Basics of R for 6020PSY Students | A brief, low pain intro to just doing the stats

Stefano Occhipinti

2020-04-25

Contents

/raggedright

1 Module 1: Prelude and Introduction

1.1 Is this for you?

R is an amazingly powerful, open-source statistical computing environment, *that also can be used for data analysis*. You have probably at least heard about R. Today, knowing R is also a very well-regarded professional skill. Possessing demonstrated skills in working with R (usually but not always in *R Studio*) is a very strong addition to your CV, especially coming from a social/behavioral sciences background. People who have done a psychology or related degree, especially to honours level, are expected to know quite a bit about applied statistics and methodology. There are numerous cases of psychology graduates getting a foothold into an organisation (e.g., a government department, a not-for-profit, or a private consultancy) by landing a research-oriented, entry level job. I can tell you that if anyone asks me - an academic who teaches statistics in psychology - about recent graduates, they now almost *always* want to know if they have R skills (also, in public health it's good to develop some Stata skills). SPSS skills don't really differentiate people anymore. R is not that hard to learn. You don't have to become a programmer.

1.2 Why this guide?

There are many, many R tutorials and video courses (e.g., Datacamp, plain YouTube) out there. Many are free. Some are quite good. If you already have a background in programming or computing generally, you will probably get a lot out of such materials but you probably are not even reading this guide. You're very welcome, but it's not intended for you. Matt Stainer and I (and originally also Caley Tapp) decided to set up workshops leading to the present guide because nothing really met the requirements of psych students and researchers like the anticipated typical users of this guide.

- Most courses spend a great deal of time on the programming basics and it can take a long time, possibly by which the initially keen learner has given up, to get to the productive statistics.
  - N.b., I learned R a lot like this from what *could* be seen as a very dry book with Notepad on Windows and I enjoyed it! But I don't expect *you* to. And I'm no programmer!
- Much of the time, free courses begin by generating random variables within R itself and running very simple descriptives and increasingly pretty plots on them and this is of almost zero utility for someone who needs to analyse a *real* dataset that they already have, usually in Excel form or in another stats package form (and who has plenty of experience with data already).
  - I suspect the current popularity of data science as latest world saviour has attracted many non-statistical data visualisers for whom this approach might be beneficial.
- Few courses spend much time addressing the practicalities of dealing with different variable types in data frames and that is a pressing skill that you need to have *before* you can do the analyses you need. It's astonishingly easy to read an SPSS file using the `foreign` package but what you get at the other end *can* be confusing and frustrating if you don't know a few basic principles.
  - You already have a strong schema for what datasets look like. You don't need baby data so you can discover a frequency polygon.
  - E.g., the public domain passenger manifest of the Titanic, the actual ship from 1912, is given a furious beating in many of these.
- In sum, most courses are not geared for people like you who already know much more advanced statistics than the general population but need to know enough R to work productively and efficiently. As a consequence, the balance between R programming per se and practical, applied statistics is out of whack.
  - How many more times do we need to learn that men on the lower decks of the Titanic were most likely to drown!?

Assumptions of this guide. I assume that you know or are learning:

- descriptive statistics and basic inferentials such as a t-test
- ordinary and hierarchical multiple regression
- ANOVA models (oneway, factorial, repeated measures, mixed between-within Ss (therefore, basic GLM analyses)
- ideally, you will have done some SPSS syntax coding (such as in 3003PSY) so you understand that data analysts use code to give instructions to stats software so that the desired analyses will be performed on the variables
- N.b., if you can do SPSS syntax as I use in 6020PSY at MG, you can definitely do R and even if you haven't done SPSS to that level, you will still find R code very logical and clear once I go through the basics

The points I will come back to repeatedly across modules in this guide (they are like themes) are:

1. It's actually very easy to do most statistics that you already know in R
2. The part that is a little taxing *at first* is how to do the ancillary but necessary tasks, such as reading in the data, addressing variables in datasets, or putting functions together to tailor analyses to your requirements. For some, this may result in the early parts of learning R feeling like they drag on a bit.
3. However…

i. you don't need to learn *everything* (or even most things) about programming R to get going with it;
ii. the new concepts you learn (e.g., nested functions) might take a while at first if you have never thought like that but then you'll see that you will be applying them to almost all of your later work and that *saves* time
iii. R is very modular and once you have learned a thing you can plug it in almost anywhere without having to re-invent the wheel

On this note, the modules in this guide show ways to do a set of standard tasks fairly typical of a 3rd or 4th year psychology context, using *base R*. This generalises to quite a lot of data analyis and will certainly give you a strong edge over many other people who might lack your statistics and methodology training. There are many different ways to do what we cover here. I don't even touch on the *tidyverse* that is currently very popular (although there will be a *ggplot()* module from Matt Stainer). *I'm showing you some skills in base R so that you can understand R things and concepts.* In future, you will be able to understand and implement new R tips and great new packages and functions that you'll see around so much more quickly.

1.3 About R

R is open source and free. Open source software is that for which the source code is freely available to and modifiable by others for their particular requirements. This is regarded as a benefit because it helps the software to grow in capabilities and helps errors to be checked and addressed by a sort of crowd sourcing. The other meaning of *free* in this sense is free of dollar cost (to coin a phrase often used online about R: *free, as in free beer*). It has helped R grow its user base exponentially as many smaller companies and start ups have adopted it because their only costs are in expertise, not necessarily prohibitive software licensing. The software's increasing power has also meant that many companies and institutions of all sizes have adopted R. R is available for all the major operating systems (i.e., Unix/Linux, Windows, MacOS) and looks essentially the same on all of them.

People use R extensively in statistical programming. Some of the many things they may use R for are:

- to *simulate data*. Statisticians often need to simulate data with specific characteristics (e.g., normally distributed; skewed; or with very low or very high proportions of measurement error etc) to demonstrate statistical points. This is very easy to do in R.

- to *run Monte Carlo studies*. In connection with simulation, Monte Carlo studies examine how statistics work with known distributions or in known conditions. (E.g., David MacKinnon and colleagues reported on a Monte Carlo study that suggested that bootstrapping approaches had optimal statistical characteristics in assessing mediation and later that work was taken up by Preacher and Hayes.)
- to do *bootstrapping* per se. As discussed above, bootstrapping is increasingly used in everyday statistical analyses such as mediation. But that is a tiny part of its usage and R allows extensive bootstrapping of many models.
- to *build new apps* on top of R. You may have heard of *jamovi* and *JASP*. These are menu-based stats apps that use R as the "engine" to run the actual analyses. I won't be covering these here as they have their own help and instructional materials. As useful as those programs are, it is much more important at your level to gain an understanding of base R and, frankly, it's not that much harder for honours students!

But some people, maybe like you, just want to read in some data and run statistical analyses like multiple regression using R. And **that's** where this guide comes in. I won't be covering programming in depth here - I couldn't begin to do that justice. I merely want to give you enough of the relevant concepts to get you rolling with actually doing the stats in R. R's great control comes at a cost of needing to be *more specific and detail-oriented than with other general purpose (GP) statistical software such as SPSS or SAS*. In this guide, I will try to impart the few specific R skills that will allow you to manipulate data and files just enough to run and report on analyses efficiently. I am not presenting a dumbed down version of R. There are plenty of learning materials about that may fulfill that dubious role. I am presenting a carefully guided version of R materials that speaks to people like you who have spent some time learning statistics with a typical GP program like SPSS. You know that computers can be used for statistics and you know what data looks like (i.e., which are the cases or participants and which are the variables). What you don't know is how to read a typical data file into R and start doing the stats you already understand, more or less. As you master those basics, you will free up cognitive space so that if you want to pursue an advanced facility with R, you will be able to and I certainly encourage you to. There are *many* materials freely available or at reasonable cost that will take you very far with R. At our basic workshops, we assure participants that if they were to go out and learn enough R to come back and teach *us* new things, we would consider that a win.

Generally, once you have learned to do a particular task in R you will have created a template from your own or others' code that you can reuse again and again with appropriate modifications on new data in different situations. You won't be beginning from scratch every time you sit down with R. You won't even need to learn to program to run a regression! In fact…
`lm(Y_var~X1 + X2 + X3, data=datafile)`… there you are… That's a re-

gression!

1.4 Brief outline

By the end of the guide in the current form (up to ANOVA), the following topics will have been covered:

- What is R
  - learning R is about changing the way you think about running stats a little
- R Studio
  - an important program that helps to control R
- R Project files
  - a special way of organizing projects within R Studio
- Reading data into R
  - some tricks
  - `here` package and why it is so fundamental in a learning environment
- Running basic descriptives in R
  - Describing a dataset and describing a specific set of variables
- Performing typical OLS regression diagnostics
  - labelling and dropping cases
- Running OLS regression and its variations
  - transformations
  - moderation
  - mediation
- Brief guide to implementing ANOVA in R
  - there are some controversies here…

N.b., as more modules are added this outline will be expanded accordingly. This is a dynamic document.

2 Module 2: Basics

2.1 Tips for the road ahead

The initial parts (including the present one) are longer than later ones. This is by design. To touch on points discussed in more detail in Module 1, this guide is designed for people who have studied or are studying statistics and methodology in a typical undergraduate sequence in psychology (see *Assumptions* in Part 1). You will already have a schema for a data file and other statistical concepts. You will most likely have used another statistics package. At Griffith (as at many other places) that will be SPSS. However, the principles (e.g., differences between such packages and R) will be the same. The aim of this guide is to equip you with enough of the basic tools (and the *understanding* to use them) required to do the stats you already know or are in the process of learning. For this reason, most of the functions are from `base R` or very widely used packages, such as `car`. Having mastered these, people who need and want to go on to learn to use contemporary tools such as the whole `tidyverse` can do so and will probably find it easier to pick up the new packages and functions. **It is safe to assume that every skill shown in this guide will have at least 1 or 2 different ways to be implemented**. This is how R works. Once you have the key, *you* can decide for yourself which packages and approaches to use or you can adapt to the norms of a new lab or workplace.

Because some of the concepts covered in this module of the guide, such as *objects* and *functions*, are so fundamental (i.e., they will be applied to virtually everything you will do with R), I will spend a comparatively long time and devote plenty of words to them. This will make this first module seem to go a little slowly. My advice is to take it at exactly the pace that feels comfortable to you (as quickly or as slowly as that is). I've done this because you are honours students and you can devote your cognitive space to understanding the concepts. You will then continue to learn them by putting them into practice in almost every later piece of R code you write or run from this point. You may return to consult this module while doing later ones. The benefit of spending time discussing these concepts here is that there is much less that needs to be covered in later modules and I there can spend almost all the time (and your concentration) discussing how to do the actual stats, rather than the intricacies of getting R to send plots to PDF files**.

2.2 R versus SPSS and other GP stats software

SPSS is similar to many other GP packages (e.g., some widely used ones are SAS and Stata). Of these, SPSS is by far the least flexible, but they are all a bit constrained and this is by design. R has taken a different path. By contrast, GP software is almost totally geared to doing stats in a basic way that facilitates a certain workflow:

- When you read in data, rows are cases (e.g., people) and columns are variables by default.
- When you run a line of code (e.g., in SPSS syntax: *reg var= intervention...*), something generally happens and formatted output is sent *automatically* to the output window so you just need to go to the output window and look through it. **In R, this process is not so automatic, but it turns out that's better in the long run!**

It is possible to overcome the limitations of GP software but often this is very cumbersome or not feasible for the standard user (e.g., trying to bootstrap mediation analyses in SPSS without the *process* macro didn't really used to be feasible for the typical user).

R is completely flexible, but this is both a blessing and curse for

students like you. The trick is to keep the blinkers on and stick to doing things in a fixed way at first. R is object-oriented and this tends to make it very modular. If you spend time learning basic skills, you can then easily build on those as you become more advanced and confident and take off the blinkers, as it were. A good plan here is to use R to solve meaningful problems so that you gradually gain understanding in the context of trying to achieve feasible goals. *Many complex tasks in R are dealt with by connecting a series of smaller, simpler functions rather than by trying to make a single, complex one.*

2.3 R is entirely code-based

Code is the specific language in which you write a computer program. SPSS syntax is a very rudimentary coding language you've already learned a little. Consider this, in almost every aspect SPSS syntax is a more flexible and efficient way to do tasks than the SPSS "menus" (i.e., the GUI). Other software like SAS and Stata have even more sophisticated and powerful coding languages. In the world of R, there are programs such as jamovi and JASP and others that sit on top of R and actually generate R code and run it in the background when you make menu selections (as I wrote above, I won't discuss this here). R code is probably the most fine grained of these examples because R is designed for statistical computing. It means all sorts of stats are possible and if you gain some understanding of coding you can even program your own stats or other tasks. However, it also means you will need to use *some coding conventions* for tasks such as reading in the data and running the stats. This is not nearly as complex as you might imagine and once I have shown you how to do it, you'll easily be able to adapt it for new files.

Advantages of coding (for all GP software) are:

- it allows *fine-grained control* and forces the data analyst to be specific about what they are doing rather than click-and-hope
- it automatically leaves an *audit trail* so you (and the rest of us) can keep track of what you have been doing. You can make it even better by leaving careful *comments* in your code **(very important advantage)**
- it *facilitates getting help* with your trickier problems from others who are more expert
  - can get others' code online and adapt to solve your problem
  - F2F or online consultants/guides (or interested onlookers!) can see what you are trying to do by inspecting your code and make suggestions (e.g., StackOverflow)—can't do this easily with menus as virtually no one can figure out what you've been clicking

In this guide, I will try to give you the templates of basic code needed to read in data and run some stats. The **good news** is that the actual code for the stats is the simplest bit!

Below you will see some code to run a multiple linear regression on variables you may recognise from the BDRV data. It's quite a bit more fleshed out than the simple example code snippet in Module 1, so from now on, focus on each new example as it comes up. The box underneath the code in which each line is prefixed with ## shows the output that R produces. In an actual data analysis session, the R output isn't prefixed like that in the console, but doing so is a convention used in guides like this one. *Note that this convention will be the same throughout this and any later modules* but for some output that would be too large to include in its entirety, I have opted for partial screenshots from an actual R session.

Remember, you could re-use this line in your own R session with an appropriately set up project and the only parts you would need to alter are the actual variable names to match your data.

```r
regint <- lm(intervention~age + sex + hospitality + conservative, data=binge)
summary(regint)
```

```
##
## Call:
## lm(formula = intervention ~ age + sex + hospitality + conservative,
##     data = binge)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.3854 -0.6485  0.0366  0.7241  2.7071
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)      2.44444    0.16079  15.202  < 2e-16 ***
## age              0.02695    0.00275   9.799  < 2e-16 ***
## sexfemale        0.24741    0.07617   3.248  0.00121 **
## hospitalityyes -0.10715    0.11840  -0.905  0.36577
## conservative     0.45265    0.05564   8.135 1.64e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.052 on 773 degrees of freedom
## Multiple R-squared:  0.208,  Adjusted R-squared:  0.2039
## F-statistic: 50.76 on 4 and 773 DF,  p-value: < 2.2e-16
```

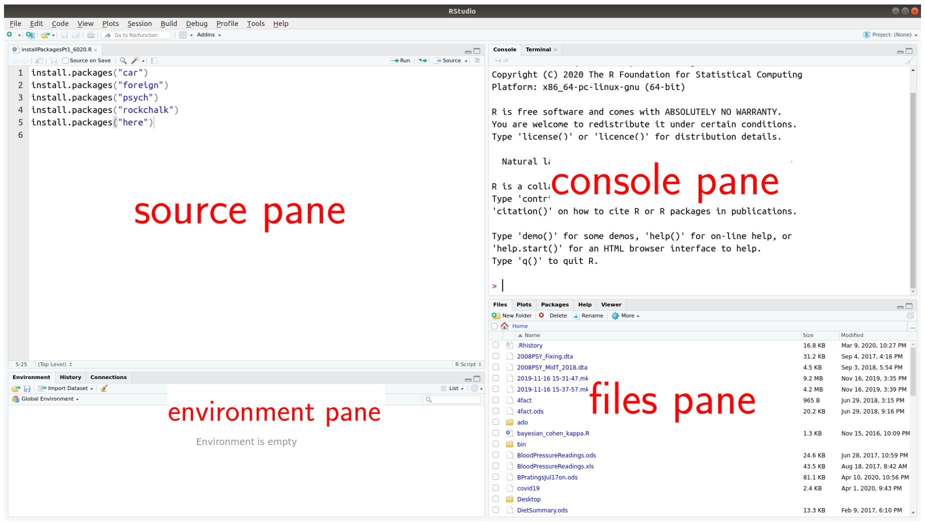
Note that this is just the default print out of a regression object in R. You could do many many things with this output with just a few lines of code. You could even use R functionality to generate APA tables directly from this output and save them as Word files or PDFs. But first things first…

2.4 R Studio: An app for efficiently controlling R

When you start R directly from the task bar or start menu in Windows or Mac equivalent, or the terminal in Linux, a single window, called the *console* appears. This is often described colloquially as *native R*. Running native R in the console is a little... sparse... If you had anything to do with computers and stats or programming up to about the 80s, you will recognise the non-graphics-based computing feel. When R starts or after a command has run successfully, the user is presented with a ">" prompt at the start of the new line. That is R essentially sitting there asking, "What do you want now? Be specific!".

Instead, most data analysts will use a program/app called *R Studio* for a more useful and pleasant control panel. This type of app is known as an integrated development environment (IDE). Programmers use IDEs to make their work more efficient and if you want to waste a few lazy and incomprehensible hours you might listen in on web conversations about the relative merits of these. Around the web, you'll find analogies made about IDEs such as R Studio that compare them to the cockpit of a plane or to a steering wheel, with R being the plane or the rest of the car, and so forth. You probably get the picture. I personally like the analogy where R is like a luge sled (from the Winter Olympics). It's lean, fast, and incredibly efficient at what it does but you *really* need to know what you are doing and there is no effective margin for error. R Studio is more like heading down the same track in a bobsled (ignore the other riders for this). It's still really fast and you still need a measure of specific skills, but you have a lot more control and a bit more margin for error. Perhaps as you go through this guide, you too can come up with a cheesy analogy.

R Studio, like R, is open source and free of cost. There are paid for versions but these are for corporate/production or server versions and the cost is mainly for support and server-related things. *The basic, free version has full functionality for data analysis and it is all you will need, likely ever.* The app organises your R work into a group of windows, called *panes* around an R console that is itself now a pane within R Studio! Essentially, an instance of R is running inside R Studio in the latter's *console* pane and as you proceed in a data analysis session, the console pane will contain a print out both of the R code sent to it (called, *echoing the code*) and of any output that arises, along with any error messages. R Studio adds a *source* pane which is where you actually write the code before running it, and panes for *graphics* (e.g., some R functions output plots and figures here), *help*, and other things. Some panes, like the *source* pane, have only one function. Other panes, like the *files/plots/packages/help/viewer* pane, have many functions that can be selected with tabs. In practice, this is not actually that complex. Figure 1 shows a screenshot of an R Studio session with the overall R Studio window containing all the typical panes. This is my own installation of R Studio, somewhat modified while writing this guide. When you first install it, your version will appear slightly different. *It's not an error* and later I'll discuss how to change it.



```
install.packages("car")
install.packages("foreign")
install.packages("psych")
install.packages("rockchalk")
install.packages("here")
```

```
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (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 la

R is a colla
Type 'contri
'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.

> 
```

Name	Size	Modified
.Rhistory	16.8 KB	Mar 9, 2020, 10:27 PM
2008PSY_Fixing.dta	31.2 KB	Sep 4, 2017, 4:16 PM
2008PSY_MidT_2018.dta	4.5 KB	Sep 3, 2018, 5:54 PM
2019-11-16 15-31-47.mk	9.2 MB	Nov 16, 2019, 3:35 PM
2019-11-16 15-37-57.mk	4.2 MB	Nov 16, 2019, 3:39 PM
4fact	965 B	Jun 29, 2018, 3:15 PM
4fact.ods	20.2 KB	Jun 29, 2018, 9:16 PM
ado		
bayesian_cohen_kappa.R	1.3 KB	Nov 15, 2016, 10:09 PM
bin		
BloodPressureReadings.ods	24.6 KB	Jun 28, 2017, 10:59 PM
BloodPressureReadings.xls	43.5 KB	Aug 18, 2017, 8:42 AM
BPratingsJul17on.ods	81.1 KB	Apr 10, 2020, 10:56 PM
covid19	2.4 KB	Apr 1, 2020, 9:43 PM
Desktop		
DietSummary.ods	13.3 KB	Feb 9, 2017, 6:10 PM

Figure 1: **Figure 1** | Typical R Studio Session Window with active panes labelled

I strongly encourage you to use R Studio. All examples in this guide will be presented assuming the user is working through R Studio. The way that I will use R Studio to work in R is as follows:

1. Write code and edit as needed in the R Studio *source pane*.
2. Select the code fragment with mouse as usual.
3. **Ctrl + Enter** = run code.
4. Check output that will be found in the R Studio *console pane*.

i. continue on; or
ii. go back and fix it up.

5. Goto 1)

This makes the workflow very similar to what you are used to with SPSS

2.4.1 Customising your installation of R Studio and fixing a couple of problems with the default setup

You can go about and customize as much or as little as you like. Click on the **Tools/Global Options** menu and you can change colour scheme and much more in *Appearance* and you might also want to modify (at least while following this guide) the *Pane Layout* (I show the steps for this below). Modifications are easy and reversible. For example, for the sake of readable figures, I temporarily made the font much larger for some screenshots while writing this guide and Matt and I did this when running workshops and sharing or projecting our screens. I will leave you to explore that yourselves. But first, here are a couple of **very important** and useful options changes. Go to **Tools/Global Options/General** (*General* should be the first tab and visible by default; see Figure 2).

1. Clear the checkbox for **Workspace** (*Restore .RData into workspace at startup*) and make sure the *Save workspace...* option is set to *Never*.

- This is an option that even Native R always asks you when you quit and you would virtually never say *Yes*, so this is just a handy change. Saying *yes* would have R save the previous session's workspace and reopen it in the new one. But, trust me, you *don't* want old sessions and their objects potentially messing up new sessions when you are doing stats with R.

2. Clear both of the History boxes. (i.e., uncheck *Always save history...* and *Remove duplicates...*).

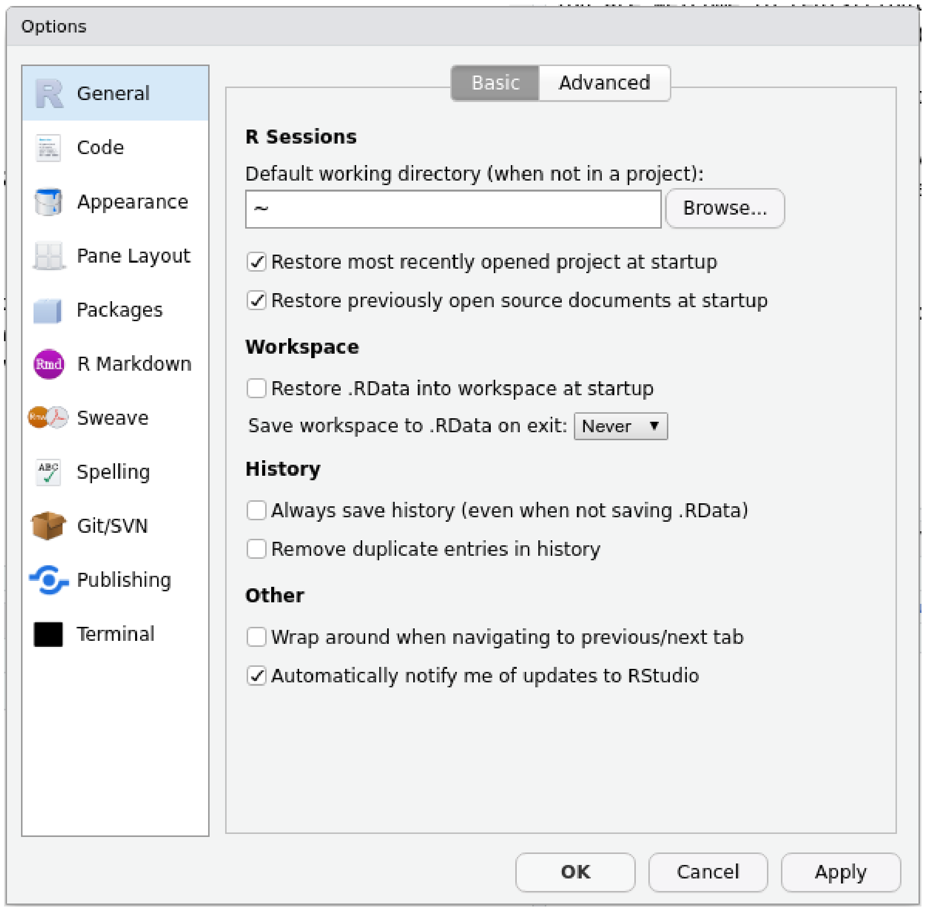
- Many R geeks have pointed out that depending on what data you are reading and working on with R (e.g., personal health identifiers in health surveys or databases), R might save items such as hospital record numbers and other important characteristics in the History in a way that you might not have noticed and this is obviously a risk. Get rid of it! The typical user would essentially never need to inspect their R history.

You'll find a version of these tips all over the web. *I didn't think of them but I definitely use them.* [hint: if you have found .RHistory files in your folders/directories, you can safely delete them and you should do so]

If you 're wondering about the colour schemes in *Appearance*, I use a light background scheme for teaching and projecting/screen sharing as it seems to work better with the text on R Studio panes. The current guide will have screenshots mainly in *Chrome*. For my own work, I have a preference for dark backgrounds and *Twilight Night Bright* is my usual go to, for the contrast!

2.4.2 Pesky but important change to R Studio default pane layout that will be "set and forget"

Another change that I made early on in my use of R Studio (I believe I saw it on a Youtube vid) is to shift the standard pane layout. I do this any time I install or re-install R Studio. By default, R Studio places the *console* pane (where the output goes) on the bottom left under the *source* pane (where the R code sits). When you first install R Studio and open the app, everything seems correct. The thing you would recognise as the native R Console seems to be in the most important and largest pane on the left. But that is only because there is no *source* pane open yet. The *source* pane is the most important pane as all of your work writing code and asking for stats is there and it should be set up to easily have the most space when required. I modify this default using **Tools/Global Options/Pane Layout** so that the *console* pane is on the top right, next to the *source* pane and the *environment/history/connections* pane is on the bottom



R Sessions

Default working directory (when not in a project):

~

[✓] Restore most recently opened project at startup

[✓] Restore previously open source documents at startup

Workspace

[ ] Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

History

[ ] Always save history (even when not saving .RData)

[ ] Remove duplicate entries in history

Other

[ ] Wrap around when navigating to previous/next tab

[✓] Automatically notify me of updates to RStudio

Figure 2: **Figure 2** | R Studio Options dialogue

left. You will note that that is how my R Studio is set up in Figure 1 above. The reason is that it is efficient to have whatever output that R will generate pop up right next to the R code you just ran. Also, it is often very useful to be able to lengthen the source pane as the length of the R code that you are writing grows (if you hover your mouse over the pane borders in the usual way you will see the pointer becomes a little grabbing fist and you can move them up and down and left to right). The *environment* pane is not something you need to constantly see in its entirety and using this layout, you can shrink or minimise it to give more space for code. If you need to check the environment (e.g., to see if you have correctly created an object), it is very easy to do so by scrolling in the shrunken pane or bringing it back if minimised. **I strongly suggest you make this modification**. Any online sessions or screencasts that I run will have this layout and it will be helpful for you to be able to see the same layout on your own screen. Plus, it's just better. You can always change it back later if you wish. Finally for this topic, using the R Studio *View* menu or any associated keyboard shortcuts, you can zoom or maximise any pane you like, but most of the time, beginners will have all 4 panes visible, hence the modifications.

2.5 R Project Files

R Studio has a special type of file, called an *R Project File*, that helps to organise R projects. These are like virtual containers that organize all files/folders to do with a... project! This will also help to avoid the use of clunky and nonportable file paths in code, which is a nontrivial benefit that addresses a problem that occurs not just in R. How many times have you had hassles because files and folders you created or worked on at university in the labs wouldn't work as planned or were hard to find at home? There is a simple but annoying reason for that that I will try to explain below. The important thing is that the use of R Project files neatly solves 99% of these issues.

The R Project "exists" within R Studio. To explain this without getting esoteric and to keep to the *just enough code to do the stats* approach, consider the following. Almost all of us have folders on our PC/Mac/Linux/Whatever machines specifically devoted to things like pictures, videos, downloads, and so forth. Similarly, when a new project pops up, you may have a folder in your filesystem on your machine that you may have created or devoted to it. (E.g., how many people reading this will already have a thesis folder?)

When you set up an R Project, you either create a new folder for it or designate an existing one to it and R Studio inserts a special *project* file (and a bunch of other things that are ordinarily hidden and can stay hidden) that works like a map of your folders. **The project file tells R Studio what it should consider to be a part of that project**. R Studio then ring fences these folders in its map to mark them off from everything else whenever it is dealing with *the specific project.* Then, it doesn't matter what else is on your machine/hard disk outside the project. For example, a typical Windows PC will provide a directory/library called *My Documents*. However, the actual full path of the My Documents directory is something like: "C:\Users\stefano\Documents" and so a 6020PSY folder with *your* R learning in your My Documents at home could be something like "C:\Users\You\Documents\UniStuff\6020PSY\R_course" and every time you copied this folder onto a USB and brought it to another computer everything up to *R_course* would change. Instead, if this *R_course* folder was an R project, R Studio would insert a special R Project file in it so that when you copied the whole thing onto a USB as usual and brought it from your computer to another one you use (even from PC to Mac or vice versa) and you opened it in R Studio again, everything would work the same. R Studio would ignore *all the files and folders outside the R Project folder* and only focus on what is inside it. You as the user don't have to worry about any of that as it is all handled inside R Studio. Naturally, if you try hard enough you will defeat any system, including this one, but that is outside the scope of this guide.

The best way to gain an understanding of and some familiarity with R Project files is to just start using a basic form of them. As usual, it probably takes longer to describe than to use them. You can create a new R Project file from scratch (see below) or convert existing folder/directory to an R Project just as easily. It's a good idea to designate a folder and corresponding R Project for every actual project you do right from the start. After the information below, I will spend some time explaining why and how to set up your folder to gain the maximum benefit.

Creating an R project file (two different ways)

1. **To create a new folder**, in R Studio: *File/New Project/New Directory/New Project*

- Name it carefully, briefly, and avoid spaces (use underscore _ or hyphen - instead or UpperAndLowerCase)
- Browse for parent directory (where you want to put it) then Create Project
- If you haven't yet created a folder/directory for your work on this R learning guide, do so now. Call it whatever you like, respecting the norms above. I will tell you below what else to put inside it. As you will see, R Studio and the R Project file will take care of the rest!

2. **To use an existing directory** (e.g., a directory you may have created for this course, or a thesis data analysis directory etc), in R Studio: *New Project/Existing Directory/Browse*, then *Create*

In both cases above, you will have a folder in your filesystem that you can treat like any other folder: copy it and save it to another place; save files to it in other programs; and so forth. You can even delete it, but that might be a bit counterproductive (even if harmless). As described above, the folder contains a special R Project file that only R Studio uses when you open the R Project. You can see the project file in your folder on the computer like any other file. It's not hidden.

2.6 Preparing the R project directory

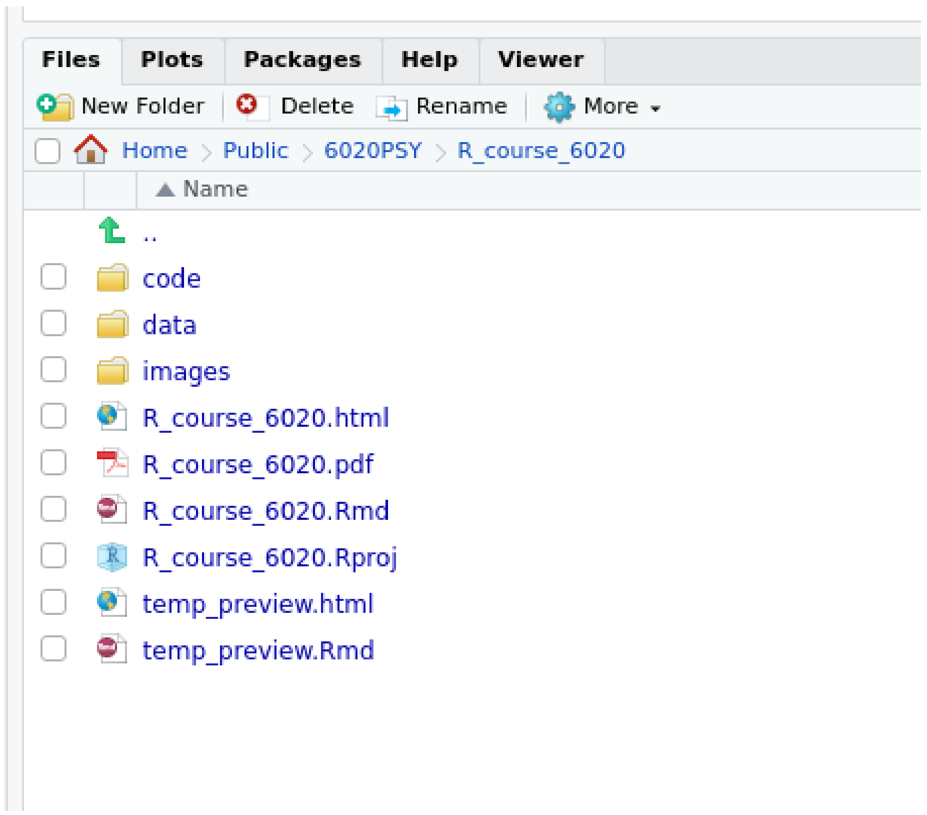
This is how I set up new directories/folders. It's typically anal in that psych stats way. For now, use the names (in lower case; **R is case sensitive**) that I use below, so that the code I provide will always work. Don't cut corners. When you have more confidence, by all means, come up with and use your own way of naming things if that works better for you. Whatever you do, just be consistent about it!

Create two directories inside your R project directory (i.e., the directory you most likely just created above) named as shown below. If you are already in the root project directory in R Studio, you can do this with the *New Folder* button located in the top left of the *files* pane, that is itself located in the bottom right of the R Studio window. However, I suggest you just go into your new folder on the filesystem of your PC (e.g., File Explorer) or your Mac and create the new folders in there as usual. I want you to see that you can work seamlessly and transparently with the filesystem and in R Studio without any problems. Then, as always, once you have mastered a skill, go for the variations. As soon as you make the new folders, when you return to R Studio and click on the *files* pane (the will be a tab in the pane under where your *console* pane should now be!), you will find them listed there.

- Root Project Directory *[whatever directory you created for your work above]*
  - data directory *[i.e., use lower case and call it "data"]*
  - code directory *[i.e., call it "code"]*

Later, you can add more directories if you like but be consistent and do not use file/directory/folder names with spaces. Spaces are for Macs and Windows, not the real world! For example, you might want directories for graphics output, directories for writeups etc—anything to do with a project and not just R. Again, the R Studio project file just maps out the folders for R Studio and tells it what to focus on and what to ignore for a given project. You can have folders full of JPGs or PNGs or a folder with Microsoft Office docs or even a folder full of your increasingly redundant SPSS files. R Studio Projects are relaxed and tolerant. They will just show you the files in a given folder in the *file* pane and not blink an eyelid. Below is a figure showing an example set of folders from a project called *R_course_6020*. This person has an *images* subdirectory in their project and there are also some other non-R files sitting in the root directory (PDF, HMTL etc). As long as *your* project contains the *data* and *code* folders, you are good to go with this guide. Anything else is up to you.

All of my project directories follow the same naming convention. I even use a standardised naming system for code files for particular jobs (e.g., reading and fixing up raw data; scoring instruments and scales and setting up the variables for analyses; analyses that use exploratory or confirmatory factor analyses; analyses that use regression models etc; yes, I am definitely *that* guy, but I can always



code
data
images
R_course_6020.html
R_course_6020.pdf
R_course_6020.Rmd
R_course_6020.Rproj
temp_preview.html
temp_preview.Rmd

Figure 3: **Figure 3** | Example folders in an R Project

find an analysis!). If you have named your two new folders as above, later, you will be able to use R techniques I'll cover soon to reference datasets very easily in your R code, avoiding errors such as from mistyping long pathnames or from shifting systems from uni to home (Win/Mac/*nix etc), or modifying your own computer's filesystem (maybe even just changing the names or locations of a couple of folders). *Free tip*: this is not just for R, I do this for Stata etc too. It's just common sense!

2.6.1 Before going on, I'd like you to now copy the course files into the R project directories.

These will either be in the 6020PSY Microsoft Teams files tab or at a link that I will have shared. I want you to put any data files, ending in either *.sav* or *.csv*, into the *data* directory you created and any R code files, ending in *.R* into the *code* directory you created. You can put anything else you like in any other folders you may have created. As long as the two specified directories exist and contain the data and code files, respectively, it will be OK.

2.7 A Schema for How R Works

2.7.1 R has a lean and efficient philosophy

When you first start R up on your computer (whether via native R or in R Studio), it already has the capabilities to do all the stats you would need in a course like 6020PSY and all of its prerequisites. This is often referred to as *base R*, the version of R as is installed before any *extra components* are added. But... R is programmable. Users have written tens of thousands of additional modules, called *packages*, that significantly extend the capabilities of R. Some packages have been written by the actual statisticians who developed the new procedures. Some have been written by keen R users of differing levels of experience who have come up with useful additions to the capabilities of R. *Packages typically contain 2 or more functions.* As will be discussed below, the functions are the things that actually do the tasks in R. Packages, as the name suggests, are collections of usually specially written functions, and sometimes also data, to achieve an overarching goal. Often, packages will *depend* on other existing packages to perform their functions and by default such *dependencies* are checked and installed if necessary. Some examples of the types of things R packages can do are:

- new stats procedures (pretty much hot off the press);
- systems to optimise statistical workflow for specific disciplines (e.g., we do stats slightly differently in psychology vs disciplines like epidemiology and there is a fantastic package called `psych` by Bill Revelle at Northwestern for us)

- really cool things: e.g., take your regression output and make an APA table from it that you can output as a Word file or as a PDF
- put together a bunch of datasets that are related for some specific purpose so when you load the package they are all instantly available

There are countless more things. It is impossible for one person to keep track of all of them. In later modules, I'll discuss how to access R communities around the world to hear about cool new things that are taking off.

After you have installed R for the first time on a new system, you can install new packages at any time. It doesn't matter whether you remember that you will need or even that you know to install specific packages before you start. Many times:

1. people have become stuck doing an analysis,
2. Googled a solution that points to an R package,
3. that they then install on the fly,
4. and start using then and there.

Once installed, you can update packages as updates become available (I'll show you how). You can uninstall them too. When you upgrade your installation of R with a new version as these become available (this is a regular occurrence) depending on the OS you can have all your packages copied over into the new system. R Studio also makes it very easy to install, keep track of, and update packages. As you will see, the R code to install new packages is very simple and many users keep a file that lists the installation of all the packages they come to use most and any time they need to (re-)install R somewhere new, they run this code file immediately and their packages are all set up. Again depending on the OS, some users maintain scripts that locate and install and modify the user's preferred setup for R and R Studio any time they configure a new system.

A key part of the philosophy of R is that:

1. it is a lean system where only enough to do the jobs at hand is installed;
2. but that can be upgraded very easily to do more specialised jobs.

If you know you will want to do certain tasks that base R isn't so clearly set up for (e.g., structural equation modelling; SEM) you will want to install the appropriate package (e.g., `lavaan`). Once the package is *installed* in R on your system it will be available to you. However, when you start R, it won't *immediately* be available to your code without another step. In fact, this would be contrary to the philosophy above.

An R user will probably not do something as involved as SEM every single time they use R. Therefore, why use up system resources and computer memory for a thing that often is not needed? The solution is to have packages loaded up into memory only during the session you are going to use them. This is similar to the way we all have to load up the *process.sps* macro in an SPSS session before we can use the process syntax to do mediation or moderation. In R, this becomes such a natural action that you find yourself doing it automatically (see below). It's just not that complex in practice. In summary:

- Only bother **installing** a package (e.g., `install.packages("lavaan")`) if you know you want to use it at some point
- Only bother **loading** a package (e.g., `library(lavaan)`) if you are planning to use it **in that session**.
    - but if you have installed a package that you definitely want to have loaded *every* time you use R, you can set that up too!

2.7.2 Things you will only do once each time you (re-)install R on a computer

1. Download R from CRAN (where it lives) and install it

- see the **Install Tips** module for more detail
- upgrading base R with a new version (i.e., as of this writing, it is 3.6.3) varies a little by system
    - *Linux*: automatic by including appropriate PPA as usual (see CRAN)
    - *Windows*: the `installr` package has *updater()* function for fairly painless upgrading (run native R as an adminstrator for this, not from inside R Studio!)
    - *Mac*: it's a manual affair (like so many Mac things), but rest easy, it's not very arduous.

2. Download and install individual packages that you have decided you need

- each new package needs to be installed only once
- you will probably install some packages immediately and others many months or years later-it's very dynamic
- updating packages varies a little by OS type but R Studio makes it very quick to upgrade all if necessary
    - the *packages* pane has an Update button, click this and either select all or specify through checkboxes as usual
    - in R code itself, the *update.packages()* function can be called for very fine-grained use.

2.7.3 Things you will only need to do once in any new R session on any type of computer

- Load up the required package(s) before you actually include their functions in your code
    - if you are definitely planning to use a package (e.g., the `foreign` pacakge to read in SPSS data files), load it at the top of the code file. It's just more cognitively efficient like that
    - but you can also do it later, as the system is very flexible

    * e.g., you noodle away and occasionally find and load packages you didn't realise you'd need
    * when you have sorted everything out, you should go back and tidy up code as required
  - *advanced point*: Sometimes two or more packages that are widely used clash in that they have identically named functions that perform similar or different tasks. This can cause a few issues with written code but they are *very* easily fixed. For this reason, some packages, experience shows which, are more easily loaded just before use (or may not even need to be loaded to be used!). This is not a major issue and it is unlikely to cause problems for typical beginning users like people using this guide.

2.8 OBJECTS and FUNCTIONS

2.8.1 The last important R concepts before looking at some basic code

R is classed as an object-oriented language. Whether or not you do any programming, the practical aspect of this is that R uses *objects* extensively in normal data analysis and this is a **big difference between R and most GP stats software like SPSS**. At first it may feel clunky, but it will swiftly become second nature. Again, in practice, it's not that big a deal. It takes more to describe it! For example, you may recognise the code snippet below as doing a regression with variables from the BDRV data. Beliefs in intervention efficacy are regressed onto participant age, sex, whether they worked in licenced hospitality jobs (a dummy), and their social conservatism.

```
lm(intervention~age + sex + hospitality + conservative, data=binge)
```

If you run just this line as is, you will see fairly sparse output:

```
##
## Call:
## lm(formula = intervention ~ age + sex + hospitality + conservative,
##     data = binge)
##
## Coefficients:
##   (Intercept)            age      sexfemale  hospitalityyes    conservative
##       2.44444        0.02695        0.24741        -0.10715         0.45265
```

R has done *exactly* as asked. It has run a regression and echoed the code and some basic output into the *console*… hmm… that's R!? so what…? Instead, there is an R convention of sending the regression output into a container called an *object*. Note the tiny addition to the code.

```r
regint <- lm(intervention~age + sex + hospitality + conservative, data=binge)
```

This line will run a regression and store everything about it in an *object* called *regint*. You could have called the object anything you liked. Really experienced users often name such objects with highly abbreviated names like *fit* or *mod* or *f*! This is sort of analogous to using *compute* to create new variables in SPSS syntax. But here you are creating a new object. To begin with, you should stick with short object names that you can decipher as you go (and use lots of comments!) In the examples I'll include in this and later modules, you will see the obvious object names I've used. Again, it is not that complex in practice.

This approach is very powerful. Objects can be: the output of procedures; datasets; new groups of variables; and more. R is really clever about figuring out what sort of object you have given it so 99% of the time, you won't need to worry about keeping careful track of object type as you just won't need to use it. It's easier just to accept that there are objects and set about using them as I will have you do from here on. For example, the *lm()* (i.e., regression output) object created above (i.e., that I have called *regint*) can itself be fed into the input of other packages and functions to process it further. Working through this guide and further modules, you will see how extensively this approach is used. It is a key aspect of how data analysis will change by using R compared to what you are used to in SPSS. Using objects and functions like this, one can obtain:

- new stats
- specialised formatting of output
- useful plots
- and much more

In fact, one of the first forays into programming you might make will probably be to write your *own* functions to help with repetitive or detailed tasks. It derives seamlessly from the R code you will use here.

2.8.2 Clearing objects: a line of code you will see again and again

Because you are likely to create many objects in a typical R session, when you create R code files you should always start with a line that clears out the workspace. In this way, you will avoid issues with pre-existing objects from previous work in that same R session interfering with new work. The line below basically says, "make a list of every object in the workspace and then remove every object on that list!" You can copy it directly from here or from any of the R files associated with this guide and probably you could google it up and find it in R code around the world!

```r
rm(list=ls())
```

2.9 In R, everything we do is a *function()*

2.9.1 There are exceptions to the below but this will get everyone on the same page to begin with

John Chambers of Stanford University, the inventor of S, the parent language of R, is credited with a very clever saying that you will soon experience! In R:

- Everything that exists is an object.
- Everything that happens is a function call.

Have you noticed all the parentheses in the snippets of R code so far? For example, have a look at the *rm(list=ls())* above or the code that creates the *regint* object in the previous section. These parentheses (not brackets-[ ]!) usually mean there is a *function* involved. **Everything command that you give R consists of at least one function**. And because most functions will ultimately send their output to objects, with few exceptions, you will often see a structure like the below:

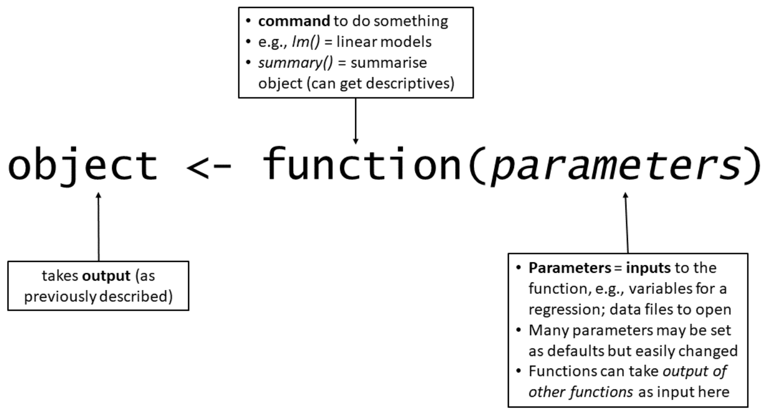

• **command** to do something
• e.g., *lm()* = linear models
• *summary()* = summarise object (can get descriptives)
object <- function(*parameters*)
takes **output** (as previously described)
• **Parameters** = **inputs** to the function, e.g., variables for a regression; data files to open
• Many parameters may be set as defaults but easily changed
• Functions can take *output of other functions* as input here

Figure 4: **Figure 4** | Schema for *functions* in R

If you have a look above at the code snippets for the regression examples, you will see examples of objects and functions. The *lm()* function takes as input parameters the variables in the model (i.e., Y and X variables) and the name of the data object (i.e., binge) and sends its output to the *regint* object created in that code. You will see this same pattern over and over and it will make the reading of perhaps previously forbidding R code actually quite staightforward and increasingly so.

As you have seen already, if an object is not specified, the output of functions is spat out into the console pane and this is often not optimal. Therefore, mostly you will specify an object (new or existing one) to receive output of the function. The way to do that is via the *assignment operator*. This is used so frequently that it is almost never referred to explicitly-it's just there! In context, it looks like this:

```
Object <- function()
```

(Yes, it's a less than symbol followed by a hyphen)

It would be read as *Object takes the output of Function* but most people just think of it as an arrow and rely on their nonverbal reasoning. If you wanted to, you could even turn it around:

```
function() -> Object
```

But that really isn't necessary for most of us. Some people use = for assignment but I **strongly** discourage you from doing this. An R convention is to prefer the arrow for assignment (i.e., send the output of *this* to *that*; *regint <- lm(...)*) and the = for setting parameters *inside* functions (i.e., make the value of *this* equal to *that*; *data = binge*). I will always follow that convention so if you want your *source* pane to match my examples, you need to do it the same way, otherwise you will be on your own.

If the output of a function, such as *lm()* for a regression, is sent to an object that already exists, R will overwrite the object with whatever the new output is. It won't ask. But if the code has been written sensibly, it's very easy, even trivial, to fix it up and rerun as required. So if you realise you created an object and sent the wrong regression to it, just fix your code and send the correct regression. R will overwrite it and nothing else is required of you. Remember, this system of functions (and objects) may seem a little complex at first but it ultimately makes things a lot easier and strangely intuitive.

A powerful tool is the ability to *nest* 2 or more functions within each other. Functions are nested in R like Russian matryoshka dolls (see Figure 5).

The output of one function (e.g., a statistical analysis) is passed immediately to the input of the next function (e.g., a plotting function) with no intervention needed by the user. In R, one can easily nest 3 or 4 or more functions. This isn't for fun but because that is how to build code to perform seemingly complex tasks from a series of steps. Each separate function can be simple in itself but nested together they perform a complex task. Usually, to figure out what is going on, start at the inside of the nested group and work back towards the outside. Some of the snippets of code that I'll show you contain nested functions that you can paste into your own work. You will only need to change a couple of details to use them and by pasting them over and over you will start to understand what is going on with each separate bit.

Figure 5: **Figure 5** | Russian *matryoshka* dolls that illustrate the concept of nesting; Original photo: User:Fanghong; Derivative work: User:Gnomz007 - removed background from File:Russian-Matroshka2.jpg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=227816

2.10 End of module

3 Module 3: Addressing files, reading in data, and basic statistics

N.b., the *basics.R* file has the R code needed for this module. My advice is to have it open in R Studio and step through it as you work through this material. You can navigate to your R Project *code* folder via the Files pane in R Studio then click once on the appropriate .R file. The first tasks explain to you how to step through the R code. The code files I will use are extensively commented and this should complement what is here. After this module, I will leave you a small series of exercises to get you to apply your learning from the module to new material.

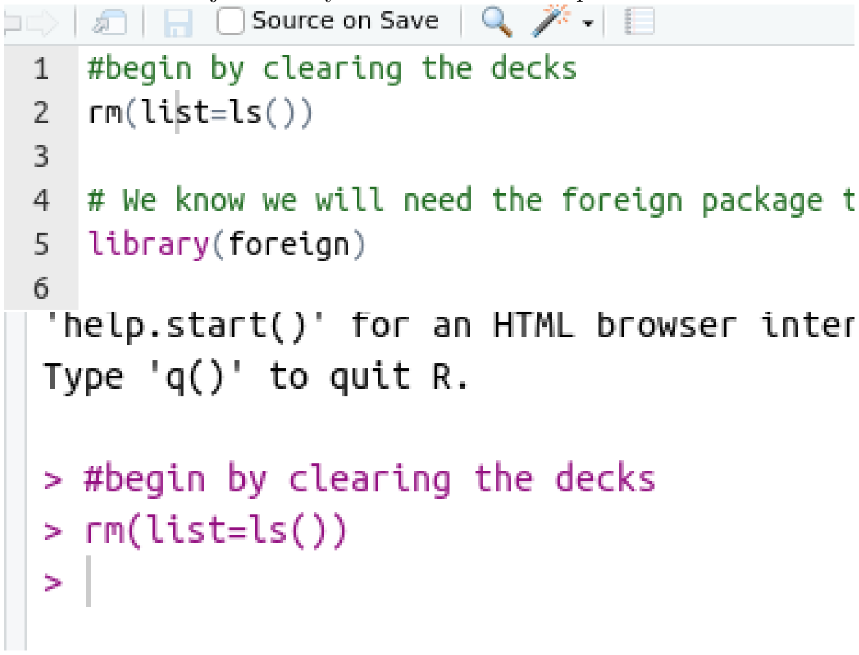
3.1 Reading data into R—using nesting!

Let's begin to apply the knowledge gained so far. You will immediately use nested functions to read in some data (by first seeing what happens when you don't nest). For this task, I'll be using a slightly simplified version of the BDRV data that we use for 6020PSY. There is already an SPSS version of this data that you will have copied into your R Project as I asked you earlier. (N.b., *don't* use SPSS data file from 6020PSY tutes. The data is the same but the format may confuse you.)

3.1.1 First, clear the workspace with the appropriate code at the top of the file.

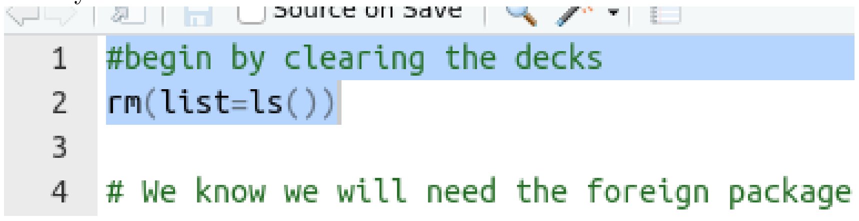
In R Studio you can run code in two pretty much equivalent ways.

1. **Place the cursor anywhere in a line of code**, the first line of code you want to run, and press **Ctrl + Enter** (all OS). This will automatically select the line of code and *any* comments before it and run them. You can check this because the R console pane will echo the code sent to it and any output generated. In this case, there is no further ouput as the *rm()* function just removes things. As it happens, some of the widely used, basic functions I will show you first don't necessarily need to be sent to an object as they send informative output direct to the console!



```
#begin by clearing the decks
rm(list=ls())

# We know we will need the foreign package t
library(foreign)

'help.start()' for an HTML browser inter
Type 'q()' to quit R.

> #begin by clearing the decks
> rm(list=ls())
> 
```

2. Select the specific section of code (with or without comments) and press **Ctrl + Enter** (all OS). As you might imagine, this will run the entire block you have selected.



```
#begin by clearing the decks
rm(list=ls())

# We know we will need the foreign package
```

That's how you will run every bit of code in R. An advantage of the first method above is that after you have run the code, the cursor is already poised at the next line. In this way, by pressing **Ctrl + Enter** in sequence, you will step through many lines of R code in order, line by line, so you can pause and inspect output or other things after each line. This is particularly useful if you have a well laid out, sequenced code file. An advantage of the second method is that it allows very precise sections of code to be run all at once and when you know a whole block "just works" and will run a regression, display some output and generate some plots, you can immediately run it. R is extremely fast-much faster than SPSS in most things, so you will be surprised how quickly output and graphics might pop up. Neither is the *correct* method. Use each in the appropriate circumstances.

3.1.2 Next, you will load the required package.

You are going to read an SPSS file. Base R is set up to read either CSV files (comma-separated values; essentially, the text file version of a spreadsheet that are used universally) or RData files (R's own format that virtually no one uses). To read *external* data formats, such as SPSS, or Stata, or SAS files, base R needs the `foreign` package (don't worry, R isn't racist). Do this with the *library()* function. Each package is loaded in a separate call to *library()*. No output is generated by this function and there is no need to assign it to an object. Run the appropriate line in the *basics.R* file.

```r
library(foreign)
```

3.1.3 Now, you will try to read in the data and you will see how nested functions and objects work in practice.

The *read.spss()* function comes from the `foreign` package, and its name is what it does. Assuming you put the BDRV data into the *data* folder of your project as directed, if you run the next line you will command R to read the data file.

```
read.spss("data/BDRVdataR.sav")
```

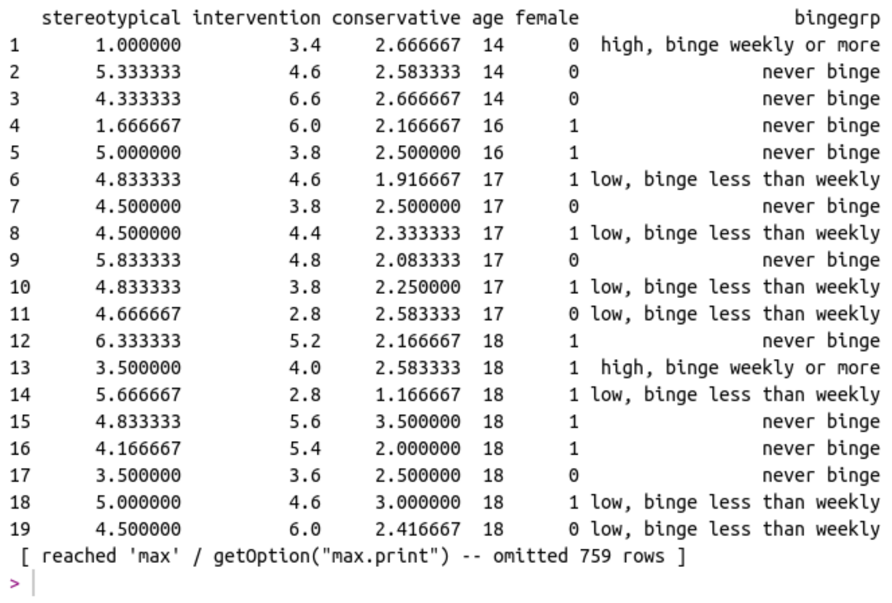
But the problem is that that is all it does and then it spits the data it has read haphazardly out into the console-and there is a lot of it!

It's clear there are variables from the BDRV dataset but the presentation is quite garbled. Next, you'll try another function and **nest** *read.spss()* inside it.

```
data.frame(read.spss("data/BDRVdataR.sav"))
```

```
   "Social Conservatism"

                     age

                      ""

                  female

                      ""

                bingegrp
                                                                                "Categorical version of reporte
d nhmrc binge frequency"
attr(,"codepage")
[1] 65001
>
```

Figure 6: **Figure 6** | Partial output from a poorly planned piece of code



```
  stereotypical intervention conservative age female                   bingegrp
1      1.000000          3.4     2.666667  14      0  high, binge weekly or more
2      5.333333          4.6     2.583333  14      0                never binge
3      4.333333          6.6     2.666667  14      0                never binge
4      1.666667          6.0     2.166667  16      1                never binge
5      5.000000          3.8     2.500000  16      1                never binge
6      4.833333          4.6     1.916667  17      1 low, binge less than weekly
7      4.500000          3.8     2.500000  17      0                never binge
8      4.500000          4.4     2.333333  17      1 low, binge less than weekly
9      5.833333          4.8     2.083333  17      0                never binge
10     4.833333          3.8     2.250000  17      1 low, binge less than weekly
11     4.666667          2.8     2.583333  17      0 low, binge less than weekly
12     6.333333          5.2     2.166667  18      1                never binge
13     3.500000          4.0     2.583333  18      1  high, binge weekly or more
14     5.666667          2.8     1.166667  18      1 low, binge less than weekly
15     4.833333          5.6     3.500000  18      1                never binge
16     4.166667          5.4     2.000000  18      1                never binge
17     3.500000          3.6     2.500000  18      0                never binge
18     5.000000          4.6     3.000000  18      1 low, binge less than weekly
19     4.500000          6.0     2.416667  18      0 low, binge less than weekly
 [ reached 'max' / getOption("max.print") -- omitted 759 rows ]
>
```

This time it is better in that it clearly has organised the data file a lot more. In R, data sets (i.e., what is in the data window of SPSS) are referred to as *data frames*. But note that the *data.frame()* function still outputs all the data into the console. Again, R is merely doing what you commanded. To fix this requires only one more element and that is an *object*.

```
binge <- data.frame(read.spss("data/BDRVdataR.sav"))
```

This has run and you will see there is no output. Part by part, this line means:

- binge <- | *assign* the output of the following into an object called *binge*
- data.frame(read.spss("data/BDRVdataR.sav")) | read the SPSS data file called *BDRVdataR.sav* in the directory called *data* off the root project

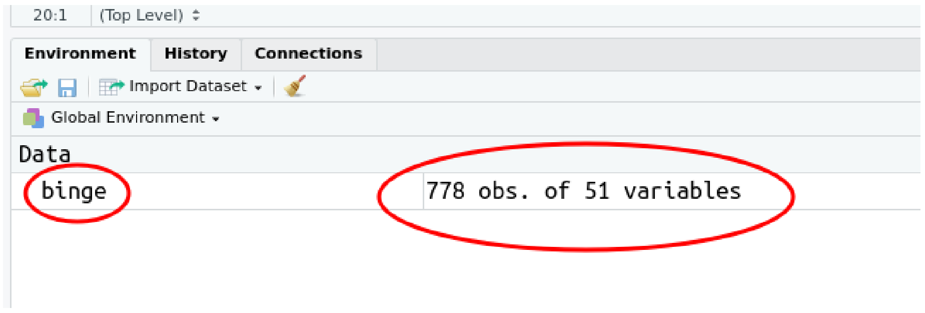
directory and output that into the function that creates an R *data frame*

The whole line:

1. reads an SPSS data file and
2. converts it to an R-formatted data frame which it
3. feeds into an object called *binge* that can now be used in the rest of the R session.

I chose the name *binge* arbitrarily because it fits, it's short, and it has no spaces! Some hard core R instructional materials are full of examples where they call every single data frame object *df*. Don't use that in your actual work!

When this line runs, you can see there is no output besides the code that has been echoed back. This is because the output of the nested functions in the code has been fed directly into the object called *binge*. If you check your *environment* pane, that should now be below the *source* pane on the left, you will see the binge object and further that it contains 778 observations (i.e., rows or participants) of 51 variables (see Figure 7. Because binge is a data frame object, R automatically presents it with the appropriate terms, rather than a generic statement like *rows = 778, columns = 51*. This is a very important aspect of R, that the user can code *generic* operations like assignment to objects and R is programmed to recognise any necessary variations.



```
binge                          778 obs. of 51 variables
```

Figure 7: **Figure 7** | *Environment* pane showing newly created *binge* object

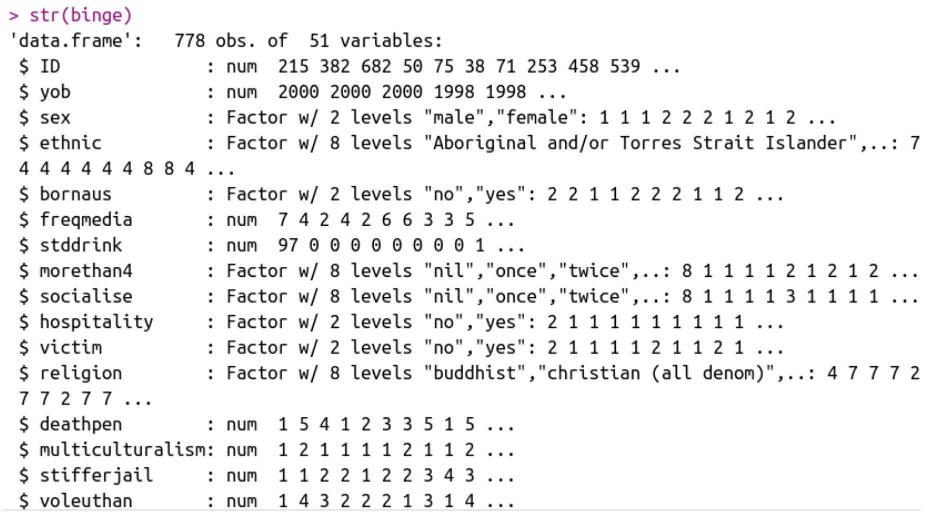
Next, I will show you how to work directly with this object. After the next section, I will show you a more efficient means to address files in R Projects.

3.2 Describing data frames and variables

3.2.1 Quickly summarise a whole data frame object

You may want to check that you have read in the data set correctly into the R data frame or you may just want a quick summary of a data set before working with it. In these cases, use the *str()* function (stands for *str*ucture). The code is very simple and the output is not even assigned to an object because it is useful to have the output echoed to the console pane. Below, I have placed a screenshot of the top portion of the output. Your output may differ slightly because I have increased font size for the screenshot. The numbers will be exactly the same. When you actually run the line, you will see that it summarises every variable in the data frame.

```
str(binge)
```



```
> str(binge)
'data.frame':   778 obs. of  51 variables:
 $ ID              : num  215 382 682 50 75 38 71 253 458 539 ...
 $ yob             : num  2000 2000 2000 1998 1998 ...
 $ sex             : Factor w/ 2 levels "male","female": 1 1 1 2 2 2 1 2 1 2 ...
 $ ethnic          : Factor w/ 8 levels "Aboriginal and/or Torres Strait Islander",..: 7
 4 4 4 4 4 4 8 8 4 ...
 $ bornaus         : Factor w/ 2 levels "no","yes": 2 2 1 1 2 2 2 1 1 2 ...
 $ freqmedia       : num  7 4 2 4 2 6 6 3 3 5 ...
 $ stddrink        : num  97 0 0 0 0 0 0 0 0 1 ...
 $ morethan4       : Factor w/ 8 levels "nil","once","twice",..: 8 1 1 1 1 2 1 2 1 2 ...
 $ socialise       : Factor w/ 8 levels "nil","once","twice",..: 8 1 1 1 1 3 1 1 1 1 ...
 $ hospitality     : Factor w/ 2 levels "no","yes": 2 1 1 1 1 1 1 1 1 1 ...
 $ victim          : Factor w/ 2 levels "no","yes": 2 1 1 1 1 2 1 1 2 1 ...
 $ religion        : Factor w/ 8 levels "buddhist","christian (all denom)",..: 4 7 7 7 2
 7 7 2 7 7 ...
 $ deathpen        : num  1 5 4 1 2 3 3 5 1 5 ...
 $ multiculturalism: num  1 2 1 1 1 1 2 1 1 2 ...
 $ stifferjail     : num  1 1 2 2 1 2 2 3 4 3 ...
 $ voleuthan       : num  1 4 3 2 2 2 1 3 1 4 ...
```

Figure 8: **Figure 8** | Partial output from running function *str()* on *binge* object

Note the format of the output The **$** denotes a variable which is named in the next column. After the colon ":" come, respectively, the variable type (numeric vs Factor, see below) and some information about the variable which is different according to type. For **numerics**, *str()* has output the values contained in the first few observations. For **Factors**, it tells us how many levels the Factor has, any associated labels (in SPSS-speak, these would be the value labels) it orders these alphabetically, and then prints out the first few actual values of the variable. For example, *bornaus* is a Factor with 2 levels (i.e., *no* and *yes*) and the first few values are *2 2 1 1 2 2 2 1 1 2*. Numeric variables, such as *deathpen* and *multicultualism* have no labels and only the first few values are output (e.g., *deathpen*: *1 5 4 1 2 3 3 5 1 5*). Below, I will discuss how to obtain numeric and Factor variables in different circumstances. I have modified the original BDRV data in SPSS to make this clearer for this guide.

The benefit of *str()* is that it very quickly and efficiently summarises our entire data set. You can also literally copy-and-paste the output into a text editor like Notepad in Windows or Mac equivalent (try it now) to create a document that summarises the data and can be shared with the whole lab or other colleagues. There are also easy tricks to make it very nice, even using text-to-tables, in word processors.

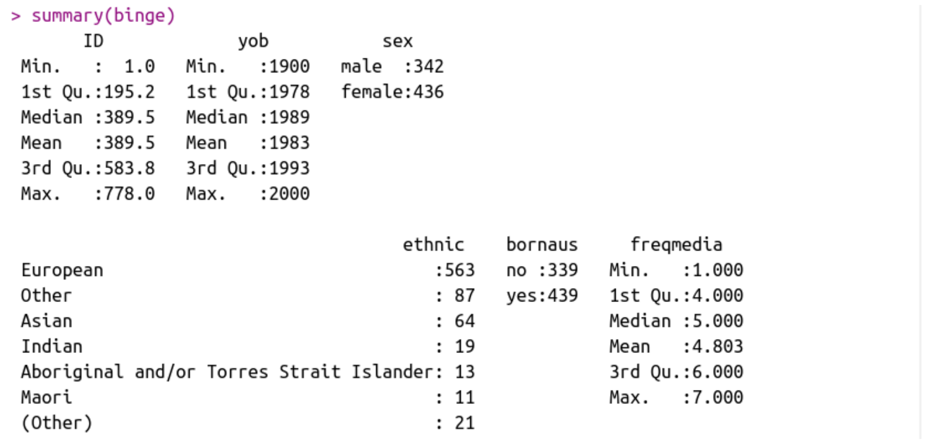
Whatever you *thought* was going in, *str()* shows what *is actually in* the created data frame object and that is what counts for R. If this doesn't match expectations, you'll need to investigate. There are functions that produce similar, but less detailed information, but I like the "just right" complexity that *str()* has. You'll probably only need to run it once when you first create a data frame object. Once you know the code creates the expected data frame from the data source, you can run it again and again in each R session to get the same data frame. Here is another R philosophy: **R users don't tend to save data frames. They save the correct code to generate them**. You'll see this in action soon. It's actually a very efficient way to avoid having 6 files called some variation of *thesis_data_final* in your folders! Leave the data as is and create and save the code that manipulates it and puts it into an R object that is used for further analysis. Because it's code and will have an audit trail, it's easy to recreate your steps each time and arrive at the same object.

3.2.2 Summarising variables with descriptive statistics

Study the next piece of code very carefully. It's very complex. To create summary descriptives as appropriate for each variable in a data frame, use the following code:

`summary(binge)`

Again, I have screenshotted a portion (Fig. 9); this time a portion from the top and then the bottom of the output generated.



```
> summary(binge)
       ID              yob            sex
 Min.   :  1.0   Min.   :1900   male  :342
 1st Qu.:195.2   1st Qu.:1978   female:436
 Median :389.5   Median :1989
 Mean   :389.5   Mean   :1983
 3rd Qu.:583.8   3rd Qu.:1993
 Max.   :778.0   Max.   :2000

                                       ethnic    bornaus     freqmedia
 European                                 :563   no :339   Min.   :1.000
 Other                                    : 87   yes:439   1st Qu.:4.000
 Asian                                    : 64             Median :5.000
 Indian                                   : 19             Mean   :4.803
 Aboriginal and/or Torres Strait Islander: 13             3rd Qu.:6.000
 Maori                                    : 11             Max.   :7.000
 (Other)                                  : 21
```

* * *

And again, R knows what to do with the data frame object that is input to *summary()*. It gives the numbers in each category for the Factors; and the usual descriptive statistics for the numerics. The are plenty of packages that provide

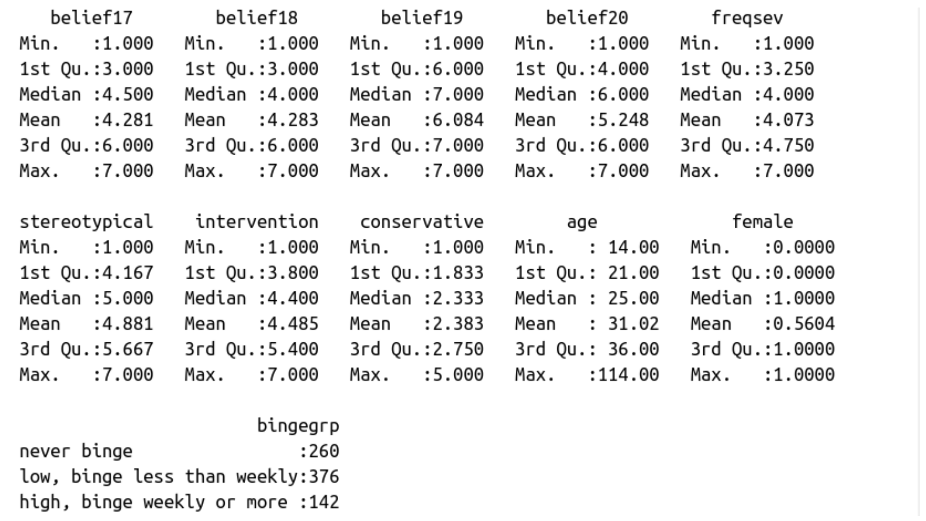

```
   belief17        belief18        belief19        belief20        freqsev     
 Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.000  
 1st Qu.:3.000   1st Qu.:3.000   1st Qu.:6.000   1st Qu.:4.000   1st Qu.:3.250  
 Median :4.500   Median :4.000   Median :7.000   Median :6.000   Median :4.000  
 Mean   :4.281   Mean   :4.283   Mean   :6.084   Mean   :5.248   Mean   :4.073  
 3rd Qu.:6.000   3rd Qu.:6.000   3rd Qu.:7.000   3rd Qu.:6.000   3rd Qu.:4.750  
 Max.   :7.000   Max.   :7.000   Max.   :7.000   Max.   :7.000   Max.   :7.000  

 stereotypical    intervention    conservative        age             female      
 Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   : 14.00   Min.   :0.0000  
 1st Qu.:4.167   1st Qu.:3.800   1st Qu.:1.833   1st Qu.: 21.00   1st Qu.:0.0000  
 Median :5.000   Median :4.400   Median :2.333   Median : 25.00   Median :1.0000  
 Mean   :4.881   Mean   :4.485   Mean   :2.383   Mean   : 31.02   Mean   :0.5604  
 3rd Qu.:5.667   3rd Qu.:5.400   3rd Qu.:2.750   3rd Qu.: 36.00   3rd Qu.:1.0000  
 Max.   :7.000   Max.   :7.000   Max.   :5.000   Max.   :114.00   Max.   :1.0000  

                        bingegrp  
 never binge                :260  
 low, binge less than weekly:376  
 high, binge weekly or more :142  
```

Figure 9: **Figure 9** | Partial output from running function *summary()* on *binge*

summary stats in different ways and that is one of the benefits of R. Users have created packages with functions that may be helpful to other users and tens of thousands of these are available for free from CRAN, where you first downloaded R. Many stats books have specially created R packages that contain all the data sets and usually functions to do the different analyses. This is really a vast topic and I won't just skim it here. I want to make sure you can at least do the task in one basic, dependable way first.

Now you are going to flex your R muscles a tiny bit. Do you always want to get summary statistics for every variable in a data frame? Here is how you specify one single variable. And as a bonus, you are learning the generic way to identify a specific variable in a specific data frame. You might easily have multiple data frames in your environment with similarly named variables (perhaps to compare them) and you want to be able to specify, *this variable* in *this data frame*. The appropriate lines of code are below (this time I have echoed the entire output into the document).

```
summary(binge$ethnic)
```

```
## Aboriginal and/or Torres Strait Islander
##                                        13
##                                   African
##                                        10
##                                     Asian
##                                        64
##                                  European
##                                       563
##                                                     Indian
##                                                         19
##                                                      Maori
##                                                         11
##                                             Middle Eastern
##                                                         11
##                                                      Other
##                                                         87
```

```
summary(binge$belief9)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    4.00    5.00    4.91    6.00    7.00
```

1. Note that this code uses exactly the same function but different parameters are fed to it. Instead of a whole data frame object, a variable is specified. In the first line, *ethnic* is a Factor and triggers appropriate output. In the second line, *belief9* is a numeric (i.e., continuous scale item) and that also triggers appropriate output. R figures it out.
2. Note the "$" operator. This essentially means, *you are about to specify a subpart of an object.* In this case, the object is a dataframe and the subpart is a variable. The generic template is:

```
data_frame$variable
```

There are many types of objects in R and whole chapters and books have been written about this. For now, you just need to use this tiny bit of coding language to deal with your data frames!

Many times in real data analysis, you will want to specify a group or subset of variables, such as the items of a scale. It would be tedious to type out a separate line such as `summary(binge$belief1)` for every item in a 20-item scale and you won't need to. There is a subsetting function whose job is to select subsets of parts (e.g., variables; observations) of an object (e.g., a data frame). Below, I use it to select subsets and nest it in the *summary()* function.

```
summary(subset(binge, select=belief1:belief20))
```

```
##     belief1         belief2         belief3         belief4
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.000
##  1st Qu.:4.000   1st Qu.:1.000   1st Qu.:4.000   1st Qu.:2.000
##  Median :5.000   Median :2.000   Median :5.000   Median :3.000
##  Mean   :4.968   Mean   :2.611   Mean   :5.004   Mean   :3.514
##  3rd Qu.:6.000   3rd Qu.:4.000   3rd Qu.:6.000   3rd Qu.:5.000
##  Max.   :7.000   Max.   :7.000   Max.   :7.000   Max.   :7.000
##     belief5         belief6         belief7         belief8         belief9
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.00
##  1st Qu.:2.000   1st Qu.:4.000   1st Qu.:4.000   1st Qu.:4.000   1st Qu.:4.00
##  Median :4.000   Median :5.000   Median :5.000   Median :5.000   Median :5.00
##  Mean   :3.711   Mean   :4.891   Mean   :4.995   Mean   :4.963   Mean   :4.91
##  3rd Qu.:5.000   3rd Qu.:6.000   3rd Qu.:6.000   3rd Qu.:6.000   3rd Qu.:6.00
##  Max.   :7.000   Max.   :7.000   Max.   :7.000   Max.   :7.000   Max.   :7.00
##     belief10        belief11        belief12        belief13       belief14
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.00   Min.   :1.000
##  1st Qu.:4.000   1st Qu.:2.000   1st Qu.:2.000   1st Qu.:2.00   1st Qu.:4.000
##  Median :6.000   Median :4.000   Median :4.000   Median :3.00   Median :5.500
##  Mean   :5.246   Mean   :3.746   Mean   :3.653   Mean   :3.26   Mean   :5.192
##  3rd Qu.:6.000   3rd Qu.:6.000   3rd Qu.:5.000   3rd Qu.:4.00   3rd Qu.:7.000
##  Max.   :7.000   Max.   :7.000   Max.   :7.000   Max.   :7.00   Max.   :7.000
##     belief15        belief16        belief17        belief18
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.000
##  1st Qu.:4.000   1st Qu.:4.000   1st Qu.:3.000   1st Qu.:3.000
##  Median :5.000   Median :5.000   Median :4.500   Median :4.000
##  Mean   :5.031   Mean   :4.865   Mean   :4.281   Mean   :4.283
##  3rd Qu.:6.000   3rd Qu.:6.000   3rd Qu.:6.000   3rd Qu.:6.000
##  Max.   :7.000   Max.   :7.000   Max.   :7.000   Max.   :7.000
##     belief19        belief20
##  Min.   :1.000   Min.   :1.000
##  1st Qu.:6.000   1st Qu.:4.000
##  Median :7.000   Median :6.000
##  Mean   :6.084   Mean   :5.248
##  3rd Qu.:7.000   3rd Qu.:6.000
##  Max.   :7.000   Max.   :7.000
```

The same function can be used to select specific groups of observations (e.g., just the female participants) or subsets of observations and variables together, by varying the parameters that are set. The *subset()* function is clever enough to recognise that *sex* is a variable from the named data frame object *binge* in the input. This also demonstrates that functions may have many *possible* parameters but it is rarely necessary to specify any but a few key ones when they are actually used in code.

```
summary(subset(binge, sex == "female", select=belief1:belief5))
```

```
##     belief1         belief2         belief3         belief4
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.000
##  1st Qu.:4.000   1st Qu.:1.000   1st Qu.:4.000   1st Qu.:2.000
##  Median :5.000   Median :2.000   Median :5.000   Median :3.000
##  Mean   :4.989   Mean   :2.615   Mean   :5.085   Mean   :3.367
##  3rd Qu.:6.000   3rd Qu.:4.000   3rd Qu.:6.000   3rd Qu.:5.000
##  Max.   :7.000   Max.   :7.000   Max.   :7.000   Max.   :7.000
##     belief5
##  Min.   :1.000
##  1st Qu.:2.000
##  Median :4.000
##  Mean   :3.732
##  3rd Qu.:5.000
##  Max.   :7.000
```

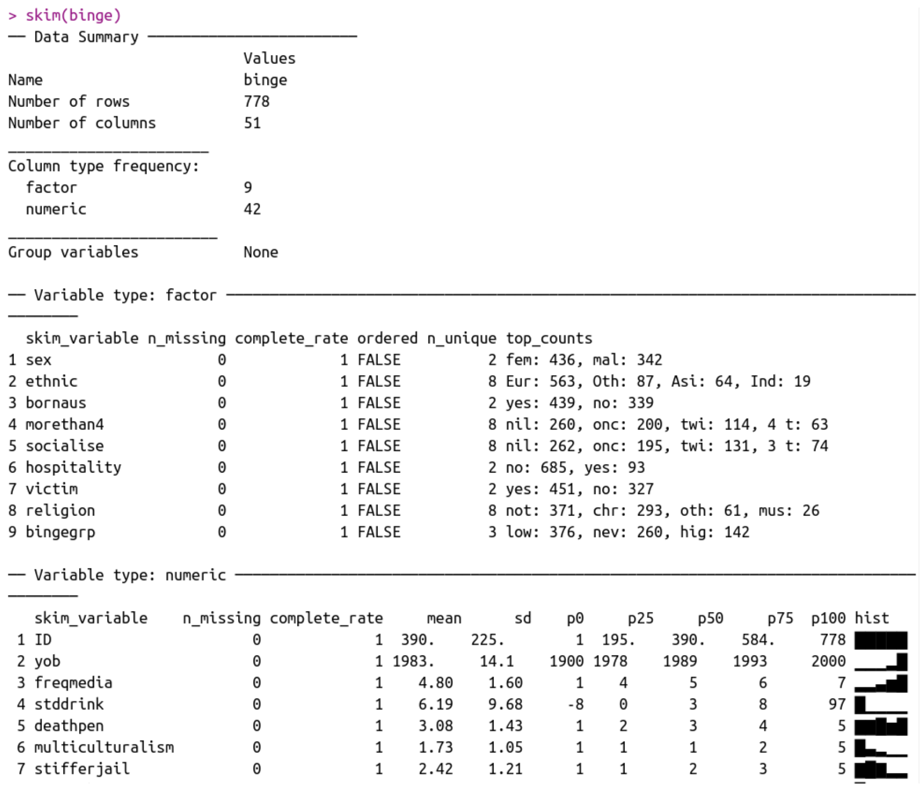
This output prints out the descriptive statistics for items 1 to 5 of the belief scale just for female participants. There is much more complexity possible with the *subset()* function. Note again that nothing has changed about the *summary()* function; instead, the *subset()* function has been nested inside to change the input that *summary()* gets. In turn, you could use *subset()* nested appropriately inside *lm()* to specify the observations to be used for a regression (e.g., run this regression just for the female participants; or the participants who score more than a cutoff score on another instrument). The system is **extremely** modular. **As much as possible, use the same functions to do similar tasks and R can almost always figure out how to adapt them** This is not just "cool" but it helps to avoid silly errors that get in the way of data analysis.

In R, there are often many ways to do the same basic task: another useful package and function for descriptive and summary statistics. There is always another way to do the same thing in R and this is both a benefit for people doing data analysis and a potential pitfall for people learning R. To illustrate a little how this is possible, consider a particularly useful function that was pointed out by a recent workshop participant. It's the *skim()* function from the `skimr` package. This package is designed with respect to a whole system of doing data transformations in R called the tidyverse. But R is so modular, one can easily just grab this one package and use it without needing to buy into the whole thing. Before using the function, you will need to load the package. And if you hadn't installed it yet, you would need to do so via the *packages* pane in R Studio (self-explanatory) or via code: `install.packages("skimr")`

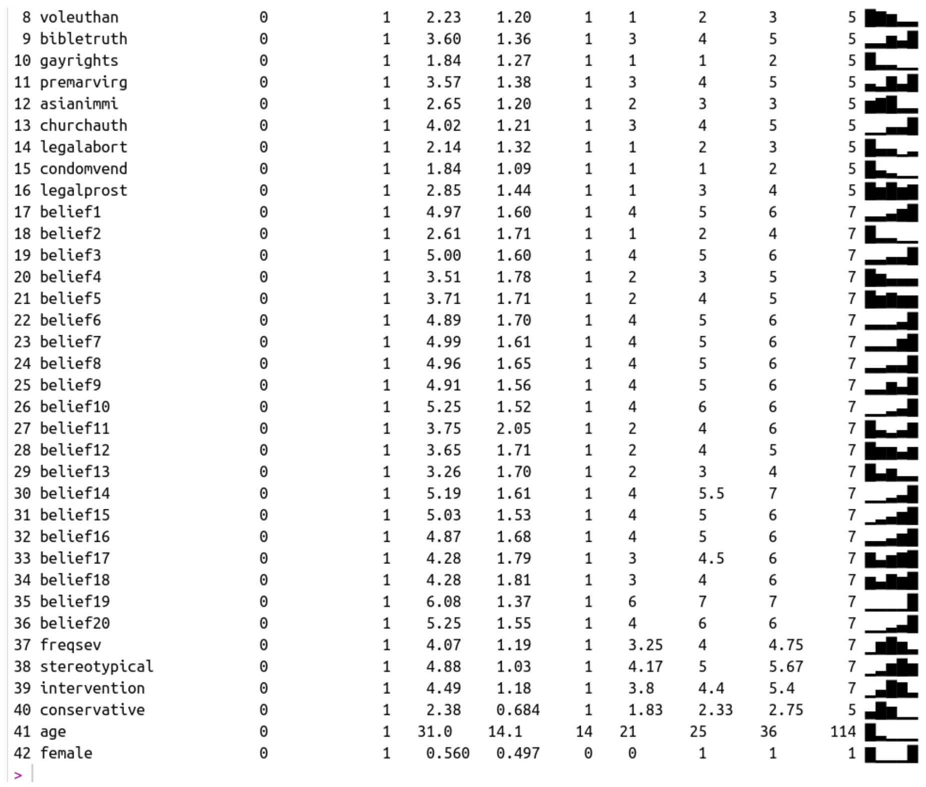
Then, to load the package: `library(skimr)`

Once loaded, the *skim()* function is available to the R session. All of the output is shown below. To have yours look the same in the R Studio console pane, you will probably need to widen the console pane considerably by dragging with the mouse or the output will wrap. You can run and re-run until you get it right! When you do, you can once again select it all and paste it into a text editor for a perfectly proportioned table that includes the histograms (or see below for word processor instructions).

```
skim(binge)
```



```
> skim(binge)
── Data Summary ────────────────────────
                           Values
Name                       binge
Number of rows             778
Number of columns          51
_______________________
Column type frequency:
  factor                   9
  numeric                  42
________________________
Group variables            None

── Variable type: factor ──────────────────────────────────────────────────────────────────────
────────
  skim_variable n_missing complete_rate ordered n_unique top_counts
1 sex                   0             1 FALSE          2 fem: 436, mal: 342
2 ethnic                0             1 FALSE          8 Eur: 563, Oth: 87, Asi: 64, Ind: 19
3 bornaus               0             1 FALSE          2 yes: 439, no: 339
4 morethan4             0             1 FALSE          8 nil: 260, onc: 200, twi: 114, 4 t: 63
5 socialise             0             1 FALSE          8 nil: 262, onc: 195, twi: 131, 3 t: 74
6 hospitality           0             1 FALSE          2 no: 685, yes: 93
7 victim                0             1 FALSE          2 yes: 451, no: 327
8 religion              0             1 FALSE          8 not: 371, chr: 293, oth: 61, mus: 26
9 bingegrp              0             1 FALSE          3 low: 376, nev: 260, hig: 142

── Variable type: numeric ─────────────────────────────────────────────────────────────────────
────────
   skim_variable    n_missing complete_rate     mean      sd    p0    p25     p50     p75  p100 hist
 1 ID                       0             1  390.    225.        1  195.    390.    584.    778 ▇▇▇▇▇
 2 yob                      0             1 1983.     14.1    1900 1978    1989    1993    2000 ▁▁▁▂▇
 3 freqmedia                0             1    4.80    1.60      1    4       5       6       7 ▂▂▃▅▇
 4 stddrink                 0             1    6.19    9.68     -8    0       3       8      97 ▇▁▁▁▁
 5 deathpen                 0             1    3.08    1.43      1    2       3       4       5 ▇▆▇▅▇
 6 multiculturalism         0             1    1.73    1.05      1    1       1       2       5 ▇▃▂▁▁
 7 stifferjail              0             1    2.42    1.21      1    1       2       3       5 ▇▇▅▂▂
                                                                                                 ▁
```



```
 8 voleuthan        0        1   2.23   1.20    1   1     2     3      5
 9 bibletruth       0        1   3.60   1.36    1   3     4     5      5
10 gayrights        0        1   1.84   1.27    1   1     1     2      5
11 premarvirg       0        1   3.57   1.38    1   3     4     5      5
12 asianimmi        0        1   2.65   1.20    1   2     3     3      5
13 churchauth       0        1   4.02   1.21    1   3     4     5      5
14 legalabort       0        1   2.14   1.32    1   1     2     3      5
15 condomvend       0        1   1.84   1.09    1   1     1     2      5
16 legalprost       0        1   2.85   1.44    1   1     3     4      5
17 belief1          0        1   4.97   1.60    1   4     5     6      7
18 belief2          0        1   2.61   1.71    1   1     2     4      7
19 belief3          0        1   5.00   1.60    1   4     5     6      7
20 belief4          0        1   3.51   1.78    1   2     3     5      7
21 belief5          0        1   3.71   1.71    1   2     4     5      7
22 belief6          0        1   4.89   1.70    1   4     5     6      7
23 belief7          0        1   4.99   1.61    1   4     5     6      7
24 belief8          0        1   4.96   1.65    1   4     5     6      7
25 belief9          0        1   4.91   1.56    1   4     5     6      7
26 belief10         0        1   5.25   1.52    1   4     6     6      7
27 belief11         0        1   3.75   2.05    1   2     4     6      7
28 belief12         0        1   3.65   1.71    1   2     4     5      7
29 belief13         0        1   3.26   1.70    1   2     3     4      7
30 belief14         0        1   5.19   1.61    1   4     5.5   7      7
31 belief15         0        1   5.03   1.53    1   4     5     6      7
32 belief16         0        1   4.87   1.68    1   4     5     6      7
33 belief17         0        1   4.28   1.79    1   3     4.5   6      7
34 belief18         0        1   4.28   1.81    1   3     4     6      7
35 belief19         0        1   6.08   1.37    1   6     7     7      7
36 belief20         0        1   5.25   1.55    1   4     6     6      7
37 freqsev          0        1   4.07   1.19    1   3.25  4     4.75   7
38 stereotypical    0        1   4.88   1.03    1   4.17  5     5.67   7
39 intervention     0        1   4.49   1.18    1   3.8   4.4   5.4    7
40 conservative     0        1   2.38   0.684   1   1.83  2.33  2.75   5
41 age              0        1  31.0   14.1    14  21    25    36    114
42 female           0        1   0.560  0.497   0   0     1     1      1
>
```

Things to note.

1. *skim()* seems to combine the information from both *str()* and *summary()* into a very useful format. Someone, a user, has programmed R using underlying functions to do that.
2. Instead of either the statistics or histograms, *skim()* provides *both* and places a miniature histogram of each numeric (perfect for *skim*ming the data) on the same line as the descriptives.
3. Again, the function neatly deals with Factors and numerics in our data frame.
4. The output is very neat and looks like it could easily be converted into a PDF or other document. In fact, it looks like it was composed using tools from *markdown* (the mark up language in which this guide is written). This gives the neat tables and carefully justified tabs. Perusal of the *help()* entry for the function suggests it is indeed designed to work with tools like R Markdown and other functions that would take *skim()* output to create very elegant tables. In fact, *skim()* objects can be fed into other R functions from user packages that facilitate printing and saving very clean tables such as the one in this guide. Even without that knowledge (yet) you can cheat by selecting the *skim()* output, pasting into a word processor, selecting it all again and choosing a font such as Courier (or Lucida Sans Typewriter) that is non-proportional (fixed width). You will find that

with font size of about 9 points and narrower margins, everything lines up beautifully. Even the histograms are created by ASCII block characters that work in text files. Try it!

And just to bring the point home about how modular things are in R, consider the code snippet below. What do you think it will do? Run it and see if you were right.

```
skim(subset(binge, select=belief1:belief20))
```

3.3 Reading files with R by using the `here` package

So far, I have had you address the data file *BDRVdataR.sav* with its full R Project path. It is in the *data* subdirectory that sits directly under the root project directory, so the full path *data/BDRVdataR.sav* works. But what if you wanted to change the directory structure and introduce an intermediate directory between the root and *data*? This would instantly break all the file path statements in code and for a large project this might be considerable. As well, getting these paths in the first place can be a trial. You almost certainly will not have noticed, but the paths in the R code snippets in this guide have used forward slashes "/". Some filesystems use forward slashes in paths (e.g., linux ad all Unix derivatives) and some use backslashes "\" (e.g., Windows). For most of us, this is a tiny detail that is easy to forget. But R *always uses forward slashes*. If you were to copy a bunch of Windows-formatted file paths (e.g., *C:\Users\You\Documents\UniStuff\6020PSY\R_course*) and paste them into R code, they wouldn't work and you'd have to search-and-replace all the back slashes with forward slashes. Clearly, this is not a good way to work and introduces many chances for error.

Enter the `here` package. This package takes care of paths for users. As long as you specify one (sometimes two) unambigous folders on the path that contains the target file, `here` will sort it out. An example will explain better than many paragraphs.

```
binge <- data.frame(read.spss(here("data", "BDRVdataR.sav")))
```

You should recognise this as being very similar to the same code snippet used above to finally read the BDRV data into an object called *binge*. Another function has been nested in it. That's 3 nested functions and it is nothing out of the ordinary. It seems that the *here()* function takes as input some parts of the filename and *path* of the BDRV data and then passes this onto *read.spss()* which in turn passes its output onto *data.frame()* which finally assigns the output to the *binge* object. You may suspect this code has read the BDRV data from SPSS again in a different way and you would be correct.

`here("data", "BDRVdataR.sav")` basically says, "in a path with a folder called *data* in it, there is a file called *BDRVdataR.sav*". Note that both the filename and the folder are in double quotes, "like", "this"? This is to make the parts of the path clear for *here()*. It seems silly to do this if the exact path is just *data/BDRVdataR.sav*. But what if in a month's time you decided to restructure the project folder because it had grown too much and you put a folder called *base_files* off the root and then *data* off *base_files*? The correct path would then be *base_files/data/BDRVdataR.sav* and any code with the old, simple path would stop working and would need to be fixed. However, the *here()* version,

```
binge <- data.frame(read.spss(here("data", "BDRVdataR.sav")))
```

would keep working unmodified. The *here()* function is such an important function for locating files consistently (mostly data but plenty of other types as you go) that it is often written in a special way. And you should do this too:

```
binge <- data.frame(read.spss(here::here("data", "BDRVdataR.sav")))
```

Note that I have written *here::here*. What this means is that in the code above, R should specifically use the *here()* function from the `here` package *whether or not it is loaded*. If someone else were to have written another package that does something else entirely but happens to have a function called *here()* in it, and you were to have loaded that package without realising this, then if you just used *here()*, as above, R would use the wrong version of *here()* and that would break the code. Writing *here::here* is sort of like using both a first name and a surname to be clear who you mean. You are saying, "not just any *here()* function that may be in the workspace, but only the *here()* function from the `here` package!" You need to have *installed* the package into your installation (as above), but *here::here* will definitely work. It's another modular thing. So, `package::function` will always work but it is a tiresome way to have to write all code like this so it is reserved for infrequent but important situations, or when quickly sketching out an example, as you will see below in this guide. This is one of those things that you *just do* and slowly the meaning will seep into your mind.

Your new mantra: **When I use R, I set up an R Project, as shown above, and I put my data files in the *data* directory and I use a line like: `binge <- data.frame(read.spss(here::here("data", "BDRVdataR.sav")))` to read them.**

3.4 Reading CSV files and turning them into data frame objects

The examples above worked with a data frame object that had been created from an SPSS file (you may recall that at the beginning of the session the code loaded the `foreign` package to allow the use of the *read.spss()* function). Often, data has not been set up as an SPSS file and is in a CSV (comma separated value) format file. Some examples:

- data from an external database, extracted and provided to you as a *flat file* spreadsheet which will almost certainly be a CSV file

- data that was gathered in a program like Redcap or Qualtrics that output CSV or similar files by default (in fact, even in the SPSS download option, Qualtrics gives you CSV data and a clunky SPSS syntax file to run it)
- you gathered the data yourself and have entered in a spreadsheet because you didn't have convenient access to SPSS (it's not free…) and you can save or export it as a CSV file
- many more reasons…

Base R has a function called *read.csv()* and it is just what you need. Because you now understand how functions work, I can just list it below and most of it will be clear.

```
binge2 <- data.frame(read.csv(here::here("data", "BDRVdataR.csv")))
```

You can see that the only changes in the code are that:

1. the object name is *binge2* to differentiate it from the above;
2. and *read.spss()* has become *read.csv()*.

Again, the modular nature of R and the way that nested functions are used to perform all the steps of a more complex task means these are the only changes necessary. You could use the *binge2* object in exactly the same way as the *binge* object. As the data file is identical, the object will be identical except that there are no labels as there are in the SPSS version. The statistics will be exactly the same. You can check yourself easily by running a line of code such as:

```
skim(subset(binge2, select=belief1:belief20))
```

3.5 Preparing data to have numerics and Factors as appropriate (especially when working with SPSS files)

It is clear from the above that being careful about variables that should be Factors or numerics, respectively, will go a long way to facilitating efficient work. *This is something to read over and keep in mind for setting up your own data.* I want you to be able to have access to this information but it is best that you get used to actually working with R first. The data I will provide for examples in this guide will be approporiately set up. There are two broad ways to go about this.

1. **if you start with a CSV file** created in a spreadsheet program (e.g., Microsoft Excel; LibreOffice Calc; some Mac thing) **enter numbers as numbers and words as words**. When R reads the data, it will automatically leave numbers as numerics and it will turn words (i.e., string variables) into Factors. For example, a rating scale item answered on a 1 to 7 scale will obviously become a numeric in R. By contrast, a binary question that was answered "yes" or "no", will become a Factor.

- If you are entering data, you can control what variables become Factors by making sure they have words as responses (always best to avoid spaces; DK vs "don't know"). This is also true if you are gathering data online as this export option can be sorted out in programs like Redcap and Qualtrics.

- **One crucial change may be necessary to the way that you have learned to enter data**. Where you have for years been told to assign numbers to all ordinal and nominal categorical variables and then enter those numbers into data that was destined for SPSS, you won't do that now. For example, marital status might have been coded as 1 = married/de fact, 2 = divorced, 3 = separated… and so on. When preparing data for R, *don't replace the categories with numbers*, leave them as words (without spaces). This is also true of dummy variables that already exist in the data (the ones you may create later can be dealt with then). For example, the BDRV data contained a question asking whether both of a person's parents and all four of their grandparents were born in Australia (*bornaus*) and the responses were: *yes* and *no*. Instead of coding yes = 1 and no = 0, or something similar, you would enter that as *yes* or *no*. Because of the way that R deals with Factors, you will find that your stats with linear models become amazingly efficient. *You need to trust me on this one!*
- **There may also be a situation where you find it is still useful to enter certain categorical variables with numerical codes**. Don't worry. There are, in fact, functions in base R that allow you to treat numerics as Factors (and other things too) if necessary. But you knew I was going to say that! It's called *coercion* but I am trying to avoid making your head swim. For now, for simple psych data, focus on numeric items (usually interval and ratio scales that will be DVs) and categorical Factors (usually nominal or ordinal variables that will be IVs/Factors in ANOVAs and may also be deal with by chi square).

2. **If you start with an SPSS file**, you just need to ensure that anything you want to be a Factor variable is either
    - words; or
    - has value labels.

- **Anything with value labels will be read as a Factor in R by default**. Anything that contains only numbers and has no value labels remains numeric. But it is also possible to set a parameter in the *read.spss()* function to stop this!
- A trick here is to make sure that numeric variables such as scale items don't still have value labels, even just the extremes such as *strongly agree* and *strongly disagree*. In the past, we all used to do this sometimes to help to label output for frequency tables and similar, but it is essentially pointless. In SPSS syntax, you can drop value lables easily and in one easy line. If you only use the menus, you've got a lot of clicking ahead of you…
- again, coercion is available

3.6 End of module

4 Module 4: Regression diagnostics

To begin, open the regressionDiagnostics.R file and run the first lines that clear the workspace and load packages. For future modules, I will not mention this but you should assume it. You now understand what `rm(list=ls())` and `library(foreign)` do.

The easiest place to start with statistical analyses in R is with ordinary least squares (OLS) regression and the diagnostics associated. It is the most straightofrward implementation of the general linear model (GLM). In this module, I will look at:

- Creating an *lm()* object for regression
  - initial assessment of output
- Generating:
  - CIs
  - ANOVA table
- Basic diagnostics
  - plots
  - scatterplots
  - residuals and residual plots
  - leverage statistics and plots
  - Checking for heteroscedasticity
  - Q-Q plots of the residuals
  - added-variable plots
  - dropping observations

I assume that you know what each of those things are. I will take a typical OLS regression and run it and generate quantities such as residuals and leverage and appropriate plots in a more or less classical tradition of influence and diagnostics. **All** of the tutorials you find on the web and in psych classes are reinterpretations of the classic papers of Bollen and Jackman (1985) and Stevens (1984), and probably also of Cook and Weisberg's (1981) *Regression Diagnostics*, and the present guide is no different. Occhipinti and Tapp (2018; *Data preparation* chapter in Brough, P. [Ed.] *Advanced Research Methods for Applied Psychology: Design, Analysis and Reporting*; Routledge) provide a handy, chapter length treatment of this and of the principles of preparing data for analysis (shameless plug alert) but consider looking at Bollen's and Stevens' papers directly!

4.1 OLS Regression in R

As you have seen already in Part 1, regression is easy. It makes use of the *lm()* function and a moment's deduction shows that *lm* stands for *linear model*.

It is always best to send the output of *lm()* to an object for examination via *summary()* or further processing, as I will show below.

The following line of code is in the *linearModels.R* file and it is repeated below. Before it runs, make sure to read the data into an object called *binge*. Note that the *regint* object should be in your environment after you have run it. As we have assigned the function's output to an object, you will only see the code itself echoed in the console.

```
regint <- lm(intervention~age + sex + hospitality + conservative, data=binge)
```

To reiterate the prototypical form of regression via lm(), see the below:

```
lm(DV ~ predictor + predictor ... + predictor, data=data frame
object)
```

This is very easily modified to incorporate things such as product terms for moderation or curvilinear analysis. For example, note the tiny difference (covered later in more depth):

```
lm(DV ~ predictor + predictor ... + predictor*predictor, data=data
frame object)
```

To examine the output of the regression, the base R function is again *summary()*:

```
summary(regint)
```

```
##
## Call:
## lm(formula = intervention ~ age + sex + hospitality + conservative,
##     data = binge)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.3854 -0.6485  0.0366  0.7241  2.7071
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)     2.44444    0.16079  15.202  < 2e-16 ***
## age             0.02695    0.00275   9.799  < 2e-16 ***
## sexfemale       0.24741    0.07617   3.248  0.00121 **
## hospitalityyes -0.10715    0.11840  -0.905  0.36577
## conservative    0.45265    0.05564   8.135 1.64e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.052 on 773 degrees of freedom
## Multiple R-squared:  0.208,  Adjusted R-squared:  0.2039
## F-statistic: 50.76 on 4 and 773 DF,  p-value: < 2.2e-16
```

For most uses, this will be fine as the point of checking generally is to ascertain the nature of the results before planning a next move. However, to illustrate the flexibility of R, consider that a textbook author and statistician called Julian Faraway decided that the information contained in the *summary()* output was a bit bloated and redundant when running many models and checking them as for a large analysis. He wrote his own version and called it *sumary()* [n.b., the single *m* is not a typo]. You don't need to install the `faraway` package that goes with his text (it can be very useful), but the output of his function is below. Also note how I have called the function to avoid loading library just for this aside.

```
faraway::sumary(regint)
```

```
##                  Estimate Std. Error t value  Pr(>|t|)
## (Intercept)     2.4444368  0.1607920 15.2025 < 2.2e-16
## age             0.0269503  0.0027504  9.7987 < 2.2e-16
## sexfemale       0.2474107  0.0761662  3.2483  0.001211
## hospitalityyes -0.1071476  0.1184006 -0.9050  0.365770
## conservative    0.4526499  0.0556413  8.1351 1.638e-15
##
## n = 778, p = 5, Residual SE = 1.05242, R-Squared = 0.21
```

It's certainly neater and focuses on the b-weights (i.e., *Estimate* column) and their associated standard errors, t values and p values (*Pr(>|t|)*, note exponential notation). However, it does not echo the code containing the function call, but depending on how you write and run your code, this may not matter.

4.2 Working with the *lm()* object

Apart from the summary-type functions above, you may want more information anout specific aspects of the analysis. Remember, R deliberately does not spit out everything that could possibly be generated from a statistical procedure. Consider the below:

```
confint(regint)
```

```
##                      2.5 %     97.5 %
## (Intercept)     2.12879611 2.76007747
## age             0.02155116 0.03234934
## sexfemale       0.09789357 0.39692782
## hospitalityyes -0.33957248 0.12527726
## conservative    0.34342386 0.56187589
```

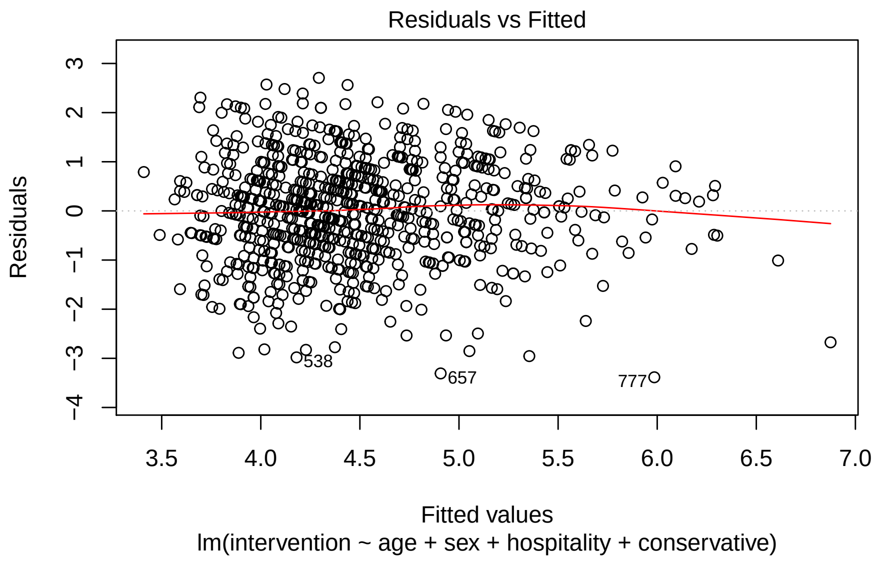
The *confint()* function gives 95% CIs by default. What if 99% CIs were required? The *level* parameter to the function can be explicitly specified to change that. Try: `confint(regint, level = .99)`.

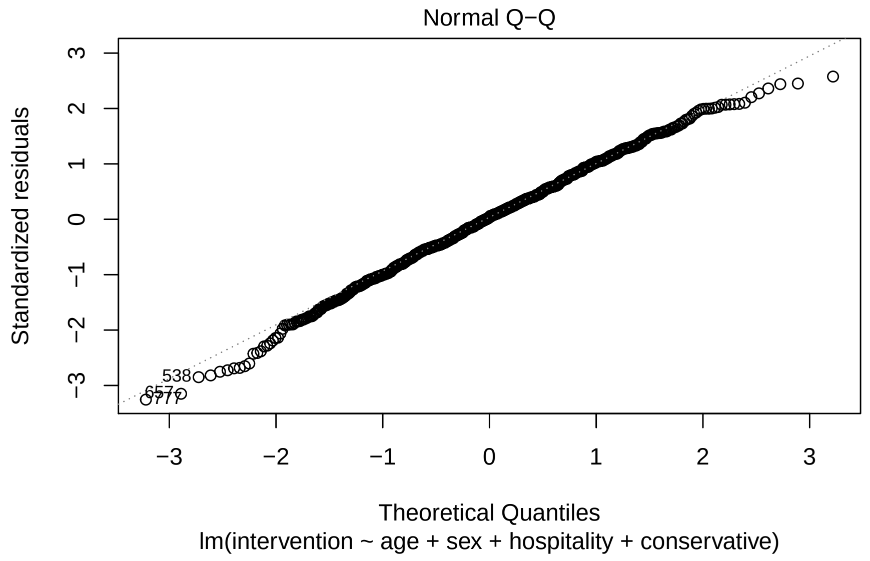
```r
anova(regint)
```

```
## Analysis of Variance Table
##
## Response: intervention
##               Df Sum Sq Mean Sq  F value    Pr(>F)
## age            1 136.78 136.777 123.4908 < 2.2e-16 ***
## sex            1  14.12  14.122  12.7503  0.000378 ***
## hospitality    1   0.68   0.683   0.6163  0.432648
## conservative   1  73.30  73.301  66.1805 1.638e-15 ***
## Residuals    773 856.16   1.108
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

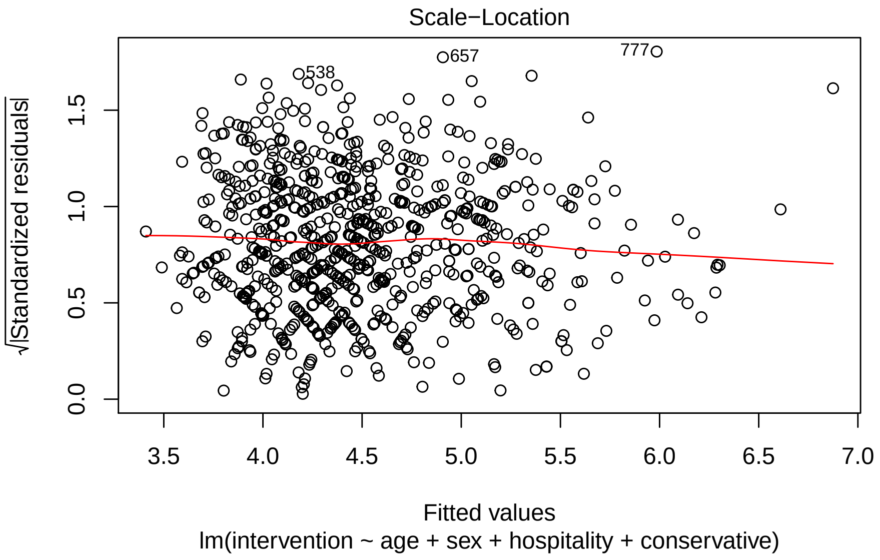
Take care when running the code below from R Studio. To see the plots, you need to click into the *console* pane of R Studio and push the **Enter** key once for each plot. The plots will pop up in the *Plots* pane below the *console*. Each can be zoomed into, exported as an image or as a PDF, or copied to the clipboard to paste into other programs such as Word, Powerpoint, or their open source equivalents. (Try it!)

```r
plot(regint)
```

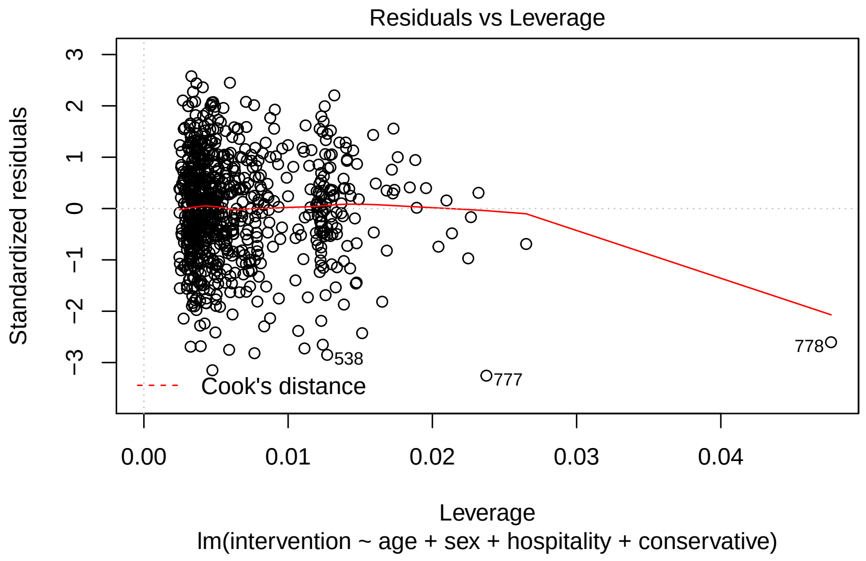




Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(intervention ~ age + sex + hospitality + conservative)
538
657
777



Scale−Location
√|Standardized residuals|
Fitted values
lm(intervention ~ age + sex + hospitality + conservative)
538
657
777

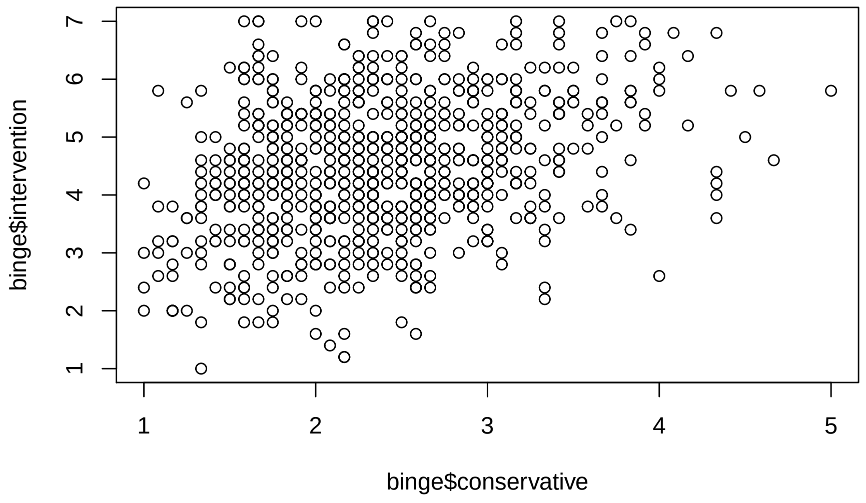


Note that these plots are the absolute basic, default level of graphics in R. It only gets better as you master other graphics functions. Also, note that in each case, you simply pass the *lm()* object *regint* into each new function and R does the necessary translation. An *lm()* object is of a special class in R but unless you are programming, you don't need to do anything about it. This is by design. R is modular like this and it makes doing stats much clearer for many people. After learning some of these concepts, the stats software stays out of the way of your learning about statistics.

4.3 Simple Diagnostics

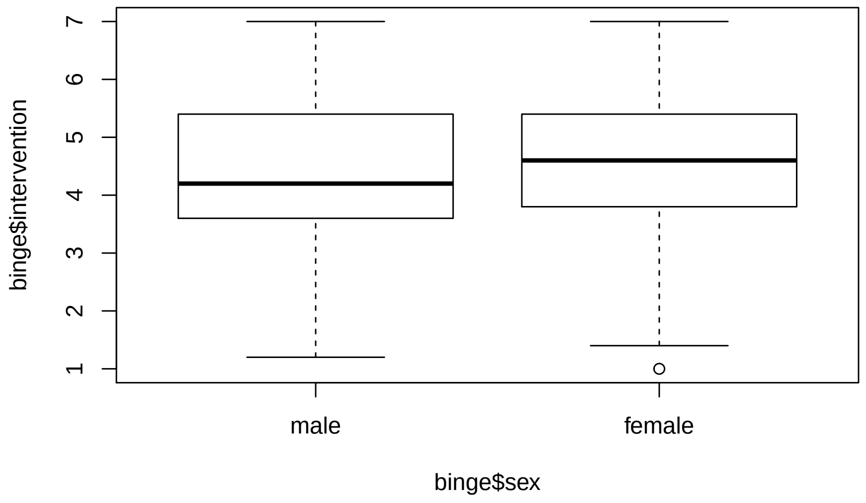
It is often useful to plot the criterion (i.e., Y variable) against the predictors (i.e., X variables). A simple bivariate plot can be obtained with the **same** *plot()* function used above. Again, note the modularity. R realises that the input to *plot()* is not an *lm()* object but is a pair of continous variable, so it produces a scatterplot:

```
plot(binge$intervention~binge$conservative)
```



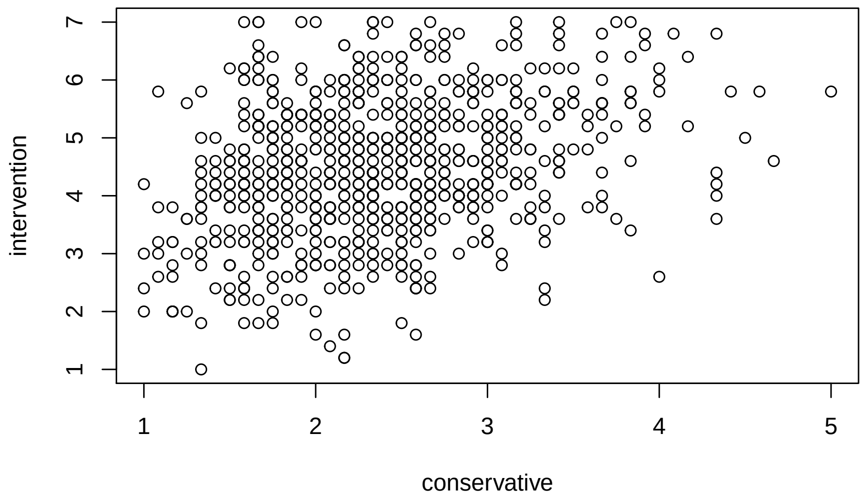
If instead *plot()* were fed a continuous Y variable and a discrete X variable:

```
plot(binge$intervention~binge$sex)
```



Further, if you wanted a few plots of variables from the BDRV data that is contained in the *binge* object and didn't want to have to type *binge$* each time, you could use the same *data* = parameter as for the *lm()* function calls above:

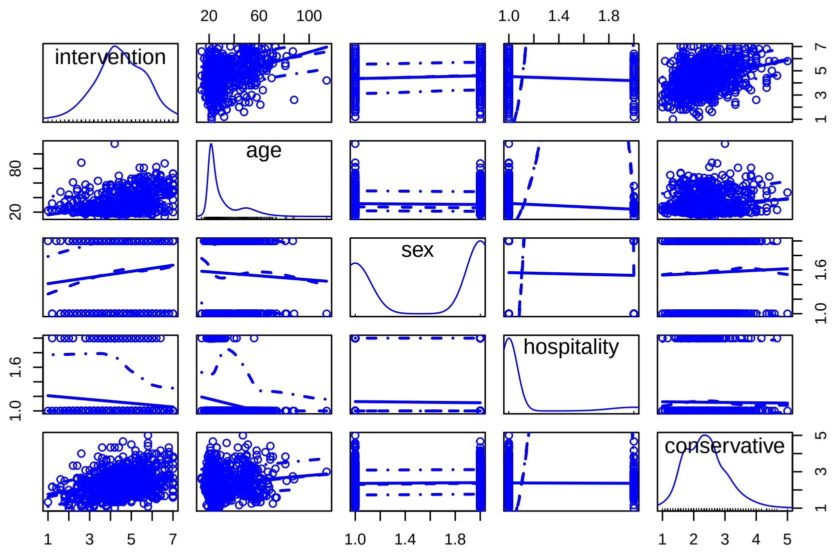
```
plot(intervention~conservative, data = binge)
```



In multiple regression, it is often of interest to obtain a series of scatterplots of all the predictors with the criterion and most software facilitates this. This function is from the `car` package, that you loaded at the head of the *linearModels.R* file.

```
## Loading required package: carData

## Registered S3 methods overwritten by 'car':
##   method                          from
##   influence.merMod                lme4
##   cooks.distance.influence.merMod lme4
##   dfbeta.influence.merMod         lme4
##   dfbetas.influence.merMod        lme4
```

```
scatterplotMatrix(~intervention + age + sex + hospitality + conservative, data=binge)
```



This is an example of a user written function that improves the way many can do statistical analysis. the `car` package was written for the textbook *Companion to Applied Regression* by Fox and Weisberg that is itsel loosely based on *Applied Regression Analysis and Generalized Linear Models*, by Fox. The `car` package is very widely used and pops up as a dependency in many other packages.

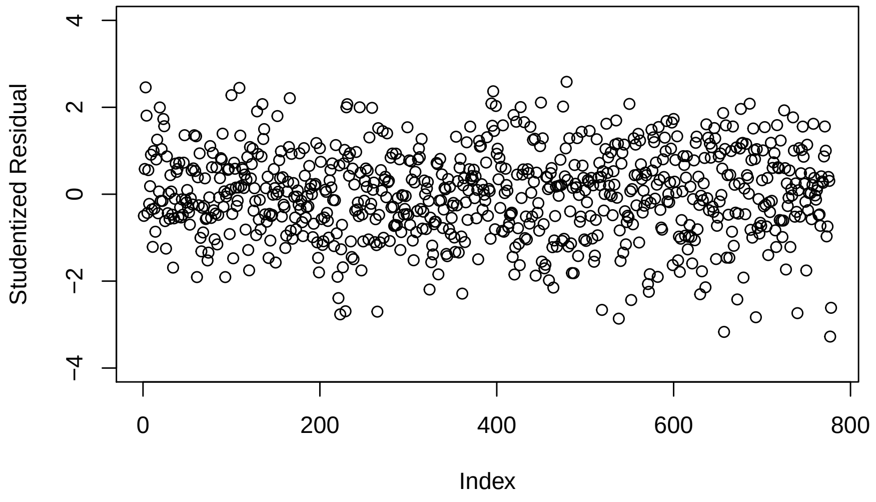
This type of figure has the (smoothed) distribution of each variable on the main diagonal and the first column (or row) has the scatterplot of the criterion variable with each predictor in turn, with fit lines and confidence bands. On the other offdiagonal elements are found the respective scatterplots of the predictors with each other. Note the distinctive pattern associated with predictors that are dummies. There is a great deal of worthwhile visual information in this figure.

Moving along with the diagnostics, the first quantity of interest is almost always the residuals. As the residuals are expressed in the scale of the criterion variable, they are usually standardised in some way to facilitate comparison with standard population values. There are various ways to standardise and in 6020PSY we teach studentized residuals with a cutoff of +/-3.3 but this is somewhat arbitraty (some authorities suggest lower cutoffs or different types of standardisation; whatever you do, be consistent!). The following line extracts the studentized residuals and assign them to a new object called *stud.res* (note the aptly named function)

```
stud.res <- rstudent(regint)
```

Your new friend the *plot()* function is used to generate the plot of residuals to check that they are centred around zero and are approximately normal (note that the residuals are on the Y axis. Again, it correctly guesses the type of plot required given the input it has received. The *ylab* function labels the Y axis. More on the *ylim* parameter, that sets the minimum and maximum *lim*its of the Y axis, below.

```r
plot(stud.res, ylab="Studentized Residual", ylim=c(-4,4))
```

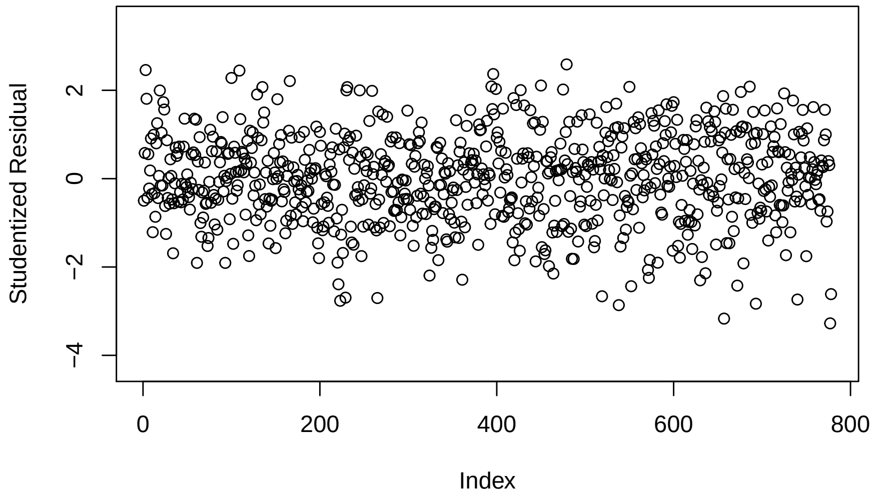


In the code snippet above, the Y axis parameter *ylim* is set with the format `c(-4, 4)`. The *c()* function is actually the way that lists of elements are input to parameters and functions as required. The c stands for *c*ombine. It is an extremely frequently used function (probably the most used in R) and has a large role in and out of statistical analysis. Using *c()* with lists and with special operators to repeat values or patterns is easily worth a chapter in a manual and is well outside the scope of this regression module. For users, when you see *c()*, you know that a list or series of elements is to be input. In the code snippet above, `c(-4, 4)` is used to give the minimum and maximum points of the Y axis in the plot. Consider what `c(1,8)` would produce. Note also that the code above shows that functions can be nested flexibly and not only as whole functions as in previous examples. `c(-4, 4)` is used to provide input to a specific parameter of *plot()* and not as the whole input.

A further modification is possible and it demonstrates how R can optimise analysis. In the above, -4 and +4 were chosen as Y axis limits as the residuals of interest in diagnostics are the extreme ones, defined here as larger than 3.3 in absolute value. Calling the *plot()* function without *ylim* makes a plot with a Y axis a tiny bit larger than the actual minimum and maximum values and using

-4/+4 creates a meaningful, symmetric frame around them. However, in this and other occasion, the user might want a plot with Y axis limits that match the actual minimum and maximum values of the data. The below makes a plot with Y axis limits 1 point larger than the actual minimum and maximum in the data, in absolute value.

```
plot(stud.res, ylab="Studentized Residual", ylim=c(min(stud.res)-1,max(stud.res)+1))
```

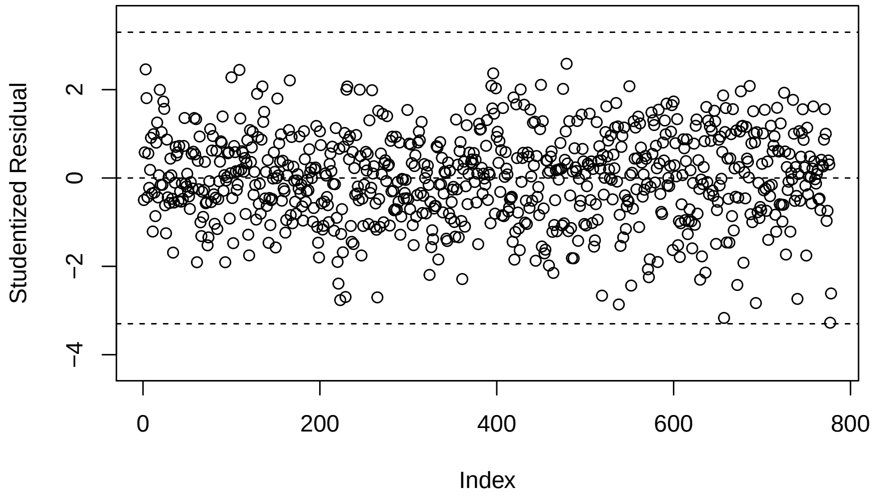


The code shows how functions can be substituted for fixed values and even used in arithmetic almost everywhere in R. `min(stud.res)-1` uses the *min()* function on the quantity that is being plotted *stud.res* and says, take 1 away from whatever the actual minimum is. The same is true analogouslyt for the scale maximum. This code snippet can be copied and pasted anywhere a plot is required that needs to be limited by the data as long as the variable *stud.res* is subsituted by the new variable. And the general principles can be applied all over R.

The important diagnostic information that is efficiently conveyed by this plot is that the positive residuals are all well within the bounds of acceptable values (in this case, they seem < 3). By contrast, the negative residuals seem to stretch out to the nonacceptable region and it seems the regression model in this data makes a strong overestimate of the true scores for some observations (i.e., remembering that as True score = Predicted Score + Residual, then Residual = True Score - Predicted Score).

However, rather than simply use a visual estimate, it is very easy in R to overlay a set of horizontal lines at milestone residual values, such as -3, 0, and +3. This is achieved with a special function called abline() that runs after plot() and overlays lines over the last plot that was called.

```
abline(h =c(-3.3,0,3.3), lty = 2)
```



This plot shows clearly that there is one point just on the -3.3 line and at least one point that is so close as to merit further assessment, depending on the remaining diagnostics.

The *abline()* function has to be run after a call to a plotting function and if this is not done it will simply produce an error message. It is worth taking this line of code apart to learn the devices in it.

- `h =` is asking for a horizontal line at Y = whatever number is given. A vertical line at X = whatever number is given would be called for with `v =`
- I could have written a single number, `h = -3.3` to get a single line but I used *c()* to make a list and *abline()* is smart enough to know that if a list is sent, then multiple lines are called for.
- `lty = 2` is a parameter set to the value for a dashed line (i.e., 2). If this parameter is left out, equivalent to adding in `lty = 1`, the default solid line is produced. A value of 3 makes a dotted line and further numbers produce variations. These are all given in the help() entries.

You could try running the *plot()* + *abline()* pair a few different times with changed options to see the effect on the plot.

Another useful device in making plots for diagnostics is to label the points with an ID or other identifying mark from the data so as to make it easier to identify any potentially influential points discovered. Some useful functions from base R and `car` may label discrepant points with R's generated row numbers from the data frame so it is handy to know how to print out each of these where required. Below, the steps to create appropriate labelling variables and then to apply these logically to later plots and diagnostics will be stepped through sequentially. Each line is a necessary logical step that ordinarily runs in direct sequence with the others. I have inserted commentary between them for the sake of learning and explanation.

The next step in diagnostic analysis is to locate potentially influential cases in the data by examining values of statistics such as leverage and Cook's D. Software such as SPSS sutomatically lists 10 largest, 10 smallest and so on of a given statistic and sometimes this shotgun approach can cause its own confusion. **In R you will learn to make your own lists and to only include the values that are needed, rather than the 10 "best values" like a Buzzfeed list**. For example, you know already that positive residuals are unlikely to be a problem in the BDRV data.

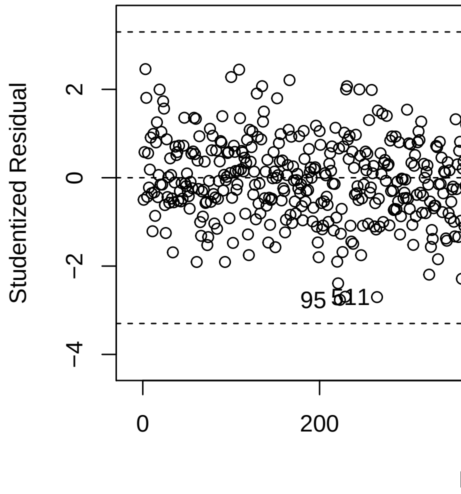
A common approach to this type of task is to make a new variable or object to hold the output of a procedure and then to list these using logical operators in R code. It is easiest to learn by doing these basic tasks.

First, collect which, if any, data points have values outside a +/- 2.7 cut-off. I chose 2.7 as the cutoff here as I know that there is at least one point just smaller than 3.3 that needs to potentially be looked at. You could tweak this value by trial and error in your own data if necessary. The row numbers of the potentially discrepant values are assigned to the new object called *index* (again, the name is arbitrary).

```r
index <- which(stud.res > 2.7 | stud.res < -2.7)
```

This code uses *which()*, a handy logical function in R that evaluates the logical truth (or falsity) of the condition in the parentheses. Only for observations where the condition is true, *which()* outputs the row name R lists in the data frame for that observation (i.e., when the condition is false, nothing happens for that observation). For the current example, `stud.res > 2.7 | stud.res < -2.7` means, "either the value of *stud.res* is greater than 2.7 or the value is less -2.7". As a logical condition, this is only true for observations with residuals outside +/- 2.7. After *which()* has run, there will be an object called *index* in the workspace and it will only contain 7 values (223, 265, 538, 657, 693, 740, 777) which are the cases with potentially extreme residuals. Check the *Environment* pane and also note how the objects are starting to build up!

I can take advantage of the logical status of *index* that *only* contains row number values for discrepant observations in the residuals by feeding it into the input of another function, called *text()*, whose job is to attach labels to points in plots.



```
text(index-30, stud.res[index] , labels = binge$ID[index])
```

- `index-30` = x co-ordinate of the labels; place the labels 30 pts to the left of the data point (i.e., arrived at by trial and error; a value between 20-40 always works; set and forget)
- `stud.res[index]` = y co-ordinate of the labels; this is slightly trickier; means "label the *stud.res* values according to the row numbers in *index*"; the trick is that *index* only exists for the discrepant observations so only they get a label
- `labels = binge$ID[index]`; use the variable *ID* in the data frame *binge* and match it with the row name from *index*. Again, *index* only exists for the discrepant observations so only they get a label

This may seem complex but it has already been written and tested! All you need to do in future is paste this code and make sure the variable and data frames match the new data you have. In the meantime, as you gain experience in R you will be able to understand the logic more clearly and will be able to build on this to do other tasks.

The remaining step is to print out both the R row numbers and the actual ID numbers with the following *print()* call:

```
print(c(index, binge$ID[index]))
```

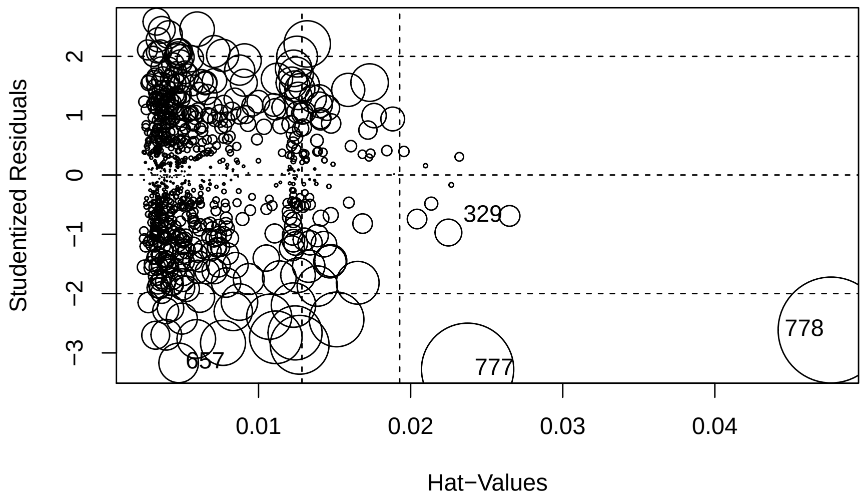
```
## 223 265 538 657 693 740 777
## 223 265 538 657 693 740 777  95 511 299 320 389  13 358
```

The first set of numbers (don't worry that there are two of them, there's a reason that isn't important!) are the **row numbers** and the second set are the **ID**

variable values for the discrepant observations. These are the values to look for in the next part.

Next, a common approach it to assess influence directly as a function of both distance (residuals)and leverage (hat). The `car` package has a very useful function called *influencePlot().* This creates the classic residual X leverage plot. If you are used to the SPSS version of this plot, the `car` version will seem on its side. The values with simultaneously high leverage and extreme residuals (as defined by Fox and Weisberg) are in the top and bottom right hand zones of the plot. The function makes use of dynamically generated dotted lines to mark out the quadrants of influence. As well, the plot markers are circles whose area is proportional to the value of Cook's D; a larger circle = more extreme. Very handily, the output of this function also contains a print out of the observations with the largest residuals and lists the corresponding values of leverage (hat) and Cook's D.

```
influencePlot(regint)
```



```
##         StudRes         Hat       CookD
## 329 -0.6901894 0.026502865 0.002595488
## 657 -3.1685132 0.004750005 0.009472262
## 777 -3.2760989 0.023745130 0.051560996
## 778 -2.6140906 0.047640429 0.067854846
```

This diagnostic suggests it is important to examine the observation with row number 778, as well as 329, 657, and 777, the latter three of which were flagged in the residuals analysis.

To check what ID number belongs to that case, the *print()* function is used again. Evidently, case 212 in the BDRV data can be added to the discrepant observations obtained from residuals analysis.

```
print(binge$ID[778])
```

```
## [1] 212
```

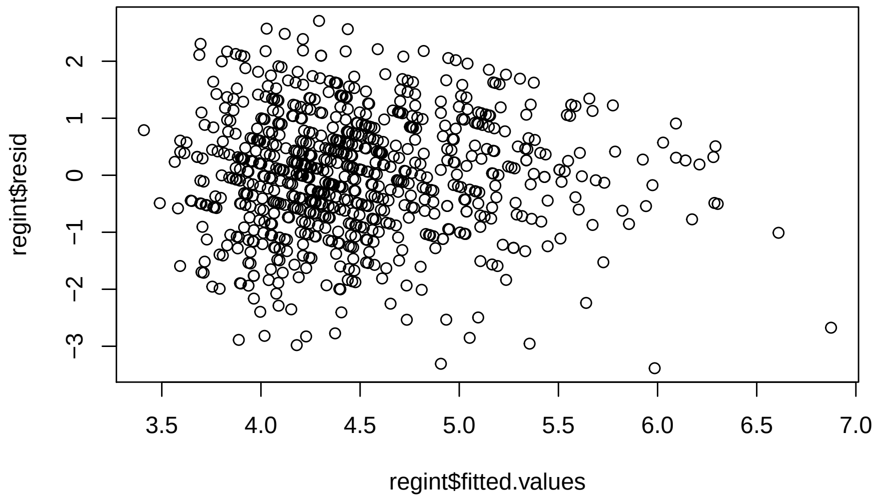
Checking for heteroscedasticity with a residual vs. fitted value plot. The function that is used is still *plot()*. The regression object generated above (i.e., *regint*) already contains the values that are needed. All that remains is to specify the input parameters. As an *lm()* object, *regint* has a known structure. You wouldn't normally bother to check, because every *lm()* object would look the same as that is the point of objects. I have listed it below with the *ls()* function.

```
ls(regint)
```

```
##  [1] "assign"        "call"          "coefficients"  "contrasts"
##  [5] "df.residual"   "effects"       "fitted.values" "model"
##  [9] "qr"            "rank"          "residuals"     "terms"
## [13] "xlevels"
```

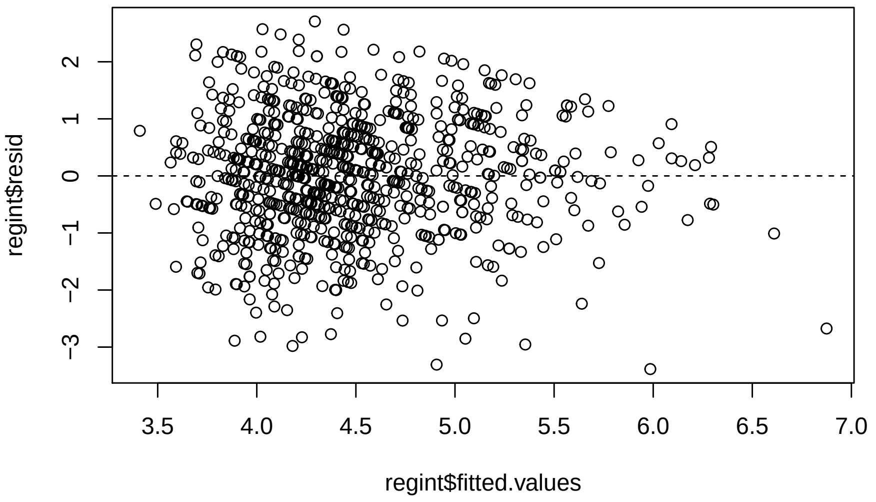
The two quantities needed for the heteroscedascticity plot are the residuals and the predicted scores. The plot() code is below. Note that the same approach (**$**) is used to specify subparts of an *lm()* object as is for a data frame object

```
plot(regint$resid ~ regint$fitted.values)
```



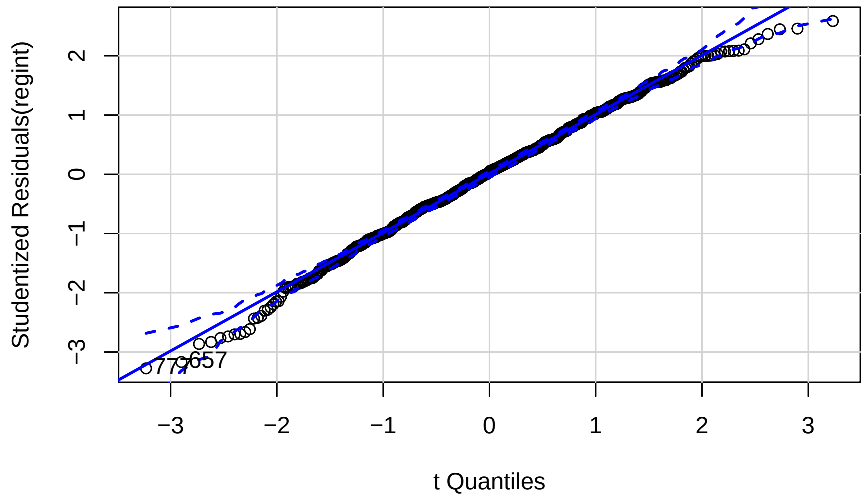
The usual interpretation of this plot (healthy rectangle or fearsome fan?) is made. As well, using the now familiar abline() function it is possible to add a horizontal line at Y = 0 to aid visual judgement (i.e., also helps in visual judgement whether the residuals are approximately centred around zero?)

```
abline(h = 0, lty = 2)
```



Making a Q-Q plot of the residuals to further assess normality. The `car` package provides a nicer and more useful Quantile-Comparison (Q-Q) plot than the base R versions (and base R takes two function calls to make something similar). The standard interpretation of these is that they show the studentized residuals plotted against the appropriate t-distribution (all this is modifiable) and if the residuals are distributed appropriately, they will sit on or very close to the main diagonal line. Any such deviations indicate nonnormality.

```r
qqPlot(regint)
```

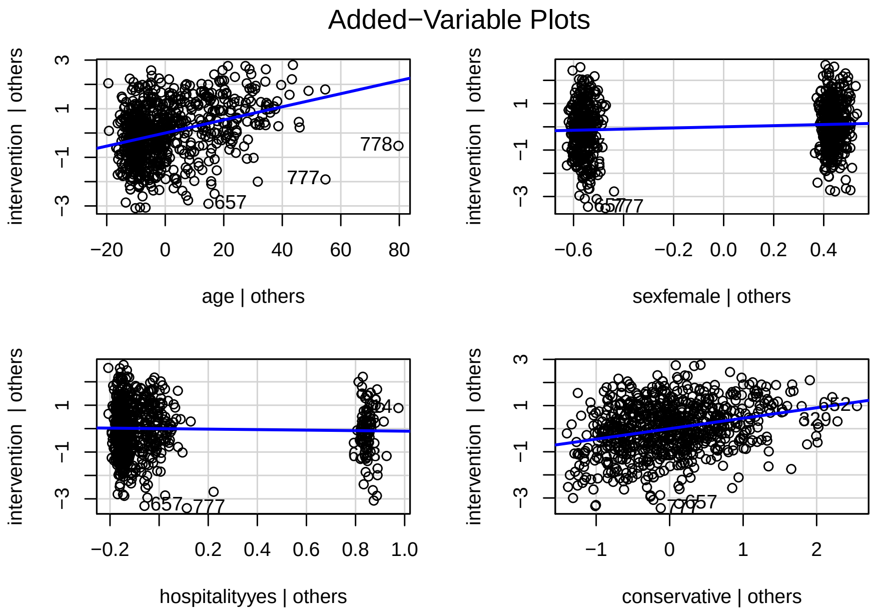


```
## [1] 657 777
```

The residuals pull away from the line at each end, characteristic of thick tailed distributions. *qqPlot()* also includes bootstrapped 95% CIs and it is of interest as to whether the points exceed this band. The output suggests that some do, around the t = -2.5 to -2.2 zone. However, the most extreme studentized residuals (row numbers = 657, 777) do not do so.

Added-variable plots, aka partial-regression plots are based on the association of a predictor with the criterion with all other predictors controlled for. Essentially, they are plots of the predictor and the criterion, residualised for all other predictors. The fit line has the same slope as for the standard regression. The really useful aspect of these plots (since the early references on diagnostics) is that they help to ascertain the observations that singly or in groups may be influencing the regression lines. In a similar way to the visual assessment of bivariate outliers, added-variable plots can emphasise the observations that sit in a position contrary to the trend described by the regression line. The *avPlots()* function from `car` presents the needed plots and include the regression lines.

```
avPlots(regint)
```



For example, the AV plot for *age* clearly suggests that observations with row numbers 777 and 778 (i.e., ID = 357 and 212, respectively) may be dragging the positive slope of the regression line down. Given that these observations have figured above in other examinations of diagnostics, it is worth checking the model with and without them included in the data. The remaining plots suggest few issues with the other observations flagged earlier, especially row number 657.

Some researchers find AV plots so useful in assessing discrepant cases in regressions (n.b., this is not just for linear regression) that it can be handy to check out a few potential models even before running the whole regression. A quick way to do this in R would be:

```
car::avPlots(lm(intervention~age + sex + hospitality + conservative,
data=binge))
```

With a large number of potential variables and models, a few checks like this would be time savers.

4.4 Running regressions dropping influential observations

If the original regression has been run already, as above, the model can be updated easily with the *update()* function (from base R) and this facilitates keeping track of the original model vs models derived without one or more influential observation and can ultimately guide changes to the data frame that will be incorporated for all future analyses. It is easiest now to demonstrate and then explain.

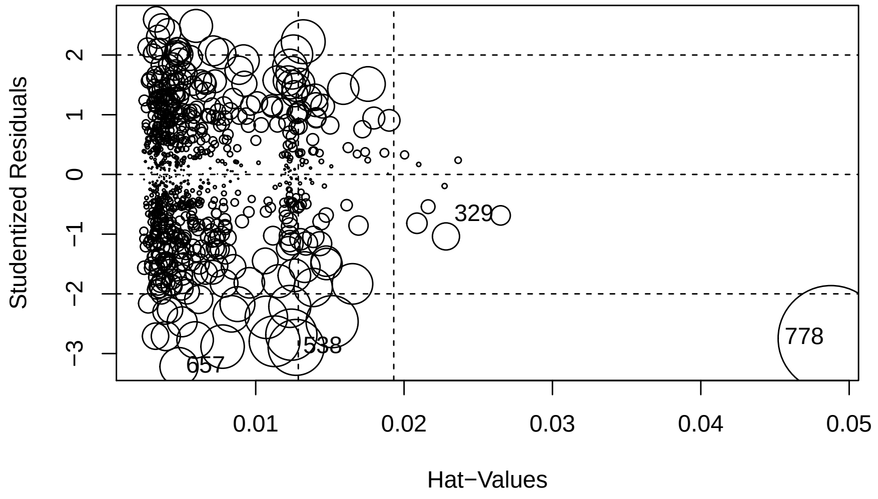
```r
regint.no777 <- update(regint, subset= -c(777))
```

- remove observation with row number 777 and re-run the regression, putting it into an aptly named new object (i.e., regint.no777)
- note the use of *subset()* again to specify which row(s) to drop;
  - note `-c`; the minus before the c tells *subset()* to keep all **but** row 777.

```r
summary(regint.no777)
```

```
##
## Call:
## lm(formula = intervention ~ age + sex + hospitality + conservative,
##     data = binge, subset = -c(777))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.3379 -0.6541  0.0428  0.7282  2.7023
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)     2.415669   0.160030  15.095  < 2e-16 ***
## age             0.028247   0.002762  10.228  < 2e-16 ***
## sexfemale       0.238829   0.075736   3.153  0.00168 **
## hospitalityyes -0.102198   0.117672  -0.869  0.38539
## conservative    0.451477   0.055295   8.165 1.31e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.046 on 772 degrees of freedom
## Multiple R-squared:  0.2163, Adjusted R-squared:  0.2122
## F-statistic: 53.27 on 4 and 772 DF,  p-value: < 2.2e-16
```

```r
influencePlot(regint.no777)
```



```
##        StudRes         Hat       CookD
## 329 -0.6867289 0.026508375 0.002570093
## 538 -2.8964020 0.012724309 0.021419354
## 657 -3.2185725 0.004829036 0.009933124
## 778 -2.7470743 0.048778274 0.076744676
```

- check the results as usual.
- note that 777 is now gone, but 778 still potentially a problem

```
regint.noinf <- update(regint, subset= -c(777, 778))
```

- can drop more than one case but must drop *all cases together*, not piecemeal
  - it's very simple: just keep adding row numbers to the *c(. . . , . . .)*

```
binge.clean <- binge[-c(777,778),]
```

- assign every row in binge except 777 and 778 to a new data frame object, *binge.clean*
- by doing so, you will have produced a dataset without influential cases without having to change the original data
- can incorporate into the original read or just include both lines at the top of new code files for analysis with this data.

```
regint.clean <- lm(intervention~age + sex + hospitality + conservative,
data=binge.clean)
```

4.5 End of module

5 Module 5: Regression and GLM procedures

To begin, open the glmModels.R file

After having assessed diagnostics, attnention can be turned to the regression analyses proper. All the analyses in this module will be based on the *binge.clean* data frame that was created at the end of Part 1b. It is the BDRV data without the observations at row numbers 777 (i.e., ID = 358) and 778 (i.e., ID = 212). The module will cover:

- multiple regression
  - use of binary and other categorical predictors
- hierarchical regression
- moderated multiple regression
- simple mediation
- some ANOVA
  - One-way B S/s
  - Factorial B S/s
  - Repeated measures and mixed factorials (limited)
    - via `afex()` package

The code in *glmModels.R* will produce an *lm()* object called *binge.clean*.

```
binge <- data.frame(read.spss(here::here("data", "BDRVdataR.sav")))
binge.clean <- binge[-c(777,778),]
regint.clean <- lm(intervention~age + sex + hospitality + conservative, data=binge.clean)
```

And the regression output is below:

```
summary(regint.clean)
```

```
##
## Call:
## lm(formula = intervention ~ age + sex + hospitality + conservative,
##     data = binge.clean)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.3726 -0.6437  0.0338  0.7252  2.7004
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)     2.369220   0.160251  14.784  < 2e-16 ***
## age             0.029882   0.002814  10.620  < 2e-16 ***
## sexfemale       0.231851   0.075460   3.072   0.0022 **
## hospitalityyes -0.093814   0.117216  -0.800   0.4238
## conservative    0.452548   0.055064   8.219 8.69e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.041 on 771 degrees of freedom
## Multiple R-squared:  0.2238, Adjusted R-squared:  0.2198
## F-statistic: 55.59 on 4 and 771 DF,  p-value: < 2.2e-16
```

```r
confint(regint.clean)
```

```
##                       2.5 %     97.5 %
## (Intercept)     2.05464114 2.68379941
## age             0.02435821 0.03540519
## sexfemale       0.08371918 0.37998270
## hospitalityyes -0.32391426 0.13628592
## conservative    0.34445464 0.56064047
```

This output corresponds to most of the output from GP software like SPSS. To reiterate some of the above:

- `Estimate` = b-weights (i.e., unstandardised regression weights)
- `Std. Error` = standard error (i.e., se) of b-weight
- `t value` = self explanatory
- `Pr(>|t|)` = p values (e.g., Sig.), note exponential notation

Of more interest is the way that R has set up the regression model in *lm()*. In particular, note that the Factor variables, *sex* and *hospitality* have been automatically turned into dummies. Because these Factors have underlying numeric values (as they came from SPSS) the lowest numeric value is the reference category. Otherwise, for text only Factors, the category that is listed first alphabetically is the reference category (n.b., in either case, this can easily be changed). As well, *lm()* output shows the actual tested catgory (e.g., sex*female*) and so the b-weight shows the difference between females compared to males as the reference category.

The creation of appropriate dummies will also occur for Factors with 3 or more levels. For example, the BDRV data contains a variable consisting of three groups on the basis of how frequently the participants reported engaging in binge behaviour (*bingegrp*: 1 = never; 2 = less than weekly; 3 = weekly or more). To briefly demonstrate how this Factor variable is treated in R, note the code below and the corresponding output.

```r
summary(lm(intervention~bingegrp, data=binge.clean))
```

```
##
## Call:
## lm(formula = intervention ~ bingegrp, data = binge.clean)
##
## Residuals:
```

```
##      Min       1Q   Median       3Q      Max
## -3.14752 -0.73977  0.05248  0.85248  2.85248
##
## Coefficients:
##                                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)                         4.93977    0.07052  70.053  < 2e-16 ***
## bingegrplow, binge less than weekly -0.63551    0.09164  -6.935 8.58e-12 ***
## bingegrphigh, binge weekly or more  -0.79225    0.11877  -6.671 4.86e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.135 on 773 degrees of freedom
## Multiple R-squared:  0.07602,    Adjusted R-squared:  0.07363
## F-statistic:  31.8 on 2 and 773 DF,  p-value: 5.36e-14
```

- Note that the Factor variable has been *automatically* converted into a pair of dummies for the purpose of the regression.
- The first category is automatically the reference category but this can be changed (either for a single regression or for all). However, if you take this into consideration when setting up response options for your data gathering, it can help your workflow considerably.
  - e.g., for the dummies in previous models, responses were no and yes. No comes alphabetically before yes and so no is the reference category.
- The special way in which Factors and numerics are dealt with in functions like lm() is one of the things that makes R so useful. Later, when looking at moderation and more complex models this approach will really bear fruit.

Transformations It is not necessary to compute transformations separately. In R, they are inserted directly into the formula in *lm()*. For example, the following computes a natural log transform of age in the regression.

```
summary(lm(intervention~log(age) + sex + hospitality + conservative, data=binge.clean))
```

```
##
## Call:
## lm(formula = intervention ~ log(age) + sex + hospitality + conservative,
##     data = binge.clean)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.3997 -0.6529  0.0488  0.7205  2.8481
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -0.22688    0.36100  -0.628  0.52988
## log(age)        1.04371    0.10205  10.227  < 2e-16 ***
## sexfemale       0.24199    0.07586   3.190  0.00148 **
## hospitalityyes -0.09577    0.11787  -0.813  0.41673
## conservative    0.45914    0.05527   8.307  4.4e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.046 on 771 degrees of freedom
## Multiple R-squared:  0.2166, Adjusted R-squared:  0.2125
## F-statistic: 53.29 on 4 and 771 DF,  p-value: < 2.2e-16
```

For a square root transformation, the code would be: `lm(intervention~sqrt(age) + sex + hospitality + conservative, data=binge.clean)` and so forth.

However, it is almost never a good idea to transform variables…

5.1 Hierarchical regression

Running a hierarchical regression is straightforward in R. as the two blocks are assigned to their own objects and then the objects are compared with the *anova()* function.

```r
# object labelled meaningfully but arbitrarily
block1 <- lm(intervention~age + sex + hospitality + conservative, data=binge.clean)
#summary(block1) *output suppressed
block2 <- lm(intervention~age + sex + hospitality + conservative + bingegrp, data=binge.clea
#summary(block2) *output suppressed

# this is the line to obtain F for change
anova(block1, block2)
```

```
## Analysis of Variance Table
##
## Model 1: intervention ~ age + sex + hospitality + conservative
## Model 2: intervention ~ age + sex + hospitality + conservative + bingegrp
##   Res.Df    RSS Df Sum of Sq      F    Pr(>F)
## 1    771 836.24
## 2    769 816.63  2    19.611 9.2338 0.0001089 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

- each block is listed as a separate regression
- the order in which the output objects are listed doesn't matter as the *anova()* function will figure out the nesting based on degrees of freedom
- need to watch for differences across the regression owing to missing data
  - create marker for sample of people who have no missing data across entire list of predictors

- this approach can be extended to multiple pairs if there are more than 2 hierarchical blocks

5.2 Moderated multiple regression

To express a moderated regression GLM in *lm()* is almost trivially easy. The product term is inserted directly into the formula passed to *lm()* as below.

```
intermod <- lm(intervention~sex + conservative*stereotypical, data=binge.clean)
summary(intermod)
```

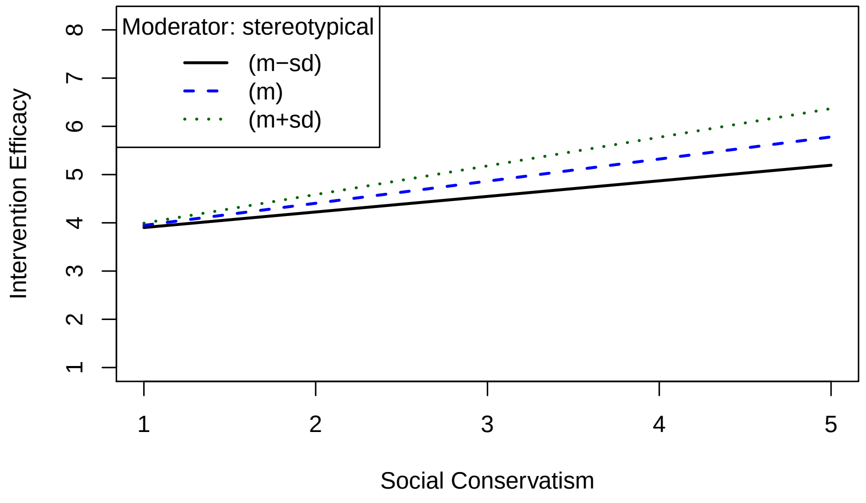
```
##
## Call:
## lm(formula = intervention ~ sex + conservative * stereotypical,
##     data = binge.clean)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.3783 -0.7044  0.0246  0.6937  2.9544
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)                 3.67836    0.65740   5.595 3.06e-08 ***
## sexfemale                   0.24205    0.07893   3.067  0.00224 **
## conservative               -0.19099    0.25795  -0.740  0.45928
## stereotypical              -0.08826    0.13178  -0.670  0.50320
## conservative:stereotypical  0.13278    0.05063   2.622  0.00890 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.088 on 771 degrees of freedom
## Multiple R-squared:  0.1526, Adjusted R-squared:  0.1482
## F-statistic: 34.72 on 4 and 771 DF,  p-value: < 2.2e-16
```

- the * operator replaces the + in these models to indicate the product term
  - *lm()* automatically inserts the effects as two separate predicors (i.e., the marginals) + the product (i.e., the interaction), as appropriate
  - use `conservative + stereotypical + conservative:stereotypical` format for explicit coding of marginals and interaction, e.g., more complex models with multiple interactions with shared marginals
    - e.g., `conservative + stereotypical + age + conservative:stereotypical + conservative:age`
- scales up to three-way effects (e.g., `conservative*stereotypical*age`)
- automatically deals with categorical IVs or moderators (think products with dummy coding)
  - below, factorial designs in ANOVA are dealt with in similar fashion

- the model can also include separate controls besides the IV and moderator if desired, as usual
- if a more detailed sequence of models was required around the moderation, hierarchical models could also be done as above, e.g.,
  - `b1 <- lm(intervention~age + hospitality, data=binge.clean)`
  - `b2 <- lm(intervention~age + hospitality + sex + conservative*stereotypical, data=binge.clean)`
  - `anova(b1, b2)`

Following up the initial test of moderation is facilitated by a package called `rockchalk`. This is a more general regression/GLM package that contains very useful functions for probing moderation. Rockchalk has extensive help in a freely available monograph.
`library(rockchalk)`

```
ps  <- plotSlopes(intermod, plotx="conservative", modx="stereotypical", xlab = "Social Conse
```



```
testps <- testSlopes(ps)
```

```
## Values of stereotypical OUTSIDE this interval:
##        lo        hi
## -9.197273  3.084799
## cause the slope of (b1 + b2*stereotypical)conservative to be statistically significant
```

- *plotSlopes()* is a function that does plotting of conditional effects
  - the code snippet above deliberately includes the key parameters required

    - *numerics* treated as continuous and *Factors* treated as categorical moderators and dealt with automatically.
- *testps()* is used to actually test conditional effects and does automatic Johnson-Neyman-type analysis on conditional slopes

5.3 Mediation

The pattern for running mediation analysis in R will bear evident similarities to the general approach that has been building where more complex analyses are set up with multiple simple objects (e.g., *lm()* fit objects) are combined by a target function in the package. The `mediation` package provides functions to test a large range of mediation models, including those with discrete mediators and outcomes (e.g., generalized models used for the discrete outcome/mediator variable and OLS regression for the continuous ones).

To illustrate mediation, an analysis is run below in which the effect of age on beliefs in the efficacy of BDRV interventions is mediated by social conservatism. Note the this data was not set up to test this research question but rather it is presented as a somewhat feasible relationship for an example. There could well be hidden confounds in the data but that is beside the point. As well, *sex* is included to show how controls are easily incorporated into the models. The code is below:

```
library(mediation)
```

```
## Loading required package: MASS

##
## Attaching package: 'MASS'

## The following object is masked from 'package:rockchalk':
##
##     mvrnorm

## Loading required package: Matrix

## Loading required package: mvtnorm

## Loading required package: sandwich

## mediation: Causal Mediation Analysis
## Version: 4.5.0
```

```
med.fit <- lm(conservative~age + sex, data= binge.clean)
dv.fit <- lm(intervention~conservative + age + sex, data= binge.clean)
med.out <- mediate(med.fit, dv.fit, treat = "age", mediator = "conservative", robustSE = TRU
summary(med.out)
```

```
##
## Causal Mediation Analysis
```

```
##
## Quasi-Bayesian Confidence Intervals
##
##                 Estimate 95% CI Lower 95% CI Upper p-value
## ACME            0.002731     0.000974         0.00  <2e-16 ***
## ADE             0.030229     0.024918         0.04  <2e-16 ***
## Total Effect    0.032959     0.027693         0.04  <2e-16 ***
## Prop. Mediated 0.082685     0.030574         0.14  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Sample Size Used: 776
##
##
## Simulations: 1000
```

- *med.fit* is clearly an *lm()* object like any other that happens to include the statistical model for the mediator. This would provide the parameter for the *a* path in Baron and Kenny notation.
- *dv.fit* is an *lm()* object that contains the full model (i.e., mediator and putative IV, with control) for the dependent variable. This would provide the parameters for the *b* and *c'* paths, respectively.
  - if desired, either of these objects could be inspected with the appropriate *summary()* lines in the code
- `sims = 1000` sets the number of repetitions to 1000 for MC or bootstrapping. This would be higher for real analyses
- the default is a bayesian approach but inserting the parameter `boot = TRUE` in the *mediate()* function and removing `robustSE = TRUE` will provide the nonparametric bootstrapping with which researchers will be familiar
- ACME stands for Average Causal Mediation Effect and this is the estimate of the indirect effect (i.e., the key statistical parameter for mediation)
- ADE stands for Average Direct Effect and this is the effect of the IV on the DV, ignoring the Mediator
  - note that these will not necessarily be exactly equal to the paths from the OLS regressions in the *lm()* objects owing to the effects of simulations (i.e., bootstrapping or MC, according to the number of repetitions)
- The *Total Effect* is the sum of the indirect and direct effects, as usual.

5.4 ANOVA

Full disclosure: ANOVA is a point of controversy in R.

5.4.1 Issue no.1

Simply put, there are various ways to calculate Sums of Squares (SS) in ANOVA when there are multiple effects. These are generally described as Type I, II, and III SS, respectively. Software like SAS and SPSS default to Type III SS for unbalanced designs. (In case you were asking, Type I = sequential SS, like a hierarchical regression; Type II = marginal SS where effects are evaluated with all other effects partialled except higher order ones that include them; Type III = effects are evaluated with *all* other effects partialled, including higher order ones that include the target effects.) Stata and R, depending on the package, go to Type II (or Type I in some cases). The statistical purist's choice is Type II. The people who maintain R are generally purists (and I am basically on their side, he said, surprising no one). However, ultimately, in balanced data, the answers should be the same; it's just that data in factorial designs are so rarely balanced in practice. Also, *some* statistical authorities argue in favour of Type III SS, but it is fair to say *most* argue against them. **In practice**, if you use the base R *aov()* function, the order in which you list effects in the formula (i.e., `DV ~ Effect1*Effect2 + Covariate` is different to `DV ~ Covariate + Effect1*Effect2` and `Effect1*Effect2` is different to `Effect2*Effect1`). This will be different to the *Anova()* function from `car` which uses Type II SS. After having mastered these functions users will probably want to make use of specialised ANOVA packages such as `ez` and `afex`.

5.4.2 Issue no. 2

Repeated measures designs are a little tricky to set up and for mixed designs (i.e., between-within designs, not mixed effects models per se) you need to be pretty clear about the appropriate effects to use as error terms. Probably the best thing ultimately is to learn to use the mixed effects modelling package `lme4` and the *lmer()* function. Otherwise, the specialised ANOVA packages such as `ez` and `afex` are of great use here as well.

What does this mean in practice? A factorial or repeated measures ANOVA run in R most likely won't give the exact same numbers as that run in SPSS or SAS but it should match Stata. If the effects are greatly different (vs just a slight difference in the value of the MS or F ratios), this is something that should be investigated thoroughly, in case there are influential data points combined with very small cells or similar. Ideally, by the time the ANOVA is run, anything potentially problematic in the data should have come to light.

Below, you will find the code for the "standard" designs. Most of these are as simple as you would expect. The spoiler is that for many, you can: a) use the *aov()* function to estimate the ANOVA directly, accepting the Type I, or sequential SS; or b) simply specify the design in *lm()* as usual with R taking care of the Factors versus the numerics and then feed that output into *anova()* or *car::Anova()* to obtain appropriate SS. If that's the sort of thing you will need to do, one of the above will be an end to it. There are packages that make that job a little easier for those who do a ANOVAs regularly. There are packages for followups that will adjust familywise error and the usual stuff.There are also packages that will provide Type III SS, just like the big players do (dubiously!).

A brief summary of packages

- base R
  - *aov()*
    - takes anova formula (specified as for *lm()*) as direct input
    - with *summary()* obtain a quick ANOVA table
    - uses Type I (i.e., sequential/hierarchical) SS and this may not exactly match a precise RQ
    - order of specification of effects in formula matters
  - *anova()*
    - takes an lm() object as input
    - Type I SS as for *aov()*
    - cannot specify formula directly as for *aov()*
    - useful to compare nested model objects etc
- `car`
  - *Anova()*
  - uses Type II (i.e., marginal) SS and this generally provide defensible inference but watch for missing data
- `ez`
  - *ezANOVA()*
  - various functions; depends on `car`
  - specially created for factorials, between and within
  - Type II SS
- `afex`
  - various functions; depends on `car`
  - Type III SS
- `emmeans`
  - provides covariate-adjusted means and predicted means in general, like the /EMMEANS subcommand in SPSS GLM (and SAS)
  - similar feel to `margins` post-estimation command in Stata
  - does simple main effects (and conditional or simple slopes)
  - runs contrasts and other followups
  - provides interaction plots
  - **n.b.,** `emmeans` can be used to followup linear models in general (e.g., mixed effects etc) and also generalized linear models with appropriate link functions if required

One-way B/Ss ANOVA and ANCOVA This is the simplest case. The *aov()* function probably gives you everything you need. Contrasts and comparisons can be made directly with `emmeans` package (the *pairs()* function defaults to tukey-style but see help() or vignette() as enormous capabilities here.

- ANOVA

```r
library(emmeans)
binge.aov <- aov(conservative~bingegrp, data = binge.clean)
summary(binge.aov)
```

```
##              Df Sum Sq Mean Sq F value   Pr(>F)    
## bingegrp      2   11.9   5.949   13.08 2.58e-06 ***
## Residuals   773  351.5   0.455                     
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
bingegrp.em <- emmeans(binge.aov, "bingegrp")
pairs(bingegrp.em)
```

```
##  contrast                                                  estimate     SE  df
##  never binge - low, binge less than weekly                    0.237 0.0545 773
##  never binge - high, binge weekly or more                     0.309 0.0706 773
##  low, binge less than weekly - high, binge weekly or more     0.072 0.0666 773
##  t.ratio p.value
##  4.353   <.0001
##  4.378   <.0001
##  1.081   0.5264
## 
## P value adjustment: tukey method for comparing a family of 3 estimates
```

- ANCOVA

```r
binge.acv <- aov(conservative~intervention + bingegrp, data = binge.clean)
summary(binge.acv)
```

```
##              Df Sum Sq Mean Sq F value Pr(>F)    
## intervention   1   34.3   34.32  81.416 <2e-16 ***
## bingegrp       2    3.6    1.82   4.327 0.0135 *  
## Residuals    772  325.4    0.42                   
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
bingegrp.cem <- emmeans(binge.acv, "bingegrp")
pairs(bingegrp.cem)
```

```
##  contrast                                                  estimate     SE  df
##  never binge - low, binge less than weekly                   0.1342 0.0540 772
##  never binge - high, binge weekly or more                    0.1808 0.0699 772
##  low, binge less than weekly - high, binge weekly or more    0.0466 0.0642 772
##  t.ratio p.value
##  2.484   0.0352
##  2.587   0.0266
##  0.726   0.7483
## 
```

```
## P value adjustment: tukey method for comparing a family of 3 estimates
```

Try also specifying the model as `conservative~bingegrp + intervention` to see the difference in SS vs the above where the covariate is partialled first.

**Between-subjects Factorial ANOVA The main issues here are the differences between the different packages and functions. All of the below generalises to higher order factorials (e.g., 2-way = `FactorA*FactorB`; 3-way = `FactorA*FactorB*FactorC`). After the basic approaches have been mastered, it is very important to examine the `emmeans` help vignettes carefully. These are extensive and serve as a set of tutorials in each of the tasks. Having a facility with how R code flows willmake it very easy to follow along. An important point here is that `emmeans` requires *aov()* objects and will not work with Anova() objects. Both `afex` and `ez` will return *aov()* objects for contrasts if required.

- aov() base R = Type I SS
- `car::Anova` and `ez` = Type II SS
  - **n.b., *Anova()* with an a capital *A* is the required function**
  - the *anova()* function is used in hierarchical regression
- `afex` package = Type III, if required
- ANOVA
  `bingesex.aov <- aov(conservative~bingegrp*sex, data = binge.clean)`
  `summary(bingesex.aov)`
  - plot interaction
    `emmip(bingesex.aov, bingegrp ~ sex)`
  - do simple contrasts of *bingesex* for each level of *sex*
    `bingesex.em <- emmeans(bingesex.aov, ~ bingegrp * sex)`
    `pairs(bingesex.em, simple = "bingegrp")`
- ANCOVA Same as for ANOVA but include covariate in formula. If using *aov()*, be sure that the order of the effects matches your RQ, otherwise use *Anova()* version

Repeated measures ANOVA (limited) Please note that `ez` and `afex` may facilitate these designs, especially larger mixed factorials. A template for a basic 2-way mixed factorial is presented below:

- `mixfac.aov <- aov(DV ~ BSs_factor * WSs_factor + Error(id_var), data = data frame)`
  - data frame needs to be reshaped into so called, *long* format, where each participant has multiple rows of data, each row represents one repeated measures occasion, variables that are measured only once, are actually repeated on each row.
  - for most, it might be easiest to enter data in spreadsheet (or SPSS) in usual *wide* format with careful variable names and use *reshape()* function from base R to convert to long. It's not complex, it's just a

little detailed. Consult *help()* for *reshape()*. Once done it is easy to move back and forth.

– each of the Factors, including the Error one, need to be Factors in R.

5.5 End of module