Contents

Part I

Getting started

1 Introduction

1.1 Why statistics?

To the surprise of many students, statistics is large part of a psychological education. To the surprise of no-one, statistics is very rarely the *favourite* part of one's psychological education. After all, if you really loved the idea of doing statistics, you'd probably be enrolled in a statistics class right now, not a psychology class. So, not surprisingly, there's a pretty large proportion of the student base that isn't happy about the fact that psychology has so much statistics in it. In view of this, I thought that the right place to start might be to answer some of the more common questions that people have about stats...

A big part of this issue at hand relates to the very idea of statistics. What is it? What's it there for? And why are scientists so bloody obsessed with it? These are all good questions, when you think about it. So let's start with the last one. As a group, scientists seem to be bizarrely fixated on running statistical tests on everything. In fact, we use statistics so often that we sometimes forget to explain to people why we do. It's a kind of article of faith among scientists – and especially social scientists – that your findings can't be trusted until you've done some stats. Undergraduate students might be forgiven for thinking that we're all completely mad, because no-one takes the time to answer one very simple question: *Why do we do statistics? Why don't scientists just use **common sense?*

It's a naive question in some ways, but most good questions are. There are many good answers to the question but perhaps the best one is the simplest one: we don't trust ourselves enough. We worry that we're human, and susceptible to all of the biases, temptations and frailties that humans suffer from. Statistics provides us with tools to help us understand our data, but it also provides us with safeguards. Using common sense to evaluate evidence means trusting "gut instincts", relying on verbal arguments and on using the raw power of human reason to come up with the right answer. As smart as we are as humans, we aren't perfect, and if we want to avoid innocent self-deception, we need something a little more rigorous.

In fact, come to think of it, this sounds a lot like a psychological question to me, and since I do work in a psychology department, it seems like a good idea to dig a little deeper here. Is it really plausible to think that this "common sense" approach is very trustworthy? Verbal arguments have to be constructed in language, and all languages have biases – some things are harder to say than others, and not necessarily because they're false (e.g., quantum electrodynamics is a good theory, but hard to explain in words). The instincts of our "gut" aren't designed to solve scientific problems, they're designed to handle day to day inferences – and given that biological evolution is slower than cultural change, we should say that they're designed to solve the day to day problems for a *different world* than the one we live in. Most fundamentally, reasoning sensibly requires people to engage in "induction", making wise guesses and going beyond the immediate evidence of the senses to make generalisations about the world. If you think that you can do that without being influenced by various distractors, well, I have a bridge in Brooklyn I'd like to sell you. Heck, as the next section shows, we can't even solve "deductive" problems (ones where no guessing is required) without being influenced by our pre-existing biases.

1.2 The parable of belief bias

People are mostly pretty smart. We're certainly smarter than the other species that we share the planet with (though many people might disagree). Our minds are quite amazing things, and we seem to be capable of the most incredible feats of thought and reason. That doesn't make us perfect though. And among the many things that psychologists have shown over the years is that we really do find it hard to be neutral, to evaluate evidence impartially and without being swayed by pre-existing biases. A good example of this is the ***belief bias effect*** in logical reasoning: if you ask people to decide whether a particular argument is logically valid (i.e., conclusion would be true if the premises were true), we tend to be influenced by the believability of the conclusion, even when we shouldn't. For instance, here's a valid argument where the conclusion is believable:

> No cigarettes are inexpensive (Premise 1)
> Some addictive things are inexpensive (Premise 2)
> Therefore, some addictive things are not cigarettes (Conclusion)

And here's a valid argument where the conclusion is not believable:

> No addictive things are inexpensive (Premise 1)
> Some cigarettes are inexpensive (Premise 2)
> Therefore, some cigarettes are not addictive (Conclusion)

The logical *structure* of argument #2 is identical to the structure of argument #1, and they're both valid. However, in the second argument, there are good reasons to think that premise 1 is incorrect, and as a result it's probably the case that the conclusion is also incorrect. But that's entirely irrelevant to the topic at hand: an argument is deductively valid if the conclusion is a logical consequence of the premises. That is, a valid argument doesn't have to involve true statements.

On the other hand, here's an invalid argument that has a believable conclusion:

> No addictive things are inexpensive (Premise 1)
> Some cigarettes are inexpensive (Premise 2)
> Therefore, some addictive things are not cigarettes (Conclusion)

And finally, an invalid argument with an unbelievable conclusion:

> No cigarettes are inexpensive (Premise 1)
> Some addictive things are inexpensive (Premise 2)
> Therefore, some cigarettes are not addictive (Conclusion)

Now, suppose that people really are perfectly able to set aside their pre-existing biases about what is true and what isn't, and purely evaluate an argument on its logical merits. We'd expect 100% of people to say that the valid arguments are valid, and 0% of people to say that the invalid arguments are valid. So if you ran an experiment looking at this, you'd expect to see data like this:

	conclusion feels true	conclusion feels false
argument is valid	100% say "valid"	100% say "valid"
argument is invalid	0% say "valid"	0% say "valid"

If the psychological data looked like this (or even a good approximation to this), we might feel safe in just trusting our gut instincts. That is, it'd be perfectly okay just to let scientists evaluate data based on their common sense, and not bother with all this murky statistics stuff. However, you guys have taken psych classes, and by now you probably know where this is going...

In a classic study, EVANS ran an experiment looking at exactly this. What they found is that when pre-existing biases (i.e., beliefs) were in agreement with the structure of the data, everything went the way you'd hope:

	conclusion feels true	conclusion feels false
argument is valid	92% say "valid"	
argument is invalid		8% say "valid"

Not perfect, but that's pretty good. But look what happens when our intuitive feelings about the truth of the conclusion run against the logical structure of the argument:

	conclusion feels true	conclusion feels false
argument is valid	92% say "valid"	**46% say "valid"**
argument is invalid	**92% say "valid"**	8% say "valid"

Oh dear, that's not as good. Apparently, when people are presented with a strong argument that contradicts our pre-existing beliefs, we find it pretty hard to even perceive it to be a strong argument (people only did so 46% of the time). Even worse, when people are presented with a weak argument that agrees with our pre-existing biases, almost no-one can see that the argument is weak (people got that one wrong 92% of the time!)[1]

If you think about it, it's not as if these data are horribly damning. Overall, people did do better than chance at compensating for their prior biases, since about 60% of people's judgements were correct (you'd expect 50% by chance). Even so, if you were a professional "evaluator of evidence", and someone came along and offered you a magic tool that improves your chances of making the right decision from 60% to (say) 95%, you'd probably jump at it, right? Of course you would. Thankfully, we actually do have a tool that can do this. But it's not magic, it's statistics. So that's reason #1 why scientists love statistics. It's just *too easy* for us to "believe what we want to believe"; so if we want to "believe in the data" instead, we're going

[1] In my more cynical moments I feel like this fact alone explains 95% of what I read on the internet.

to need a bit of help to keep our personal biases under control. That's what statistics does: it helps keep us honest.

1.3 Statistics in psychology

I hope that the discussion above helped explain why science in general is so focused on statistics. But I'm guessing that you have a lot more questions about what role statistics plays in psychology, and specifically why psychology classes always devote so many lectures to stats. So here's my attempt to answer a few of them...

Why does psychology have so much statistics?

To be perfectly honest, there's a few different reasons, some of which are better than others. The most important reason is that psychology is a statistical science. What I mean by that is that the "things" that we study are *people*. Real, complicated, gloriously messy, infuriatingly perverse people. The "things" of physics include object like electrons, and while there are all sorts of complexities that arise in physics, electrons don't have minds of their own. They don't have opinions, they don't differ from each other in weird and arbitrary ways, they don't get bored in the middle of an experiment, and they don't get angry at the experimenter and then deliberately try to sabotage the data set (not that I've ever done that...). At a fundamental level psychology is harder than physics.[2]

Basically, we teach statistics to you as psychologists because you need to be better at stats than physicists. There's actually a saying used sometimes in physics, to the effect that "if your experiment needs statistics, you should have done a better experiment". They have the luxury of being able to say that because their objects of study are pathetically simple in comparison to the vast mess that confronts social scientists. It's not just psychology, really: most social sciences are desperately reliant on statistics. Not because we're bad experimenters, but because we've picked a harder problem to solve. We teach you stats because you really, really need it.

Can't someone else do the statistics?

To some extent, but not completely. It's true that you don't need to become a fully trained statistician just to do psychology, but you do need to reach a certain level of statistical competence. In my view, there's three reasons that every psychological researcher ought to be able to do basic statistics:

- Firstly, there's the fundamental reason: statistics is deeply intertwined with research design. If you want to be good at designing psychological studies, you need to at least understand the basics of stats.
- Secondly, if you want to be good at the psychological side of the research, then you need to be able to understand the psychological literature, right? But almost every paper in the psychological literature reports the results of statistical analyses. So if you really want to understand the psychology, you need to be able to understand what other people did with their data. And that means understanding a certain amount of statistics.
- Thirdly, there's a big practical problem with being dependent on other people to do all your statistics: statistical analysis is *expensive*. If you ever get bored and want to look up how much the Australian government charges for university fees, you'll notice something interesting: statistics is designated as a "national priority" category, and so the fees are much, much lower than for any other area of study. This is because there's a massive shortage of statisticians out there. So, from your perspective as a psychological researcher, the laws of supply and demand aren't exactly on your side here! As a result, in almost any real life situation where you want to do psychological research, the cruel facts will be that you don't have enough money to afford a statistician. So the economics of the situation mean that you have to be pretty self-sufficient.

Note that a lot of these reasons generalise beyond researchers. If you want to be a practicing psychologist and stay on top of the field, it helps to be able to read the scientific literature, which relies pretty heavily on statistics.

[2] Which might explain why physics is just a teensy bit further advanced as a science than we are.

I don't care about jobs, research, or clinical work. Do I need statistics?

Okay, now you're just messing with me. Still, I think it should matter to you too. Statistics should matter to you in the same way that statistics should matter to *everyone*: we live in the 21st century, and data are *everywhere*. Frankly, given the world in which we live these days, a basic knowledge of statistics is pretty damn close to a survival tool! Which is the topic of the next section...

1.4 Statistics in everyday life

> *"We are drowning in information, but we are starved for knowledge"*
>
> -Various authors, original probably John Naisbitt

When I started writing up my lecture notes I took the 20 most recent news articles posted to the ABC news website. Of those 20 articles, it turned out that 8 of them involved a discussion of something that I would call a statistical topic; 6 of those made a mistake. The most common error, if you're curious, was failing to report baseline data (e.g., the article mentions that 5% of people in situation X have some characteristic Y, but doesn't say how common the characteristic is for everyone else!) The point I'm trying to make here isn't that journalists are bad at statistics (though they almost always are), it's that a basic knowledge of statistics is very helpful for trying to figure out when someone else is either making a mistake or even lying to you. In fact, one of the biggest things that a knowledge of statistics does to you is cause you to get angry at the newspaper or the internet on a far more frequent basis: you can find a good example of this in Section **??**. In later versions of this book I'll try to include more anecdotes along those lines.

1.5 On the limitations of statistics

Before leaving this topic entirely, I want to point out something else really critical that is often overlooked in a research methods class. Statistics only solves *part* of the problem. Remember that we started all this with the concern that Berkeley's admissions processes might be unfairly biased against female applicants. When we looked at the "aggregated" data, it did seem like the university was discriminating against women, but when we "disaggregate" and looked at the individual behaviour of all the departments, it turned out that the actual departments were, if anything, slightly biased in favour of women. The gender bias in total admissions was caused by the fact that women tended to self-select for harder departments. From a legal perspective, that would probably put the university in the clear. Postgraduate admissions are determined at the level of the individual department (and there are good reasons to do that), and at the level of individual departments, the decisions are more or less unbiased (the weak bias in favour of females at that level is small, and not consistent across departments). Since the university can't dictate which departments people choose to apply to, and the decision making takes place at the level of the department it can hardly be held accountable for any biases that those choices produce.

That was the basis for my somewhat glib remarks earlier, but that's not exactly the whole story, is it? After all, if we're interested in this from a more sociological and psychological perspective, we might want to ask *why* there are such strong gender differences in applications. Why do males tend to apply to engineering more often than females, and why is this reversed for the English department? And why is it it the case that the departments that tend to have a female-application bias tend to have lower overall admission rates than those departments that have a male-application bias? Might this not still reflect a gender bias, even though every single department is itself unbiased? It might. Suppose, hypothetically, that males preferred to apply to "hard sciences" and females prefer "humanities". And suppose further that the reason for why the humanities departments have low admission rates is because the government doesn't want to fund the humanities (Ph.D. places, for instance, are often tied to government funded research projects). Does that constitute a gender bias? Or just an unenlightened view of the value of the humanities? What if someone at a high level in the government cut the humanities funds because they felt that the humanities are "useless chick stuff". That seems pretty *blatantly* gender biased. None of this falls within the purview of statistics, but it matters to the research project. If you're interested in the overall structural effects of subtle gender

biases, then you probably want to look at *both* the aggregated and disaggregated data. If you're interested in the decision making process at Berkeley itself then you're probably only interested in the disaggregated data.

In short there are a lot of critical questions that you can't answer with statistics, but the answers to those questions will have a huge impact on how you analyse and interpret data. And this is the reason why you should always think of statistics as a *tool* to help you learn about your data, no more and no less. It's a powerful tool to that end, but there's no substitute for careful thought.

So far, most of what I've talked about is statistics, and so you'd be forgiven for thinking that statistics is all I care about in life. To be fair, you wouldn't be far wrong, but research methodology is a broader concept than statistics. So most research methods courses will cover a lot of topics that relate much more to the pragmatics of research design, and in particular the issues that you encounter when trying to do research with humans. However, about 99% of student *fears* relate to the statistics part of the course, so I've focused on the stats in this discussion, and hopefully I've convinced you that statistics matters, and more importantly, that it's not to be feared. That being said, it's pretty typical for introductory research methods classes to be very stats-heavy. This is not (usually) because the lecturers are evil people. Quite the contrary, in fact. Introductory classes focus a lot on the statistics because you almost always find yourself needing statistics before you need the other research methods training. Why? Because almost all of your assignments in other classes will rely on statistical training, to a much greater extent than they rely on other methodological tools. It's not common for undergraduate assignments to require you to design your own study from the ground up (in which case you would need to know a lot about research design), but it *is* common for assignments to ask you to analyse and interpret data that were collected in a study that someone else designed (in which case you need statistics). In that sense, from the perspective of allowing you to do well in all your other classes, the statistics is more urgent.

But note that "urgent" is different from "important" – they both matter. I really do want to stress that research design is just as important as data analysis, and this book does spend a fair amount of time on it. However, while statistics has a kind of universality, and provides a set of core tools that are useful for most types of psychological research, the research methods side isn't quite so universal. There are some general principles that everyone should think about, but a lot of research design is very idiosyncratic, and is specific to the area of research that you want to engage in. To the extent that it's the details that matter, those details don't usually show up in an introductory stats and research methods class.

2 Getting started in R

> Machine dreams hold a special vertigo. –William Gibson, *Count Zero*, 1986

In this chapter I'll discuss how to get started in R. I'll briefly talk about how to download and install R, then show you how to write R commands. The goal in this chapter is not to learn any statistical or programming concepts: we're just trying to learn how R works and get comfortable interacting with the system. We'll spend a bit of time using R as a simple calculator, since that's the easiest thing to do with R, just to give you a feel for what it's like to work in R.

2.1 Why learn R?

2.2 First steps

2.2.1 Installing R & RStudio

An important distinction to remember is between the R *programming language* itself, and the software you use to interact with R. You could choose to interact with R directly from the terminal, but that's painful, so most people use an *integrated development environment* (IDE), which takes care of a lot of boring tasks

for you. For this class, we'll use the popular Rstudio IDE. To get started, make sure you have both R and RStudio installed on your computer. Both are free and open source, and for most people they should be straightforward to install.

- **Installing R**: Go to the R website and download the installer file. Once downloaded, open the installer file and follow the instructions.
- **Installing RStudio**: Go to the R studio website, and follow the links to download RStudio. The version you want is the "RStudio Desktop". Once the installer is downloaded, open it and follow the instructions.

To get started, open the **Rstudio** application (i.e., RStudio.exe or RStudio.app), not the vanilla application (i.e., not R.exe or R.app). You should be looking at something like this:

In the bottom left hand corner you'll see a panel labelled *Console*, and a whole lot of text that doesn't make much sense. Ignore it for now! The important part is this...

```
>
```

... which has a flashing cursor next to it. That's the *command prompt*. When you see this, it means that R is waiting patiently for you to do something! So it's time to get started!

2.2.2 R commands

One of the easiest things you can do with R is use it as a simple calculator, so it's a good place to start. For instance, try typing 30, and hitting enter. When you do this, you've entered a command, and R will **execute** that command. What you see on screen now will be this:

```
10 + 20
```

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

Not a lot of surprises in this extract. But there's a few things worth talking about, even with such a simple example. Firstly, it's important that you understand how to read the extract. In this example, what I typed was the `10 + 20` part at the top, and the content below is what R produced.

Secondly, it's important to understand how the output is formatted. When you look at the output on your screen it will probably look like this `[1] 30`. Obviously, the correct answer to the sum `10 + 20` is `30`, and not surprisingly R has printed that out as part of its response. But it's also printed out this `[1]` part, which probably doesn't make a lot of sense to you right now. You're going to see that a lot. I'll talk about what this means in a bit more detail later on, but for now you can think of `[1] 30` as if R were saying "the answer to the first question you asked is 30". That's not quite the truth, but it's close enough for now. And in any case it's not really very interesting at the moment: we only asked R to calculate one thing, so obviously there's only one answer printed on the screen. Later on this will change, and the `[1]` part will start to make a bit more sense. For now, I just don't want you to get confused or concerned by it.

2.2.3 Avoid typos

Before we go on to talk about other types of calculations that we can do with R, there's a few other things I want to point out. The first thing is that, while R is good software, it's still software. It's pretty stupid, and because it's stupid it can't handle typos. It takes it on faith that you meant to type exactly what you did type. For example, suppose that you forgot to hit the shift key when trying to type `+`, and as a result your command ended up being `10 = 20` rather than `10 + 20`. Here's what happens:

```
10 = 20
```

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

What's happened here is that R has attempted to interpret `10 = 20` as a command, and spits out an error message because the command doesn't make any sense to it. When a human looks at this, and then looks down at his or her keyboard and sees that + and = are on the same key, it's pretty obvious that the command was a typo. But R doesn't know this, so it gets upset. And, if you look at it from its perspective, this makes sense. All that R "knows" is that `10` is a legitimate number, `20` is a legitimate number, and `=` is a legitimate part of the language too. In other words, from its perspective this really does look like the user meant to type `10 = 20`, since all the individual parts of that statement are legitimate and it's too stupid to realise that this is probably a typo. Therefore, R takes it on faith that this is exactly what you meant... it only "discovers" that the command is nonsense when it tries to follow your instructions, typo and all. And then it whinges, and spits out an error.

Even more subtle is the fact that some typos won't produce errors at all, because they happen to correspond to "well-formed" R commands. For instance, suppose that not only did I forget to hit the shift key when trying to type `10 + 20`, I also managed to press the key next to one I meant do. The resulting typo would produce the command `10 - 20`. Clearly, R has no way of knowing that you meant to add `20` to `10`, not subtract `20` from `10`, so what happens this time is this:

```
10 - 20
```

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

In this case, R produces the right answer, but to the the wrong question.

To some extent, I'm stating the obvious here, but it's important. The people who wrote R are smart. You, the user, are smart. But R itself is dumb. And because it's dumb, it has to be mindlessly obedient. It does exactly what you ask it to do. There is no equivalent to "autocorrect" in R, and for good reason. When doing advanced stuff – and even the simplest of statistics is pretty advanced in a lot of ways – it's dangerous to let a mindless automaton like R try to overrule the human user. But because of this, it's your responsibility to be careful. Always make sure you type exactly what you mean. When dealing with computers, it's not enough to type "approximately" the right thing. In general, you absolutely must be precise in what you say to R ... like all machines it is too stupid to be anything other than absurdly literal in its interpretation.

2.2.4 R is flexible with spacing?

Of course, now that I've been so uptight about the importance of always being precise, I should point out that there are some exceptions. Or, more accurately, there are some situations in which R does show a bit more flexibility than my previous description suggests. The first thing R is smart enough to do is ignore redundant spacing. What I mean by this is that, when I typed `10 + 20` before, I could equally have done this

```
10        + 20
```

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

and get exactly the same answer. However, that doesn't mean that you can insert spaces in any old place. For instance, when you open up R it suggests that you type `citation()` to get some information about how to cite R:

```
citation()
```

```
##
## To cite R in publications use:
##
##   R Core Team (2019). R: A language and environment for
##   statistical computing. R Foundation for Statistical Computing,
##   Vienna, Austria. URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2019},
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please
## cite it when using it for data analysis. See also
## 'citation("pkgname")' for citing R packages.
```

Okay, that's good to know. Let's see what happens when I try changing the spacing. If I insert spaces in between the word and the parentheses, or inside the parentheses themselves, then all is well. But insertinhg spaces in the middle of the commands, not so much. Try these three just to see:

```
citation ()     # works!
citation(    )  # works!
cita tion()     # doesn't work
```

2.2.5 R knows you're not finished?

One more thing I should point out. If you hit enter in a situation where it's "obvious" to R that you haven't actually finished typing the command, R is just smart enough to keep waiting. For example, if you type `10 +` and then press enter, even R is smart enough to realise that you probably wanted to type in another number. So here's what happens:

```
> 10 +
+
```

and there's a blinking cursor next to the plus sign. What this means is that R is still waiting for you to finish. It "thinks" you're still typing your command, so it hasn't tried to execute it yet. In other words, this plus sign is actually another command prompt. It's different from the usual one (i.e., the `>` symbol) to remind you that R is going to "add" whatever you type now to what you typed last time. For example, if I then go on to type `20` and hit enter, what I get is this:

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

And as far as R is concerned, this is exactly the same as if you had typed `10 + 20`. Similarly, consider the `citation()` command that we talked about in the previous section. Suppose you hit enter after typing `citation(`. Once again, R is smart enough to realise that there must be more coming – since you need to add the `)` character – so it waits. I can even hit enter several times and it will keep waiting:

```
> citation(
+
+
+)
```

Sometimes when doing this, you'll eventually get yourself in trouble (it happens to us all). Maybe you start typing a command, and then you realise you've screwed up. For example,

```
> citblation(
+
+
```

You'd probably prefer R not to try running this command, right? If you want to get out of this situation, just hit the *escape* key. R will return you to the normal command prompt (i.e. `>`) without attempting to execute the botched command.

That being said, it's not often the case that R is smart enough to tell that there's more coming. For instance, in the same way that I can't add a space in the middle of a word, I can't hit enter in the middle of a word either. If I hit enter after typing `citat` I get an error, because R thinks I'm interested in something called `citat` and can't find it:

```
citat
```

```
## Error in eval(expr, envir, enclos): object 'citat' not found
```

What about if I typed `citation` and hit enter, without the parentheses? In this case we get something very odd, something that we definitely don't want, at least not at this stage. Here's what happens:

```
citation
```

```
## function (package = "base", lib.loc = NULL, auto = NULL)
## {
##     if (!is.null(auto) && !is.logical(auto) && !any(is.na(match(c("Package",
##         "Version", "Title"), names(meta <- as.list(auto))))) &&
##         !all(is.na(match(c("Authors@R", "Author"), names(meta))))) {
##         auto_was_meta <- TRUE
##         package <- meta$Package
##     }
##     else {
##         auto_was_meta <- FALSE
##         dir <- system.file(package = package, lib.loc = lib.loc)
##         if (dir == "")
##             stop(packageNotFoundError(package, lib.loc, sys.call()))
##         meta <- packageDescription(pkg = package, lib.loc = dirname(dir))
##         citfile <- file.path(dir, "CITATION")
##         test <- file_test("-f", citfile)
##         if (!test) {
```

```
##             citfile <- file.path(dir, "inst", "CITATION")
##             test <- file_test("-f", citfile)
##         }
##         if (is.null(auto))
##             auto <- !test
##         if (!auto) {
##             return(readCitationFile(citfile, meta))
##         }
##     }
##     if ((!is.null(meta$Priority)) && (meta$Priority == "base")) {
##         cit <- citation("base", auto = FALSE)
##         attr(cit, "mheader")[1L] <- paste0("The ", sQuote(package),
##             " package is part of R.  ", attr(cit, "mheader")[1L])
##         return(.citation(cit, package))
##     }
##     year <- sub("-.*", "", meta$`Date/Publication`)
##     if (!length(year)) {
##         if (is.null(meta$Date)) {
##             warning(gettextf("no date field in DESCRIPTION file of package %s",
##                 sQuote(package)), domain = NA)
##         }
##         else {
##             date <- trimws(as.vector(meta$Date))[1L]
##             date <- strptime(date, "%Y-%m-%d", tz = "GMT")
##             if (!is.na(date))
##                 year <- format(date, "%Y")
##         }
##     }
##     if (!length(year)) {
##         date <- as.POSIXlt(sub(";.*", "", trimws(meta$Packaged)[1L]))
##         if (!is.na(date))
##             year <- format(date, "%Y")
##     }
##     if (!length(year)) {
##         warning(gettextf("could not determine year for %s from package DESCRIPTION file",
##             sQuote(package)), domain = NA)
##         year <- NA_character_
##     }
##     author <- meta$`Authors@R`
##     if (length(author)) {
##         aar <- .read_authors_at_R_field(author)
##         author <- Filter(function(e) {
##             !(is.null(e$given) && is.null(e$family)) && !is.na(match("aut",
##                 e$role))
##         }, aar)
##         if (!length(author))
##             author <- Filter(function(e) {
##                 !(is.null(e$given) && is.null(e$family)) && !is.na(match("cre",
##                   e$role))
##             }, aar)
##     }
##     if (length(author)) {
##         has_authors_at_R_field <- TRUE
##     }
```

```
##     else {
##         has_authors_at_R_field <- FALSE
##         author <- as.personList(meta$Author)
##     }
##     z <- list(title = paste0(package, ": ", meta$Title), author = author,
##         year = year, note = paste("R package version", meta$Version))
##     if (identical(meta$Repository, "CRAN"))
##         z$url <- sprintf("https://CRAN.R-project.org/package=%s",
##             package)
##     if (identical(meta$Repository, "R-Forge")) {
##         z$url <- if (!is.null(rfp <- meta$"Repository/R-Forge/Project"))
##             sprintf("https://R-Forge.R-project.org/projects/%s/",
##                 rfp)
##         else "https://R-Forge.R-project.org/"
##         if (!is.null(rfr <- meta$"Repository/R-Forge/Revision"))
##             z$note <- paste(z$note, rfr, sep = "/r")
##     }
##     if (!length(z$url) && !is.null(url <- meta$URL)) {
##         if (grepl("[, ]", url))
##             z$note <- url
##         else z$url <- url
##     }
##     header <- if (!auto_was_meta) {
##         gettextf("To cite package %s in publications use:", sQuote(package))
##     }
##     else NULL
##     footer <- if (!has_authors_at_R_field && !auto_was_meta) {
##         gettextf("ATTENTION: This citation information has been auto-generated from the package DESC
##             sQuote("help(\"citation\")"))
##     }
##     else NULL
##     author <- format(z$author, include = c("given", "family"))
##     if (length(author) > 1L)
##         author <- paste(paste(head(author, -1L), collapse = ", "),
##             tail(author, 1L), sep = " and ")
##     rval <- bibentry(bibtype = "Manual", textVersion = paste0(author,
##         " (", z$year, "). ", z$title, ". ", z$note, ". ", z$url),
##         header = header, footer = footer, other = z)
##     .citation(rval, package)
## }
## <bytecode: 0x55ecc8a05960>
## <environment: namespace:utils>
```

where the `BLAH BLAH BLAH` goes on for rather a long time, and you don't know enough R yet to understand what all this gibberish actually means. This incomprehensible output can be quite intimidating to novice users, and unfortunately it's very easy to forget to type the parentheses; so almost certainly you'll do this by accident. Do not panic when this happens. Simply ignore the gibberish.

2.3 Operations

Okay, now that we've discussed some of the tedious details associated with typing R commands, let's get back to learning how to use the most powerful piece of statistical software in the world as a $2 calculator. So far, all we know how to do is addition. Clearly, a calculator that only did addition would be a bit stupid, so

I should tell you about how to perform other simple calculations using R. But first, some more terminology. Addition is an example of an "operation" that you can perform (specifically, an arithmetic operation), and the **operator** that performs it is +. To people with a programming or mathematics background, this terminology probably feels pretty natural, but to other people it might feel like I'm trying to make something very simple (addition) sound more complicated than it is (by calling it an arithmetic operation). To some extent, that's true: if addition was the only operation that we were interested in, it'd be a bit silly to introduce all this extra terminology. However, as we go along, we'll start using more and more different kinds of operations, so it's probably a good idea to get the language straight now, while we're still talking about very familiar concepts like addition!

2.3.1 Arithmetic operations

So, now that we have the terminology, let's learn how to perform some arithmetic operations. R has operators that correspond to the basic arithmetic we learned in primary school: addition is `+`, subtraction is `-`, multiplication is `*` and division ia `/`. As you can see, R uses fairly standard symbols to denote each of the different operations you might want to perform: if I wanted to find out what `57` times `61` is (and who wouldn't?), I can use R instead of a calculator, like so:

```
57 * 61
```

```
## [1] 3477
```

So that's handy.

There are three other arithmetic operations that I should probably mention: taking powers, doing integer division, and calculating a modulus. Of the three, the only one that is of any real importance for the purposes of this book is taking powers, so I'll discuss that one here: the other two are discussed later.

For those of you who can still remember your high school maths, this should be familiar. But for some people high school maths was a long time ago, and others of us didn't listen very hard in high school. It's not complicated. As I'm sure everyone will probably remember the moment they read this, the act of multiplying a number x by itself n times is called "raising x to the n-th power". Mathematically, this is written as x^n. Some values of n have special names: in particular x^2 is called x-squared, and x^3 is called x-cubed. One way that we could calculate 5^4 in R would be to type in the complete multiplication like so,

```
5*5*5*5
```

```
## [1] 625
```

but that does seem a bit tedious. It would be very annoying indeed if you wanted to calculate 5^{15}, since the command would end up being quite long. Therefore, to make our lives easier, we use the power operator instead. When we do that, our command to calculate 5^4 goes like this:

```
5^4
```

```
## [1] 625
```

Much easier.

2.3.2 Order of operations

Okay. At this point, you know how to take one of the most powerful pieces of statistical software in the world, and use it as a $2 calculator. And as a bonus, you've learned a few very basic programming concepts. That's not nothing (you could argue that you've just saved yourself $2) but on the other hand, it's not very much either. In order to use R more effectively, we need to introduce more programming concepts.

In most situations where you would want to use a calculator, you might want to do multiple calcula- tions. R lets you do this, just by typing in longer commands. In fact, we've already seen an example of this earlier, when I typed in `5 * 5 * 5 * 5`. However, let's try a slightly different example:

```
1 + 2 * 4
```

```
## [1] 9
```

Clearly, this isn't a problem for R either. However, it's worth stopping for a second, and thinking about what R just did. Clearly, since it gave us an answer of `9` it must have multiplied `2 * 4` (to get an interim answer of `8`) and then added `1` to that. But, suppose it had decided to just go from left to right: if R had decided instead to add `1+2` (to get an interim answer of `3`) and then multiplied by `4`, it would have come up with an answer of `12`

To answer this, you need to know the **order of operations** that R uses.[3] If you remember back to your high school maths classes, it's actually the same order that you got taught when you were at school: the BEDMAS order. Thatis, first calculate things inside **B**rackets, then calculate **E**xponents, then **D**ivision and **M**ultiplication, then **A**ddition and **S**ubtraction. So, to continue the example above, if we want to force R to calculate the `1 + 2` part before the multiplication, all we would have to do is enclose it in brackets:

```
(1 + 2) * 4
```

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

This is a fairly useful thing to be able to do. The only other thing I should point out about order of operations is what to expect when you have two operations that have the same priority: that is, how does R resolve ties? For instance, multiplication and division are actually the same priority, but what should we expect when we give R a problem like `4 / 2 * 3` to solve? If it evaluates the multiplication first and then the division, it would calculate a value of two-thirds. But if it evaluates the division first it calculates a value of siz. The answer, in this case, is that R goes from **left to right**, so in this case the division step would come first:

```
4 / 2 * 3
```

```
## [1] 6
```

All of the above being said, it's helpful to remember that *brackets always come first*. So, if you're ever unsure about what order R will do things in, an easy solution is to enclose the thing you want it to do first in brackets. There's nothing stopping you from typing `(4 / 2) * 3`. By enclosing the division in brackets we make it clear which thing is supposed to happen first. In this instance you wouldn't have needed to, since R would have done the division first anyway, but when you're first starting out it's better to make sure R does what you want!

[3] For a more precise statement, see the operator precedence for R

2.4 Functions

The symbols `+`, `-`, `*` and so on are examples of operators. As we've seen, you can do quite a lot of calculations just by using these operators. However, in order to do more advanced calculations (and later on, to do actual statistics), you're going to need to start using functions.[4] I'll talk in more detail about functions and how they work later, but for now let's just dive in and use a few. To get started, suppose I wanted to take the square root of `225`. The square root, in case your high school maths is a bit rusty, is just the opposite of squaring a number. So, for instance, since "5 squared is 25" I can say that "5 is the square root of 25". The usual notation for this is $\sqrt{25} = 5$, though sometimes you'll also see it written like this $25^{0.5} = 5$. This second way of writing it is kind of useful to "remind" you of the mathematical fact that "square root of x" is actually the same as "raising x to the power of 0.5". Personally, I've never found this to be terribly meaningful psychologically, though I have to admit it's quite convenient mathematically. Anyway, it's not important. What is important is that you remember what a square root is, since it's kind of useful in statistics!

To calculate the square root of 25, I can do it in my head pretty easily, since I memorised my multiplication tables when I was a kid. It gets harder when the numbers get bigger, and pretty much impossible if they're not whole numbers. This is where something like R comes in very handy. Let's say I wanted to calculate the square root of 225. There's two ways I could do this using R. Firstly, since the square root of 255 is the same thing as raising `225` to the power of `0.5`, I could use the power operator `^`, just like we did earlier:

```
225 ^ 0.5
```

```
## [1] 15
```

However, there's a second way to do this by using square root function `sqrt`.

2.4.1 Using functions

To calculate the square root of 255 using the `sqrt` function, the command I type is this:

```
sqrt(225)
```

```
## [1] 15
```

When we use a function to do something, we generally refer to this as **calling** the function, and the values that we type into the function (there can be more than one) are referred to as the **arguments** of that function.

Obviously, the `sqrt` function doesn't really give us any new functionality, since we already knew how to do square root calculations by using the power operator `^`. However, there are lots of other functions in R: in fact, almost everything of interest that I'll talk about in this book is an R function of some kind. For example, one function that comes in handy quite often is the **absolute value** function. Compared to the square root function, it's extremely simple: it just converts negative numbers to positive numbers, and leaves positive numbers alone. Calculating absolute values in R is pretty easy, since R provides the `abs` function that you can use for this purpose. For instance:

```
abs(-13)
```

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

[4] A side note for students with a programming background. Technically speaking, operators are functions in R: the addition operator `+` is a convenient way of calling the addition function `` `+`() ``. Thus `10+20` is equivalent to the function call `` `+`(20, 30) ``. Not surprisingly, no-one ever uses this version. Because that would be stupid.

2.4.2 Combining functions

Before moving on, it's worth noting that, in the same way that R allows us to put multiple operations together into a longer command (like `1 + 2 * 4` for instance), it also lets us put functions together and even combine functions with operators if we so desire. For example, the following is a perfectly legitimate command:

```
sqrt(1 + abs(-8))
```

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

When R executes this command, starts out by calculating the value of `abs(-8)`, which produces an intermediate value of `8`. Having done so, the command simplifies to `sqrt(1 + 8)`. To solve the square root[5] it first needs to add `1 + 8` to get `9`, at which point it evaluates `sqrt(9)`, and so it finally outputs a value of `3`.

2.4.3 Multiple arguments

There's two more fairly important things that you need to understand about how functions work in R, and that's the use of "named" arguments, and default values" for arguments. Not surprisingly, that's not to say that this is the last we'll hear about how functions work, but they are the last things we desperately need to discuss in order to get you started. To understand what these two concepts are all about, I'll introduce another function. The `round` function can be used to round some value to the nearest whole number. For example, I could type this:

```
round(3.1415)
```

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

Pretty straightforward, really. However, suppose I only wanted to round it to two decimal places: that is, I want to get `3.14` as the output. The `round` function supports this, by allowing you to input a second argument to the function that specifies the number of decimal places that you want to round the number to. In other words, I could do this:

```
round(3.14165, 2)
```

```
## [1] 3.14
```

What's happening here is that I've specified two arguments: the first argument is the number that needs to be rounded (i.e., `3.1415`), the second argument is the number of decimal places that it should be rounded to (i.e., `2`), and the two arguments are separated by a comma.

2.4.4 Argument names

In this simple example, it's not too hard to remember which argument comes first and which one comes second, but as you might imagine it starts to get very difficult once you start using complicated functions that have lots of arguments. Fortunately, most R functions use **argument names** to make your life a little

[5] A note for the mathematically inclined: R does support complex numbers, but unless you explicitly specify that you want them it assumes all calculations must be real valued. By default, the square root of a negative number is treated as undefined: `sqrt(-9)` will produce `NaN` (not a number) as its output. To get complex numbers, you would type `sqrt(-9+0i)` and R would now return `0+3i`. However, since we won't have any need for complex numbers in this book, I won't refer to them again.

easier. For the `round` function, for example the number that needs to be rounded is specified using the `x` argument, and the number of decimal points that you want it rounded to is specified using the `digits` argument. Because we have these names available to us, we can specify the arguments to the function by name. We do so like this:

```
round(x = 3.1415, digits = 2)
```

```
## [1] 3.14
```

Notice that this is kind of similar in spirit to variable assignment, except that I used `=` here, rather than `<-`. In both cases we're specifying specific values to be associated with a label. However, there are some differences between what I was doing earlier on when creating variables, and what I'm doing here when specifying arguments, and so as a consequence it's important that you use `=` in this context.

As you can see, specifying the arguments by name involves a lot more typing, but it's also a lot easier to read. Because of this, the commands in this book will usually specify arguments by name,[6] since that makes it clearer to you what I'm doing. However, one important thing to note is that when specifying the arguments using their names, it doesn't matter what order you type them in. But if you don't use the argument names, then you have to input the arguments in the correct order. In other words, these three commands all produce the same output...

```
round(3.14165, 2)
round(x = 3.1415, digits = 2)
round(digits = 2, x = 3.1415)
```

```
## [1] 3.14
## [1] 3.14
## [1] 3.14
```

but this one does not...

```
round(2, 3.14165)
```

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

2.4.5 Default values

Okay, so that's the first thing I said you'd need to know: argument names. The second thing you need to know about is default values. Notice that the first time I called the `round` function I didn't actually specify the `digits` argument at all, and yet R somehow knew that this meant it should round to the nearest whole number. How did that happen? The answer is that the `digits` argument has a **default value** of `0`, meaning that if you decide not to specify a value for `digits` then R will act as if you had typed `digits = 0`. This is quite handy: most of the time when you want to round a number you want to round it to the nearest *whole* number, and it would be pretty annoying to have to specify the `digits` argument every single time. On the other hand, sometimes you actually do want to round to something other than the nearest whole number, and it would be even more annoying if R didn't allow this! Thus, by having `digits = 0` as the default value, we get the best of both worlds.

[6] The two functions discussed previously, `sqrt` and `abs`, both only have a single argument, `x`. So I could have typed something like `sqrt(x = 225)` or `abs(x = -13)` earlier. The fact that all these functions use `x` as the name of the argument that corresponds the "main" variable that you're working with is not entirely a coincidence. That's a fairly widely used convention. Quite often, the writers of R functions will try to use conventional names like this to make your life easier. Or at least that's the theory. In practice... it doesn't always work as well as you'd hope.

2.5 RStudio helps!

At this stage you know how to type in basic commands, including how to use R functions. And it's probably beginning to dawn on you that there are a lot of R functions, all of which have their own arguments. You're probably also worried that you're going to have to remember all of them! Thankfully, it's not that bad. In fact, very few data analysts bother to try to remember all the commands. What they really do is use tricks to make their lives easier. The first (and arguably most important one) is to use the internet. If you don't know how a particular R function works, Google it. There is a lot of R documentation out there, and almost all of it is searchable! For the moment though, I want to call your attention to a couple of simple tricks that Rstudio makes available to you.

2.5.1 Tab autocomplete

The first thing I want to call your attention to is the *autocomplete* ability in Rstudio.[7] Let's stick to our example above and assume that what you want to do is to round a number. This time around, start typing the name of the function that you want, and then hit the "tab" key. Rstudio will then display a little window like the one shown here:

In this figure, I've typed the letters `rou` at the command line, and then hit tab. The window has two panels. On the left, there's a list of variables and functions that start with the letters that I've typed shown in black text, and some grey text that tells you where that variable/function is stored. Ignore the grey text for now: it won't make much sense to you until we've talked about packages. There's a few options there, and the one we want is `round`, but if you're typing this yourself you'll notice that when you hit the tab key the window pops up with the top entry highlighted. You can use the up and down arrow keys to select the one that you want. Or, if none of the options look right to you, you can hit the escape key ("esc") or the left arrow key to make the window go away.

In our case, the thing we want is the `round` option, and the panel on the right tells you a bit about how the function works. This display is really handy. The very first thing it says is `round(x, digits = 0)`: what this is telling you is that the `round` function has two arguments. The first argument is called `x`, and it doesn't have a default value. The second argument is `digits`, and it has a default value of `0`. In a lot of situations, that's all the information you need. But Rstudio goes a bit further, and provides some additional information about the function underneath. Sometimes that additional information is very helpful, sometimes it's not: Rstudio pulls that text from the R help documentation, and my experience is that the helpfulness of that documentation varies wildly. Anyway, if you've decided that `round` is the function that you want to use, you can hit the enter key and Rstudio will finish typing the rest of the function name for you.

2.5.2 The history pane

One thing R does is keep track of your "command history". That is, it remembers all the commands that you've previously typed. You can access this history in a few different ways. The simplest way is to use the up and down arrow keys. If you hit the up key, the R console will show you the most recent command that you've typed. Hit it again, and it will show you the command before that. If you want the text on the screen to go away, hit escape. Using the up and down keys can be really handy if you've typed a long command that had one typo in it. Rather than having to type it all again from scratch, you can use the up key to bring up the command and fix it.

The second way to get access to your command history is to look at the history panel in Rstudio. On the upper right hand side of the Rstudio window you'll see a tab labelled "History". Click on that, and you'll see a list of all your recent commands displayed in that panel: it should look something like this:

If you double click on one of the commands, it will be copied to the R console. You can achieve the same result by selecting the command you want with the mouse and then clicking the "To Console" button.

[7] Okay, this isn't just an Rstudio thing. If you're running R in a terminal window, tab autocomplete still works, and does so in the way you'd expect

2.6 Exercises

Two exercises that uses skills that you've already acquired...

- Calculate the number of seconds in a year, on the simplifying assumption that a year contains exactly 365 days.
- Calculate the square root of your favourite number

Two more that foreshadow where we're going...

- Type `rep("hello!",100)` and take a look at the output. Try to work out what it means
- Type `hist(x=rnorm(300))` and see what happens. What do you think R has done?

2.7 Project oriented workflow

> It's not a put down I put my foot down And then I make some love, I put my root down Like Sweetie Pie by the Stone Alliance Everybody knows I'm known for dropping science   –The Beastie Boys

Part II

Exploring data

3 Pretty pictures

> Let them eat cake (first) - Mine Cetinkaya-Rundel

One of the primary tasks facing a scientist or any other data analyst is to *make sense of data*. There are so many different sources of data in the world, but if our goal is to learn things and make scientific progress, we need to transform data (raw information) into knowledge (human understanding). There are many different aspects to this "exploratory data analysis" process.

Sometimes exploration is like cleaning the house: there are tedious jobs to be done to organise your data into a comprehensible form, to document it properly for other people to read and reuse, and so on. Just as cleaning the house and running a household are undervalued domestic job, data cleaning, data manipulation and documentation are often neglected in statistics textbooks. I don't intend to repeat that error in this book, so I'll talk a lot about that process later.

But there's something terribly depressing about starting out with the least exciting parts of the job. Because sometimes, data exploration is like eating cake. Sweet, fun and makes you feel so so good. Very much as Mine Cetinkaya-Rundel suggests, I think data visualisation is the cake. It's your reward for cleaning the house and paying the bills, but if my job is to sell you on the virtues of working with data, it makes sense to start by getting you addicted to the cake – data visualisation is *fun*.

Okay lovely reader, let's get this party started.

```
library(tidyverse)
library(tidylsr)
```

3.1 R graphics

What does it mean for a machine to draw a picture? It's a weird question to ask, but it's a surprisingly complicated one. Reduced to its simplest form, you can think of R graphics like a painting. You start out with an empty canvas. Every time you use a graphics function, it paints some new things onto your canvas. Later you can paint more things over the top if you want, layering new information over the old. This way of thinking about plotting data is referred to as the **painter's model** - the key thing to keep in mind is because the plot is constructed sequentially, the order in which you do things matters.

We can extend the painting metaphor a little. If you want to paint a picture, you need to paint it on something. In real life, you can paint on lots of different things. Painting onto canvas isn't the same as painting onto paper, and neither is the same as painting on a wall. In R, the thing that you paint onto is called a **graphics device**. In RStudio, the default graphics device is `RStudioGD` and it corresponds to the "plot" pane. If you were using the basic R program for Windows (i.e., `R.exe`) the default device is `windows`, on the Mac application (`R.app`) it's called `quartz`, etc. However, from the computer's perspective there's nothing terribly special about drawing pictures on screen, and so R is quite happy to paint pictures directly into a file. R can paint several different types of image files: `jpeg`, `png`, `pdf`, `postscript`, `tiff` and `bmp` files are all available as graphics devices and you can write plots directly to those using those[8]

Secondly, when you paint a picture you need to paint it with something. Maybe you want to do an oil painting, but maybe you want to use watercolour. And, generally speaking, you pretty much have to pick one or the other. The analog to this in R is a "graphics system". A graphics system defines a collection of graphics commands about what to draw and where to draw it. Something that surprises most new R users is the discovery that R actually has several mutually incompatible graphics systems. The two of most interest to us are the **base graphics** system that comes with R and the **ggplot2** system[9] that forms part of **tidyverse**. There's quite a difference of opinion among R users about the relative merits of these two systems. You can get started in base graphics really easily. To see just how easy it is, let's load a new data set and try to draw a picture.

3.2 The data set

The `tidylsr` package comes with a data set called `samplingframes`, a relatively simple experiment I ran a little while ago (it's experiment two from this paper). What we were interested in was understanding how people use statistical information to guide inductive inferences. For example, suppose you observe a sample of "robins" that have "plaxium blood" (whatever that is). How likely is it that "sparrows" will possess plaxium blood? Or "cows"? Does it matter how many robins you have seen? Does it matter whether you specifically selected robins and they turned out to have plaxium blood (category sampling) as opposed to detecting animals with plaxium blood that then turned out to all be robins (property sampling)? In that paper we had a computational model of inductive reasoning that made specific predictions about how the sample size (number of robins) and sampling method (property or category) would inflence people's judgments.

In this particular experiment we didn't show people animals (though we have done those too!) we just showed them small "alien rocks" called "sodor spheres", and asked people to make guesses about new rocks of different sizes: `test_loc` values of 1 and 2 were very similar to the items they were shown during training, whereas value 7 was quite dissimilar. The number of training observations ranged from 2 (`sample_size = "small"`) to 12 (`sample_size = "large"`) and was varied within-subject. So everyone saw two observations, made some generalization judgments (`response` on a scale from 0 to 9), then saw more training observations and so on. Participants were randomly assigned to a `"property"` sampling `condition` or to a category sampling one. We also recorded `age`, `gender`, and assigned each person a unique `id`. So here's the data

[8] If you are interested, take a look at the various `dev` functions for working with graphics devces, such as `dev.new`, `dev.print`, `dev.off` etc. I won't talk about those here because I'm focusing on **tidyverse** but they're handy in other situations

[9] It's worth acknowledging that **ggplot2** and builds off the **grid** graphics system that we very briefly encountered when using **TurtleGraphics**

```
samplingframes
```

```
## # A tibble: 4,725 x 8
##       id gender   age condition sample_size n_obs test_item response
##    <dbl> <chr>  <dbl> <chr>     <chr>       <dbl>     <dbl>    <dbl>
##  1     1 male      36 category  small           2         1        8
##  2     1 male      36 category  small           2         2        7
##  3     1 male      36 category  small           2         3        6
##  4     1 male      36 category  small           2         4        6
##  5     1 male      36 category  small           2         5        5
##  6     1 male      36 category  small           2         6        6
##  7     1 male      36 category  small           2         7        3
##  8     1 male      36 category  medium          6         1        9
##  9     1 male      36 category  medium          6         2        7
## 10     1 male      36 category  medium          6         3        5
## # ... with 4,715 more rows
```

Oh dear this has rather a lot of observations. Every single response by every single participant is recorded as a separate row, and there are several thousand of those in total. Drawing pretty pictures might require us to simplify this a little bit... but I promised you cake, so I don't want to bore you (yet!) with the data wrangling operations required to do this. But it's not too hard... to foreshadow the content to come in the next chapter, here's the code I use to create a `frames_small` data set that contains one row per *person*...

```
frames_small <- samplingframes %>%
  group_by(id, gender, age, condition) %>%
  summarise(response = mean(response)) %>%
  ungroup()
```

and here's what that data set looks like...

```
frames_small
```

```
## # A tibble: 225 x 5
##       id gender   age condition response
##    <dbl> <chr>  <dbl> <chr>        <dbl>
##  1     1 male      36 category      5.33
##  2     2 male      46 category      7.05
##  3     3 female    33 property      4.86
##  4     4 female    71 property      3.86
##  5     5 female    23 property      9
##  6     6 female    31 category      7.90
##  7     7 male      23 property      3.76
##  8     8 female    31 property      4
##  9     9 female    37 category      3.38
## 10    10 female    46 category      5.86
## # ... with 215 more rows
```

Let's see if we can understand what's going on with this data set by drawing pictures, and also learn about R graphics in the process.

3.3 Histograms

When analysing the data from this kind of experiment, one of the first things I check is that the `age` and `gender` variables are actually irrelevant. Honestly, I only bother to measure them at all because people seem to expect it in the literature but for the kind of questions I care about there's never any effects of age or gender. Still, you have to check just to make sure nothing weird is goind on.

With that in mind, let's give ourselves the task of drawing a histogram of the `age` variable in this data frame. We don't need to make it pretty, we just want to plot the frequency distribution for the number of points scored by the home team. As I mentioned earlier, we could do this using base graphics or we could do it the tidyverse way. Let's see how it plays out both ways.

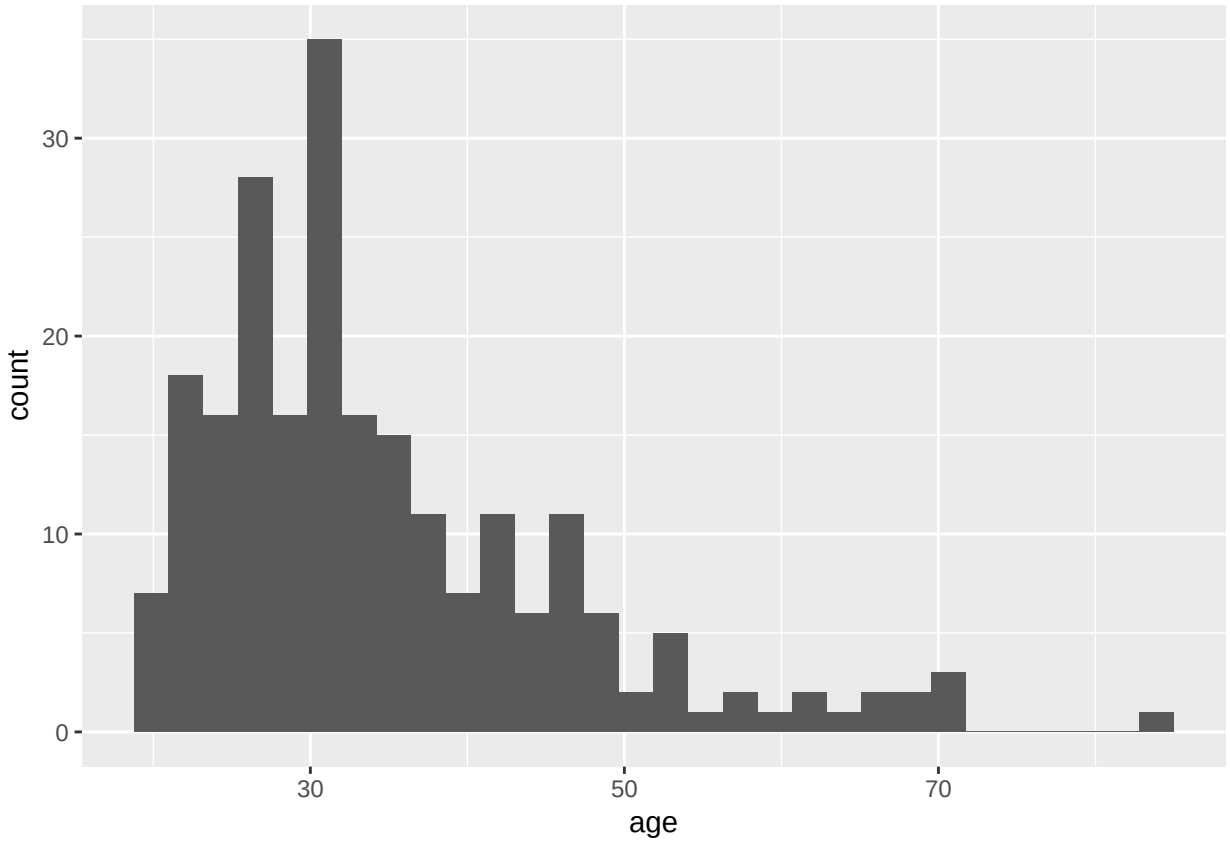
The function to do this in base graphics is called `hist` and here it is:

```
hist(frames_small$age)
```

It's not exactly pretty – though it's way nicer than the default plots that I remember seeing when I was younger – but it gets the job done.

Okay, how do I create a histogram the tidyverse way? Much like base graphics, **ggplot2** recognises that histograms are such a fundamentally useful thing that there exists a "geom" (a term I'll explain in a moment) for them, but there's no way to draw a plot without going through a somewhat more convoluted process:

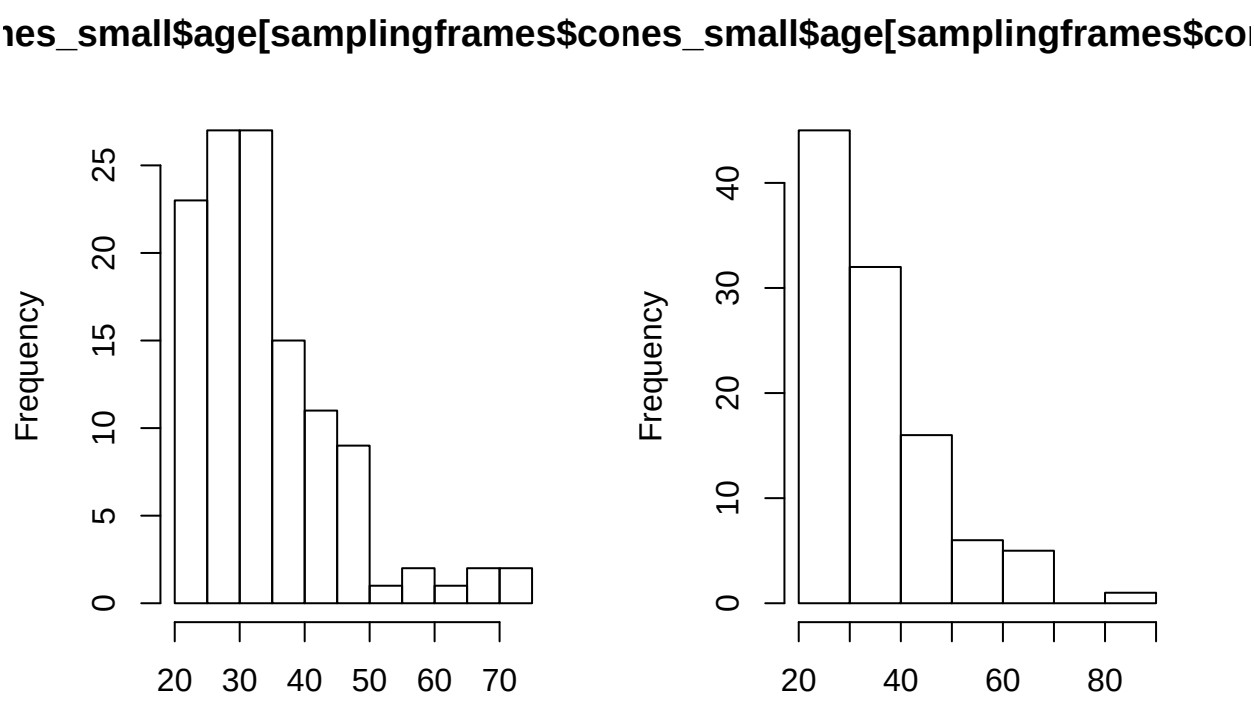
```
frames_small %>%
  ggplot(aes(x = age)) +
  geom_histogram()
```



The output is prettier, admittedly, but our goal here wasn't to be pretty. Our goal was to be simple. For this comparison, the tidyverse approach is not as straightforward as base graphics.

Where the tidyverse versions shines is when we want to do something a little bit more complicated. Suppose I wanted two histograms side by side, plotting the age distribution separately by `condition`. In base graphics, it's a little cumbersome. What I have to do here is manually control the "layout" of the plot, dividing it into two separate panels and then drawing the histogram separately into each panel. That gives me code that looks like this:

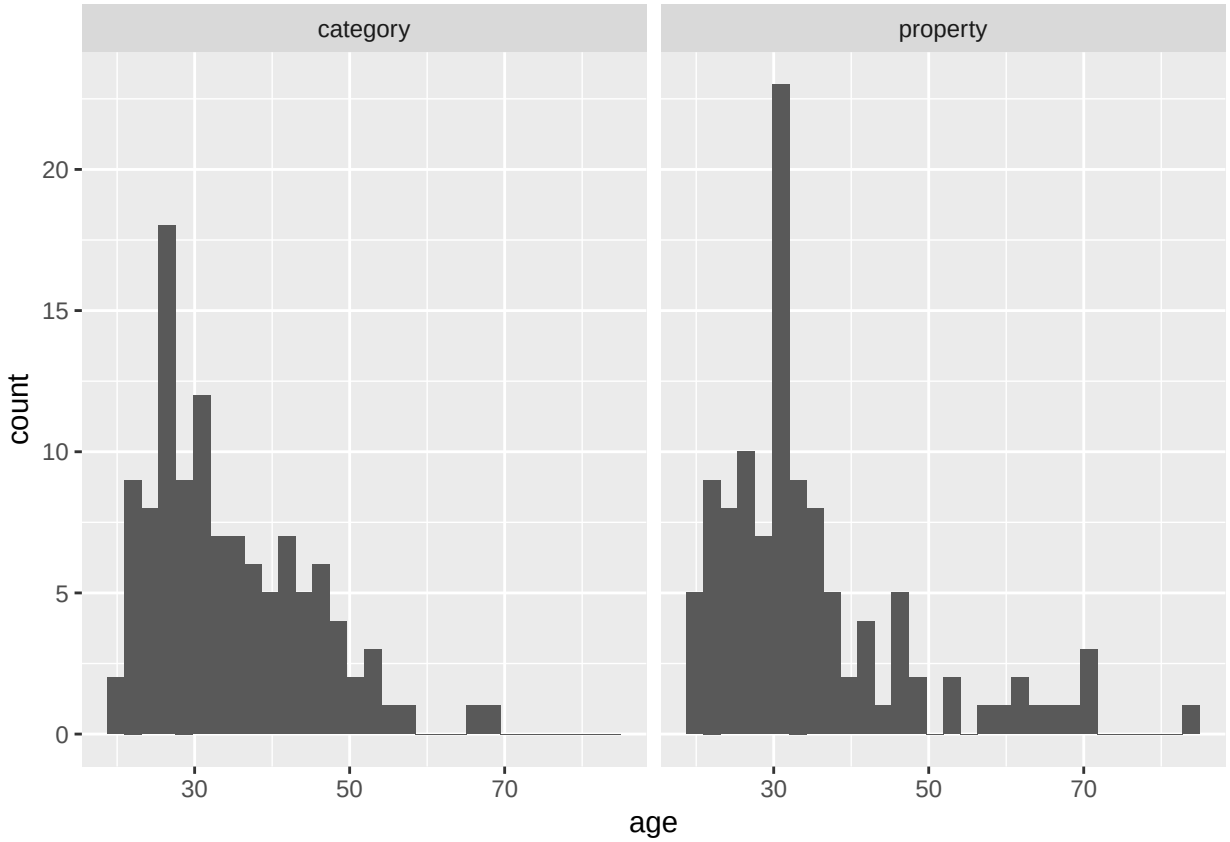
```
layout(matrix(1:2, 1, 2))
hist(frames_small$age[samplingframes$condition == "category"])
hist(frames_small$age[samplingframes$condition == "property"])
```



I need a lot more code as I did for the original version and I've ended up with an ugly plot. Of course, base graphics absolutely does allow me to do a better job than this, but I hope you can see that it will take quite a bit of effort to turn this into something readable.

What about the **ggplot2** version? It turns out that it's extremely easy. Splitting an existing plot into "facets" is a basic operation within **ggplot2** so all I need to do is add one line of code that specifies which variable to use to make the facets! The result is actually kind of nice:

```
frames_small %>%
  ggplot(aes(x = age)) +
  geom_histogram() +
  facet_wrap(~condition)
```



For me at least, this is the big advantage to the **ggplot2** approach. There's a certain amount of effort required to construct the basic plot, but once that is done, you can modify or manipulate that plot in an extremely flexible fashion with very little effort indeed. What I've found in practice is that the low-effort to making changes makes me much more willing to "play around" with different ways of visualising the data. So, while I'll admit that there are some situations where I resort to using base graphics (mostly when I have a very unconventional graph to draw), I tend to find the tidyverse approach works better in the majority of cases.

As far as what the data visualisation is telling us: the distribution of ages was mostly in the 20-40 age range with a modest positive skew. That's pretty typical of these studies. More importantly, it's pretty clear from inspection that there's not much of a difference in the age distribition across conditions, which is what I'd hope to see given that people were assigned randomly.

3.4 Scatter plot

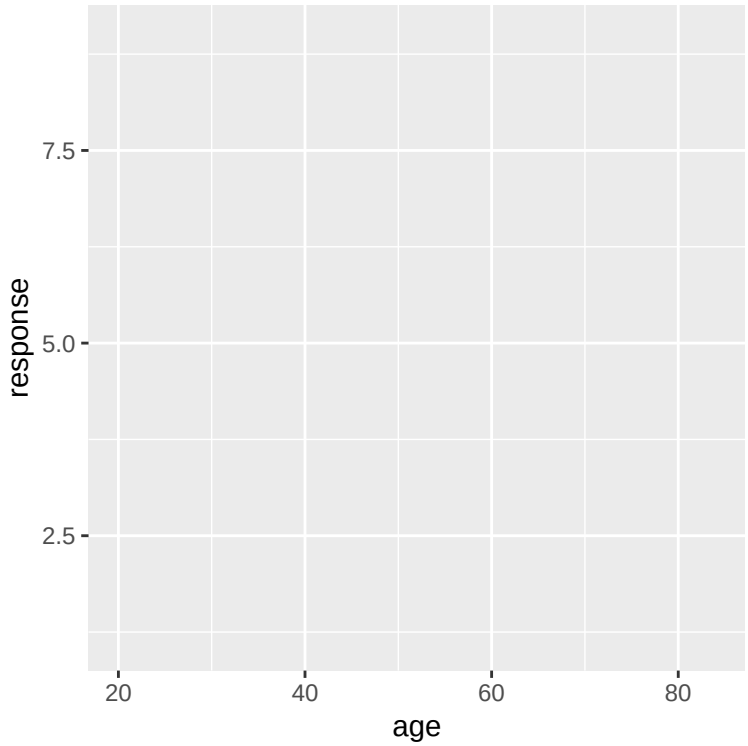
The second kind of plot I'll talk about is a scatter plot, in which each observation is drawn as a point, and the graph shows the values on one variable against the values on another. It's such a simply plot that I'll use it as a mechanism to illustrate the key ideas in **ggplot2**.

Lets build our plot piece by piece. The data we want to use here come from `frames_small`, so the first thing we'll do is pipe this data set to the `ggplot()` function and see what it produces:

```
frames_small %>%
  ggplot()
```

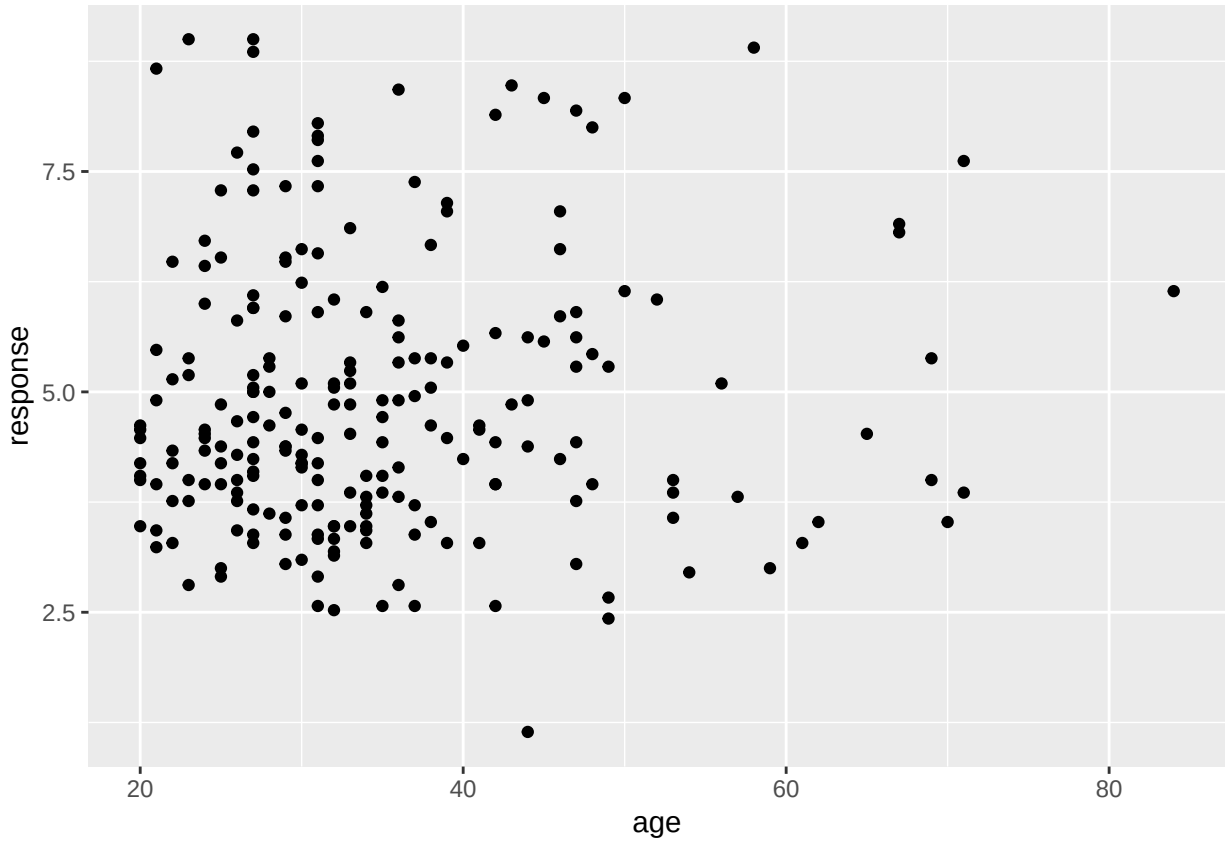
In retrospect that's unsurprising. Although we've passed the data to `ggplot()`, we haven't told R what it ought to *do* with these data, so the output is simply a blank canvas! To make some progress, the next step is to specify a `mapping` for the plot that will tell R something about what roles the different variables play. In **ggplot2** these mappings are described as a set of "*aesthetics*", defined using the `aes()` function. There are many different aesthetics that can be used, but for a scatter plot the only things we really *need* to specify as aesthetics are the variable on the `x` axis and the variable on the `y` axis. So if we wanted to plot `y = response` against `x = age`, this is the mapping we would need:

```
frames_small %>%
  ggplot(mapping = aes(x = age, y = response))
```



Hm. That's clearly some progress. The canvas now has the axis labels and gridlines reflecting the fact that R now knows which variables we're hoping to ploy. However, we still don't have any data, because we haven't told R what it should *do* to render the data. This is the role played by *geoms*, which specify different ways in which data can be displayed. Since we're trying to draw a scatter plot, I'm going to use the simplest possible *geom* function, namely `geom_point()`. All it does is draw a dot for each data point:

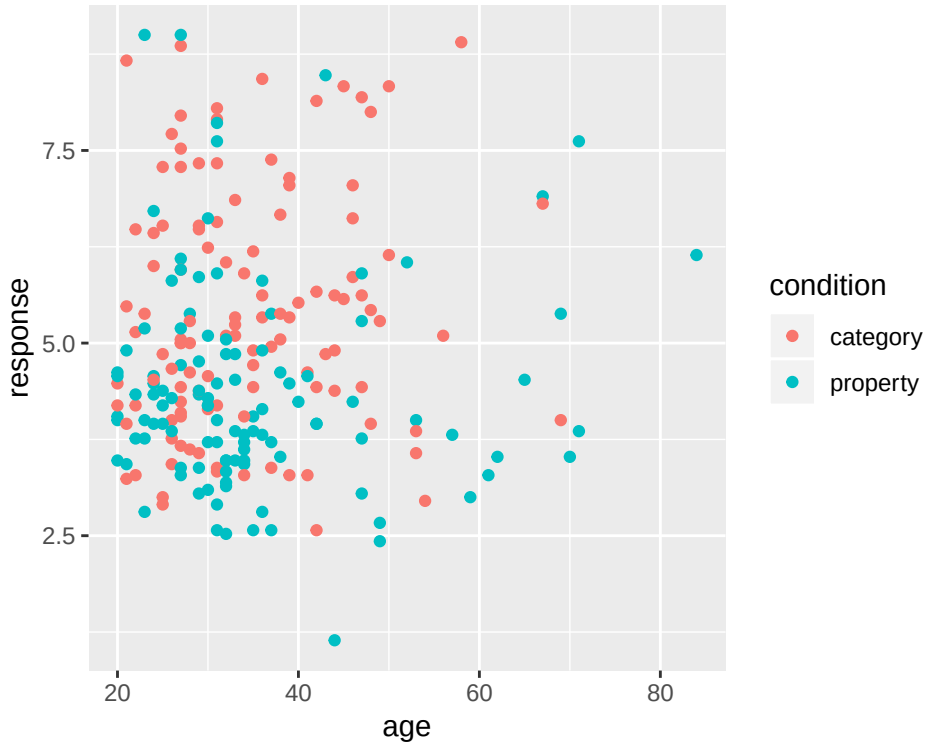
```
frames_small %>%
  ggplot(mapping = aes(x = age, y = response)) +
  geom_point()
```



Now we have our scatter plot! From visual inspection there doesn't seem to be any strong relationship between the `age` of a participant and the `response` they give. Again, that's not surprising, but useful to check.

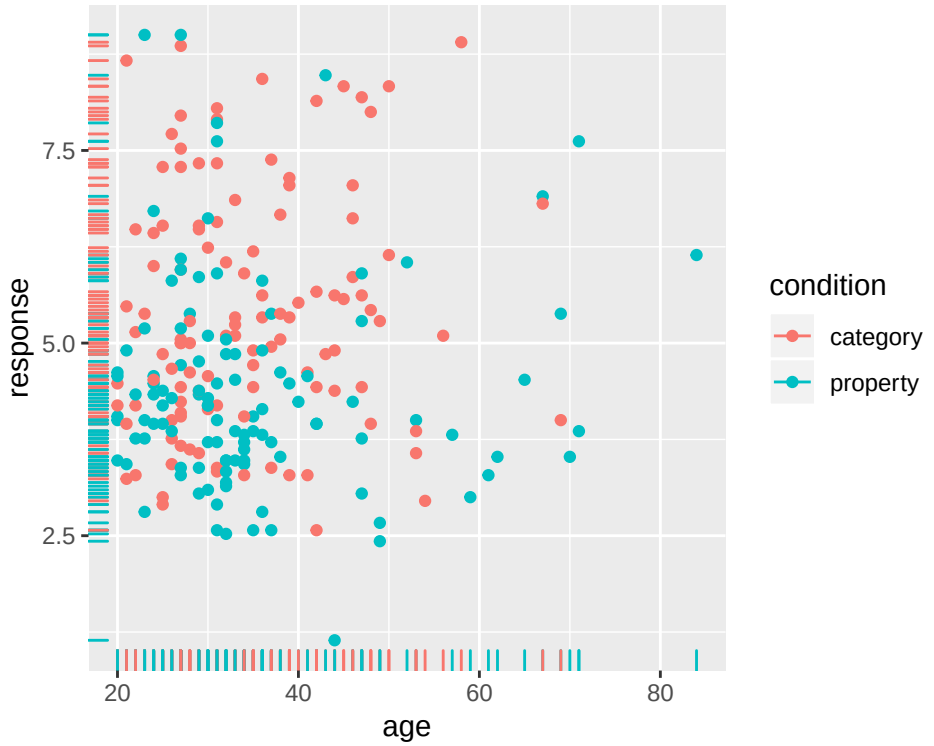
It's worth highlighting the different roles played by aesthetics and geoms. For instance, I could have expanded the list of aesthetics to include `colour = condition`, which would indicate that the colour of each dot should indicated which condition the participant in question was assigned to:

```
frames_small %>%
  ggplot(mapping = aes(x = age, y = response, colour = condition)) +
  geom_point()
```



However, I can also add new geoms that will draw new layers to the plot. For example, `geom_rug` adds a visual representation of the marginal distribution of the data on both axes, like this:

```
frames_small %>%
  ggplot(mapping = aes(x = age, y = response, colour = condition)) +
  geom_point() +
  geom_rug()
```



Notice the style here. A pretty typical way to build a visualisation is to construct it in layers, *adding* new geoms, aesthetics and other plot customisations as you go. So you'll often end up with code structured like this:[10]

```
DATA %>%
  ggplot(aes( LIST_OF_AESTHETICS )) +
  A_GEOM +
  ANOTHER_GEOM +
  ETC
```

In any case, looking at these scatter plots there's nothing that immediately suggests and differential patterns of responding as a function of age but there is a hint that responses are lower in the property sampling condition (blue) than in the category sampling condition (red).

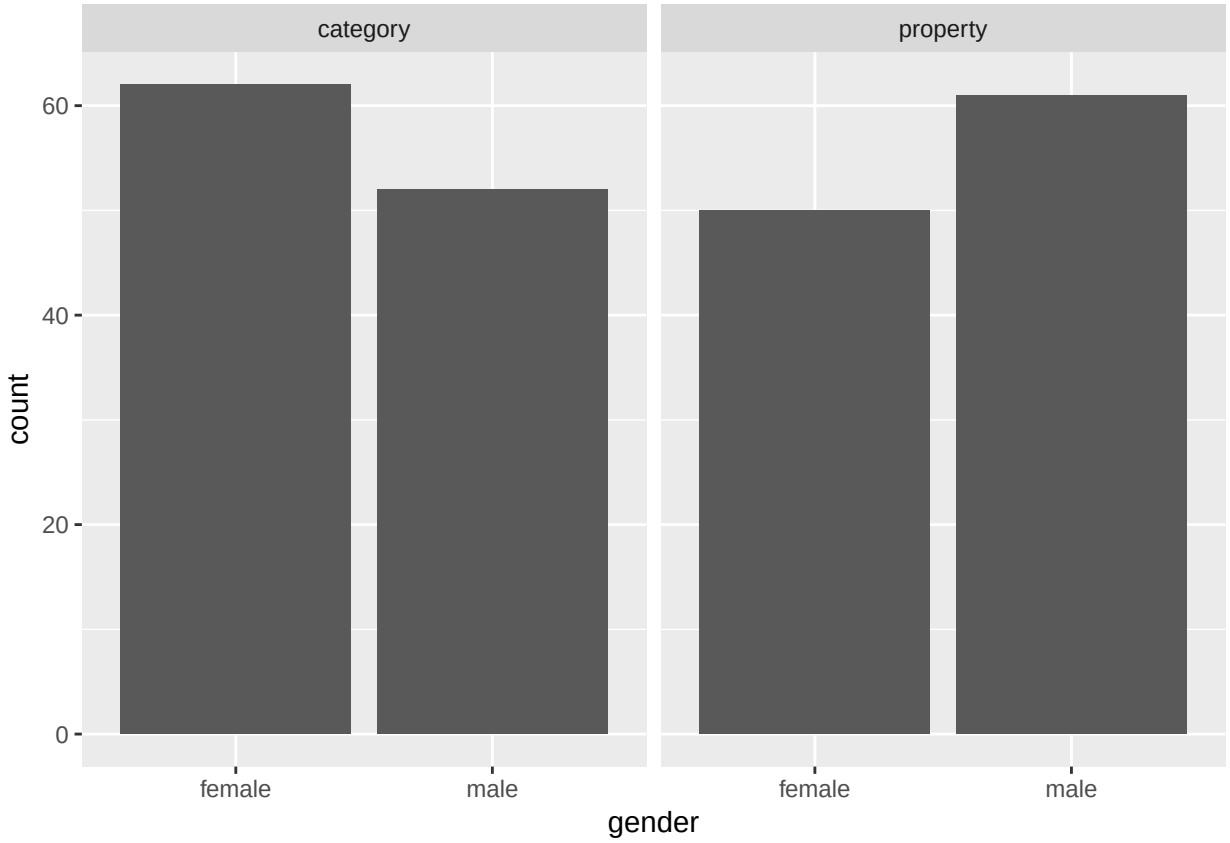
3.5 Bar graphs

The humble bar graph is a staple in the scientific literature, and can be rendered by **ggplot2** with the assistance of the `geom_bar()` function. Suppose I want to check that the gender distribution for this study was relatively even across conditions.[11] To do that, I'll set up my mapping with `gender` on the x-axis (using `ggplot` to initialise the plot and `aes` to specify the aesthetics), create separate panels for each `condition` (using `facet_wrap` which I'll explain in more detail below), and then use `geom_bar()` to draw the plot:

[10] There is a little bit of weirdness to this. Once you've gotten accustomed to using the pipe operator `%>%` it starts to feel like we should be writing code like `data %>% ggplot() %>% geom() %>% etc` rather than `data %>% (ggplot() + geom() + etc)`. This is purely for historical reasons: **ggplot2** predates the pipe operator and so unlike everything else in **tidyverse** it's not very compatible with `%>%`.

[11] In this study I asked people to respond with a three value questionnaire, with options for "male", "female" and "other" as possible gender identities. I'm not entirely satisfied with this as a measurement method though, and the question wasn't compulsory to answer.

```
frames_small %>%
  ggplot(aes(x = gender)) +  # set up the mapping
  facet_wrap(~condition) +   # split it into plots
  geom_bar()                 # add the bars!
```



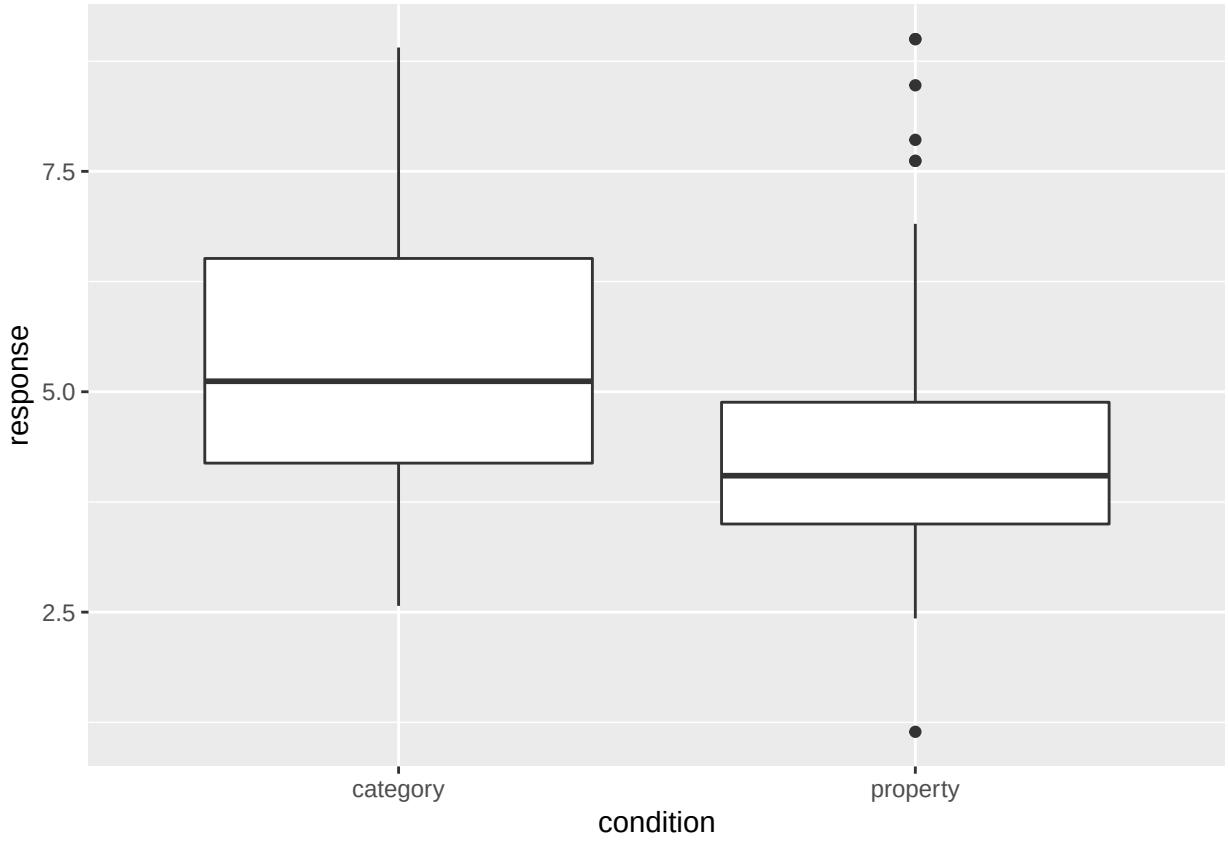
As one would expect for random assignment there's a little unevenness but not very much. That looks totally fine to me, and at this point I've convinced myself that the `age` and `gender` variables really don't matter very much, so I can start digging into the details about how the manipulated variables of theoretical interest (`sample_size`, `condition` and `test_loc`) relate to the inductive generalisation `response` judgment that people make...

3.6 Box plots

Another classic data visualisation used in psychology is the box plot (due to John Tukey), which presents a schematic representation of the distribution of responses. Box plots can be drawn in many different forms, but the most conventional version the median value (50th percentile) as a thick horizontal line, contained within a box that spans the range from 25th percentile to the 75th percentile. Above and below the box are the "whiskers" which extend to cover the full range of the data (after excluding outliers). Outliers are plotted as individual dots. Note that in this context an outlier is conventionally defined as a point that lies more than 1.5 times the interquartile range from the median: this is often a convenient heuristic to use but it's not magical. Don't read too much into the fact that a data set includes outliers!

To construct a boxplot using **ggplot2** all we need to do is add a `geom_boxplot()` to the graph. So if I want to plot the distribution of `response` values across participants (averaged across test items) separately for each `condition`, I could do this:

```
frames_small %>%
  ggplot(aes(x = condition, y = response)) +
  geom_boxplot()
```



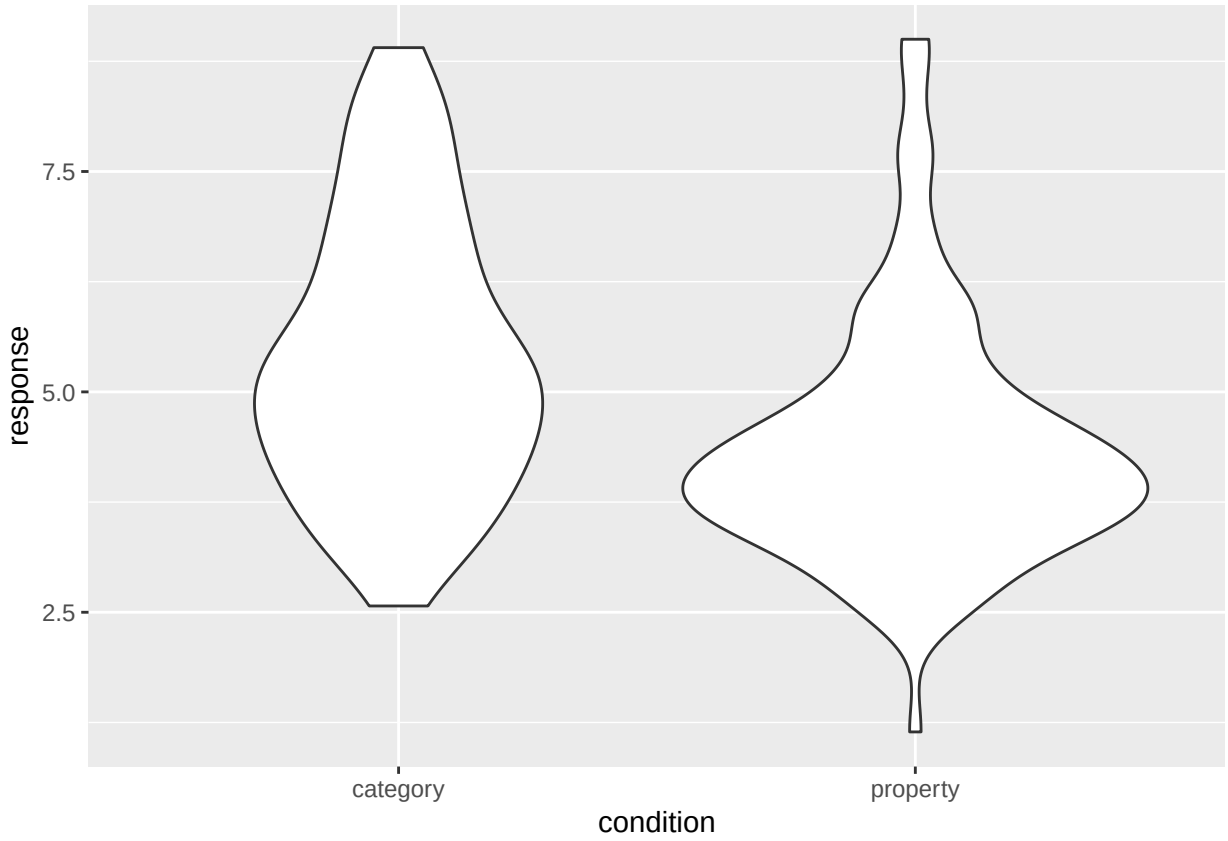
This plot is easy to interpret and provides our first hint that there is in fact something of interest going on in these data. On a first pass, it rather looks like people are less willing to make inductive generalisations (i.e., give lower scores) in the property sampling condition than in the category sampling condition!

3.7 Violin plots

Although the box plot is an old method for visualising the key characteristics of distribution it's a very good one. It's visually simple, meaning that you can put a lot of box plots in a single graph without causing too much clutter, yet it still conveys a lot of information about the distribution. So there are many situations in which they are worth including (we'll see a good example in a moment)

On the other hand, for the specific situation here where there are only two disributions to be displayed, it does seem like we could do better. With the state of computing having advanced quite dramatically over recent decades, it is extremely easy to construct more complicated *kernel density estimates* of the shape of the population from which the sample is drawn. The *violin plot* provides a method for visually displaying a kernel density estimate. If we switch from `geom_boxplot()` to `geom_violin()` the resulting plot looks like this:

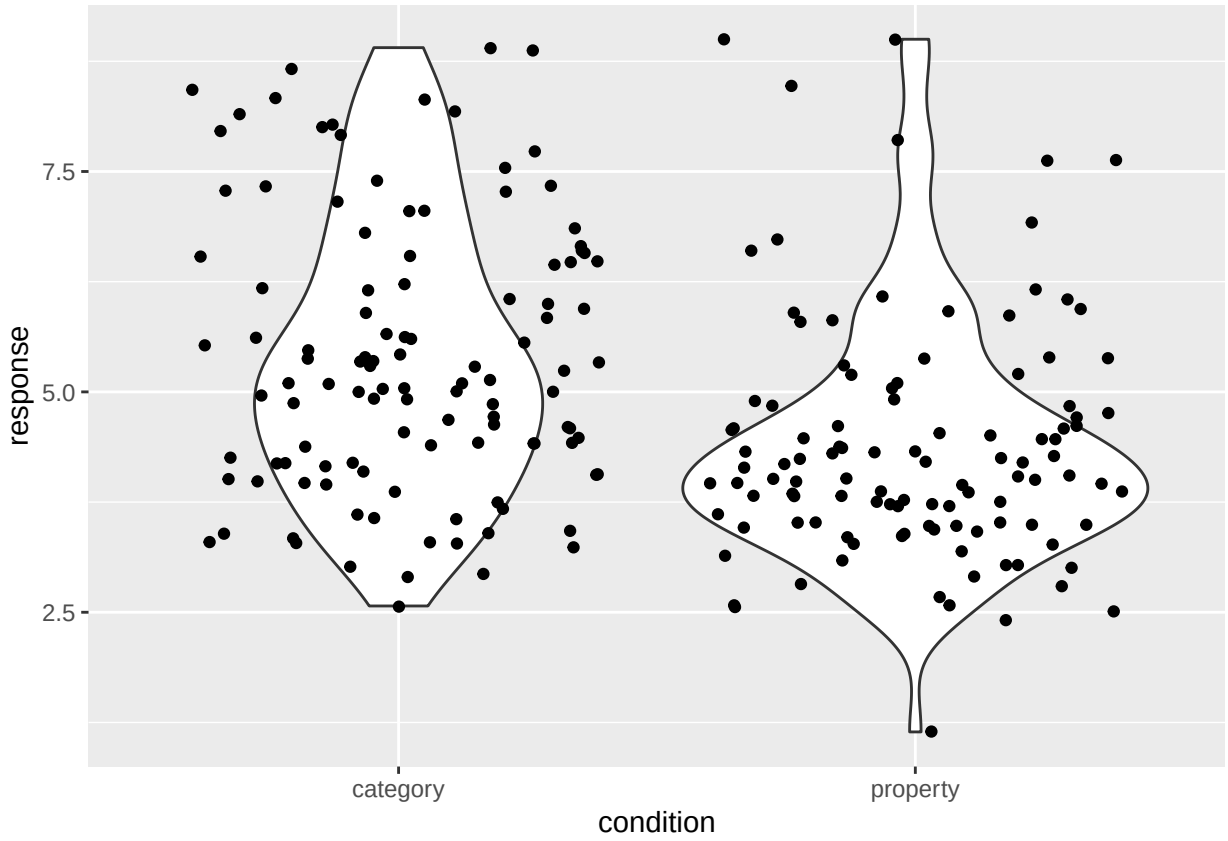
```
frames_small %>%
  ggplot(aes(x = condition, y = response)) +
  geom_violin()
```



It's intuitively obvious what this plot is doing: the "width" of the violin[12] at each point shows the estimated "density" of the population distribution at that point. To make it a little clearer what this is showing, let's overlay the raw data on the plot, using the `geom_jitter()` method to make it a little easier to see:

```
frames_small %>%
  ggplot(aes(x = condition, y = response)) +
  geom_violin() +
  geom_jitter()
```

[12]Yeah, they never actually look like violins. We should just give up and call them blob plots



As you can see, in the property sampling condition the data are concentrated pretty tightly around 4.5, whereas in the category sampling condition the data are shifted upwards and spread out. That's what produces the different shapes in the violin plots.

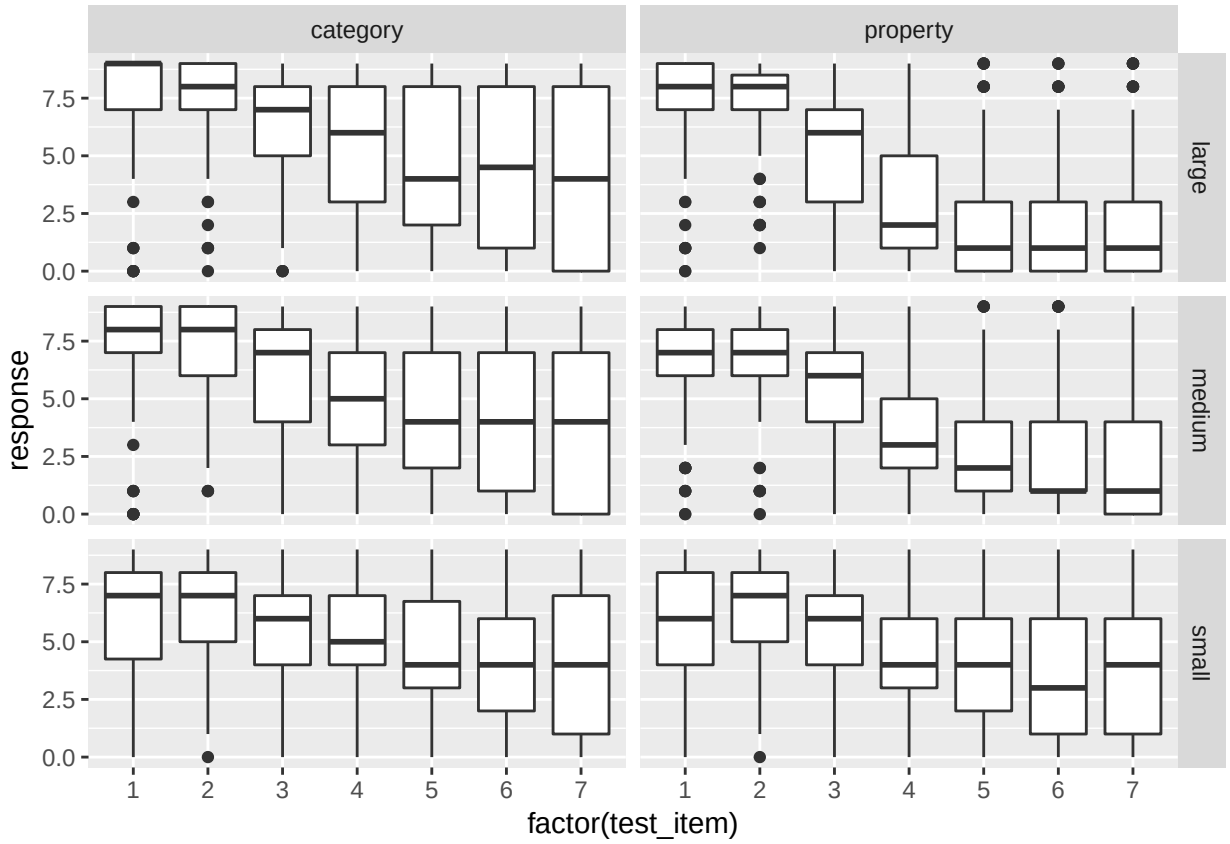
3.8 Facetted plots

The big problem I have with the data visualisations we've drawn so far is that we're really not getting a very fine grained story. When we constructed the `response` variable for the `frames_small` data frame we averaged over all three values of `sample_size` and across all seven values of `test_loc`. That's fine as a first pass if we want to take a quick look at the difference across `condition`, but in real world data analysis we need to hold ourselves to higher standards! So lets go back to the raw `frames` data, and see if we can find a better visualisation. This brings us naturally to the topic of *faceting* a plot...

A common task in data visualisation is to construct facet plots that display an existing plot separately for each group. We've seen this twice already, once in the histogram at the start of this section and previously in the prelude. In both cases I used `facet_wrap()` to do the work, using the `~` operator to specify a one-sided formula (e.g., `~condition`). This function splits the plot by the levels of a factor, and then "wraps" them around to keep the plot tidy. For example, if there were 16 levels of the `condition` variable then `facet_wrap()` would probably give us a 4x4 grid.

If you want more precise control over the faceting, an alternative approach is to use `facet_grid()`. For instance, let's suppose I want to draw separate boxplots for the `response` variable for every possible `test_item`, broken down by `condition` and `sample_size`. That's going to give us 42 boxplots, so some degree of care is required here! Here's my first attempt:

```
samplingframes %>%                          # start with the full data set!
  ggplot(aes(
    x = factor(test_item),                  # treat "test_item" as categorical
    y = response)) +                        # y variable is "response"
  facet_grid(sample_size ~ condition) +     # add faceting
  geom_boxplot()                            # oh, right... add the boxplots :-)
```



That's not bad as a first pass. There's a few things I'd like to do to tidy it though:

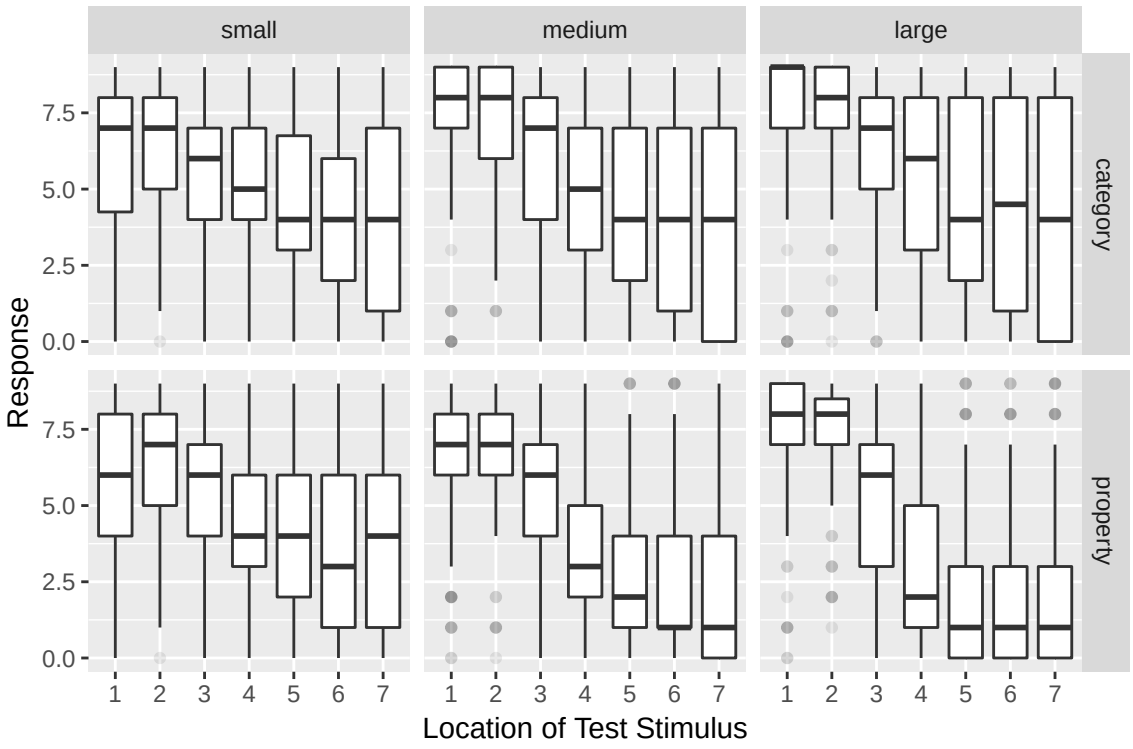
- Flip the grid by changing the formula to `condition ~ sample_size`
- Reorder the sample size facets by changing `sample_size` to a factor
- Adding nicer axis labels using `xlab()` and `ylab()`
- Make the "outliers" less obtrusive using transparency

To do this, the first thing I'm going to do is use `dplyr::mutate` – discussed in the previous section – to convert the `sample_size` variable to a factor, and make sure the levels are specified in the order I want them to appear in the plot:

```
samplingframes <- samplingframes %>%
  mutate(sample_size = factor(sample_size, levels = c("small","medium","large")))
```

Now my command to draw the plot looks like this:

```
samplingframes %>%
  ggplot(aes(
    x = factor(test_item),
    y = response)) +
  facet_grid(condition ~ sample_size) +  # reversed faceting
  geom_boxplot(outlier.alpha = 0.1) +    # alpha sets the transparency
  xlab("Location of Test Stimulus") +    # add x-label
  ylab("Response")                       # add y-label
```



It's essentially the same as before, but we've switched the ordering of variables in `facet_grid()`, added the `outlier.alpha = 0.1` argument to `geom_boxplot()` to make the outliers fade into the background, and then specified some nicer axis titles. The resulting plot is a fair bit more readable.

As for what we're seeing in the data, there are a few different things to notice:

- Within every facet the responses tend to shift *downwards* from left to right: as the test item becomes less similar to the training items, people are less willing to make generalisations. This pattern of similarity-based generalisation is unsurprising and it's a finding that has been replicated many, many times in the literature.
- The effect of sample size is inhomogeneous. For stimuli that are very similar to the training items (test locations 1 and 2), increasing the sample size pushes the generalisation *upwards*, regardless of whether category sampling or property sampling is applied
- For stimuli that are very dissimilar (especially test items 6 and 7), the effect of sample size depends on the sampling method. For category sampling, basically *nothing happens*: the box plots for large sample sizes aren't any different to those for small sample sizes. Yet for property sampling, there's a systematic tendency for these to shift *downwards*

Assuming that these findings replicate[13] it looks like we have a three-way interaction of some kind!

[13]They appear to – we ran 9 experiments in total and while some effects are a little fragile and I wouldn't put too much

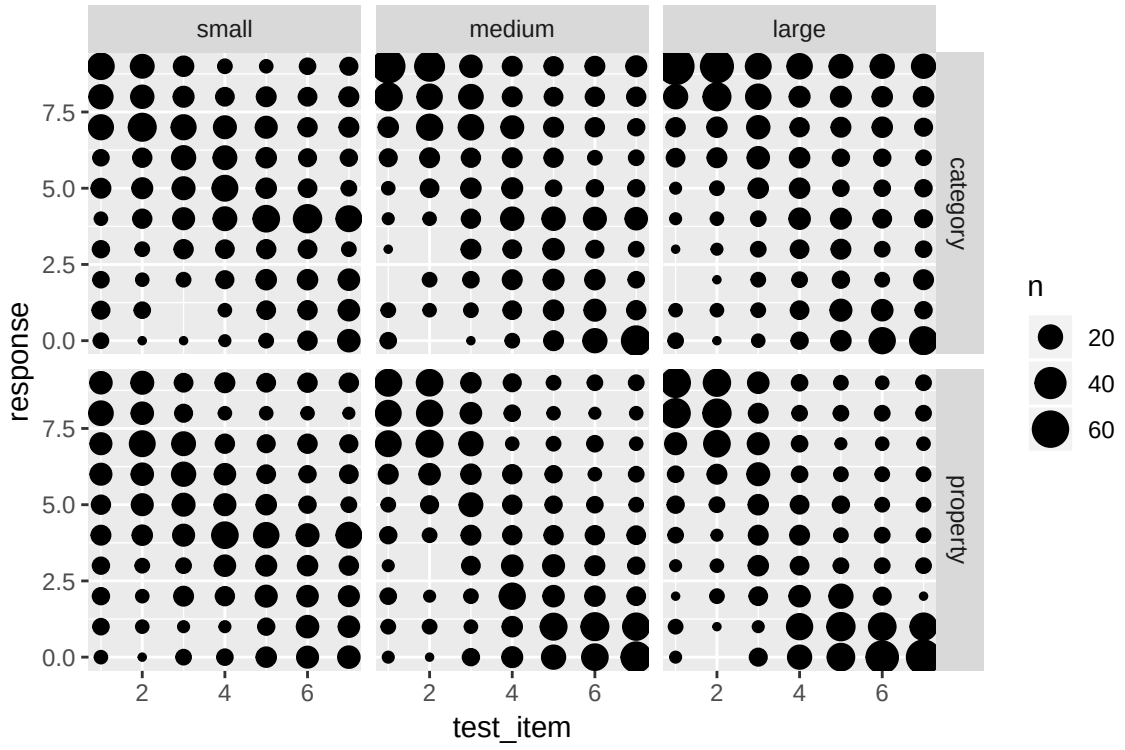
To be honest though, I'm still not pleased with this graph. I think we can do better.

3.9 Bubble plots

One difficulty with box plots is that they only work as distributional summaries when the data are unimodal. If most people respond with extreme values, the box plot ends up being a little misleading. Violin plots are often better at capturing multimodality, but they're designed to work for continuous variables and often behave poorly when applied to data that fall in a small number of ordered categories. Psychological data from questionnaires or – as in the `frames` data set - other ordinal response methods often have exactly this characteristic. Neither box plots nor violin plots are ideally suited to this situation.

Fortunately a bubble plot can help out here. This takes the form of a scatter plot, but instead of plotting every data point as a unique dot, we plot dots at every location whose size (area) is proportional to the number of cases at that location.[14] To do this with **ggplot2** all we need to do is use the `geom_count()` function. When applied to the `frames` data, we get this:

```
samplingframes %>%
  ggplot(aes(x = test_item, y = response)) +
  facet_grid(condition ~ sample_size) +
  geom_count()
```



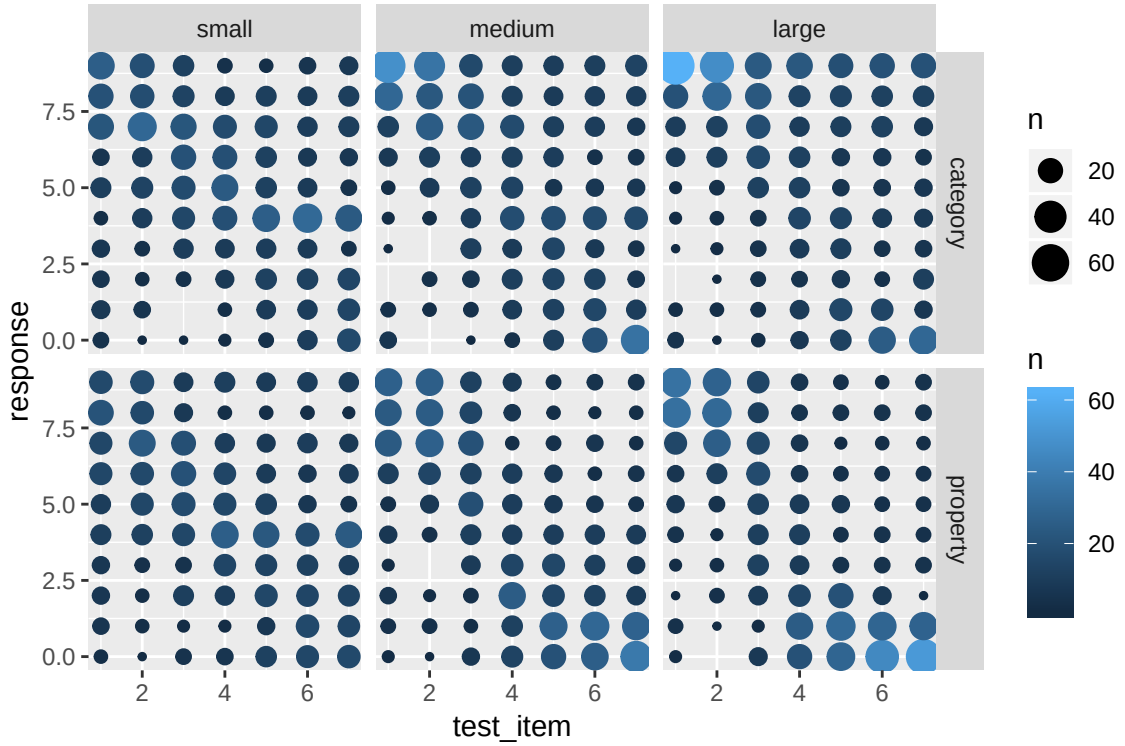
faith in their ability to replicate, the core effect in which there are systematic differences in how statistical information (e.g., sample size, base rate, auxiliary data) interacts with the sampling mechanism replicated every time; oh and if you go read the paper the model predictions correlate with human data very highly, always produce the same qualitative patterns regardless of parametre values... etc, etc. This isn't a cognitive science paper so I won't bore you with details, but the reason this works as cleanly as it does is that we didn't choose these manipulations arbitrarily. We had a proper theory to guide us!

[14] There are drawbacks to bubble plots. The psychophysical function for area isn't linear, so subjective area doesn't scale perfectly with actual area. If you need people to be able to make fine-grained comparisons between quantities don't use bubble plots. However, my goal here is different - I don't mind if we can't quite see the "perfect" relationship, I just want the "gist" to come through properly.

This version of the plot highlights a failure of the box plot version. Compare the distribution of responses to test item 7 in the category sampling condition (top row). In the `small` sample size, the modal response is 4, with most people using the middle category, whereas for the `large` sample size most people are using the extremes, responding with either 0 or 9. However, both cases have the same median and very similar inter-quartile ranges, so the box plots look almost the same!

On the other hand, this plot is a little hard to read. We've added a lot of new detail about the low frequency responses and in doing so learned more about the overall distribution of responding, but that's come at the cost of making it harder to see what the most typical responses are! A compromise that I am often fond of is using shading to slightly de-emphasize the low frequency cases. Larger bubbles should stand out (dark colour), but low frequency responses should fade into the light grey background. It's actually pretty easy to do that: `geom_count()` keeps track of the number of observations in each cell, and so we can add a new aesthetic to the `geom_count()` layer, simply by specifying `colour = ..n..`:

```
samplingframes %>%
  ggplot(aes(x = test_item, y = response)) +
  facet_grid(condition ~ sample_size) +
  geom_count(aes(colour = ..n..))
```

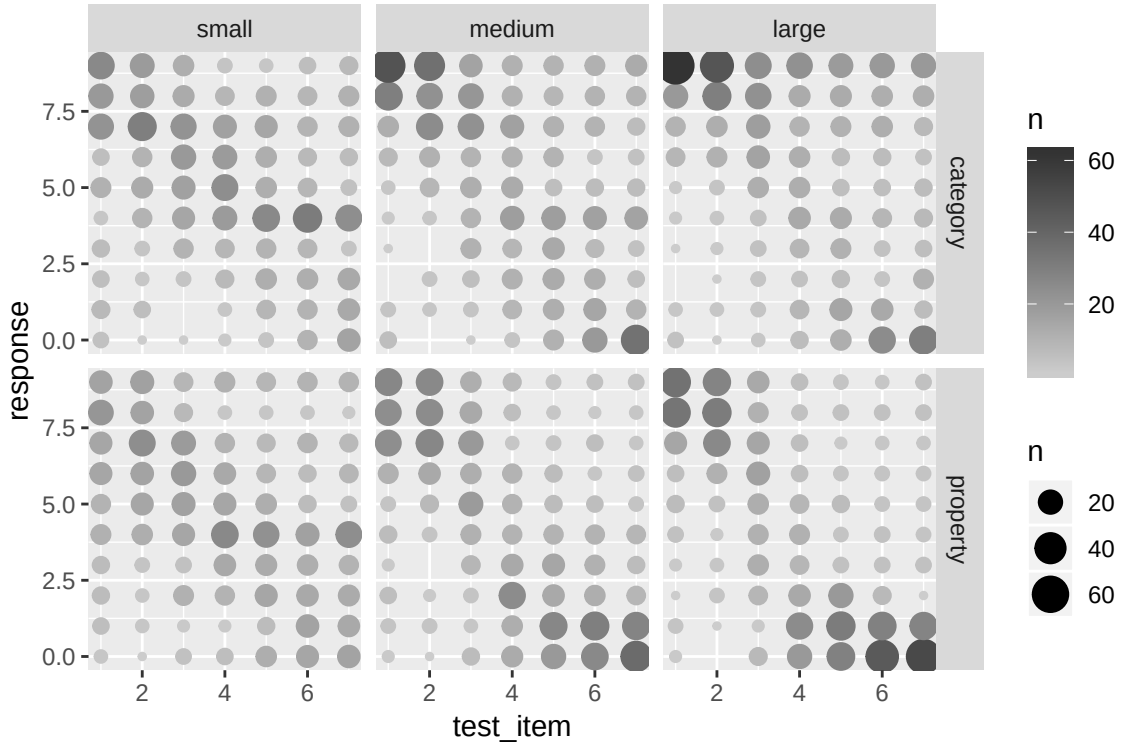


That's a little nicer, but in this case I don't want or need the pretty blue colouring. What I want to do is specify my own grey scale, where larger dots are coloured closer to black (e.g., upper bound is `"grey20"`) and the smaller dots are coloured in a light grey (e.g., lower bound is `"grey80"`).[15] Then all I have to do is this:

```
samplingframes %>%
  ggplot(aes(x = test_item, y = response)) +
```

[15] R is quite flexible in how it handles colours. For most purposes you can use an English colour word and R will have a rough idea what you mean. Type `colours()` at the console to see a list of the colour names it knows: there are a lot! Alternatively you can specify a hexadecimal string like `#88398A`, or use a function like `rgb()` or `hsv()` to construct the colour you want.

```
  facet_grid(condition ~ sample_size) +
  geom_count(aes(colour = ..n..)) +
  scale_colour_gradient(low = "grey80", high = "grey20")
```



To my mind that's much more readable! It captures the multimodality of the responding without overwhelming the viewer with too much detail. There are still some issues that you might want to care about in real life – the figure is a little too low contrast for people with less than perfect vision, for instance. But it comes a lot closer to what we need.

3.10 Error bars

As much as I love the bubble plot above, academic journals often expect a more compressed representation of the data, especially when the manuscript you're submitting involves a large number of experiments. It's often the case that we want to summarise the data from a single condition using a "mean plus error bar" style plot. A common format is to plot the mean for every experimental condition and have error bars plotting the 95% confidence interval for the mean. To do that for the `frames` data, let's begin by constructing a summary data set

```
frames_mean <- samplingframes %>%
  group_by(condition,sample_size,test_item) %>%
  summarise(
    mean_response = mean(response),
    lower = lsr::ciMean(response)[1],
    upper = lsr::ciMean(response)[2]
  )
frames_mean
```

```
## # A tibble: 42 x 6
```

```
## # Groups:   condition, sample_size [6]
##    condition sample_size test_item mean_response lower upper
##    <chr>     <fct>           <dbl>         <dbl> <dbl> <dbl>
##  1 category  small               1          6.07  5.55  6.59
##  2 category  small               2          6.26  5.84  6.69
##  3 category  small               3          5.87  5.50  6.24
##  4 category  small               4          5.11  4.74  5.49
##  5 category  small               5          4.55  4.12  4.99
##  6 category  small               6          4.16  3.69  4.63
##  7 category  small               7          3.98  3.44  4.52
##  8 category  medium              1          7.32  6.85  7.78
##  9 category  medium              2          7.17  6.80  7.54
## 10 category  medium              3          5.98  5.54  6.42
## # ... with 32 more rows
```

In this case I'm using the `ciMean()` function from the **lsr** package to compute the 95% confidence intervals. Just so that you can see how this function works – without all the additional complications from calling it within the `summarise()` operation – here's a simple example. I'll generate 20 numbers from a normal distribution with true mean 100 and true standard deviation 15 (following the convention used for IQ scores), and then use the `ciMean()` function to calculate a 95% confidence interval:

```
fake_iq <- rnorm(n = 20, mean = 100, sd = 15)  # normally distributed data
lsr::ciMean(fake_iq)                           # 95% confidence interval
```

```
##             2.5%    97.5%
## fake_iq 84.83717 100.5382
```

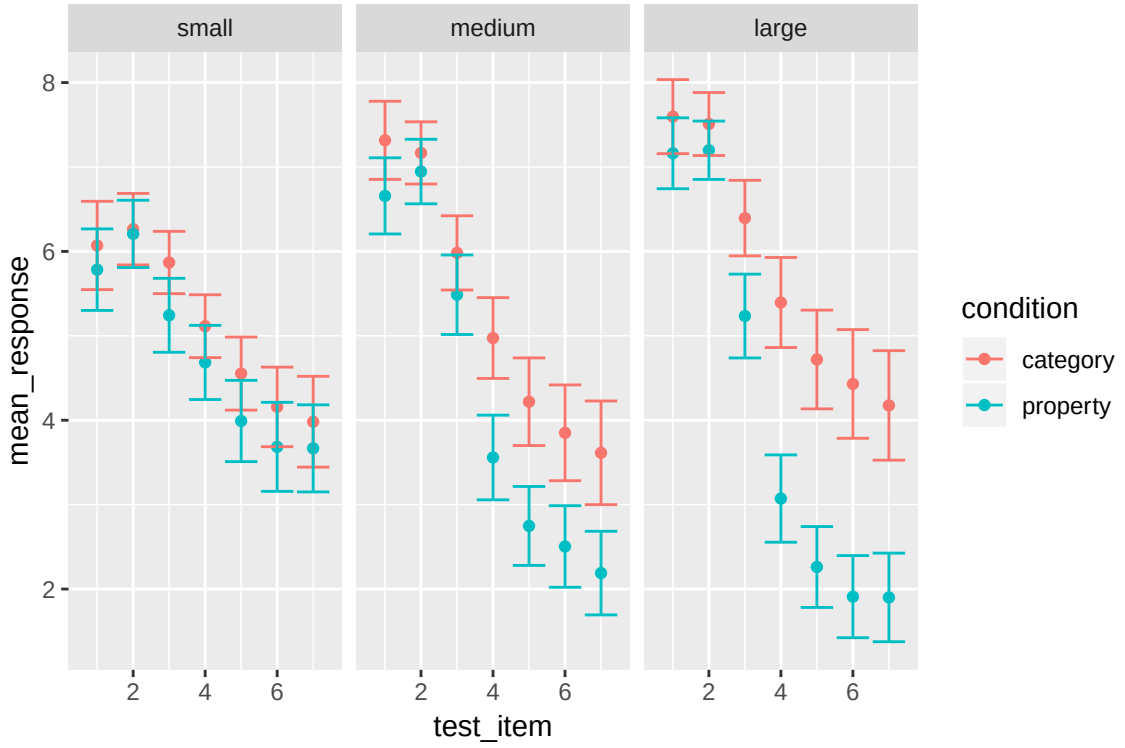
So the `lower` variable in `frames_mean` is the lower bound on the 95% confidence interval, and `upper` is the upper bound. In any case, now that we have data in an appropriate format we can create the plot.

To add error bars is no more difficult than anything else in **ggplot2**. There is a geom called `geom_errorbar()` that will draw those for us. However, it does need some new information. In our previous graphs we've specifed three aesthetics at the beginning of our graphing exercise, namely `x = test_item`, `y = mean_response` and `colour = condition`. That works fine for `geom_point()` because doesn't need anything else, but `geom_errorbar()` also needs to know which variables specify the `ymin` and `ymax` values that will be displayed in the error bars. If I wanted to, I could could include these aesthetics within the original call to `ggplot()` but I'll do something a little different this time. Individual geoms can have their own unique aesthetics, so I can insert a new call to `aes()` within the `geom_errorbar()` call. Conceptually I find this neater, because it makes clear that `ymin` and `ymax` are aesthetics that apply to the error bars, but not to other geoms in the plot. Anyway here's the command:

```
my_pic <- frames_mean %>%
  ggplot(aes(x = test_item, y = mean_response, colour = condition)) +
  geom_point() +
  geom_errorbar(aes(ymin = lower, ymax = upper)) +  # add the error bars!
  facet_wrap(~sample_size)
```

Hey, where's my plot? Oh right... notice that I assigned the result to a variable called `my_pic`. Under the hood, `my_pic` is essentially a big list of information that specifies everything R needs to know to draw the plot. So if we want to *see* the plot, all we need to do is call `print(my_pic)` or `plot(my_pic)`, or even more simply I can do this:

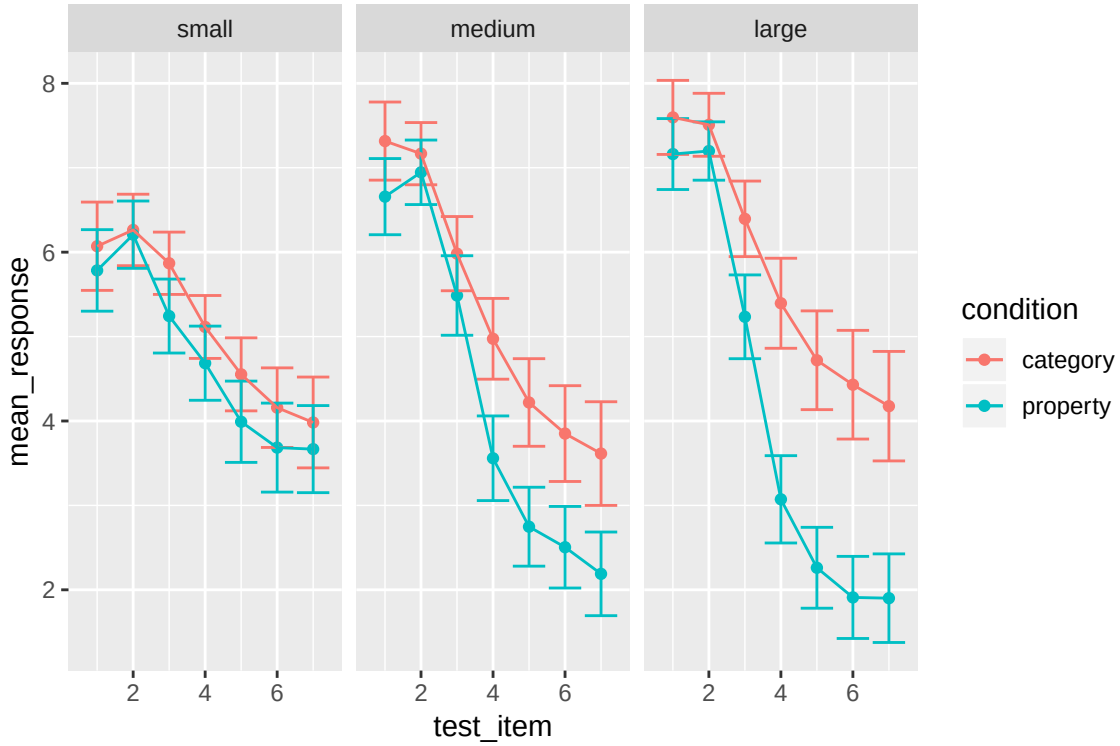
```
my_pic
```



Clearly, when compared to the bubble plot we are losing a lot of information – all we have left is information about the distributional mean – but it's still fairly informative about what is going on across the diferent conditions.

Another thing worth noting. Because the test items vary along a continuous scale, it's appropriate to connect the points with lines. I can do this in a straightforward way simply by adding `geom_line()` to the plot:

```
my_pic + geom_line()
```

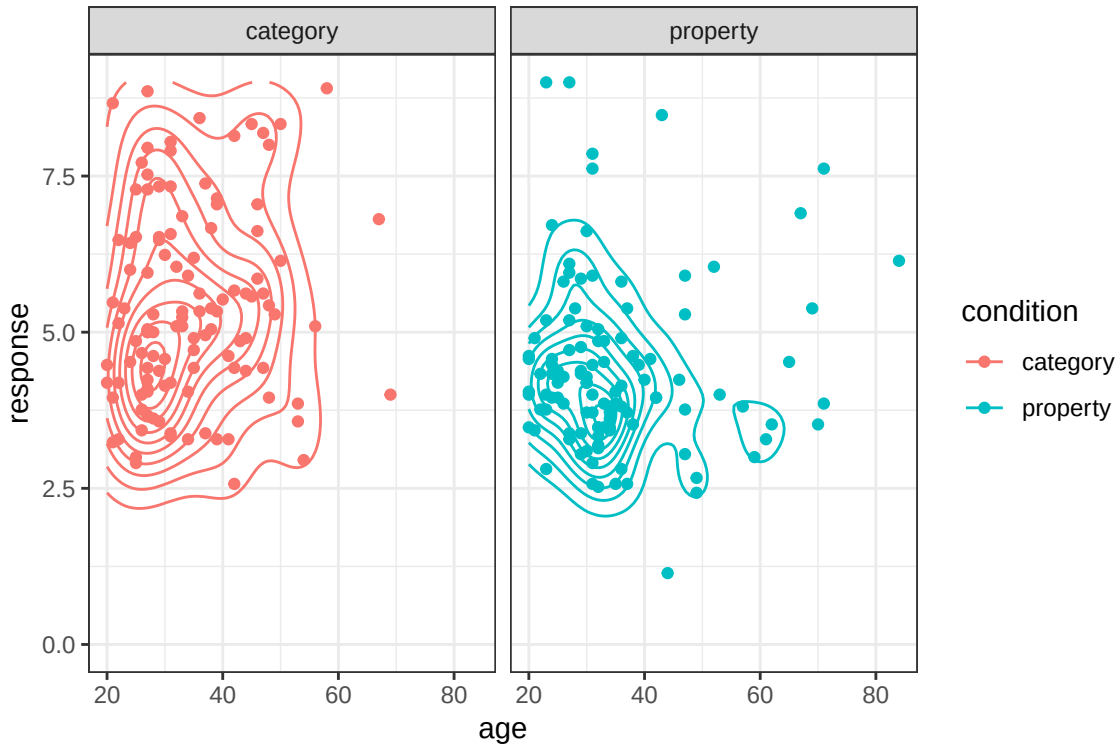


Notice that I can add `geom_line()` directly to `my_pic`. That can be extremely handy if you've worked out what the core features of your plot will be and want to play around with a few possible additions to the plot.

3.11 Other possibilities

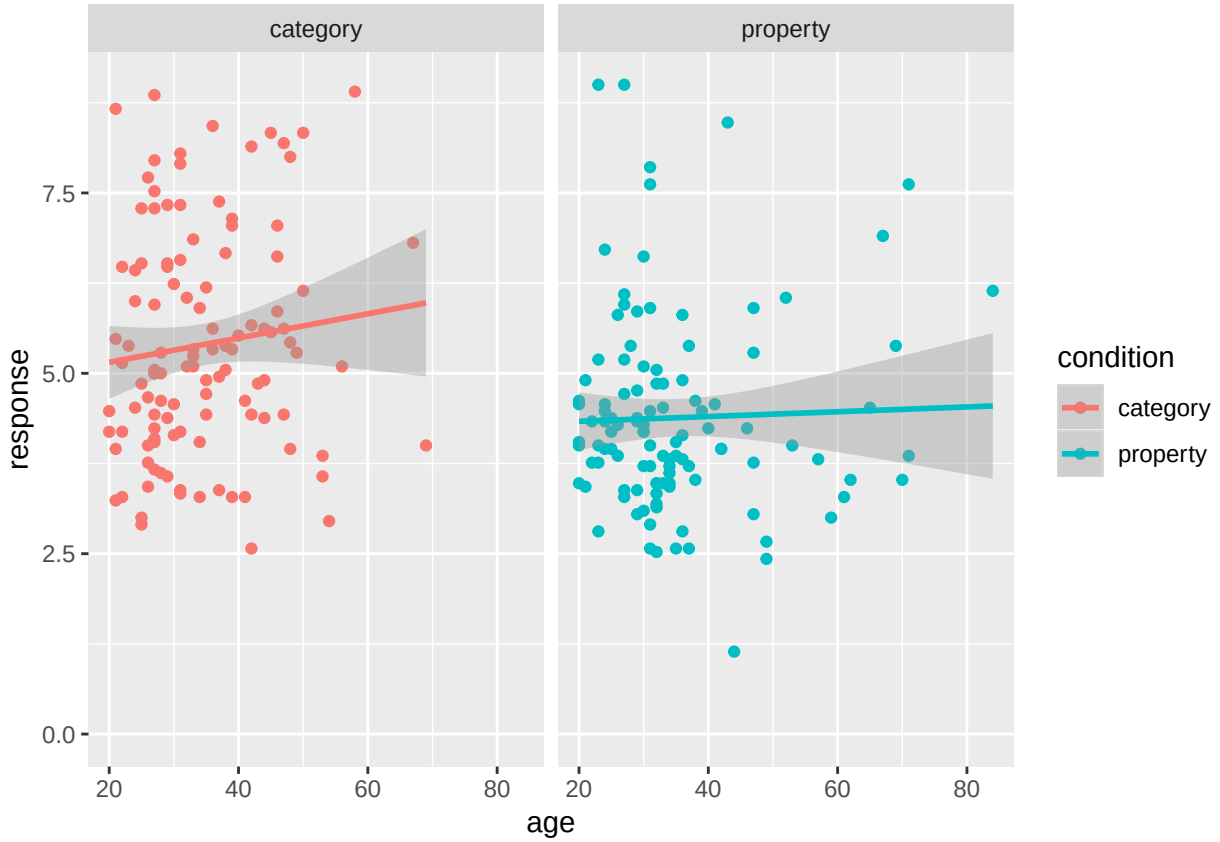
As you can see, there are quite a few things you can do with this package. However, the **ggplot2** package provides many more plotting tools than I've described so far, and it was designed to be extensible so there are many other packages that extend it in interesting ways. Even without looking at other packages, there are various neat possibilities. Just to give you a bit of a sense of this, here are a few more! If I want to add contouring to a scatter plot, I can do this using the `geom_density_2d()` function:

```
frames_small %>%
  ggplot(mapping = aes(x = age, y = response, colour = condition)) +
  geom_point() +
  theme_bw() +
  geom_density_2d() +
  facet_wrap(~condition) +
  ylim(0,9)
```



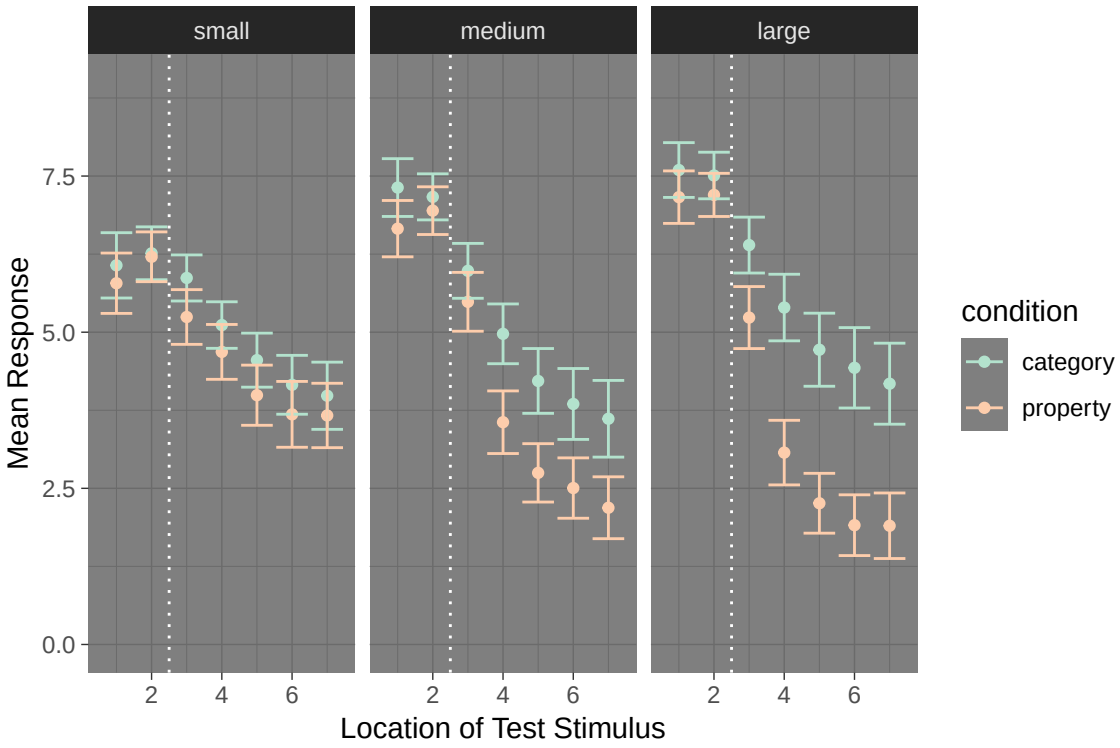
Alternatively, if I want to add regression lines to this plot I can use `geom_smooth()`. By default the `geom_smooth()` function applies a non-linear method (loess regression), which we saw previously in the prelude. This is also customisable. If I want to use a simpler linear model to provide my regression line, all I need to do is specify `method = "lm"` and I get a plot like this:

```
frames_small %>%
  ggplot(mapping = aes(x = age, y = response, colour = condition)) +
  geom_point() +
  geom_smooth(method = "lm") +
  facet_wrap(~condition) +
  ylim(0,9)
```



Similarly, plots are customisable in other ways. We can use `theme_dark()` to switch to a dark theme or `theme_bw()` to switch to a black and white scheme. We can add horizontal lines to plots with `geom_hline()`, vertical lines with `geom_vline()` and diagonal lines with `geom_abline()`. The colour palette can be adapted in arbitrary ways (e.g. with `scale_color_brewer`). We can control the axis scales with `xlim()` and `ylim()`. We can add text to a plot with `geom_text()`. Here's an example showing some of those customisation:

```
my_new_pic <- my_pic +
  theme_dark() +
  scale_color_brewer(palette = "Pastel2") +
  ylim(0, 9) +
  geom_vline(xintercept = 2.5, colour = "white", lty = "dotted") +
  xlab("Location of Test Stimulus") +
  ylab("Mean Response")

my_new_pic
```



The possibilities are limited mostly by your imagination, and by common sense. The **ggplot2** system is so rich and flexible that it's easy to go overboard with creating pretty pictures. Always remember that there's a human reader at the other end of your picture, and they're more interested in what your data visualisation can tell them about *the data*, not what it says about your R skills! :-)

3.12 Saving images

The last topic I want to cover here is how to save your plot to an image file. Suppose I want to take `my_new_plot` and save it to a PNG file that is 16cm wide and 8cm high. The `ggsave()` function allows me to do that, and the command is pretty intuitive:

```
ggsave(
  filename = "./output/frame_pic.png",
  plot = my_new_pic,
  width = 16,
  height = 8,
  units = "cm"
)
```

The function is smart enough to guess from the `filename` argument what kind of image file to create (e.g., PNG, JPG, etc), and so as long as the filename specifies a file extension that `ggsave()` can handle it will automatically create the right kind of image. The result in this case is this image file.

3.13 Further reading

- Data visualisation: A practical introduction. Kieran Healy
- ggplot2: Elegant graphics for data analysis. Hadley Wickham.

Part III
Learning from data

Part IV
Tools of the trade

A Data types in R

> He divided the universe in forty categories or classes, these being further subdivided into differences, which was then subdivided into species. He assigned to each class a monosyllable of two letters; to each difference, a consonant; to each species, a vowel. For example: `de`, which means an element; `deb`, the first of the elements, fire; `deba`, a part of the element fire, a flame. ... The words of the analytical language created by John Wilkins are not mere arbitrary symbols; each letter in them has a meaning, like those from the Holy Writ had for the Cabbalists. –Jorge Luis Borges, The Analytical Language of John Wilkins

```
library(tidyverse)
library(tidylsr)
```

A.1 Vectors

When I introduced variablesx, I showed you how we can use them to store a single number. In this section, we'll extend this idea and look at how to store multiple numbers within the one variable. In R the name for a variable that can store multiple values is a **vector**. So let's create one.

A.1.1 Character vectors

Let's return to the example we were working with in the previous section on variables. We're designing a survey, and we want to keep track of the responses that a participant has given. This time, let's imagine that we've finished running the survey and we're examining the data. Suppose we've administered the Depression, Anxiety and Stress Scale (DASS) and as a consequence every participant has scores for on the *depression*, *anxiety* and *stress* scales provided by the DASS. One thing we might want to do is create a single variable called `scale_name` that identifies the three scales. The simplest way to do this in R is to use the *combine* function, `c`.[16] To do so, all we have to do is type the values we want to store in a comma separated list, like this

```
scale_name <- c("depression","anxpiety","stress")
scale_name
```

```
## [1] "depression" "anxpiety"   "stress"
```

[16] Notice that I didn't specify any argument names here. The `c` function is one of those cases where we don't use names. We just type all the numbers, and R just dumps them all in a single variable.

To use the correct terminology here, we have a single variable here called `scale_name`: this variable is a **vector** that has three **elements**. Because the vector contains text, it is a character vector. You can use the `length` function to check the length, and the `class` function to check what kind of vector it is:

```
length(scale_name)
```

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

```
class(scale_name)
```

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

A.1.2 Numeric vectors

As you might expect, we can define numeric or logical variables in the same way. For instance, we could define the raw scores on the three DASS scales like so:

```
raw_score <- c(12, 3, 8)
raw_score
```

```
## [1] 12  3  8
```

We'll talk about logical vectors in a moment.

A.1.3 Extracting an element

If I want to extract the first element from the vector, all I have to do is refer to the relevant numerical index, using square brackets to do so. For example, to get the first element of `scale_name` I would type this

```
scale_name[1]
```

```
## [1] "depression"
```

The second element of the vector is

```
scale_name[2]
```

```
## [1] "anxpiety"
```

You get the idea.[17]

A.1.4 Extracting multiple elements

There are a few ways to extract multiple elements of a vector. The first way is to specify a vector that contains the indices of the variables that you want to keep. To extract the first two scale names:

[17] Note that the square brackets here are used to index the elements of the vector, and that this is the same notation that we see in the R output. That's not accidental: when R prints `[1] "depression"` to the screen what it's saying is that `"depression"` is the first element of the output. When the output is long enough, you'll often see other numbers at the start of each line of the output.

```
scale_name[c(1,2)]
```

```
## [1] "depression" "anxpiety"
```

Alternatively, R provides a convenient shorthand notation in which `1:2` is a vector containing the nubmers from 1 to 2, and similarly `1:10` is a vector containing the numbers from 1 to 10. So this is also the same:

```
scale_name[1:2]
```

```
## [1] "depression" "anxpiety"
```

Notice that order matters here. So if I do this

```
scale_name[c(2,1)]
```

```
## [1] "anxpiety"   "depression"
```

I get the same numbers, but in the reverse order.

A.1.5 Removing elements

Finally, when working with vectors, R allows us to use negative numbers to indicate which elements to remove. So this is yet another way of doing the same thing:

```
scale_name[-3]
```

```
## [1] "depression" "anxpiety"
```

Notice that done of this has changed the original variable. The `scale_name` itself has remained completely untouched.

```
scale_name
```

```
## [1] "depression" "anxpiety"   "stress"
```

A.1.6 Editing vectors

Sometimes you'll want to change the values stored in a vector. Imagine my surprise when a student points out that `"anxpiety"` is not in fact a real thing. I should probably fix that! One possibility would be to assign the whole vector again from the beginning, using `c`. But that's a lot of typing. Also, it's a little wasteful: why should R have to redefine the names for all three scales, when only the second one is wrong? Fortunately, we can tell R to change only the second element, using this trick:

```
scale_name[2] <- "anxiety"
scale_name
```

```
## [1] "depression" "anxiety"    "stress"
```

That's better.

Another way to edit variables in is to use the `edit` function. I won't go into that here, but if you're curious, try typing a command like this:

```
edit(scale_name)
```

A.1.7 Naming elements

One very handy thing in R is that it lets you assign meaningul *names* to the different elements in a vector. For example, the `raw_scores` vector that we introduced earlier contains the actual data from a study but when you print it out on its own

```
raw_score
```

```
## [1] 12  3  8
```

its not obvious what each of the scores corresponds to. There are several different ways of making this a little more meaningful (and we'll talk about them later) but for now I want to show one simple trick. Ideally, what we'd like to do is have R remember that the first element of the `raw_score` is the "depression" score, the second is "anxiety" and the third is "stress". We can do that like this:

```
names(raw_score) <- scale_name
```

This is a bit of an unusual looking assignment statement. Usually, whenever we use `<-` the thing on the left hand side is the variable itself (i.e., `raw_score`) but this time around the left hand side refers to the names. To see what this command has done, let's get R to print out the `raw_score` variable now:

```
raw_score
```

```
## depression    anxiety     stress
##         12          3          8
```

That's a little nicer. Element names don't just look nice, they're functional too. You can refer to the elements of a vector using their names, like so:

```
raw_score["anxiety"]
```

```
## anxiety
##       3
```

A.1.8 Vector operations

One really nice thing about vectors is that a lot of R functions and operators will work on the whole vector at once. For instance, suppose I want to normalise the raw scores from the DASS. Each scale of the DASS is constructed from 14 questions that are rated on a 0-3 scale, so the minimum possible score is 0 and the maximum is 42. Suppose I wanted to rescale the raw scores to lie on a scale from 0 to 1. I can create the `scaled_score` variable like this:

```
scaled_score <- raw_score / 42
scaled_score
```

```
## depression    anxiety     stress
## 0.28571429 0.07142857 0.19047619
```

In other words, when you divide a vector by a single number, all elements in the vector get divided. The same is true for addition, subtraction, multiplicattion and taking powers. So that's neat.

Suppose it later turned out that I'd made a mistake. I hadn't in fact administered the complete DASS, only the first page. As noted in the DASS website, it's possible to fix this mistake (sort of). First, I have to recognise that my scores are actually out of 21 not 42, so the calculation I should have done is this:

```
scaled_score <- raw_score / 21
scaled_score
```

```
## depression    anxiety     stress
##  0.5714286  0.1428571  0.3809524
```

Then, it turns out that page 1 of the full DASS is *almost* the same as the short form of the DASS, but there's a correction factor you have to apply. The depression score needs to be multiplied by 1.04645, the anxiety score by 1.02284, and stress by 0.98617

```
correction_factor <- c(1.04645, 1.02284, 0.98617)
corrected_score <- scaled_score * correction_factor
corrected_score
```

```
## depression    anxiety     stress
##  0.5979714  0.1461200  0.3756838
```

What this has done is multiply the first element of `scaled_score` by the first element of `correction_factor`, multiply the second element of `scaled_score` by the second element of `correction_factor`, and so on.

I'll talk more about calculations involving vectors later, because they come up a lot. In particular R has a thing called the *recycling rule* that is worth knowing about.[18] But that's enough detail for now.

A.1.9 Logical vectors

I mentioned earlier that we can define vectors of logical values in the same way that we can store vectors of numbers and vectors of text, again using the `c` function to combine multiple values. Logical vectors can be useful as data in their own right, but the thing that they're expecially useful for is extracting elements of another vector, which is referred to as *logical indexing*.

Here's a simple example. Suppose I decide that the stress scale is not very useful for my study, and I only want to keep the first two elements, depression and anxiety. One way to do this is to define a logical vector that indicates which values to `keep`:

```
keep <- c(TRUE, TRUE, FALSE)
keep
```

[18] The recycling rule: if two vectors are of unequal length, the values of shorter one will be "recycled". To get a feel for how this works, try setting `x <- c(1,1,1,1,1)` and `y <- c(2,7)` and then getting R to evaluate `x + y`

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

In this instance the `keep` vector indicates that it is `TRUE` that I want to retain the first two elements, and `FALSE` that I want to keep the third. So if I type this

```
corrected_score[keep]
```

```
## depression    anxiety
##  0.5979714  0.1461200
```

R prints out the corrected scores for the two variables only. As usual, note that this hasn't changed the original variable. If I print out the original vector...

```
corrected_score
```

```
## depression    anxiety     stress
##  0.5979714  0.1461200  0.3756838
```

... all three values are still there. If I *do* want to create a new variable, I need to explicitly assign the results of my previous command to a variable.

Let's suppose that I want to call the new variable `short_score`, indicating that I've only retained some of the scales. Here's how I do that:

```
short_score <- corrected_score[keep]
short_score
```

```
## depression    anxiety
##  0.5979714  0.1461200
```

A.1.10 Comment

At this point, I hope you can see why logical indexing is such a useful thing. It's a very basic, yet very powerful way to manipulate data. For intance, I might want to extract the scores of the adult participants in a study, which would probably involve a command like `scores[age > 18]`. The operation `age > 18` would return a vector of `TRUE` and `FALSE` values, and so the the full command `scores[age > 18]` would return only the `scores` for participants with `age > 18`. It does take practice to become completely comfortable using logical indexing, so it's a good idea to play around with these sorts of commands. Practice makes perfect, and it's only by practicing logical indexing that you'll perfect the art of yelling frustrated insults at your computer.

A.1.11 Exercises

- Use the combine function `c` to create a numeric vector called `age` that lists the ages of four people (e.g., 19, 34, 7 and 67)
- Use the square brackets `[]` to print out the `age` of the second person.
- Use the square brackets `[]` to print out the `age` of the second person and third persons
- Use the combine function `c` to create a character vector called `gender` that lists the gender of those four people
- Create a logical vector `adult` that indicates whether each participant was 18 or older. Instead of using `c`, try using a logical operator like `>` or `>=` to automatically create `adult` from `age`
- Test your logical indexing skills. Print out the `gender` of all the `adult` participants.

A.2 Factors

As psychological research methodology classes are at pains to point out, the data we analyse come in different kinds. Some variables are inherently *quantitative* in nature: response time (RT) for instance, has a natural interpretation in units of time. So when I defined a response time variable in the previous section, I used a numeric vector. To keep my variable names concise, I'll define the same variable again using the conventional RT abbreviation:

```
RT <- c(420, 619, 550, 521, 1003, 486, 512, 560, 495, 610)
```

A response time of 1500 milliseconds is indeed 400 milliseconds slower than a response time of 1100 milliseconds, so addition and subtraction are meaningful operations. Similarly, 1500 milliseconds is twice as long as 750 milliseconds, so multiplication and division are also meaningful. That's not the case for other kinds of data, and this is where **factors** can be useful...

A.2.1 Unordered factors

Some variables are inherently *nominal* in nature. If I recruit participants in an online experiment I might see that their place of residence falls in one of several different regions. For simplicity, let's imagine that my study is designed to sample people from one of four distinct geographical regions: the United States, India, China or the European Union, which I'll represent using the codes `"us"`, `"in"`, `"ch"` and `"eu"`. My first thought would be to represent the data using a character vector:

```
region_raw <- c("us","us","us","eu","in","eu","in","in","us","in")
```

This seems quite reasonable, but there's a problem: as it happens there is nobody from China in this sample. So if I try to construct a frequency table of these data – which I can do using the `table()` function in R – the answer I get omits China entirely:

```
table(region_raw)
```

```
## region_raw
## eu in us
##  2  4  4
```

Intuitively it feels like there should be a fourth entry here, indicating that we have 0 participants from China. R has a natural tool for representing this idea, called a **factor**. First, we'll create a new variable using the `factor()` function that contains the same information but represents it as a factor:

```
region <- factor(region_raw)
region
```

```
##  [1] us us us eu in eu in in us in
## Levels: eu in us
```

This looks a much the same, and not surprisingly R still doesn't know anything about the possibility of participants from China. However, notice that the bottom of the output lists the *levels* of the factor. The levels of a factor specify the set of values that variable could have taken. By default, `factor()` tries to guess the levels using the raw data, but we can override that manually, like this:

```
region <- factor(region_raw, levels = c("ch","eu","in","us"))
region
```

```
##  [1] us us us eu in eu in in us in
## Levels: ch eu in us
```

Now when we tabulate the `region` variable, we obtain the right answer:

```
table(region)
```

```
## region
## ch eu in us
##  0  2  4  4
```

Much nicer.

A.2.2 Ordered factors

There are two different types of factor in R. Until now we have been discussing *unordered* factors, in which the categories are purely nominal and there is no notion that the categories are arranged in any particular order. However, many psychologically important variables are inherently *ordinal*. Questionnaire responses often take this form, where participants might be asked to endorse a proposition using verbal categories such as *"strongly agree", "agree", "neutral", "disagree"* and *"strongly disagree"*. The five response categories can't be given any sensible numerical values[19] but they can be ordered in a sensible fashion. In this situation we may want to represent the responses as an **ordered factor**.

To give you a sense of how these work in R, suppose we've been unfortunate enough to be given a data set that encodes ordinal responses numerically. In my experience that happens quite often. Let's suppose the original survey asked people how strongly they supported a polticial policy. Here we have a variable consisting of Likert scale data, where (let's suppose) in the original questionnaire 1 = *"strongly agree"* and 7 = *"strongly disagree"*,

```
support_raw <- c(1, 7, 3, 4, 4, 4, 2, 6, 5, 5)
```

We can convert this to an ordered factor by specifying `ordered = TRUE` when we call the `factor()` function, like so:

```
support <- factor(
  x = support_raw,            # the raw data
  levels = c(7,6,5,4,3,2,1),  # strongest agreement is 1, weakest is 7
  ordered = TRUE              # and it's ordered
)
support
```

```
##  [1] 1 7 3 4 4 4 2 6 5 5
## Levels: 7 < 6 < 5 < 4 < 3 < 2 < 1
```

[19] For example, suppose we decide to assign them the numbers 1 to 5. If we take these numbers literally, we're implicitly assuming that is the psycholigical difference between *"strongly agree"* and *"neutral"* is the same in "size" as the difference between *"agree"* and *"disagree"*. In many situations this is probably okay to a first approximation, but in general it feels very strange.

Notice that when we print out the ordered factor, R explicitly tells us what order the levels come in.

Because I wanted to order my levels in terms of *increasing* strength of endorsement, and because a response of 1 corresponded to the strongest agreement and 7 to the strongest disagreement, it was important that I tell R to encode 7 as the lowest value and 1 as the largest. Always check this when creating an ordered factor: it's very easy to accidentally encode your data with the levels reversed if you're not paying attention. In any case, note that we can (and should) attach meaningful names to these factor levels by using the `levels` function, like this:

```
levels(support) <- c(
  "strong disagree", "disagree", "weak disagree",
  "neutral", "weak agree", "agree", "strong agree"
)
support
```

```
##  [1] strong agree    strong disagree weak agree      neutral
##  [5] neutral         neutral         agree           disagree
##  [9] weak disagree   weak disagree
## 7 Levels: strong disagree < disagree < weak disagree < ... < strong agree
```

A nice thing about ordered factors is that some analyses in R automatically treat ordered factors differently to unordered factors, and generally in a way that is more appropriate for ordinal data.

A.3 Data frames / tibbles

We now have three variables that we might plausibly have encountered as the result of some study, `region`, `support` and `RT`.[20] At the moment, R has no understanding of how these variables are related to each other. Quite likely they're ordered the same way, so that the data stored in `region[1]`, `support[1]` and `RT[1]` all come from the same person. That would be sensible, but R is a robot and does not possess common sense. To help a poor little robot out (and to make our own lives easier), it's nice to organise these three variable into a tabular format. We saw this in the last section, in which the AFL data was presented as a table. This is where **data frames** – and the tidyverse analog **tibbles** – are very useful.

A.3.1 Making a data frame

So how do we create a data frame (or tibble)? One way we've already seen: if we import our data from a CSV file, R will create one for you. A second method is to create a data frame directly from some existing variables using the `data.frame` function. In real world data analysis this method is less common, but it's very helpful for understanding what a data frame actually is, so that's what we'll do in this section.

Manually constructing a data frame is simple. All you have to do when calling `data.frame` is type a list of variables that you want to include in the data frame. If I want to store the variables from my experiment in a data frame called `dat` I can do so like this:

```
dat <- data.frame(region, support, RT)
dat
```

```
##    region         support   RT
## 1      us    strong agree  420
## 2      us strong disagree  619
```

[20] Admittedly it would be a strange study that produced only these three variables, but I hope you'll forgive the lack of realism on this point.

```
## 3      us      weak agree  550
## 4      eu         neutral  521
## 5      in         neutral 1003
## 6      eu         neutral  486
## 7      in           agree  512
## 8      in        disagree  560
## 9      us   weak disagree  495
## 10     in   weak disagree  610
```

Note that `dat` is a self-contained variable. Once created, it no longer depends on the variables from which it was constructed. If we make changes to the original `RT` variable, these will not influence the copy in `dat` (or vice versa). So for the sake of my sanity I'm going to remove all the originals:

```
rm(region_raw, region, support_raw, support, RT)
show_environment()
```

```
## # A tibble: 8 x 3
##   variable          class      size
##   <chr>             <chr>      <chr>
## 1 corrected_score   numeric    length: 3
## 2 correction_factor numeric    length: 3
## 3 dat               data.frame rectangular: 10 by 3
## 4 keep              logical    length: 3
## 5 raw_score         numeric    length: 3
## 6 scale_name        character  length: 3
## 7 scaled_score      numeric    length: 3
## 8 short_score       numeric    length: 2
```

As you can see, our workspace has only a single variable, a data frame called `dat`. In this example I constructed the data frame manually so that you can see *how* a data frame is built from a set of variables, but in most real life situations you'd probably load your data frame directly from a CSV file or similar.

A.3.2 Making a tibble

Constructing a tibble from raw variables is essentially the same as constructing a data frame, and the function we use to do this is `tibble`. If I hadn't deleted all the raw variables in the previous section, this command would work:

```
tib <- tibble(region, support, RT)
```

Alas they are gone, and I will have to try a different method. Fortunately, I can **coerce** my existing data frame `dat` into a tibble using the `as_tibble()` function, and use it to create a tibble called `tib`. I'm very imaginative :-)

```
tib <- as_tibble(dat)
tib
```

```
## # A tibble: 10 x 3
##    region support            RT
##    <fct>  <ord>           <dbl>
##  1 us     strong agree      420
```

```
##  2 us     strong disagree   619
##  3 us     weak agree        550
##  4 eu     neutral           521
##  5 in     neutral          1003
##  6 eu     neutral           486
##  7 in     agree             512
##  8 in     disagree          560
##  9 us     weak disagree     495
## 10 in     weak disagree     610
```

Coercion is an important R concept, and one that we'll talk about again at the end of this section. In the meantime, there are some nice things to note about the output when we print `tib`. It states that the variable is a tibble with 10 rows and 3 columns. Underneath the variable names it tells you what type of data they store: `region` is a factor (`<fct>`), `support` is an ordered factor (`<ord>`) and RT is numeric (`<dbl>`, short for "double")[21].

A.3.3 Tibbles are data frames

Under the hood, tibbles are essentially the same thing as data frames and are designed to behave the same way. In fact, if we use the `class()` function to see what R thinks `tib` really is...

```
class(tib)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

... it agrees that in addition to being a tibble, `tib` is also a data frame! We can check this more directly using the `is.data.frame()` function:

```
is.data.frame(tib)
```

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

That being said, there are one or two differences between tibbles and pure data frames. For the most part, my impression has been that whenever they differ, the behaviour of tibbles tends to be more intuitive. With this in mind, although I'll tend to use the terms "data frame" and "tibble" interchangeably in these notes, for the rest of these notes I'm going to work with tibbles like `tib` rather than pure data frames like `dat`.

A.3.4 Using the `$` operator

At this point our workspace contains a data frame called `dat`, a tibble called `tib`, but no longer contains the original variables. That's okay because the tibble (data frame) is acting as a container that keeps them in a nice tidy rectangular shape. Conceptually this is very nice, but now we have a practical question ... how do we get information out again? There are two qualitatively different ways to do this,[22] reflecting two different ways to think about your data:

- Your data set is a *list of variables* (... use `$`)
- Your data set is a *table of values* (... use `[ ]`)

[21] The origin of the term "double" comes from double precision floating point the format in which numeric variables are represented internally

[22] Technically this is a lie: there are many more ways to do this, but let's not make this any more difficult than it needs to be, yeah?

Both perspectives are valid, and R allows you to work with your data both ways.

To start with, let's think of `tib` as a list of variables. This was the perspective we took when constructing `dat` in the first place: we took three different vectors (`region`, `support`, `RT`) and bound them together into a data frame, which we later coerced into the tibble `tib`. From this perspective, what we want is an operator that will extract one of those variables for us. This is the role plaed by `$`. If I want to refer to the `region` variable contained *within* the `tib` tibble, I would use this command:

```
tib$region
```

```
##  [1] us us us eu in eu in in us in
## Levels: ch eu in us
```

As you can see, the output looks exactly the same as it did for the original variable: `tib$region` *is a vector* (an unordered factor in this case), and we can refer to an element of that vector in the same way we normally would:

```
tib$region[1]
```

```
## [1] us
## Levels: ch eu in us
```

Conceptually, the metaphor here is `dataset$variable[value]`. The table below illustrates this by showing what type of output you get with different commands:

data frame command	data frame output	tibble command	tibble output
dat	data frame	tib	tibble
dat$RT	vector	tib$RT	vector
dat$RT[1]	element	tib$RT[1]	element

As you can see, the `$` operator works the same way for pure data frames as for tibbles. This is not quite the case for when using square brackets `[ ]`, as the next section demonstrates...

A.3.5 Using square brackets

The second way to think about a tibble is to treat it as a fancy table. There is something appealing about this, because it emphasises the fact that the data set has a *case by variable* structure:

```
tib
```

```
## # A tibble: 10 x 3
##    region support            RT
##    <fct>  <ord>           <dbl>
##  1 us     strong agree      420
##  2 us     strong disagree   619
##  3 us     weak agree        550
##  4 eu     neutral           521
##  5 in     neutral          1003
##  6 eu     neutral           486
##  7 in     agree             512
##  8 in     disagree          560
##  9 us     weak disagree     495
## 10 in     weak disagree     610
```

In this structure each row is a person, and each column is a variable. The square bracket notation allows you to refer to entries in the data set by their row and column number (or name). As such, the reference looks like this:

```
dataset[row,column]
```

R allows you to select multiple rows and columnns. For instance if you set `row` to be `1:3` then R will return the first three cases. Here is an example where we select the first three rows and the first two columns:

```
tib[1:3, 1:2]
```

```
## # A tibble: 3 x 2
##   region support
##   <fct>  <ord>
## 1 us     strong agree
## 2 us     strong disagree
## 3 us     weak agree
```

If we omit values for the rows (or columms) *while keeping the comma* then R will assume you want all rows (or columnns). So this returns every row in `tib` but only the first two columns:

```
tib[, 1:2]
```

```
## # A tibble: 10 x 2
##    region support
##    <fct>  <ord>
##  1 us     strong agree
##  2 us     strong disagree
##  3 us     weak agree
##  4 eu     neutral
##  5 in     neutral
##  6 eu     neutral
##  7 in     agree
##  8 in     disagree
##  9 us     weak disagree
## 10 in     weak disagree
```

An important thing to recognise here is that – for tibbles – the metaphor underpinning the square bracket system is that your data have a rectangular shape that is imposed by the fact that your variable is a tibble, and no matter what you do with the square brackets the result will **always remain a tibble**. If I select just one row...

```
tib[5,]
```

```
## # A tibble: 1 x 3
##   region support    RT
##   <fct>  <ord>   <dbl>
## 1 in     neutral  1003
```

the result is a tibble. If I select just one column...

```
tib[,3]
```

```
## # A tibble: 10 x 1
##       RT
##    <dbl>
##  1   420
##  2   619
##  3   550
##  4   521
##  5  1003
##  6   486
##  7   512
##  8   560
##  9   495
## 10   610
```

the result is a tibble. Even if I select a single value...

```
tib[5,3]
```

```
## # A tibble: 1 x 1
##      RT
##   <dbl>
## 1  1003
```

the result is a tibble. For the square bracket system the rule is very simple: **tibbles stay tibbles**

Annoyingly, this is not the case for a pure data frame like `dat`. For a pure data frame, any time it is possible for R to treat the result as something else, it does: if I were to use the same commands for the data frame `dat`, the results would be different in some cases. This has caused my students (and myself) no end of frustration over the years because everyone forgets about this particular property of data frames and stuff breaks. In the original version of these notes published in *Learning Statistics with R* I had a length explanation of this behaviour. Nowadays I just encourage people to use tibbles instead. For what it's worth, if you are working with pure data frames, here's a summary of what to expect:

data frame command	data frame output	tibble command	tibble output
dat[1,1]	element	tib[1,1]	tibble
dat[1,]	data frame	tib[1,]	tibble
dat[,1]	vector	tib[,1]	tibble
dat[2:3,]	data frame	tib[2:3,]	tibble
dat[,2:3]	data frame	tib[,2:3]	tibble

I **like** tibbles.[23]

A.4 Matrices

Data frames and tibbles are mostly used to describe data that take the form of a *case by variable* structure: each row is a case (e.g., a participant) and each column is a variable (e.g., measurement). Case by variable structures are fundamentally asymmetric because the rows and columns have qualitatively different meaning.

[23] Just FYI: you can make a pure data frame behave like a tibble. If you use `dat[,1,drop=FALSE]` you can suppress this weird thing and make R return a one-column data frame instead of a vector, but that command is so unbearably cumbersome that everyone forgets to use it.

Two participants who provide data will always provide data in the same format (if they don't then you can't organise the data this way), but two variables can be different in many different ways: one column might be numeric, another is a factor, yet another might contains dates. Many psychological data sets have this characteristic. Others do not, so it is worth talking about a few other data structures that arise quite frequently!

Much like a data frame, a **matrix** is basically a big rectangular table of data, and there are similarities between the two. However, matrices treat columns and rows in the same fashion, and as a consequence every entry in a matrix has to be of the same type (e.g. all numeric, all character, etc). Let's create a matrix using the *row bind* function, `rbind`, which combines multiple vectors in a row-wise fashion:

```
row1 <- c(2, 3, 1)          # create data for row 1
row2 <- c(5, 6, 7)          # create data for row 2
mattie <- rbind(row1, row2) # row bind them into a matrix
mattie
```

```
##      [,1] [,2] [,3]
## row1    2    3    1
## row2    5    6    7
```

Notice that when we bound the two vectors together R turned the names of the original variables into row names.[24] To keep things fair, let's add some exciting column names as well:

```
colnames(mattie) <- c("col1", "col2", "col3")
mattie
```

```
##      col1 col2 col3
## row1    2    3    1
## row2    5    6    7
```

A.4.1 Matrix indexing

You can use square brackets to subset a matrix in much the same way that you can for data frames, again specifying a row index and then a column index. For instance, `mattie[2,3]` pulls out the entry in the 2nd row and 3rd column of the matrix (i.e., 7), whereas `mattie[2,]` pulls out the entire 2nd row, and `mattie[,3]` pulls out the entire 3rd column. However, it's worth noting that when you pull out a column, R will print the results horizontally, not vertically.[25]

[24] We could delete these if we wanted by typing `rownames(mattie)<-NULL`, but I generally prefer having meaningful names attached to my variables, so I'll keep them.

[25] The reason for this relates to how matrices are implemented. The original matrix `mattie` is treated as a *two-dimensional* object, containing two rows and three columns. However, whenever you pull out a single row or a single column, the result is considered to be a vector, which has a *length* but doesn't have dimensions. Unless you explictly coerce the vector into a matrix, R doesn't really distinguish between row vectors and column vectors. This has implications for how matrix algebra is implemented in R (which I'll admit I initially found odd). When multiplying a matrix by a vector using the `%*%` operator, R will attempt to interpret the vector as either a row vector or column vector, depending on whichever one makes the multiplication work. That is, suppose $\mathbf{M}$ is 2×3 matrix, and v is a 1×3 row vector. Mathematically the matrix multiplication $\mathbf{M}v$ doesn't make sense since the dimensions don't conform, but you can multiply by the corresponding column vector, $\mathbf{M}v^T$. So, if I set `v <- mattie[2,]`, the object that R returns doesn't technically have any *dimensions* only a *length*. So even though `v` was behaving like a row vector when it was part of `mattie`, R has forgotten that completely and only knows that `v` is length three. So when I try to calculate `mattie %*% v`, which you'd think would fail because I didn't transpose `v`, it actually works. In this context R treated `v` as if it were a column vector for the purposes of matrix multiplication. Note that if both objects are vectors, this leads to ambiguity since vv^T (inner product) and v^Tv (outer product) yield different answers. In this situation `v %*% v` returns the inner product. You can obtain the outer product with `outer(v,v)`. The help documentation may come in handy!

```
mattie[2,]
```

```
## col1 col2 col3
##    5    6    7
```

```
mattie[,3]
```

```
## row1 row2
##    1    7
```

This can be a little confusing for novice users: because it is no longer a two dimensional object R treats the output as a regular vector.[26]

A.4.2 Matrices vs data frames

As mentioned above difference between a data frame and a matrix is that, at a fundamental level, a matrix really is just *one* variable: it just happens that this one variable is formatted into rows and columns. If you want a matrix of numeric data, every single element in the matrix *must* be a number. If you want a matrix of character strings, every single element in the matrix *must* be a character string. If you try to mix data of different types together, then R will either complain or try to transform the matrix into something unexpected. To give you a sense of this, let's do something silly and convert one element of `mattie` from the number `5` to the character string `"five"`...

```
mattie[2,2] <- "five"
mattie
```

```
##      col1 col2   col3
## row1 "2"  "3"    "1"
## row2 "5"  "five" "7"
```

Oh no I broke `mattie` – she's all text now! I'm so sorry `mattie`, I still love you.

A.4.3 Other ways to make a matrix

When I created `mattie` I used the `rbind` command. Not surprisingly there is also a `cbind` command that combines vectors column-wise rather than row-wise. There is also a `matrix` command that you can use to specify a matrix directly:

```
matrix(
  data = 1:12, # the values to include in the matrix
  nrow = 3,    # number of rows
  ncol = 4     # number of columns
)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

[26] You can suppress this behaviour by using a command like `mattie[,3,drop=FALSE]`. It's unpleasant though. Also be warned: data frames do this too when you select one column using square brackets. Tibbles don't. One of the reasons I like tibbles actually.

The result is a 3×4 matrix of the numbers 1 to 12, listed column-wise.[27] If you need to create a matrix row-wise, you can specify `byrow = TRUE` when calling `matrix()`.

A.5 Arrays

When doing data analysis, we often have reasons to want to use higher dimensional tables (e.g., sometimes you need to cross-tabulate three variables against each other). You can't do this with matrices, but you can do it with arrays. An **array** is just like a matrix, except it can have more than two dimensions if you need it to. In fact, as far as R is concerned a matrix is just a special kind of array, in much the same way that a data frame is a special kind of list. I don't want to talk about arrays too much, but I will very briefly show you an example of what a three dimensional array looks like.

```
arr <- array(
  data = 1:24,
  dim = c(3,4,2)
  )
arr
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

Of course, calling an array `arr` just makes me think of pirates.

A.5.1 Array indexing

Array indexing is a straightforward generalisation of matrix indexing, so the same logic applies. Since `arr` is a three-dimensional $3 \times 4 \times 2$ array, we need three indices to specify an element:

```
arr[2,3,1]
```

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

Omitted indices have the same meaning that they have for matrices, so `arr[,2]` is a two dimensional slice through the array, and as such R recognises it as a matrix even though the full three dimensional array `arr` does not count as one.[28] Here's what `arr[,2]` returns:

[27] I won't go into details, but note that you can refer the elements of a matrix by specifying only a single index. For a 3×4 matrix `M`, `M[2,2]` and `M[5]` refer to the same cell. This method of indexing assumes column-wise ordering regardless of whether the matrix `M` was originally created in column-wise or row-wise fashion.

[28] To see this, compare `is.matrix(arr)` to `is.matrix(arr[,2])`

```
arr[,,2]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

A.5.2 Array names

As with other data structures, arrays can have names for specific elements. In fact, we can assign names to each of the *dimensions* too. For instance, suppose we have another array – that we affectionately call `cubie` – it is also a three dimensional array, but that has the shape that it does because it represents a (3 genders) × (4 seasons) × (2 times) structure. We could specify the dimension names for `cubie` like this:

```
cubie <- array(
  data = 1:24,
  dim = c(3,4,2),
  dimnames = list(
    "genders" = c("male", "female", "nonbinary"),
    "seasons" = c("summer", "autumn", "winter", "spring"),
    "times" = c("day", "night")
    )
  )
cubie
```

```
## , , times = day
##
##            seasons
## genders     summer autumn winter spring
##   male           1      4      7     10
##   female         2      5      8     11
##   nonbinary      3      6      9     12
##
## , , times = night
##
##            seasons
## genders     summer autumn winter spring
##   male          13     16     19     22
##   female        14     17     20     23
##   nonbinary     15     18     21     24
```

I find the output for `cubie` easier to read than the one for `arr` – it's usually a good idea to label your arrays! Plus, it makes it a little easier to extract information from them too, since you can refer to elements by names. So if I just wanted to take a slice through the array corresponding to the `"nonbinary"` values, I could do this:

```
cubie["nonbinary",,]
```

```
##         times
## seasons  day night
##   summer   3    15
```

```
##   autumn   6    18
##   winter   9    21
##   spring  12    24
```

A.6 Lists

The next kind of data I want to mention are lists. Lists are an extremely fundamental data structure in R, and as you start making the transition from a novice to a savvy R user you will use lists all the time. Most of the advanced data structures in R are built from lists (e.g., data frames are actually a specific type of list), so it's useful to have a basic understanding of them.

Okay, so what is a list, exactly? Like data frames, lists are just "collections of variables." However, unlike data frames – which are basically supposed to look like a nice "rectangular" table of data – there are no constraints on what kinds of variables we include, and no requirement that the variables have any particular relationship to one another. In order to understand what this actually means, the best thing to do is create a list, which we can do using the `list` function. If I type this as my command:

```
starks <- list(
  parents = c("Eddard", "Catelyn"),
  children = c("Robb", "Jon", "Sansa", "Arya", "Brandon", "Rickon"),
  alive = 8
)
```

I create a list `starks` that contains a list of the various characters that belong to House Stark in George R. R. Martin's *A Song of Ice and Fire* novels. Because Martin does seem to enjoy killing off characters, the list starts out by indicating that all eight are currently alive (at the start of the books obviously!) and we can update it if need be. When a character dies, I might do this:

```
starks$alive <- starks$alive - 1
starks
```

```
## $parents
## [1] "Eddard"  "Catelyn"
##
## $children
## [1] "Robb"    "Jon"     "Sansa"   "Arya"    "Brandon" "Rickon"
##
## $alive
## [1] 7
```

I can delete whole variables from the list if I want. For instance, I might just give up on the parents entirely:

```
starks$parents <- NULL
starks
```

```
## $children
## [1] "Robb"    "Jon"     "Sansa"   "Arya"    "Brandon" "Rickon"
##
## $alive
## [1] 7
```

You get the idea, I hope. The key thing with lists is that they're flexible. You can construct a list to map onto all kinds of data structures and do cool things with them. At a fundamental level, many of the more advanced data structures in R are just fancy lists.

A.6.1 Indexing lists

In the example above we used `$` to extract named elements from a list, in the same fashion that we would do for a data frame or tibble. It is also possible to index a list using square brackets, though it takes a little effort to get used to. The elements of a list can be extracted using single brackets (e.g., `starks[1]`) or double brackets (e.g., `starks[[1]]`). To see the difference between the two, notice that the single bracket version returns a list *containing* only a single vector

```
starks[1]
```

```
## $children
## [1] "Robb"    "Jon"     "Sansa"   "Arya"    "Brandon" "Rickon"
```

This output is a list that contains one vector `starks$children`. In contrast, the double bracketed version returns the `children` vector itself:

```
starks[[1]]
```

```
## [1] "Robb"    "Jon"     "Sansa"   "Arya"    "Brandon" "Rickon"
```

If this seems complicated and annoying... well, yes. Yes it is!

I find it helps me to think of the list as a container. When we use single brackets, the result is still inside its container; when we use double brackets we remove it from the container. This intuition is illustrated nicely in the image below, tweeted by Hadley Wickham:

In this example `x` is a container (list) containing many pepper sachets (elements). When we type `x[1]` we keep only one of the sachets of pepper, but it's still inside the container. When we type `x[[1]]` we take the sachet out of the container.

The final panel highlights how lists can become more complicated.^[Speaking of complications... Under the hood, data frames and tibbles are secretly lists, so you can use the list indexing methods for them and so, for example, `dat[[3]]` is the same as `dat$RT`. Probably best not to worry too much about that detail right now! Lists are just fancy containers, and there's no reason why lists can't contain other lists. In the pepper shaker scenario, if each sachet is itself a list, we would need to type `x[[1]][[1]]` to extract the tasty, tasty pepper!

A.7 Dates

Dates (and time) are very annoying types of data. To a first approximation we can say that there are 365 days in a year, 24 hours in a day, 60 minutes in an hour and 60 seconds in a minute, but that's not quite correct. The length of the solar day is not exactly 24 hours, and the length of solar year is not exactly 365 days, so we have a complicated system of corrections that have to be made to keep the time and date system working. On top of that, the measurement of time is usually taken relative to a local time zone, and most (but not all) time zones have both a standard time and a daylight savings time, though the date at which the switch occurs is not at all standardised. So, as a form of data, times and dates are just awful to work with. Unfortunately, they're also important. Sometimes it's possible to avoid having to use any complicated system for dealing with times and dates. Often you just want to know what year something happened in, so you can just use numeric data: in quite a lot of situations something as simple as declaring that `this_year` is 2019, and it works just fine. If you can get away with that for your application, this is probably the best thing to do. However, sometimes you really do need to know the actual date. Or, even worse, the actual time. In this section, I'll very briefly introduce you to the basics of how R deals with date and time data. As with a lot of things in this chapter, I won't go into details: the goal here is to show you the basics of what

you need to do if you ever encounter this kind of data in real life. And then we'll all agree never to speak of it again.

To start with, let's talk about the date. As it happens, modern operating systems are very good at keeping track of the time and date, and can even handle all those annoying timezone issues and daylight savings pretty well. So R takes the quite sensible view that it can just ask the operating system what the date is. We can pull the date using the `Sys.Date` function:

```
today <- Sys.Date()  # ask the operating system for the date
print(today)         # display the date
```

```
## [1] "2019-07-14"
```

Okay, that seems straightforward. But, it does rather look like today is just a character string, doesn't it? That would be a problem, because dates really do have a quasi-numeric character to them, and it would be nice to be able to do basic addition and subtraction with them. Well, fear not. If you type in `class(today)`, R will tell you that the `today` variable is a `"Date"` object. What this means is that, hidden underneath this text string, R has a numeric representation.[29] What that means is that you can in fact add and subtract days. For instance, if we add `1` to `today`, R will print out the date for tomorrow:

```
today + 1
```

```
## [1] "2019-07-15"
```

Let's see what happens when we add 365 days:

```
today + 365
```

```
## [1] "2020-07-13"
```

R provides a number of functions for working with dates, but I don't want to talk about them in any detail, other than to say that the **lubridate** package (part of the tidyverse) makes things a lot easier than they used to be. A little while back I wrote a blog post about lubridate and may fold it into these notes one day.

A.8 Coercion

Sometimes you want to change the variable class. Sometimes when you import data from files, it can come to you in the wrong format: numbers sometimes get imported as text, dates usually get imported as text, and many other possibilities besides. Sometimes you might want to convert a data frame to a tibble or vice versa. Changing the variable in this way is called **coercion**, and the functions to coerce variables are usually given names like `as.numeric()`, `as.factor()`, `as_tibble()` and so on. We've seen some explicit examples in this chapter:

- Coercing a data frame to a tibble
- Coercing a character vector to a factor

There are many other possibilities. A common situation requiring coercion arises when you have been given a variable `x` that is *supposed* to be representing a number, but the data file that you've been given has encoded it as text.

[29] Date objects are coded internally as the number of days that have passed since January 1, 1970.

```
x <- c("15","19")  # the variable
class(x)           # what class is it?
```

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

Obviously, if I want to do mathematical calculations using `x` in its current state R wil get very sad. It thinks `x` is text and it won't allow me to do mathematics with text! To coerce `x` from "character" to "numeric", we use the `as.numeric` function:

```
x <- as.numeric(x)  # coerce the variable
class(x)            # what class is it?
```

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

```
x + 1               # hey, addition works!
```

```
## [1] 16 20
```

Not surprisingly, we can also convert it back again if we need to. The function that we use to do this is the `as.character` function:

```
x <- as.character(x)   # coerce back to text
class(x)               # check the class
```

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

There are of course some limitations: you can't coerce `"hello world"` into a number because there isn't a number that corresponds to it. If you try, R metaphorically shrugs its shoulders and declares it to be missing:

```
x <- c("51", "hello world")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] 51 NA
```

Makes sense I suppose!

Another case worth talking about is how R handles coercion with logical variables. Coercing text to logical data using `as.logical()` is mostly intuitive. The strings `"T"`, `"TRUE"`, `"True"` and `"true"` all convert to `TRUE`, whereas `"F"`, `"FALSE"`, `"False"`, and `"false"` all become `FALSE`. All other strings convert to `NA`. When coercing from logical to test using `as.character`, `TRUE` converts to `"TRUE"` and `FALSE` converts to `"FALSE"`.

Converting numeric values to logical data – again using `as.logical` – is similarly straightforward. Following the standard convention in the study of Boolean logic `0` coerces to `FALSE`. Everything else is `TRUE`. When coercing logical to numeric, `FALSE` converts to `0` and `TRUE` converts to `1`.

B Programming in R

When I introduced the idea of working with scripts I said that R starts at the top of the file and runs straight through to the end of the file. That was a tiny bit of a lie. It is true that unless you insert commands to explicitly alter how the script runs, that is what will happen. However, you actually have quite a lot of flexibility in this respect. Depending on how you write the script, you can have R repeat several commands, or skip over different commands, and so on. This topic is referred to as **flow control**.

B.1 If/else

One kind of flow control that programming languages provide is the ability to evaluate **conditional statements**. Unlike loops, which can repeat over and over again, a conditional statement only executes once, but it can switch between different possible commands depending on a condition that is specified by the programmer. The power of these commands is that they allow the program itself to make choices, and in particular, to make different choices depending on the context in which the program is run. The most prominent of example of a conditional statement is the `if` statement, and the accompanying `else` statement.[30]

The basic format of an `if` statement in R is as follows:

```
if ( CONDITION ) {
  STATEMENT1
  STATEMENT2
  ETC
}
```

And the execution of the statement is pretty straightforward. If the condition is `TRUE`, then R will execute the statements contained in the curly braces. If the condition is `FALSE`, then it does not. So the way R processes an `if` statement is illustrated by this schematic:

If you want to, you can extend the `if` statement to include an `else` statement as well, leading to the following syntax:

```
if ( CONDITION ) {
  STATEMENT1
  STATEMENT2
  ETC
} else {
  STATEMENT3
  STATEMENT4
  ETC
}
```

As you'd expect, the interpretation of this version is similar. If the condition is `TRUE`, then the contents of the first block of code (i.e., *statement1*, *statement2*, etc) are executed; but if it is `FALSE`, then the contents of the second block of code (i.e., *statement3*, *statement4*, etc) are executed instead. So the schematic illustration of an if-else construction looks like this:

In other words, when we use an if-else pair, we can define different behaviour for our script for both cases.

[30] There are other ways of making conditional statements in R. In particular, the `ifelse` function and the `switch` functions can be very useful in different contexts.

B.1.1 Example 1:

To give you a feel for how you can use `if` and `else`, the example that I'll show you is a script (feelings.R) that prints out a different message depending the day of the week. Here's the script:

```
if(today == "Monday") {
  print("I don't like Mondays")
} else {
  print("I'm a happy little automaton")
}
```

So let's set the value of `today` to `Monday` and source the script:

```
today <- "Monday"
source("./scripts/feelings.R")
```

```
## [1] "I don't like Mondays"
```

That's very sad. However, tomorrow should be better:

```
today <- "Tuesday"
source("./scripts/feelings.R")
```

```
## [1] "I'm a happy little automaton"
```

B.1.2 Example 2:

One useful feature of `if` and `else` is that you can chain several of them together to switch between several different possibilities. For example, the more_feelings.R script contains this code:

```
if(today == "Monday") {
  print("I don't like Mondays")

} else if(today == "Tuesday") {
  print("I'm a happy little automaton")

} else if(today == "Wednesday") {
  print("Wednesday is beige")

} else {
  print("eh, I have no feelings")
}
```

B.1.3 Exercises

- Write your own version of the "feelings" script that expresses your opinions about summer, winter, autumn and spring. Test your script out.
- Expand your script so that it loops over vector of four `seasons` and prints out your feelings about each of them

The solutions for these exercises are here.

B.2 Loops

The second kind of flow control that I want to talk about is a **loop**. The idea is simple: a loop is a block of code (i.e., a sequence of commands) that R will execute over and over again until some *termination criterion* is met. To illustrate the idea, here's a schematic picture showing the difference between what R does with a script that contains a loop and one that doesn't:

Looping is a very powerful idea, because it allows you to automate repetitive tasks. Much like my children, R will execute an continuous cycle of *"are we there yet? are we there yet? are we there yet?"* checks against the termination criterion, and it will keep going forever until it is finally *there* and can break out of the loop. Rather unlike my children, however, I find that this behaviour is actually helpful.

There are several different ways to construct a loop in R. There are two methods I'll talk about here, one using the `while` command and another using the `for` command.

B.2.1 The `while` loop

A `while` loop is a simple thing. The basic format of the loop looks like this:

```
while ( CONDITION ) {
  STATEMENT1
  STATEMENT2
  ETC
}
```

The code corresponding to `condition` needs to produce a logical value, either `TRUE` or `FALSE`. Whenever R encounters a while statement, it checks to see if the condition is `TRUE`. If it is, then R goes on to execute all of the commands inside the curly brackets, proceeding from top to bottom as usual. However, when it gets to the bottom of those statements, it moves back up to the while statement. Then, like the mindless automaton it is, it checks to see if the condition is `TRUE`. If it is, then R goes on to execute all the commands inside ... well, you get the idea. This continues endlessly until at some point the `condition` turns out to be `FALSE`. Once that happens, R jumps to the bottom of the loop (i.e., to the `}` character), and then continues on with whatever commands appear next in the script.

B.2.2 A simple example

To start with, let's keep things simple, and use a `while` loop to calculate the smallest multiple of 179 that is greater than or equal to 1000. This is of course a very silly example, since you can calculate it using simple arithmetic, but the point here isn't to do something novel. The point is to show how to write a `while` loop. Here's the code in action:

```
x <- 0
while(x < 1000) {
  x <- x + 179
}
print(x)
```

```
## [1] 1074
```

When we run this code, R starts at the top and creates a new variable called `x` and assigns it a value of 0. It then moves down to the loop, and "notices" that the condition here is `x < 1000`. Since the current value of `x` is zero, the condition is `TRUE`, so it enters the body of the loop (inside the curly braces). There's only one

command here, which instructs R to increase the value of `x` by 179. R then returns to the top of the loop, and rechecks the condition. The value of `x` is now 179, but that's still less than 1000, so the loop continues. Here's a visual representation of that:

To see this in action, we can move the `print` statement inside the body of the loop. By doing that, R will print out the value of `x` every time it gets updated. Let's watch:

```
x <- 0
while(x < 1000) {
  x <- x + 179
  print(x)
}
```

```
## [1] 179
## [1] 358
## [1] 537
## [1] 716
## [1] 895
## [1] 1074
```

Truly fascinating stuff.

B.2.3 Mortgage calculator

To give you a sense of how you can use a `while` loop in a more complex situation, let's write a simple script to simulate the progression of a mortgage. BLEGH COME UP WITH SOMETHING BETTER

A happy ending! Yaaaaay!

B.2.4 The `for` loop

The `for` loop is also pretty simple, though not quite as simple as the `while` loop. The basic format of this loop goes like this:

```
for ( VAR in VECTOR ) {
  STATEMENT1
  STATEMENT2
  ETC
}
```

In a `for` loop, R runs a fixed number of iterations. We have a vector which has several elements, each one corresponding to a possible value of the variable `var`. In the first iteration of the loop, `var` is given a value corresponding to the first element of vector; in the second iteration of the loop `var` gets a value corresponding to the second value in vector; and so on. Once we've exhausted all of the values in the vector, the loop terminates and the flow of the program continues down the script.

B.2.5 Multiplication tables

When I was a kid we used to have multiplication tables hanging on the walls at home, so I'd end up memorising the all the multiples of small numbers. I was okay at this as long as all the numbers were smaller than 10. Anything above that and I got lazy. So as a first example we'll get R to print out the multiples of 137. Let's say I want to it to calculate 137×1, then 137×2, and so on until we reach 137×10. In other

words what we want to do is calculate `137 * value` for every `value` within the range spanned by `1:10`, and then print the answer to the console. Because we have a *fixed* range of values that we want to loop over, this situation is well-suited to a `for` loop. Here's the code:

```
for(value in 1:10) {
  answer <- 137 * value
  print(answer)
}
```

```
## [1] 137
## [1] 274
## [1] 411
## [1] 548
## [1] 685
## [1] 822
## [1] 959
## [1] 1096
## [1] 1233
## [1] 1370
```

The intuition here is that R starts by setting `value` to 1. It then computes and prints `137 * value`, then moves back to the top of the loop. When it gets there, it increases `value` by 1, and then repeats the calculation. It keeps doing this until the `value` reaches 10 and then it stops. That intuition is essentially correct, but it's worth unpacking it a bit further using a different example where R loops over something other than a sequence of numbers...

B.2.6 Looping over other vectors

In the example above, the `for` loop was defined over the numbers from 1 to 10, specified using the R code `1:10`. However, it's worth keeping in mind that as far as R is concerned, `1:10` is actually a vector:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

So in the previous example, the intuition about the `for` loop is slightly misleading. When R gets to the top of the loop the action it takes is "*assigning* `value` *equal to the next element of the vector*". In this case it turns out that this action causes R to "*increase* `value` *by 1*", but that's not true in general. To illustrate that, here's an example in which a `for` loop iterates over a character vector. First, I'll create a vector of `words`:

```
words <- c("it", "was", "the", "dirty", "end", "of", "winter")
```

Now what I'll do is create a `for` loop using this vector. For every word in the vector of `words` R will do three things:

- Count the number of letters in the word
- Convert the word to upper case
- Print a nice summary to the console

Here it is:

```
for(this_word in words) {
  n_letters <- nchar(this_word)
  block_word <- toupper(this_word)
  cat(block_word, "has", n_letters, "letters\n")
}
```

```
## IT has 2 letters
## WAS has 3 letters
## THE has 3 letters
## DIRTY has 5 letters
## END has 3 letters
## OF has 2 letters
## WINTER has 6 letters
```

From the perspective of the R interpreter this is what the code four the `for` loop is doing. It's pretty similar to the `while` loop, but not quite the same:

B.2.7 Vectorised code?

> Somebody has to write loops: it doesn't have to be you - Jenny Bryan

Of course, there are ways of doing this that don't require you to write the loop manually. Because many functions in R operate naturally on vectors, you can take advantage of this. Code that bypasses the need to write loops is called *vectorised* code, and there are some good reasons to do this (sometimes) once you're comfortable working in R. Here's an example:

```
chars <- nchar(words)
names(chars) <- toupper(words)
print(chars)
```

```
##     IT    WAS    THE  DIRTY    END     OF WINTER
##      2      3      3      5      3      2      6
```

Sometimes vectorised code is easy to write and easy to read. I think the example above is pretty simple, for instance. It's not always so easy though!

When you go out into the wider world of R programming you'll probably encounter a lot of examples of people talking about how to vectorise your code to produce better performance. My advice for novices is not to worry about that right now. Loops are perfectly fine, and it's often more intuitive to write code using loops than using vectors. Eventually you'll probably want to think about these topics but it's something that you can leave for a later date!

B.2.8 Exercises

To start with, here are some exercises with `for` loops and turtles: [WE'RE DROPPING TURTLES]

- Use **TurtleGraphics** to draw a square rather than a hexagon
- Use **TurtleGraphics** to draw a triangle.
- Is there a way in which you can get R to automatically work out the `angle` rather than you having to manually work it out?

As an exercise in using a `while` loop, consider this vector:

```
telegram <- c("All","is","well","here","STOP","This","is","fine")
```

- Write a `while` loop that prints words from `telegram` until it reaches `STOP`. When it encounters the word `STOP`, end the loop. So what you want is output that looks like this.

```
## [1] "All"
## [1] "is"
## [1] "well"
## [1] "here"
## [1] "STOP"
```

B.3 Functions

In this section I want to talk about functions again. We've been using functions from the beginning, but you've learned a lot about R since then, so we can talk about them in more detail. In particular, I want to show you how to create your own. To stick with the same basic framework that I used to describe loops and conditionals, here's the syntax that you use to create a function:

```
FNAME <- function( ARG1, ARG2, ETC ) {
  STATEMENT1
  STATEMENT2
  ETC
  return( VALUE )
}
```

What this does is create a function with the name `fname`, which has arguments `arg1`, `arg2` and so forth. Whenever the function is called, R executes the statements in the curly braces, and then outputs the contents of value to the user. Note, however, that R does not execute the function commands inside the workspace. Instead, what it does is create a temporary local environment: all the internal statements in the body of the function are executed there, so they remain invisible to the user. Only the final results in the `value` are returned to the workspace.

B.3.1 A boring example

To give a simple example of this, let's create a function called `quadruple` which multiplies its inputs by four.

```
quadruple <- function(x) {
  y <- x * 4
  return(y)
}
```

When we run this command, nothing happens apart from the fact that a new object appears in the workspace corresponding to the `quadruple` function. Not surprisingly, if we ask R to tell us what kind of object it is, it tells us that it is a function:

```
class(quadruple)
```

```
## [1] "function"
```

And now that we've created the `quadruple()` function, we can call it just like any other function:

```
quadruple(10)
```

```
## [1] 40
```

An important thing to recognise here is that the two internal variables that the `quadruple` function makes use of, `x` and `y`, stay internal. At no point do either of these variables get created in the workspace.

B.3.2 Default arguments

Okay, now that we are starting to get a sense for how functions are constructed, let's have a look at a slightly more complex example. Consider this function:

```
pow <- function(x, y = 1) {
  out <- x ^ y  # raise x to the power y
  return(out)
}
```

The `pow` function takes two arguments `x` and `y`, and computes the value of x^y. For instance, this command

```
pow(x = 4, y = 2)
```

```
## [1] 16
```

computes 4 squared. The interesting thing about this function isn't what it does, since R already has has perfectly good mechanisms for calculating powers. Rather, notice that when I defined the function, I specified `y=1` when listing the arguments? That's the default value for `y`. So if we enter a command without specifying a value for `y`, then the function assumes that we want `y=1`:

```
pow(x = 3)
```

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

However, since I didn't specify any default value for `x` when I defined the `pow` function, the user must input a value for `x` or else R will spit out an error message.

B.3.3 Unspecified arguments

The other thing I should point out while I'm on this topic is the use of the `...` argument. The `...` argument is a special construct in R which is only used within functions. It is used as a way of matching against multiple user inputs: in other words, `...` is used as a mechanism to allow the user to enter as many inputs as they like. I won't talk at all about the low-level details of how this works at all, but I will show you a simple example of a function that makes use of it. Consider the following:

```
doubleMax <- function(...) {
  max.val <- max(...)  # find the largest value in ...
  out <- 2 * max.val   # double it
  return(out)
}
```

The `doubleMax` function doesn't do anything with the user input(s) other than pass them directly to the `max` function. You can type in as many inputs as you like: the `doubleMax` function identifies the largest value in the inputs, by passing all the user inputs to the `max` function, and then doubles it. For example:

```
doubleMax(1, 2, 5)
```

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

B.3.4 More on functions?

There's a lot of other details to functions that I've hidden in my description in this chapter. Experienced programmers will wonder exactly how the scoping rules work in R, or want to know how to use a function to create variables in other environments, or if function objects can be assigned as elements of a list and probably hundreds of other things besides. However, I don't want to have this discussion get too cluttered with details, so I think it's best – at least for the purposes of the current book – to stop here.

B.4 Rescorla-Wagner model

At this point you have all the tools you need to write a fully functional R program. To illustrate this, let's write a program that implements the Rescorla-Wagner model of associative learning, and apply it to a few simple experimental designs.

B.4.1 The model itself

The Rescorla-Wagner model provides a learning rule that describes how associative strength changes during Pavlovian conditioning. Suppose we take an initially neutral stimulus (e.g., a tone), and pair it with an outcome that has inherent value to the organism (e.g., food, shock). Over time the organism learns to associate the tone with the shock and will respond to the tone in much the same way that it does to the shock. In this example the shock is referred to as the *unconditioned stimulus* (US) and the tone is the *conditioned stimulus* (CS).

Suppose we present a compound stimulus AB, which consists of two things, a tone (A) and a light (B). This compound is presented together with a shock. In associative learning studies, this kind of trial is denoted AB+ to indicate that the outcome (US) was present at the same time as the two stimuli that comprise the CS. According to the Rescorla-Wagner model, the rule for updating the associative strengths v_A and v_B between the originally neutral stimuli and the shock is given by:

$$\begin{aligned} v_A &\leftarrow v_A + \alpha_A \beta_U (\lambda_U - v_{AB}) \\ v_B &\leftarrow v_B + \alpha_B \beta_U (\lambda_U - v_{AB}) \end{aligned}$$

where the associative value v_{AB} of the compound stimulus AB is just the sum of the values of the two items individually. This is expressed as:

$$v_{AB} = v_A + v_B$$

To understand this rule, note that:

- λ_U is a variable that represents the "reward value" (or "punishment value") of the US itself, and as such represents the maximum possible association strength for the CS.
- β_U is a learning rate linked to the US (e.g. how quickly do I learn about shocks?)
- α_A is a learning rate linked to the CS (e.g, how quickly do I learn about tones?)

- α_B is also a learning rate linked to the CS (e.g, how quickly do I learn about lights?)

So this rule is telling us that we should adjust the values of v_A and v_B in a fashion that partly depends on the learning rate parameters (α and β), and partly depends on the *prediction error* ($\lambda - v_{AB}$) corresponding to the *difference* between the actual outcome value λ and the value of the compound v_{AB}.

The Rescorla-Wagner successfully predicts many phenomena in associative learning, though it does have a number of shortcomings. However, despite its simplicity it can be a little difficult at times to get a good intuitive feel for what the model predicts. To remedy this, lets implement this learning rule as an R function, and then apply it to a few experimental designs.

B.4.2 R implementation

To work out how to write a function that implements the Rescorla-Wagner update rule, the first thing we need to ask ourselves is *what situations do we want to describe?* Do we want to be able to handle stimuli consisting of only a single feature (A), compounds with two features (AB), or compounds that might have any number of features? Do we want it to handle any possible values for the parameters α, β and λ or just some? For the current exercise, I'll try to write something fairly general-purpose!

B.4.3 The skeleton

To start with, I'll create a skeleton for the function that looks like this:

```
update_RW <- function(value, alpha, beta, lambda) {
}
```

My thinking is that `value` is going to be a vector that specifies the associative strength of association between the US and each element of the CS: that is, it will contain the values v_A, v_B, etc. Similarly, the `alpha` argument will be a vector that specifies the various salience parameters (α_A, α_B, etc) associated with the CS. I'm going to assume that there is only ever a single US presented, so we'll assume that the learning rate β and the maximum associability λ associated with the US are just numbers.

B.4.4 Make a plan

So now how do I fill out the contents of this function? The first thing I usually do is add some comments to scaffold the rest of my code. Basically I'm making a plan:

```
update_RW <- function(value, alpha, beta, lambda) {
  # compute the value of the compound stimulus
  # compute the prediction error
  # compute the change in strength
  # update the association value
  # return the new value
}
```

B.4.5 Fill in the pieces

Since the stimulus might be a compound (e.g. AB or ABC), the first thing we need to do is calculate the value (V_{AB}) of the compound stimulus. In the Rescorla-Wagner model, the associative strength for the compound is just the sum of the individual strengths, so I can use the `sum` function to add up all the elements of the `value` argument:

```
update_RW <- function(value, alpha, beta, lambda) {

  # compute the value of the compound stimulus
  value_compound <- sum(value)

  # compute the prediction error
  # compute the change in strength
  # update the association value
  # return the new value
}
```

The `value_compound` vector plays the same role in my R function that V_{AB} plays in the equations for the Rescorla-Wagner model. However, if we look at the Rescorla-Wagner model, it's clear that the quantity that actually drives learning is the *prediction error*, $\lambda_U - V_{AB}$, namely the difference between the maximum association strength that the US supports and the current association strength for the compound. Well that's easy... it's just subtraction:

```
update_RW <- function(value, alpha, beta, lambda) {

  # compute the value of the compound stimulus
  value_compound <- sum(value)

  # compute the prediction error
  prediction_error <- lambda - value_compound

  # compute the change in strength
  # update the association value
  # return the new value
}
```

Now we have to multiply everything by α and β, in order to work out how much learning has occurred. In the Rescorla-Wagner model this is often denoted Δv. That is:

$$\begin{aligned} \Delta v_A &= \alpha_A \beta_U (\lambda_U - v_{AB}) \\ \Delta v_B &= \alpha_B \beta_U (\lambda_U - v_{AB}) \end{aligned}$$

Within our R function, that's really simple because that's just multiplication. So let's do that, and while we're at it we'll update the `value` (that's just addition) and return the new association value...

```
update_RW <- function(value, alpha, beta, lambda) {

  # compute the value of the compound stimulus
  value_compound <- sum(value)

  # compute the prediction error
  prediction_error <- lambda - value_compound

  # compute the change in strength
  value_change <- alpha * beta * prediction_error

  # update the association value
  value <- value + value_change
```

```
  # return the new value
  return(value)
}
```

B.4.6 Tidying

Depending on your personal preferences, you might want to reorganise to make this a little shorter. You could do this by shortening the comments and moving them to the side. You might also want to set some sensible default values, as I have done here:

```
update_RW <- function(value, alpha=.3, beta=.3, lambda=1) {
  value_compound <- sum(value)                    # value of the compound
  prediction_error <- lambda - value_compound     # prediction error
  value_change <- alpha * beta * prediction_error # change in strength
  value <- value + value_change                   # update value
  return(value)
}
```

All done! Yay!

B.4.7 Model predictions

Okay, now that we have a function `update_RW` that implements the Rescorla-Wagner learning rule, let's use it to make predictions about three learning phenomena: *conditioning*, *extinction* and *blocking*.

B.4.8 Conditioning

For the first "experiment" to simulate, we'll pair a simple CS (i.e. not compound) with a US for 20 trials, and examine how the association strength changes over time. So get started, we'll specify the number of trials

```
n_trials <- 20
```

and we'll create a numeric `strength` vector that we will use to store the association strengths. The way we'll do that is like this:

```
strength <- numeric(n_trials)
strength
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

As you can see, the `numeric` function has created a vector of zeros for us. When trial 1 begins association strength is in fact zero, so that much is correct at least, but of course we'll need to use the `update_RW` function to fill in the other values correctly. To do that, all we have to do is let the experiment run! We set up a loop in which we "present" the CS-US pairing and update the association strength at the end of each trial:

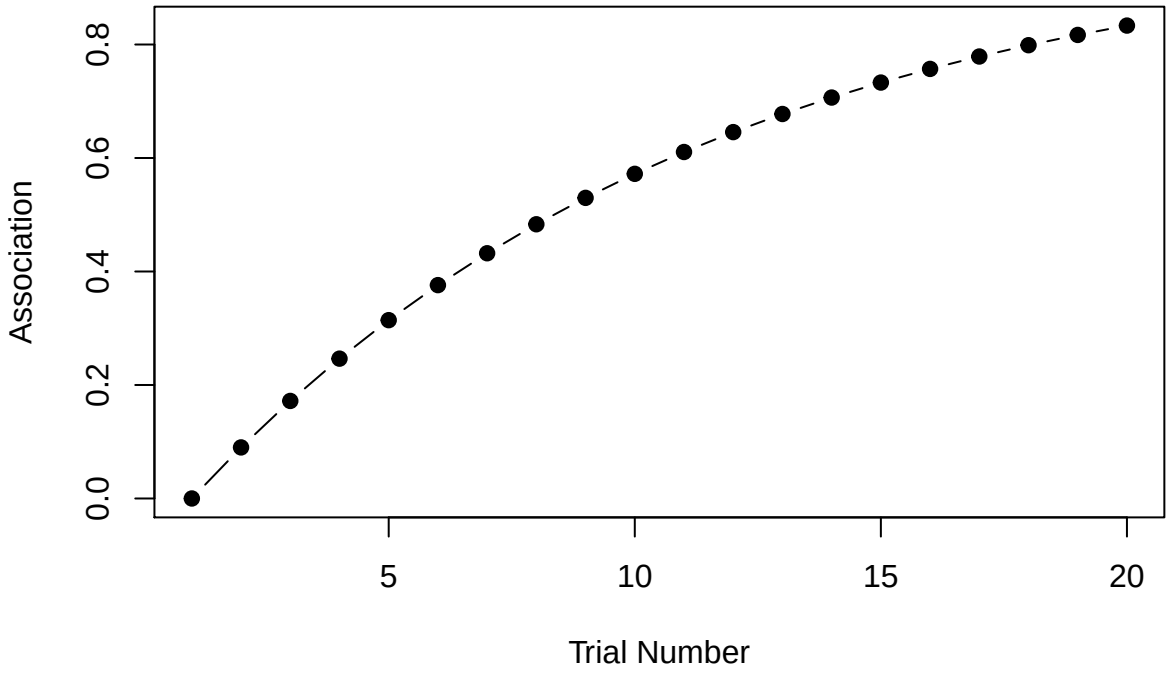
```
for(trial in 2:n_trials) {
  strength[trial] <- update_RW( strength[trial-1] )
}
```

That's it! Now we print out the association strength:

```
print(strength)
```

```
##  [1] 0.0000000 0.0900000 0.1719000 0.2464290 0.3142504 0.3759679 0.4321307
##  [8] 0.4832390 0.5297475 0.5720702 0.6105839 0.6456313 0.6775245 0.7065473
## [15] 0.7329580 0.7569918 0.7788626 0.7987649 0.8168761 0.8333572
```

You can see in the output that with repeated stimulus presentations, the strength of the association rises quickly. It's a little easier to see what's going on if we draw a picture though:



I've hidden the R command that produces the plot, because we haven't covered data visualisation yet. However, if you are interested in a sneak peek, the source code for all the analyses in this section are here.

B.4.9 Extinction

For the second example, let's consider the extinction of a learned association. What we'll do this time is start out doing the same thing as last time. For the first 25 trials we'll present a CS-US trial that pairs a tone with a shock (or whatever) and over that time the association for the CS will rise to match the reward "value" ($\lambda = .3$) linked to the US. Then for the next 25 trials we will present the CS alone with no US present. We'll capture this by setting $\lambda = 0$ to reflect the fact that the "value" to be predicted is now zero (i.e. no shock). For simplicity, we'll leave the learning rate β the same for shock and no-shock.

Okay here goes. First, let's set up our variables:

```
n_trials <- 50
strength <- numeric(n_trials)
lambda <- .3 # initial reward value
```

Now we have to set up our loop, same as before. This time around we need to include an `if` statement in the loop, to check whether we have moved from the learning phase (trials 1 to 25) to the extinction phase (trials 26 to 50), and adjust the value of λ accordingly.

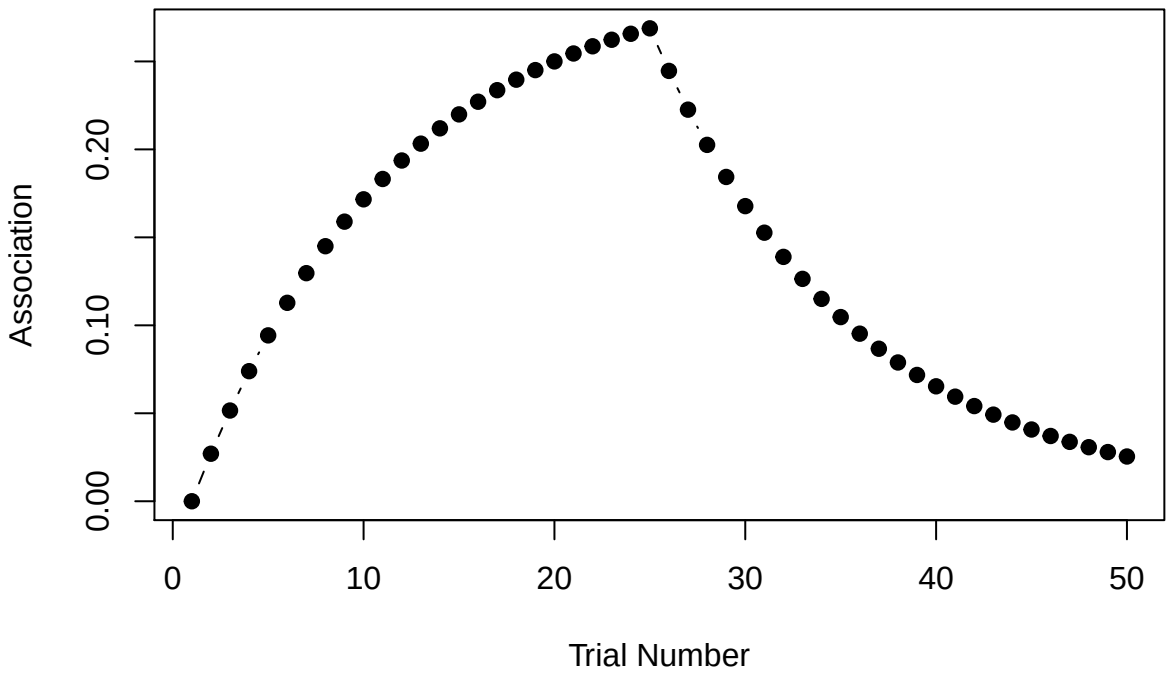
```
for(trial in 2:n_trials) {

  # remove the shock after trial 25
  if(trial > 25) {
    lambda <- 0
  }

  # update associative strength on each trial
  strength[trial] <- update_RW(
    value = strength[trial-1],
    lambda = lambda
  )
}
```

What we expect to see in this situation is that after trial 25 when the shock is removed, the association strength starts to weaken because the learner is now associating the CS with no-shock (i.e. λ has dropped to zero and so the association v is slowly reverting to that value). Here's the raw numbers:

```
print(strength)
```

```
##  [1] 0.00000000 0.02700000 0.05157000 0.07392870 0.09427512 0.11279036
##  [7] 0.12963922 0.14497169 0.15892424 0.17162106 0.18317516 0.19368940
## [13] 0.20325735 0.21196419 0.21988741 0.22709755 0.23365877 0.23962948
## [19] 0.24506283 0.25000717 0.25450653 0.25860094 0.26232685 0.26571744
## [25] 0.26880287 0.24461061 0.22259565 0.20256205 0.18433146 0.16774163
## [31] 0.15264488 0.13890684 0.12640523 0.11502876 0.10467617 0.09525531
## [37] 0.08668234 0.07888093 0.07178164 0.06532129 0.05944238 0.05409256
## [43] 0.04922423 0.04479405 0.04076259 0.03709395 0.03375550 0.03071750
## [49] 0.02795293 0.02543716
```

Here they are as a pretty picture:



That looks right to me! Extinction is initially effective at removing the association, but it's effectiveness declines over time, so that by the end of the task there's still some association left.

B.4.10 Blocking

For the final example, consider a blocking paradigm. In this design we might initially pair a tone with a shock (A+ trials) for a number of trials until an association is learned. Then we present a compound stimulus AB (tone plus light) together with a shock (AB+). During the first phase, the learner quickly acquires a strong association between A and the shock, but then during the second phase they don't learn very much about B, because A already predicts the shock.[31]

Because we are presenting a compound stimulus, the values that we pass to the `update_RW` function can be vectors. But that's okay, we designed our function to be able to handle that. So let's start by setting up our modelling exercise:

```
# total number of trials across
# both phases of the task
n_trials <- 50

# vectors of zeros
strength_A <- rep(0,n_trials)
strength_B <- rep(0,n_trials)
```

[31] In a real blocking study there would be various control conditions but I'm not going to bother modelling those here. I just want to show how our code works for the important one!

There are two strength vectors here, one for the tone (A) and one for the light (B). Of course, during the first phase of the task the light isn't actually present, which we can capture by setting the relevant learning rate (or salience) parameter α to 0:

```
alpha <- c(.3, 0)
```

This means that at the start of the task, the model will learn about the tone but not the light. After trial 15, however, both stimuli will be present. For simplicity I'll assume they're equally salient, so after trial 15 the α value becomes .3 for both stimuli.

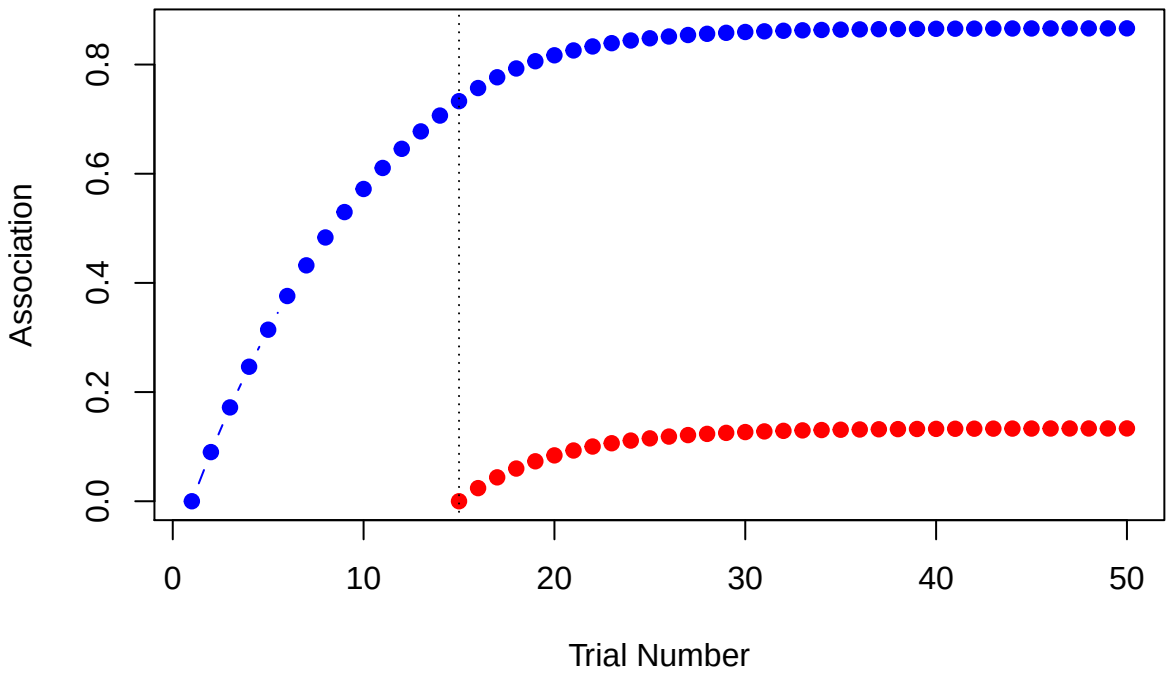
As before we construct a loop over the trials:

```
for(trial in 2:n_trials) {

  # after trial 15, both stimuli are present
  if(trial > 15) alpha <- c(.3, .3)

  # vector of current associative strengths
  v_old <- c(strength_A[trial-1], strength_B[trial-1])

  # vector of new associative strengths
  v_new <- update_RW(
    value = v_old,
    alpha = alpha
   )

  # record the new strengths
  strength_A[trial] <- v_new[1]
  strength_B[trial] <- v_new[2]
}
```

It's a little more complex this time because we have read off two strength values and pass them to the `update_RW` function.[32] Hopefully it is clear what this code is doing.

As with the previous two examples we could print out `strength_A` and `strength_B`, but realistically no-one likes looking at long lists of numbers so let's just draw the picture. In the plot below, the blue line shows the associative strength to A and the red line shows the associative strength to B:

[32] In fact, we could make this code a lot simpler if we'd recorded the strengths as a **matrix** rather than two vectors, but since I haven't introduced matrices yet I've left it in this slightly more clunky form with vectors



That's the blocking effect, right there! The model learns a strong association between the tone and the shock (blue) but the association it learns between the light and the shock (red) is much weaker.

B.4.11 Done!

If you've been following along yourself you are now officially a computational modeller. Nice work!