Data Science for Biologists - 5023Y

Philip Leftwich

2021-09-27

Contents

Chapter 1

Introduction

1.1 Approach and style

This book is designed to accompany the module BIO-5023Y for those new to R looking for best practices and tips. So it must be both accessible and succinct. The approach here is to provide just enough text explanation that someone very new to R can apply the code and follow what the code is doing. It is not a comprehensive textbook.

A few other points:

This is a code reference book accompanied by relatively brief examples - not a thorough textbook on R or data science

This is intended to be a living document - optimal R packages for a given task change often and we welcome discussion about which to emphasize in this handbook

Top tips for the course:

DON'T worry if you don't understand everything

DO ask lots of questions!

1.2 Teaching

We have:

- one lecture per week
- one workshop per week

These are both timetabled in-person sessions, and you should check Timetabler for up to-date information on scheduling. However, all lessons can be accessed

remotely through Collaborate, and **everything** you need to complete workshops will be available on this site.

If you feel unwell, or cannot attend a session in-person because you need to self-isolate then don't worry you can access everything, and follow along in real time, or work at your own pace.

Questions/issues/errors can all be posted on our Yammer page.

1.2.1 Workshops

The workshops are the best way to learn, they teach you the practical skills you need to become an R wizard

Figure 1.1: courtesy of Allison Horst

1.3 Introduction to R

R is the name of the programming language itself and RStudio is a convenient interface.], which we will be using throughout the course in order to learn how to organise data, produce accurate data analyses & data visualisations.

Eventually we will also add extra tools like GitHub and RMarkdown for data reproducibility and collaborative programming, check out this short (and very cheesy) intro video.], which are collaboration and version control systems that we will be using throughout the course. More on this in future weeks.

By the end of this module I hope you will have the tools to confidently analyze real data, make informative and beautiful data visuals, and be able to analyse lots of different types of data.

The taught content this autumn will be given to you in several **worksheets**, these will be added to this dynamic webpage each week.

1.4 Getting around on RStudio

All of our sessions will run on cloud-based software. All you have to do is make a free account, and join our Workspace `BIO-5023Y` the sharing link is here.

Once you are signed up - you will see that there are two `Spaces`

- Your workspace
- BIO-5023Y

Make sure you are working in the class workspace - there is a limit to the hours/month on your workspace, so all assigments and project work should take place in the `BIO-5023Y space`.

Watch these short explainer videos to get used to navigating the environment.

1.4.1 An intro to RStudio

RStudio

> Note - people often mix up R and RStudio. R is the programming language (the engine), RStudio is a handy interface/wrapper that makes things a bit easier to use.

1.4.2 Using R Studio Cloud

RStudio Cloud works in exactly the same way as RStudio, but means you don't have to download any software. You can access the hosted cloud server and your projects through any browser connection (Chrome works best), from any computer.

1.5 Reading

There are lots of useful books and online resources to help develop and improve your R knowledge. Throughout this webpage I will be adding useful resources for you.

The core textbook you might want to bookmark is R for Data Science (Hadley Wickham, 2020) but we will add others throughout the course, and their is a bibliography at the end which collects everything together!

1.6 Get Help!

There are a **lot** of sources of information about using R out there. Here are a few helpful places to get help when you have an issue, or just to learn more

- The R help system itself - we cover this in Week one Error
- Vignettes - type `browseVignettes()` into the console and hit Enter, a list of available vignettes for all the packages we have will be displayed
- Cheat Sheets - available at RStudio.com. Most common packages have an associate cheat sheet covering the basics of how to use them. Download/bookmark ones we will use commonly such as `ggplot2`, `Data transformation with dplyr`, `Data tidying with tidyr` & `Data import`.
- Google - I use Google constantly, because I continually forget how to do even basic tasks. If I want to remind myself how to round a number, I might type something like `R round number` - if I am using a particular package I should include that in the search term as well
- Ask for help - If you are stuck, getting an error message, can't think what to do next, then ask someone. It could be me, it could be a classmate. When you do this it is very important that you **show the code**, include the **error message**. "This doesn't work" is not helpful. "Here is my code, this is the data I am using, I want it to do X, and here's the problem I get".

  Note - It may be daunting to send your code to someone for help. It is natural and common to feel apprehensive, or to think that your code is really bad. I still feel the same! But we learn when we share our mistakes, and eventually you will find it funny when you look back on your early mistakes, or laugh about the mistakes you still occasionally make!

Chapter 2

Getting to know R - Week One

Go to RStudio Cloud and enter the Project labelled `Week One` - this will clone the project and provide you with your own workspace.

Follow the instructions below to get used to the R command line, and how R works as a language.

2.1 Your first R command

In the RStudio pane, navigate to the console (bottom left) and `type or copy` the below it should appear at the >

Hit Enter on your keyboard.

```
10 + 20
```

You should now be looking at the below:

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

The first line shows the request you made to R, the next line is R's response

You didn't type the `>` symbol: that's just the R command prompt and isn't part of the actual command.

It's important to understand how the output is formatted. 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.

You can think of `[1] 30` as if R were saying "the answer to the 1st question you asked is 30".

2.1.1 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`. Try it for yourself and replicate this error message:

```
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.

2.1.2 More simple arithmetic

One of the best ways to get to know R is to play with it, it's pretty difficult to break it so don't worry too much. Type whatever you want into to the console and see what happens.

If the last line of your console looks like this

```
> 10+
+
```

and there's a blinking cursor next to the plus sign. 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, type `20` and hit enter, then it finishes the command:

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

Alternatively hit escape, and R will forget what you were trying to do and return to a blank line.

2.1.3 Try some maths

```
1+7
```

```
13-10
```

```
4*6
```

```
12/3
```

Raise a number to the power of another

```
5^4
```

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. So, the 4th power of 5 is calculated like this:

$$5^4 = 5 \times 5 \times 5 \times 5$$

2.1.4 Perform some combos

Perform some mathematical combos, noting that the order in which R performs calculations is the standard one.

That is, first calculate things inside **B**rackets `()`, then calculate **O**rders of (exponents) `^`, then **D**ivision `/` and **M**ultiplication `*`, then **A**ddition `+` and **S**ubtraction `-`.

Notice the different outputs of these two commands.

```
3^2-5/2
```

```
(3^2-5)/2
```

Similarly if we want to raise a number to a fraction, we need to surround the fraction with parentheses ()

```
16^1/2
```

```
16^(1/2)
```

The first one calculates 16 raised to the power of 1, then divided this answer by two. The second one raises 16 to the power of a half. A big difference in the output.

> **Note - While the cursor is in the console, you can press the up arrow to see all your previous commands. You can run them again, or edit them. Later on we will look at scripts, as an essential way to re-use, store and edit commands.

2.1.5 Storing outputs

With simple questions like the ones above we are happy to just see the answer, but our quesitons are often more complex than this. If we need to take multiple steps, we benefit from being able to store our answers and recall them for use in later steps. This is very simple to do we can *assign* outputs to a name:

```
a <- 1+2
```

This literally means please *assign* the value of `1+2` to the name `a`. We use the **assignment operator** `<-` to make this assignment.

> Note the shortcut key for <- is Alt + - (Windows) or Option + - (Mac)

If you perform this action you should be able to do two things

- You should be able to see that in the top right-hand pane in the `Environment` tab their is now an `object` called a with the value of 3.
- You should be able to look at what a is by typing it into your Console and pressing Enter

```
a
```

```
> a
[1] 3
```

You can now call this object at any time during your R session and perform calculations with it.

```
2 * a
```

```
[1] 6
```

What happens if we assign a value to a named object that already exists in our R environment??? for example

```
a <- 10
a
```

```
[1] 10
```

You should see that the previous assignment is lost, *gone forever* and has been replaced by the new value.

We can assign lots of things to objects, and use them in calculations to build more objects.

```
b <- 5
c <- a + b
```

Note that if you now change the value of b, the value of c does *not* change. Objects are totally independent from each other once they are made

```
b <- 7
b
c
```

Look at the environment tab again - you should see it's starting to fill up now!

> Note - RStudio will by default save the objects in its memory when you close a session. These will then be there the next time you

logon. It might seem nice to be able to close things down and pick up where you left off, but its actually quite dangerous. It's messy, and can cause lots of problems when we work with scripts later, so don't do this!!! To stop RStudio from saving objects by default go to the Preferences option and change "Save workspace to .RData on exit" to "Never". Instead we are going to learn how to use scripts to quickly re-run analyses we have been working on.

2.1.6 Choosing names

- Use informative variable names. As a general rule, using meaningful names like `orange` and `apple` is preferred over arbitrary ones like `variable1` and `variable2`. Otherwise it's very hard to remember what the contents of different variables actually are.
- Use short variable names. Typing is a pain and no-one likes doing it. So we much prefer to use a name like `apple` over a name like `pink_lady_apple`.
- Use one of the conventional naming styles for multi-word variable names. R only lets you use certain things as **legal** names. Legal names must start with a letter **not** a number, which can then be followed by a sequence of letters, numbers, ., or _. R does not like using spaces. Upper and lower case names are allowed, but R is case sensitive so `Apple` and `apple` are different.
- My favourite naming convention is `snake_case` short, lower case only, spaces between words are separated with a _. It's easy to read and easy to remember.

2.2 Writing scripts

Until now we have been typing words directly into the Console. This is fine for short/simple calculations - but as soon as we have a more complex, multi-step process this becomes time consuming, error-prone and *boring*. **Scripts** are a document containing all of your commands (in the order you want them to run), they are *repeatable, shareable, annotated records of what you have done*. In short they are incredibly useful - and a big step towards **open** and **reproducible** research.

To create a script go to File > New File > R Script.

This will open a pane in the top-left of RStudio with a tab name of `Untitled1`. In your new script, type some of the basic arithmetic and assignment commands you used previously. When you write a script, make sure it has all of the commands you need to complete your analysis, *in the order you want them to run.*

2.2.1 Commenting on scripts

Annotating your instructions provides yourself and others insights into why you are doing what you are doing. This is a vital aspect of a robust and reproducible workflow. And when you come back to a script, one week, one month or one year from now you will often wonder what a command was for. It is very, very useful to make notes for yourself, and its useful in case anyone else will ever read your script. Make these comments helpful they are for humans to read.

In R we signal a comment with the # key. Everything in the line after a # is ignored by R and won't be treated as a command. You should see that it is marked in a different colour in your script.

Put the following comment in your script. Try adding a few comments to your previous lines of code

```
# I really love R
```

2.2.2 Running your script

To run the commands from your script, we need to get it into the Console. You could select and copy/paste this into the Console. But there are a couple of faster shortcuts.

- Hit the Run button in the top right of the script pane. Pressing this will run the line of code the cursor is sitting on.
- Pressing Ctrl+Enter will do the same thing as hitting the Run button
- If you want to run the whole script in one go then press Ctrl+A then either click Run or press Ctrl+Enter

2.2.3 Saving your script

Our script now contains code and comments from our first workshop. We need to save it.

Alongside our data, our script is the most precious part of our analysis. We don't need to save anything else, any outputs etc. because our script can always be used to generate everything again. Note the colour of the script - the name changes colour when we have unsaved changes. Press the Save button or go to File > Save as. Give the File a sensible name like "Simple commands in R" and in the bottom right pane under `Files` you should now be able to see your saved script.

You could now safely quit R, and when you log on next time to this project, your script will be waiting for you.

2.3 Error

Things will go wrong eventually, they always do…

R is *very* pedantic, even the smallest typo can result in failure and typos are impossilbe to avoid. So we will make mistakes. One type of mistake we will make is an **error**. The code fails to run. The most common causes for an error are:

- typos
- missing commas
- missing brackets

There's nothing wrong with making *lots* of errors. The trick is not to panic or get frustrated, but to read the error message and our script carefully and start to *debug*…

… and sometimes we need to walk away and come back later!

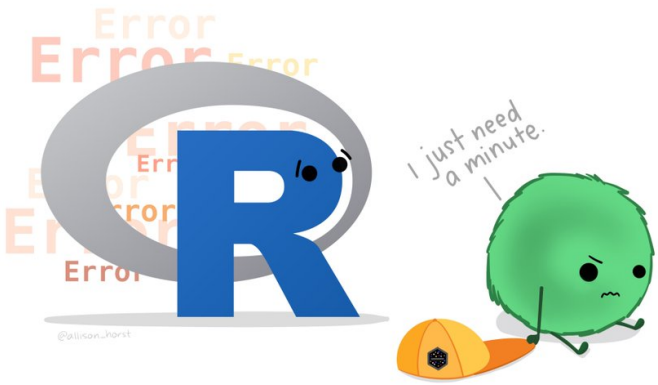

Figure 2.1: courtesy of Allison Horst

2.4 Functions

Functions are the tools of R. Each one helps us to do a different task.

Take for example the function that we use to round a number to a certain number of digits - this function is called `round`

Here's an example

```
round(x  = 2.4326782647, digits = 2)
```

We start the command with the function name `round`. The name is followed by parentheses (). Within these we place the *arguments* for the function, each of which is separated by a comma.

The arguments

- x = 2.4326782647
- digits = 2

Arguments are the information we give to a function. These arguments are in the form `name = value` the name specifies the argument, and the value is what we are providing. That is the first argument x is the number we would like to round, it has a value of 2.4326782647. The second argument digits is how we would like the number to be rounded and we specify 2.

Ok put the above commmand in your script and add a comment with # as to what you are doing.

2.4.1 Storing the output of functions

What if we need the answer from a function in a later calculation. The answer is to use the assignment operator again.

Can you work out what is going on here? If so copy this into your R script and a #comment next to each line.

```
number_of_digits <- 2
my_number <- 2.4326782647
rounded_number <- round(x  = 2.4326782647,
                        digits = 2)
```

2.4.2 More fun with functions

Check this out

```
round(2.4326782647, 2)
```

We don't *have* to give the names of arguments for a function to still work. This works because the function `round` expects us to give the number value first, and the argument for rounding digits second. *But* this assumes we know the expected ordering within a function, this might be the case for functions we use a lot. If you give arguments their proper names *then* you can actually introduce them in any order you want.

Try this:

```
round(digits = 2, x  = 2.4326782647)
```

But this gives a different answer

```
round(2, 2.4326782647)
```

Are you happy with what is happening here? naming arguments overrides the position defaults

Ok what about this?

```
round(2.4326782647)
```

We didn't specify how many digits to round to, but we still got an answer. That's because in many functions arguments have `defaults` - the default argument here is digits = 0. So we don't have to specify the argument if we are happy for round to produce whole numbers.

How do we know argument orders and defaults? Well we get to know how a lot of functions work through practice, but we can also use the inbuilt R help. This is a function - but now we specify the name of another function to provide a help menu.

```
help(round)
```

2.5 Packages

An R package is a container for various things including functions and data. These make it easy to do very complicate protocols by using custom-built functions. Later we will see how we can write our own simple functions.

On RStudio Cloud I have already installed several add-on packages, all we need to do is use a simple function to load these packages into our workspace. Once this is complete we will have access to all the custom functions they contain.

Let's try that now:

```
library(ggplot2)
library(palmerpenguins)
```

> Errors part 2 Another common source of errors is to call a function that is part of a package but forgetting to load the package. If R says something like "Error in function-name" then most likely

the function was misspelled or the package containing the function hasn't been loaded.

Packages are a lot like new apps extending the functionality of what your phone can do. To use the functionalities of a package they must be loaded *before* we call on the funcitons or data they contain. So the most sensible place to put library calls for packages is at the very **top** of our script. So let's do that now.

2.6 My first data visualisation

Let's run our first data visualisation using the functions and data we have now loaded - this produces a plot using functions from the `ggplot2` package (Wickham et al. (2020)) and data from the `palmerpenguins` (Horst et al. (2020)) package. Use the # comments to add notes on what you are using each package for in your script.

Using these functions we can write a simple line of code to produce a figure. We specify the data source, the variables to be used for the x and y axis and then the type of visual object to produce, colouring them by the species.

Copy this into your console and hit Enter.

```
ggplot(data = penguins,aes(x = bill_length_mm, y = bill_depth_mm)) + geom_point(aes(colour=specie
```

The above command can also be written as below, its in a longer style with each new line for each argument in the function. This style can be easier to read, and makes it easier to write comments with #. Copy this longer command into your `script` then run it by either highlighting the entire command or placing the cursor in the first line and then hit Run or Ctrl+Enter.

```
ggplot(data = penguins, # calls ggplot function, data is penguins
       aes(x = bill_length_mm, # sets x axis as bill length
           y = bill_depth_mm)) + # sets y axis value as bill depth
    geom_point(aes(colour=species)) # plot points coloured by penguin species
```

2.7 Quitting

- Make sure you have saved any changes to your R script - that's all you need to make sure you've done!
- If you want me to take a look at your script let me know
- If you spotted any mistakes or errors let me know
- Close your RStudio Cloud Browser

- Go to Blackboard to complete a short quiz!

*R Cheat Sheets

Bibliography

Hadley Wickham, G. G. (2020). *R for Data Science*. Chapman and Hall/CRC, Boca Raton, Florida.

Horst, A., Hill, A., and Gorman, K. (2020). *palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. R package version 0.1.0.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2020). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.3.2.