Teach Yourself or Others R

Adam Rawles

Contents

Chapter 1

teacheR

1.1 Overview

This book is a collection of training materials for an introduction to the R statistical computing programming language. Broken down into chapters, I've aimed to cover most of the basics. Alongside each chapter is a xaringan presentation. This can help for those looking to learn, but can also function as a first step for those looking to begin teaching R but who don't have the time to fully develop their own training modules. Hopefully, all the topics covered in the text version will be covered in the presentation and vice versa, so if you learn visually then you can rest easy knowing that you're not missing out! You can find a link to each presentation on the first page of each chapter.

This is a work in progress, and so I would greatly appreciate any feedback. Anything from typos to content suggestions, feel free to raise a GitHub issue if you feel something should be changed.

1.2 About Me

I began using R in my second year of university, during an internship looking at publication bias correction methods. I was under the tutorship of a member of staff who helped me immensely, but I must confess that I have never taken an official course in R, online or in person. I like to think, however, that this is not always a bad thing. Learning from the bottom up and struggling along the way is a fantastic way to acquire knowledge and instills a very important lesson:

You're not going to know everything there is to know about R. Ever. But that's okay.

I'm now 4 years into my R career and I use R every day. With that in mind, I don't think there has ever been a day where I haven't referred to a tutorial, or Stack Overflow, or even just Googled the name of a function that I've used 1000 times before. There is a great repository of knowledge for R and it's one of the things I love most about the R community. So please never feel as though you're an impostor in a world of R gurus. In reality, everyone else is just as lost as you. But if you keep ticking along and never feel that learning something new in R isn't worth your time, you'll end up doing some great things.

And in a roundabout way, that is part of the reason I decided to develop these materials. I don't pretend to be the ultimate R programmer, because I still know what it's like to learn something from the start. And everyone has to start somewhere. So I hope that I can help impart some of the lessons that I've learnt over the 4 years to anyone who's looking to learn R in a way that won't leave you feeling lost.

The only final note I have before we start learning how to use R is another bit of advice:

Don't believe everything you read

Whilst this is probably a good thing to keep in mind for any type of training, I feel it's particularly relevant with R for two reasons. Firstly, when it comes to programming languages, lots of people have opinions. Some are true, most are not. Most things you read are a mix between fact and opinion, so take everything with a pinch of salt. For example, the developers of the `ggplot2` package are fervently against arbitrary second axes and so support for them in `ggplot2` is limited. I also share this view, but that doesn't mean that I'm right - read, learn, but question and make your own mind up.

Secondly, R and particuarly all of its packages are prone to change. For this reason, people may make statements relative to one version of R that aren't necessarily true in the future. Things have changed over the years, and so answers from a 10 year-old Stack Overflow question may not still be true when you come across them. A microcosmic version of this are some recent changes in the `tidyr` package. Historically, converting data from/to long and short formats was done using the `spread()` and `gather()` functions. However, in newer releases, these functions are deprecated in favour of `pivot_wider()` and `pivot_longer()`, which provide the same functionality but also some extra bits. The practical implication of this suggestion is don't always read one tutorial on a subject before you dive in.

Chapter 2

Introduction to R

Chapters 2-7 cover R and its basics. The xaringan presentation for this module is here.

2.1 What is R?

R is a public license programming language. More specifically, it's a statistical programming language meaning that it's often used for statistical analysis rather than software development. R is also a functional programming, rather than an object-oriented programming language like Python. This means that operation in R are primarily performed by functions (input, do something, output), but more about that later.

Strictly speaking, R is not just a functional programming language. In reality, a language is never purely one type and R is no exception. There are object-oriented systems in R (three main ones), meaning that object-oriented programming is possible and relatively straightforward in R. Having said that, I feel now is a good time to refer back to the mantra from the Overview chapter:

Don't believe everything you read

So basically, R is a functional programming language with some object-oriented systems. If that means very little to you, don't worry. For the vast majority of users, this is a purely academic definition.

One important attribute about R however that may affect you, is that R is an interpreted language. This essentially means that when you send someone some R code, they need R installed to be able to run it. This means that making full programs is difficult. Later on, we'll look at the `shiny` package, which can be used to quickly make web apps based on R code. These apps are no different in

the sense that they also need R installed to be able to run, but because they are web-based, they are significantly easier to share.

For the most part however, if you want to share R code with colleagues, they'll need to have R installed as well.

2.2 Should I use R?

People will forever argue about which is better, R or Python or Java or C or writing down mathematical equations on a piece of paper and handing it to a monkey to solve. I imagine you're reading this because you heard that R was good for data analysis, and it absolutely is. And so is Python. They're just... different. Personally, I prefer to use R but I understand that other people don't.

Importantly though, never feel as though you've missed a trick by picking a particular language. Programming is not just a practice, it's a way of thinking, and experience is almost always transferable across languages.

For reference however, here are a few of the things that you can use R for:

- data analysis
- reporting and writing
- web apps
- text analysis

If you're interested in any of these, then you're in the right place.

2.3 Using R

R is very simple. There is a console where you type commands and get responses. Like the classic command-line interfaces you see when the stereotypical nerd has to hack into the FBI database, you type commands, one at a time into the console, R processes it, and then produces a response if appropriate. For example, if you type `2 + 2` into the R console and hit enter, you'll get `4`.

Writing commands out one at a time can be quite time consuming if you want to make changes however. So we use scripts to store multiple lines of code that can then be run altogether. When you execute a script, each line gets passed one by one to the console and executed. For example, I might make a script with this code:

```
variable1 <- 2 + 2
variable1 / 10
```

```
## [1] 0.4
```

So when I run the script, it will run the first line, then the second without me having to type anything else in.

2.4 RStudio

RStudio is separate from R. R is a programming language and RStudio is an integrated development environment or *IDE*. This means that RStudio doesn't actually run any code, it just passes it to R for you, meaning that you'll need R to really use RStudio.

RStudio is a massive part of how you interact with R however. For example, with the exception of a few days when I was waiting for RStudio to be installed, I can't ever remember using R without RStudio.

In the previous section, we talked very briefly of the R console and scripts. RStudio helps with this workflow. It makes it easier to create scripts, providing extra tools to help write code quicker, and then acts as a window to R when you want to execute the script.

2.4.1 What is an IDE?

At its simplest definition, an IDE helps you get work done in your programming language of choice. It can help you save blocks of code, organise projects, save plots and everything in between. R comes with a basic user interface when you install it, but RStudio provides lot more functionality to help you interact with the R console.

2.4.2 RStudio Panes

TO DO

2.5 Packages

As we know, R is a functional programming language, meaning that we rely on functions to do our work for us. And when you install R, you'll have access to thousands of functions that come bundled with it. However, these functions have been chosen because they're more generalisable and basic. Including functions for everything that could be done in R with the base version would result in it being unnecessarily large.

So instead of them being available from the start, people create sets of functions that usually are used for a particularly tasks and then distribute them as a

package. You can install this package and have access to all these great functions that someone has written for you.

Some great examples of R packages are:

- ggplot2 for creating plots
- dplyr for data manipulation
- shiny for creating web apps
- BMRSr for extracting energy data
- Truth be told, this isn't a *great* example of a package, it's just the one I've made so that's why it's here.

The thing to remember is that R has a fantastic open source community and if you need to do something in R, somebody has probably written a package to help you out.

2.5.1 Installing packages

You can think of installing a package as a bit like installing a program on your computer. You only need to install it once, but then you'll need to open it each time you use it.

Installing packages is really easy; you just use the `install.packages()` function:

```
install.packages("BMRSr")
```

You can choose where the package installs by suppling a path to the `lib` parameter (e.g. `lib = "C:/me/desktop"`), but by default it will install it into your default library folder. You can find the path to this default library folder with the `.libPaths()` function.

Once the package has been installed, you only need to reinstall it if there's a problem or you want to update it. Otherwise, you just need to load it every time you want to use it. The logic behind this is that you may have hundreds of packages installed and you don't need all of them for every project you do. So instead, we load specific packages we want each time.

To load a package, use the `library()` function. Place this somewhere near the start of your script so that it's obvious which packages someone will need if they're reading your code and want to do it for themselves. This will then load in all of the functions from that package for you to use.

But Adam, what happens if I load two packages that have functions with the same name? Ah, that's a great question. When R loads your packages, it will do it in a specific order. The later on the package is loaded, the higher it's precedence. That means that when you try and use a function with a name that's in more than one of the packages you've loaded, it will default to the latest package you've loaded.

To help avoid these situations, there are some things in place. Firstly, when you load a package that includes function names that are already used, you'll get a conflict warning. Secondly, to avoid this confusion altogether, you can be explicit about which package your function came from. To do that, just prefix your function with the package name and `::` when you use it, like this:

```
dplyr::mutate()

mutate()
```

Finally, there is a package called `conflicted` that you can use to avoid these issues. Rather than giving a certain package precedence because it was loaded later, attempting to use a function that could come from more than one package will cause an error and you'll need to be explicit with which one you mean.

Personally, I use the prefix method. Yes it's a little bit more verbose, but it makes it 100 times easier for someone else to know exactly where your function has come from without the need for extra packages. This also tends to be the approach that I take because there's nothing worse than coming back to a script a year later and forgetting which packages you need because you didn't include the correct library calls.

Chapter 3

Operators

3.1 Arithmetic operators

At the base of lots of programming languages are the arithmetic operators. These are your symbols that perform things like addition, subtraction, multiplication, etc. Because these operations are so ubiquitous however, the symbols that are used are often very similar across languages, so if you've used Excel or Python or SPSS or anything similar before, then these should be fairly straightforward.

Here are the main operators in use:

```
2 + 2  # addition
```

```
## [1] 4
```

```
10 - 5 # subtraction
```

```
## [1] 5
```

```
5 * 4 # multiplication
```

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

```
100 / 25 # division
```

```
## [1] 4
```

3.2 Logical operators

Logical operators are slightly different to arithmetic operators - they are used to evaluate a particular criteria. For example, are two values equal. Or, are two values equal *and* two other values different.

To compare whether two things are equal, we use two equal signs (==) in R:

```
1 == 1 # equal
```

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

Why two I hear you say? Well, a bit later on we'll see that we use a single equals sign for something else.

To compare whether two things are different (not equal), we use !=:

```
1 != 2 # not equal
```

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

The ! sign is also used in other types of criteria, so the best way to think about it is that it inverts the criteria you're testing. So in this case, it's inverting the "equals" criteria, making it "not equal".

Testing whether a value is smaller or larger than another is done with the < and > operators:

```
2 > 1  # greater than
```

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

```
2 < 4 # less than
```

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

Applying our logic with the ! sign, we can also test whether something is *not* smaller or *not* larger:

```
1 >! 2 # not greater than
```

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

```
2 <! 4 # not less than
```

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

Why is the ! sign before the equals sign in the "not equal" to code, but after the "less than/greater than" sign? No idea. It'd probably make more sense if they were the same, but I suppose worse things happen at sea.

There are three more logical operators, and they are the "and", "or", and "xor" operators. These are used to test whether at least one or more than one or only one of the logical comparisons are true or false:

```
1 == 1 | 2 == 3 # or (i.e. are either of these TRUE)
```

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

```
1 == 1 & 2 == 3 # and (i.e. are these both TRUE)
```

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

The xor operator is a bit different:

```
xor(1 == 1, 2 == 3) # TRUE because only 1 is
```

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

```
xor(1 == 1, 2 == 2) # FALSE because both are
```

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

For `xor()`, you need to provide your criteria in brackets, but this will make much more sense once we look at functions.

Chapter 4

Variable assignment

Do you ever tell a story to a friend, and then someone else walks in once you've finished and so you have to tell the whole thing again?

Well, imagine after the second friend walks in, another friend comes in, and you have to start the story over again, and then another friend comes in and so on and so forth. What would be the best way to save you repeating yourself? As weird as it would look, if you wrote the story down then anyone who came in could just read it, rather than you having to go through the effort of explaining the whole thing each time.

This is essentially what we can do in R. Sometimes you'll use the same value again and again in your script. For example, say you're looking at total expenditure over a year, the value for the amount spent would probably come up quite a lot. Now, you could just type that value in every time you need it, but what happens if the value changed? You'd then have to go through and change it every time it appears.

Instead, you could store the value in a variable, and then reference the variable every time you need it. This way, if you ever have to change the value, you only need to change it once.

Creating variables in R is really easy. All you need to do is provide a valid name, use the `<-` symbol, and then provide a value to assign:

```
hello_im_a_variable <- 100
hello_im_a_variable
```

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

Now, whenever you want to use your variable, you just need to provide the variable name in place of the value:

```
hello_im_a_variable / 10
```

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

You can even use your variable to create new variables:

```
hello_im_another_variable <- hello_im_a_variable / 20
hello_im_another_variable
```

```
## [1] 5
```

When you come across other people's work, you may see that they use `=` instead of `<-` when they create their variables. Even though it's not the end of the world if you do do that, I would recommend getting into the habit of using `<-`. `<-` is purely used for assignment, whereas `=` is actually also used when we call functions, and so it can get a bit confusing if you use them interchangeably.

As a side note, you'll see that the value of the variable isn't outputted when we assign it. If we want to see the value, we need just the name.

Variables are very flexible. You can overwrite a previously defined variable just be reassigning a new value to the same name:

```
variable_1 <- 100
variable_1 <- "I'm not 100 anymore"
variable_1
```

```
## [1] "I'm not 100 anymore"
```

R will also give the variable an appropriate *type* based on the value you assign. So for example, if you assign `20` to a variable, then that variable will be stored as a number. If you assign something in quotation marks like `"hello"`, then R will store it as text.

Let's look in a bit more detail at the different data types...

Chapter 5

Data types

Data can be stored in lots of different forms. For example, `"TRUE"` and `TRUE` are stored as two different types, even though they look very similar to us.

The main different data types are:

- logical
  - TRUE
  - FALSE
- double (numeric)
  - 12.5
  - 19
  - 99999
- integer (numeric)
  - 2L
  - 34L
- character
  - hello"
  - my name is"
- factors
- dates
  - 2019-06-01
- datetime (POSIXct)
  - 2019-06-01 12:00:00

Let's have a look at each one in detail:

5.1 Logical

A logical variable can only have two *real* values, `TRUE` or `FALSE`. I say two *real* values, because you can also have things like `NA`, but that's true of any data type.

Logical variables are used a lot in response questionnaires, where the answer to the question is either "Yes" or "No" (`TRUE` or `FALSE`). I would recommend converting any character strings like "Yes" or "No" or "TRUE" or "FALSE" to a logical variable rather than leaving them as characters, because it'll make your analysis less verbose (use fewer lines of code), even if it doesn't change the underlying logic.

To test whether something is stored as logical, we use the `is.logical()` function:

```
is.logical(TRUE)
```

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

```
is.logical("TRUE")
```

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

To convert a value to logical, use the `as.logical()` function:

```
as.logical(1)
```

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

```
as.logical(0)
```

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

```
as.logical("TRUE")
```

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

```
as.logical("FALSE")
```

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

Be careful though, just because a conversion seems obvious to you, doesn't mean you'll get the expected result! For instance, what do you think `as.logical(2)` should return? See for yourself.

5.2 Double

The best way to think of a `double` value is as a number. It can be a whole number (but see Integers) or a decimal. R will often take care of any implicit number conversion that needs to be done under the hood, so the only thing you

really need to keep in mind is that when you assign a number, be it a whole number or decimal, it will be stored as double by default.

As an aside, it's called double because it's stored using double precision.

To check whether a value is stored as double (or more generally numeric), use the `is.double()` and `is.numeric()` functions:

```
is.double(2)
```

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

```
is.numeric("not numeric")
```

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

```
is.double(2L) # see the next section for why this returns FALSE
```

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

To convert a value to a double, use the `as.double()` or `as.numeric()` functions:

```
as.double("5")
```

```
## [1] 5
```

```
as.numeric("10")
```

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

```
as.double("im going to cause a warning")
```

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

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

5.3 Integer

Whilst also storing numeric data (like double), integers are specific to whole numbers. Also, by default, even when you assign a whole number, like this: , R will store that value as double rather than as an integer. To store something explicitly as an integer, suffix the value with an `L`, like this: . Attempting to store something that isn't an integer as an integer will result in a warning:

```
1.5L
```

```
## [1] 1.5
```

For the most part, I let R take care of how it stores numbers, unless I explicitly need it to be of a certain type. This is pretty rare though.

To check if something is an integer, use the `is.integer()` function:

```r
is.integer(2)
```

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

```r
is.integer(2L)
```

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

To convert to an integer, use the `L` suffix or the `as.integer()` function:

```r
1L
```

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

5.4 Character

Sometimes called characters, or character strings, or just strings, characters store text. If you assign a value within quotation marks, regardless of what's inside the quotation marks, it will be stored as character. For example, 5 stores a character string with the text "5", not the number 5. This is particularly important when you want to start combining variables. For example, "`{r, error = TRUE} "5" + 5"` doesn't work, because you're trying to add text to a number, which doesn't make sense.

To check whether something is stored as a character, use the `is.character()` function:

```r
is.character("hello")
```

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

```r
is.character(5)
```

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

```r
is.character(TRUE)
```

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

To convert something to a character, use the `as.character()` function:

```r
as.character(5)
```

```
## [1] "5"
```

```r
as.character(TRUE)
```

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

5.5 Factors

Factors are a unique but useful data type in R. Essentially, factors store different levels that represent some sort of grouping. For example, say you were collecting some information on people from different countries, the column that holds which country the respondent is from could be stored as a factor, with the levels England, Spain, France, etc.

A factor level is made up of two things. A label and a number that represents that group. For example, in my countries example, our factor would have the labels "England", "Spain", "France" and the values 1, 2, 3. This means that internally, a factor is essentially a collection of integers representing the level position and character strings representing the level label.

To create a factor, we just use the `factor()` function:

```
factor(c("England", "France", "Spain"))
```

```
## [1] England France  Spain
## Levels: England France Spain
```

It's also worth remembering that you can have levels that don't appear in the data you have. For example, in a questionnaire, you may provide the options "None", "Some", "All". But in your responses, you may see that no one chose the "None" option. In that case, you would still create a factor with three levels, even though only two of them appear.

You can also specify whether a factor is *ordered*. You would use an ordered factor when the levels have meaningful order. For instance, in the above example, it would make sense that "Some" is better than "None", and "All" is better than "Some". To create an ordered factor, just specify `ordered = TRUE` in your function. By default, the factor will be ordered in the ordered the values appear, unless you specify levels (see below).

To convert something to a factor, use the `factor()` function if you want to specify levels and labels, or `as.factor()` to do it for you:

```
factor(c("Some", "All"), levels = c("None", "Some", "All"))
```

```
## [1] Some All
## Levels: None Some All
```

```
factor(c("Some", "All"), levels = c("None", "Some", "All"), ordered = TRUE)
```

```
## [1] Some All
## Levels: None < Some < All
```

```
as.factor(c("Some", "All"))
```

```
## [1] Some All
## Levels: All Some
```

Notice the difference in the output of those three lines. The first allows us to specify the levels (i.e. the values that were possible). The second does the same but we also specify the ordering of the levels, and the third just converts the provided values and generates the levels based on that data.

Note: An important change in R version 4.0.0 is that R will no longer automatically convert strings (characters) to factors when you import data using `data.frame()` or `read.table()`. Prior to 4.0.0, it would automatically convert strings to characters unless otherwise specified.

5.5.1 Converting from factors

Sometimes you'll need to convert data from a factor to something else, usually a character. This is fairly straightforward using the tools we've already seen:

```
as.character(factor(c("Some", "None", "All")))
```

```
## [1] "Some" "None" "All"
```

5.6 Dates

Dates in any language are tricky. Different countries store dates in different formats and different bits of software stores dates in different ways (looking at you Excel). This can make storing values as dates tough.

The most common way of creating a date is to use the `as.Date()` function. To use this function, you just need to provide your date as a character string:

```
as.Date("2019/01/01")
```

```
## [1] "2019-01-01"
```

But Adam, how does R know which one is the month and which is the day? Good question, thank you for asking. By default, R expects your character string to be in the order "Year/Month/Day". If you don't provide it in that format, you'll get a nonsense output:

```
as.Date("01/12/2019")
```

```
## [1] "1-12-20"
```

If you're data is in a different format however, you can specify the format:

```
as.Date("01/12/2019", format = "%d/%m/%Y")
```

```
## [1] "2019-12-01"
```

Here, we're telling R that the string is in the format "Day/Month/Year". A list of the different codes that can be used in the format parameter can be found here, or by typing "R date codes" into Google.

Because nothing in life is simple, sometimes you'll get some data that has the date stored as a number. This is because the source of that data has the date stored as the number of days that have passed since an origin date. Because it's a number, our `as.Date(..., format = ...)` doesn't work. Instead, we can still use the `as.Date()` function, but we need to specify what the origin date is that the number refers to.

By default, when importing from Excel, the origin date is January 1st 1970, also known as the "epoch date". Chances are your data source is also using this date, but always check.

Anyway, to specify your origin, we use the `origin` parameter, like this:

```
as.Date(18262, origin = "1970/01/01")
```

```
## [1] "2020-01-01"
```

Notice the format I've provided the origin in. It's the same as the default that R expects, and I would recommend copying that format wherever possible. If you're someone who just wants to watch the world burn, then you can specify a format for your origin as well...'

```
as.Date(18262, origin = as.Date("01/01/1970", format = "%d/%m/%Y"))
```

```
## [1] "2020-01-01"
```

but where's the humanity in that?

Testing whether something is a date is not as simple as the other data types unfortunately. Instead, we just use the `is()` function. If the first value returned is "Date", then you know it's a date:

```
is(as.Date("2020/01/01"))
```

```
## [1] "Date"     "oldClass"
```

5.7 Datetimes (POSIXct)

If you thought dates were annoying, datetimes are like dates' little brother who keeps asking when his turn on the Xbox is. One of the reasons for this is that datetimes aren't actually called datetimes. They're called POSIXct in R. So whenever you see that dreadful word, just remember "ah, Adam told me that means datetime" and you'll be fine.

Another thing that makes datetimes tough is that in addition to dates, datetimes (as you may have guessed) also store the time. The issue with that is that time

is a more relative concept - there are lots of different time zones, so how do you know which one you're referring to. By default, R has a locale for where you currently are and will use that location for your timezone. You override that default using the `Sys.setlocale()` function, or you can use the `tz` parameter when creating your datetime as we'll see below.

With these annoyances aside however, creating datetimes isn't all that different to creating dates except that we use the `as.POSIXct()` function instead. We just provide a character string (with a `format` specification if necessary), or a number with an origin. One important departure from dates though, is that now our origin is in seconds, not days, to allow us to calculate the time.

```
as.POSIXct("2020/01/01 12:00:00")
```

```
## [1] "2020-01-01 12:00:00 UTC"
```

```
as.POSIXct("2020/01/01 12:00:00", tz = "NZ")
```

```
## [1] "2020-01-01 12:00:00 NZDT"
```

```
as.POSIXct(1577880000, origin = "1970/01/01")
```

```
## [1] "2020-01-01 12:00:00 UTC"
```

Similar to dates, there is no `as.POSIXct()` function in base R, so we use the `is()` function instead:

```
is(as.POSIXct("2020/01/01 12:00:00"))
```

```
## [1] "POSIXct"  "POSIXt"   "oldClass"
```

5.8 Technicalities

Although I've presented the above as separate data types, in reality objects and values are not represented in this distinct fashion. Instead, some of the data types can be considered sub-types of other data types. For example, if we type:

```
is(1L)
```

```
## [1] "integer"              "double"               "numeric"
## [4] "vector"               "data.frameRowLabels"
```

We see that it's an integer, but it's also a double, and numeric, and a vector. What this demonstrates is the idea that data types are not siloed, instead there is a web of *inheritance*, with types that span many sub-types. Inheritance is a complicated subject and so we won't get into it now, but it essentially just reflects that not each data sub type is an island.

Why have I presented in the way that I did then? Well, for beginners, this degree of inheritance doesn't really matter. Integers are distinct from doubles, which

are distinct from factors, and so on. And so as long as you know the general data type of your value, as well as what structure you're storing it in (which we're moving on to now), then you're fine.

Chapter 6

Data structures

It's rare that you're ever going to be working on a single value in R. Instead, you're going to want to work on collections of values, like a dataset or a list or something similar. So we need to know what data structures are available in R. Here's a list of the structures which we'll go into more detail:

- vectors
- lists
- matrices
- data frames

6.1 Vectors

Vectors are simple arrays of data in a single dimension. You can think of vectors as a like a very simple list. For instance, you can store the numbers 1 to 10 in a vector. Or each word in a string of text could be stored in a vector.

One of the main things to remember about vectors however, is that they are atomic. That's basically a fancy word to mean that each value in a vector must be a single unit. For instance, the number `1` is a single unit. But a vector containing all the numbers between 1 and 10 is not. This is in direct contrast to lists, which are recursive, and we'll look at those next.

To create a vector, we use the `c()` function, which is short for concatenate. In other words, we're pulling together lots of different values and concatenating them into one structure.

```
c(1,2,3,4)
```

```
## [1] 1 2 3 4
```

Technically speaking, even single values are stored as a vector in R, they just have length one. That's why if you type `is(1)`, the second things that pops up after "numeric" is "vector". R is telling us that `1` is a number and that it's in a vector.

All the values in a vector must be of the same type (e.g. character, numeric, etc.), but you can name the values in a vector. To give a value a name, you can simply provide one with a `=` sign when you create your vector:

```
c(this_is_the_first_value = 1, this_is_the_second_value = 2)
```

```
##  this_is_the_first_value this_is_the_second_value
##                        1                        2
```

6.2 Lists

Lists are similar to vectors in that they store values one after another. However, there are two main differences:

- Lists can contain values of any type - they are *recursive*.

Recursion is the action of doing something again and again. We call lists recursive, because we could have a list, that contains a list, that contains a list, and so on and so forth like Russian Dolls.

- Lists do not have to be made up of values of the same type.

So for instance, whilst a vector must always be the same, like `c(1,2,3)`, we could have a list that looks like this:

```
list(1, "hello", TRUE)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "hello"
##
## [[3]]
## [1] TRUE
```

As you just saw, we create lists using the `list()` function, and providing names is done the same way as it is for vectors:

```
list(
  first_value = 1,
  second_value = "hello"
)
```

```
## $first_value
## [1] 1
##
## $second_value
## [1] "hello"
```

6.3 Lists vs Vectors

Given that lists and vectors are intrinsically linked, it's very natural to wonder when to use on over the other. Well, the basic answer is to use whichever one has the requirements you need. If all of your values are of the same type and are atomic (numeric, integer, logical, etc.). If they aren't all the same, or you need to have a list of data structures like vectors and lists rather than just single values, then use a list.

I appreciate that this answer isn't particularly satisfactory however, and so let me give a real life example of when I've used each.

Vector

I was recently producing a simulation that I needed to run multiple times with a different value each time. The value itself was a single number ranging between 1 and 30. So I used a vector like so:

```
my_vector <- c(1:30)
# this is just shorthand for saying "all of the numbers from 1 to 30"
```

So when I ran my simulation, I had all the values I wanted to run it for in a single structure.

List

When doing data modelling, it can sometimes be helpful to create and evaluate multiple models. One way of doing that is to create multiple models and assign them to different variables:

```
model1 <- model(...)
model2 <- model(...)
model3 <- model(...)
```

The problem with this however, is that if I then want to compare the models, I'll have to write out `modelX` each time. If I have 50 models or similar, it may take a while. So instead, I often store all my models in a list. The values are complex (i.e. a model isn't just a numeric or character value) so they can't be stored in a vector, but they can be stored in a list. This means that I keep all my models together, and if I then decide that actually I want to add more models to my list, this is significantly easier than typing out more `modelX <- model(...)` lines and assigning each one as a new variable.

Before moving onto the other data structures, I just want to quickly mention that in my learning experience, understanding vectors and lists is one of the most important parts of getting to grips with R. R for many is about automating analysis and reducing the amount of time taken to do something. And vectors and lists are at the heart of this. Later on, we'll look at functions and for loops, which we can use to perform the same action or calculation on all the values in a list or vector. Together, these will be your strongest R tools.

6.4 Matrices

Unlike vectors, matrices are 2 dimensional. In fact, matrices resemble something a bit like a watered down version of a spreadsheet or table.

I say watered down, because matrices can only contain values of the same type (like vectors). This means that storing complex datasets in matrices isn't really very easy. Instead, matrices are an efficient way of storing and performing matrix mathematics on sets of numbers.

Creating a matrix is easy using the `matrix()` function. We provide the values we want to put into the matrix, and how many rows and columns they should be split into:

```
matrix(c(1:4), nrow = 2, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

By default, the matrix is filled by column first (i.e. it starts at column 1 and fills that column, then moves onto the next one). To change this, use `byrow = TRUE`.

6.5 Dataframes

Dataframes are the more typical dataset storage medium. They can have columns of different types (although all of types within a column need to be the same), and they resemble more of an Excel spreadsheet than matrices.

To create a dataframe, we use the `data.frame()` function. To this function, we provide our values as columns:

```
data.frame(col_1 = c(1,2,3),
           col_2 = c("hello", "world", "howsitgoing"))
```

```
##   col_1       col_2
## 1     1       hello
## 2     2       world
```

```
## 3     3 howsitgoing
```

More specifically, R stores dataframes as essentially a list of lists, with each list representing a different column. To demonstrate this, when we type...

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

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

The second value in the returned vector is "list".

So at it's heart, a dataframe is a list, and each column within a dataframe is also a list. Why is that useful to know? Well, for one, this should make things make a bit more sense when we move onto susbetting. Secondly, when you start to move onto more complicated analysis, you can utilise the features of a list to create datasets that wouldn't be possible in something like Excel. For instance, we know that we can store models in a list. Well, let's say we had a dataset that had data for lots of different countries and we wanted to create a separate model for each country. We could have a dataset that had the country in one column and then the model in another:

```
data.frame(
  country = c("England", "Spain", "France"),
  model =  I(list(model(...), model(...), model(...)))
  # The I() just tells R to leave it as a list
)
```

For now though, don't worry too much about the internals. Just remember that data frames are the most flexible dataset storage medium and they'll be what you do most of your analysis with. And if you can remember that each column is technically a list, then you're ahead of the game.

Chapter 7

Subsetting

There will be occasions where you don't want all the values in a vector/list/matrix/dataframe. Instead, you'll only want a *subset*. The way to do that is slightly different depending on the data structure you're using.

7.1 Vectors

Vectors are simple. Just use square brackets (`[]` or `[[]]`) after your vector and provide the index or indices of the values that you want:

```
c(10,20,30,40)[1]
```

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

```
c(10,20,30,40)[c(1,4)]
```

```
## [1] 10 40
```

```
c(10,20,30,40)[1:3]
```

```
## [1] 10 20 30
```

```
c(10,20,30,40)[[1]]
```

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

P.S. If you have a vector of named values, you can also use the names instead of the indices. Like `c(value_1 = 1)[["value_1"]]`.

But Adam, I hear you ask, `c(10,20,30,40)[1]` and `c(10,20,30,40)[[1]]` just gave us the same thing, so are the interchangeable?

Well, they kind of returned the same thing, but they didn't. So no, they're not interchangeable.

Essentially, `[]` returns the *container* at the provided index, where `[[]]` returns the *value* at the provided index. Let's see a practical example of how these are different:

```
c(value_1 = 10,
  value_2 = 20)[1]
```

```
## value_1
##      10
```

```
c(value_1 = 10,
  value_2 = 20)[[1]]
```

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

In the first call, we get the name of the value and the value itself. In other words, rather than just returning the value at that index, we've essentially just chopped up the vector to only returning everything from the first position. Conversely, in the second call, we've just been given the value. What we've done here is extracted the value out from that position.

/TO DO

I think some kind of analogy could be useful here to drive home the difference.

It's a very subtle difference, but it is an important one. Make sure that if you want the value, use `[[]]`, and if you want the whole part of the vector, use `[]`.

7.2 Lists

Lists can be subsetted in the same way as vectors - `[]` returns the container at the index provided and `[[]]` returns the value:

```
list(
  value_1 = c(1,2,3),
  value_2 = c("hello", "there", "everyone")
)[[1]]
```

```
## [1] 1 2 3
```

```
list(
  value_1 = c(1,2,3),
  value_2 = c("hello", "there", "everyone")
)[1]
```

```
## $value_1
## [1] 1 2 3
```

A key difference with lists however, is that you can also subset based on the name of the value in the list using the $ operator:

```
list(
  value_1 = c(1,2,3),
  value_2 = c("hello", "there", "everyone")
)$value_1
```

```
## [1] 1 2 3
```

This is equivalent to:

```
list(
  value_1 = c(1,2,3),
  value_2 = c("hello", "there", "everyone")
)[["value_1"]]
```

```
## [1] 1 2 3
```

Another key difference is that lists can, of course, hold recursive values. This means that subsetting a list can return another list, that can also be subsetted and so on:

```
list(
  list_1 = list(
    list_2 = list(
      list_3 = "hello"
    )
  )
)[1][1][1]
```

```
## $list_1
## $list_1$list_2
## $list_1$list_2$list_3
## [1] "hello"
```

And of course, you can do the same thing with the [[]] operator if you only want the value and not the container.

7.3 Matrices

Matrices are two dimension, meaning they can't be subsetted with a single value. Instead, we still use the [] operator, but we provide two values: one for the row and another for the column:

```
matrix(c(1:10), nrow = 5, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    6
```

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

```
matrix(c(1:10), nrow = 5, ncol = 2)[4,1]
```

```
## [1] 4
```

7.4 Dataframes

Dataframes can be subsetted in the same way as matrices (using the `[]` operator). However, dataframes can also be subsetting (like lists), using the `$` operator and the name of the column:

```
data.frame(
  col_1 = c(1,2,3),
  col_2 = c("hello", "there", "everybody")
)$col_1
```

```
## [1] 1 2 3
```

Why does this approach work for dataframes? Well, as I alluded to before, dataframes store columns as lists. But technically, the dataframe itself is also stored as a kind of list, with each column being another entry in that list. So, just like we can subset lists using `$`, we can subset dataframes with it as well because a dataframe is like a fancy list.

Chapter 8

Functions

Being a functional programming language, functions are at the heart of R. We've already used lots of functions in the previous chapters, but now we're going to look in more detail about what a function is.

8.1 Function basics

John Chambers, creator of the S programming language upon which R is based and core member of the R programming language project, said this:

> "To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.
>
> — John Chambers"

For now, we're going to focus on that second statement. What does it mean?

Well, a function is quite simple. It has an input, it does something, and then it gives an output. A really simple example of this is just typing a number into the console and hitting enter. You've given an input, there was a calculation, and now there's an output.

If you're well versed in mathematics, you'll know that functions in maths are the same. $f(x) = 3x$ means that the to get `y`, you take `x` and multiply it by three. In this case, our input is `x`, our bit in the middle is multiplying by three, and then our output is `y`.

If you haven't used functions in mathematics then don't worry. Even by getting this far in the book, you've already used functions loads of times. For example, how do you create a vector? If you remember, you use the `c()` function, which

we know stands for "concatenate". So, every time you've created a vector, you've used a function without even knowing it. The input was whatever you provided in the brackets. The computation was to concatenate everything together. And then the output was the vector.

Similarly, whenever you created a factor or a matrix or a dataframe or whatever, you used a function. You provided an input, there was a computation to change that input, and then you got an output.

As confusing as functions will inevitably become, just try to remember the core of what a function is: There's an input. There's a computation. There's an output.

8.1.1 Functions in R

So more specifically, what do functions looks like in R? Well, a good starting point is that functions are almost always followed by brackets (`()`, not any other type of bracket) when you use them. This helps make it clear what values you're providing as your inputs. For example, the `c()` function, the `data.frame()` function, the `sum()` function are all followed by `()`, which is how you provide your inputs.

I say that nearly almost all functions are followed by `()`, because some aren't. A simple example of this is `+`. `+` is still a function:

```
is.function(`+`)
```

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

```
# the backticks just mean I'm referring
# to the + function without using it
```

But it doesn't have brackets. Instead, we can use a shorthand where we provide the values we want to give to the function either side of it (e.g. `1 + 2`). Importantly however, the logic is exactly the same, and you can still use the `+` like a normal function with brackets:

```
`+`(1,2)
```

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

It's just that this looks a little weird to us, so we often use the shorthand way.

8.1.2 Inputs

We know that to use a function in R, we have to provide inputs*. And we also know that we provide our inputs within the brackets after the function name. But how do we know what values are allowed?

* Technically, sometimes you don't have to provide an input to a function (e.g. `Sys.Date()`, which gives us the current date without putting anything in the brackets). But in the interests of clarity, just imagine that the inputs to these functions are blank rather than that they don't have any input at all.

By typing a `?` followed by the name of the function into the console (e.g. `?length()`), you'll get a help page showing you the input parameters allowed by the function. So if we use `?length()` as an example, the help page tells us that the `length()` function expects one input parameter, `x`, and that needs to be an `R` object. Nice and simple.

In some cases, you'll see a `...` as one of the input parameters. This essentially means that you can provide an indeterminate number of values for that input. I know that sounds confusing, but the `c()` function is a good way of demonstrating this. When you create a vector, you can provide an (essentially) infinite number of values to the function. So the `c()` function basically bundles everything you provide to it into that `...` parameter.

8.1.2.1 Explicit input parameters

If you type `?c()` into the console however, you'll see that there are also some other input parameters: `recursive` and `use.names`. Well Adam, if `...` just bundles everything I provide into a single input, then how do those work? Well this outlines the importance of providing **explicit** input parameters. When we're explicit, we're saying exactly which input parameter we're referring to with each value we provide. And to do this, we just provide the name of the input parameter when we give it. Let's look at the `substr()` function as an example.

The `substr()` function simply returns part of a character string that you provide. So, if I was to type:

```
substr("Hello", 1, 3)
```

```
## [1] "Hel"
```

I get the first to the third characters in the string "hello". With this function call however, I haven't been explicit. Instead, I've just provided the inputs in the order that they're listed in the documentation:

- x
- a character vector
- start
- the first element to be extracted
- stop
- the last element to be extracted

To be explicit, I need to provide the name of the input parameter that I'm referring to when I provide my inputs:

```
substr(x = "Hello", start = 1, stop = 3)
```

```
## [1] "Hel"
```

```
substr(start = 1, stop = 3, x = "Hello")
```

```
## [1] "Hel"
```

Notice how, when I'm being explicit, it doesn't matter what order I provide my inputs in, R knows which value should be mapped to which input parameter.

Also, notice how we're using `=` here and not anything else like `<-`? This is another reason why I suggest not using `=` for assignment: we use `=` when we're providing input parameters and so it's good to keep them separate.

So how does this link back with the `...`? Well, with the `c()` function, every unnamed parameter you provide is bundled into the `...` parameter. To give values for the `recursive` and `use.names` parameters, you'd need to provide them *explicitly* (e.g. `recursive = TRUE`). This will be true of many functions where you see a `...`. If you're not explicit with the parameters that you don't want to be included in the `...`, you're going to have a bad time.

8.1.2.2 Optional input parameters

For many functions, certain parameters have a predefined value that they will default to. This provides a level of flexibility whilst not requiring lines and lines of code for every function call; there's a default value, but you can override it if needed.

Optional parameters are easily distinguished in the documentation of a function because they will a value already assigned to them like this: `use.names = TRUE`.

For instance, when we create a vector using the `c()` function, there are two optional parameters (`recursive` and `use.names`) that already have the values `TRUE` and `FALSE` assigned to them. To override these defaults, we just need to provide a new value to the parameter like this:

```
c(1,10,15, use.names = FALSE)
```

```
## [1]  1 10 15
```

8.1.3 Outputs

First and foremost, in R you can have as many input as you like to a function. However, you can only ever return one *thing*. I say one *thing*, because you can return a list as your output which itself can contain multiple values, but just keep this in mind: **Functions in R have a single return value**.

8.1.3.1 Reassigning outputs

Functions in R do not edit the inputs you provide in place. Instead, they essentially work on copies of the inputs you provide. Here's a quick example:

```
x <- 1
sum(x, 1)
```

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

```
x
```

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

As you can see, when we call the `sum()` function with `x` as an input parameter, the value of `x` stays the same.

If you do want to edit your original value, you just need to reassign the output of the function call back to the variable. I know that sounds complicated, but it's quite simple:

```
x <- 1
x <- sum(x,1)
x
```

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

This works because the right-hand side of the assignment line is executed first. In other words, when the `sum(x,1)` is evaluated, `x` is still equal to one. This makes sense because otherwise it'd be very hard to keep track of what `x` was equal to!

This behaviour (not changing the input parameter value in place) is a major point of difference between functions and what are called methods in other languages. If you're coming from something like Python, you may be used to changing objects through methods: `object.AddNew` or something like that. In R, functions do not change variables in the global environment because they are executed in their own environment. But what's an environment? Well, we'll take a look at that in the environments chapter.

8.1.4 A little more on functions

If you're not too bothered about the slightly more in-depth stuff about function inputs, then feel free to skip this part and move onto the next section.

If you're interested however, there are some specifics about function inputs in R that can be good to know.

Firstly, unlike some other languages, functions do not have a specific data type tied to each input parameter. Any requirements that should be imposed on an input parameter (e.g. it should be numeric) are done by the function creator in

the body of the function. So for instance, when you try to sum character strings, the error you get occurs because of type-checking in the body of the function, not when you provide the input parameters.

Secondly, functions are technically just another object. This means that you can use functions like you would any other object. For instance, some functions will accept other functions as an input parameter. When we move onto the `apply` logic, the `lapply()` function requires a `FUN` parameter that is the function the be applied each time. This doesn't really come into play right now or even at a beginner level, but later on it can be very powerful.

Linked with the idea that functions are just another type of an object, there is an important distinction between `sum` and `sum()`. The first will return the `sum` *object*. That is, not the result of applying inputs to the `sum` function, but the function itself. If you just type the name of the function into the console, it will show you the code for that function:

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

Conversely, `sum()` will attempt to apply the `sum` function to the inputs provided in the brackets.

Again, while this is isn't really a massive point for the moment, it can help some understand the fundamental building blocks of R.

8.2 Creating functions

R and its packages give you access to hundreds of thousands of different functions, all tailored to perform a particular task. Despite this wide array to choose from however, they will always be cases where there isn't a function to do exactly what you need to do. For those of you coming over from Excel, this can often be a serious source of frustration where there isn't an Excel function for you to use and there isn't an easy way to create one without knowing VBA.

R is different. Creating functions can be very simple, and will really change the way you work.

8.2.1 Function structure

If we go back to the beginning of this chapter, we learnt that everything that exists is an object. Functions are no exception, and so we create them like we do all our other objects. There is a slight diversion however. When we define a function, we assign it to an object with the `function` keyword like this:

```
my_first_function <- function() {}
```

Notice how we've got two sets of brackets here. The first (`()`) is where we define our input parameters. The second (`{}`) is where we define the body of our function.

Let's do a simple example. Let's create a function that adds two numbers together:

```
my_sum_function <- function(x, y) {
  x + y
}
```

So in this example, I've defined that when anyone uses the function, they need to provide two input parameters named `x` and `y`. Something that people tend to struggle with is that the names of your input parameters have no implicit meaning. They are just used to reference the value provided in the body of the function and, hopefully, make it clear what kind of thing the user of the function should be providing. This is why for example in some functions that require a dataframe there will be an input parameter called `df` or similar. But importantly, these names are technically just arbitrary.

In the body of the function, we can see that we're just doing something really simple: we're adding `x` and `y` together with `+`.

Once I've run the code to **define** my function, I can then **call** it like I would any other function:

```
my_sum_function(x = 5, y = 6)
```

```
## [1] 11
```

8.2.2 Optional input parameters

When defining your function, you can define optional parameters. These will likely be values where most of the time you need it to be one thing, but there are edge cases where you need it to be something else. Defining optional parameters is really easy; whenever you define your function, just give it a value and that will be its default:

```
add_mostly_2 <- function(x, y = 2){
  x + y
}

add_mostly_2(x = 5)
```

```
## [1] 7
```

```
add_mostly_2(x = 5, y = 3)
```

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

8.2.3 ...

You'll notice a crucial distinction between R's `sum()` function and ours. The base function allows for an indeterminate number of input parameters, whereas we've only allowed 2 (`x` and `y`). This is because the base `sum()` function uses a `...`. This `...` is essentially shorthand for "as many or as few inputs as the user wants to provide". To use the `...`, just add it as in an input parameter:

```
dot_dot_dot_function <- function(x, y, ...) {
}
```

The `...` works particularly well when you might be creating a function that *wraps* around another one. A wrapping function is just a function that makes a call to another one within it, like this:

```
sum_and_add_2 <- function(...){
  sum(...) + 2
}
```

All we're basically doing in the above wrapping around the `sum()` function to add some specific functionality.

By using the `...` here, we can just pass everything that the user provides to the `sum()` function. This means we don't have to worry about copying any input parameters.

8.3 Function environments

To better understand how functions operate, we need to understand how environments and scoping works in R. There's a separate chapter on environments in this book, but we'll briefly look at how functions create environments.

Environments are hierarchical collections of objects. You can think of these environments as going from non-specific to specific. When you define a variable in a script, you are creating that variable in the *global environment*, a very non-specific environment. Functions, however, create their own more specific environment when they are called, but will inherit the values from the more general environments.

This breeds some specific behaviours. For example, say you've written a function that expects two input parameters, `x` and `y`. Well, what would happen if someone

had already defined an `x` and `y` variable in their script? Which value should R use?

Let's see what happens.

```
sum_custom <- function(x,y) {
  x + y
}

x <- 10
y <- 5

sum_custom(x = 1, y = 2)
```

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

In this case, the fact that there is already an `x` and `y` in the global environment doesn't really make much difference. The function creates its own more specific environment when it's called, and it looks for the `x` and `y` variables in here first. It finds them and uses those values (1 and 2).

But what happens if a variable doesn't exist in the more specific function environment? Let's take a look.

```
sum_custom <- function(x,y) {
  x + y + w
}

w <- 5

sum_custom(x = 1, y = 2)
```

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

In this case, the function looks in the specific environment for `w`, but it doesn't exist. The only objects that exist in the function environment are the `x` and `y` that we've provided. So when R doesn't find it in the more specific environment, it looks in the less-specific global environment. It finds, and so it uses the value it finds.

This can be a dangerous thing, so always make sure that you're function is accessing the values you think it is.

So does R work the other way? Does it ever look in a more specific environment? Nope.

```
sum_custom <- function(x,y){
  im_a_sneaky_variable <- 10
  x + y
}
```

```
im_a_sneaky_variable
```

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

Once the function is called, objects in its environment are unaccessible. The long and short of it is, R will start fromk specific environments and then look upwards, never downwards.

8.3.1 'Super' assignment

There will be occassions however when you need to make changes to the global environment. For instance, say you want to increment a counter every time a function is called, regardless of where it's called from. In these cases, you can use the controversial `<<-` operator. This is used as an assignment operator to assign a value to the global environment. Observe...

```
sum_custom <- function(x,y) {
  count <<- count + 1
  x + y
}

count <- 0

sum_custom(1,2)
```

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

```
count
```

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

```
sum_custom(2,3)
```

```
## [1] 5
```

```
count
```

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

Note how when we assign `0` to our `count` variable outside of the function, we don't need to use `<<-`. This is because we're already assigning to the global environment.

Use the `<<-` with care and only assign something to the global environment if you really need to. Otherwise, you may start overwriting variables in your global environment without ever realising it.

Chapter 9

Environments

We looked briefly at environments and how they are used with functions, but let's look at environments in a bit more detail.

9.1 Environment basics

Chapter 10

Exercises

Here are some interactive exercises covering the basic concepts we've looked at in the last X modules.

This exercises are hosted on a Shiny server here