colnames(new_hr)[1] <- "x"

hr_at_lac_thres <- new_hr[which.min(abs(new_hr$x - speed_at_lac_thres)), "y"] %>%
  round (0)
hr_at_ane_thres <- new_hr[which.min(abs(new_hr$x - speed_at_ane_thres)), "y"] %>%
  round

pi <- data.frame(Variable = c("Lactate Threshold", "Anaerobic Threshold"),
                 Speed = c(speed_at_lac_thres, speed_at_ane_thres ),
                 HR = c(hr_at_lac_thres, hr_at_ane_thres))

cat ("The speed (km/h) and heart rate (b/min) at lactate threshold are:", pi[1,2], "and", pi[1,3])
cat ("The speed (km/h) and heart rate (b/min) at anaerobic threshold are:", pi[2,2], "and", pi[2,3])

df_plot <- data.frame(speed = new_lac$x,
                      lactate = new_lac$y,
                      hr = new_hr$y,
                      speed_lac = speed_at_lac_thres,
                      speed_ane = speed_at_ane_thres) %>%
  pivot_longer(cols = lactate:hr,
               names_to = "var",
               values_to = "val")

```

```
ggplot (df_plot) +  
  geom_line (aes (x = speed, y = val)) +  
  geom_vline(xintercept = speed_at_lac_thres, color = "blue", linetype = "dashed") +  
  geom_vline(xintercept = speed_at_ane_thres, color = "red", linetype = "dashed") +  
  facet_wrap(~var, ncol = 2, scales = "free") +  
  labs (x = "Speed",  
        y = "Values") +  
  theme_bw() +  
  labs (title = "Plot of Treadmill test") +  
  theme(axis.text.x = element_text(size = 12),  
        axis.text.y = element_text(size = 12),  
        axis.title.x = element_text(size = 16),  
        axis.title.y = element_text(size = 16))
```