NZILBB R and Open Science Workshops

Joshua Wilson Black

2024-04-26

Table of contents

Preface

This 'book' will slowly evolve and add contributors as the NZILBB R and Open Science workshops develop. The hope is that this resource will allow people to study at their own pace.

Introduction

Our first order of business is a series of interlinked introductory sessions.

We will highlight *linguistic examples*.

Open Science

NZILBB values *Open Science*.

Reproducibility

Replicability

There is a vast literature on the *replication crisis*.

One aspect of the solution is to share …

Preregistration

In addition to sharing code and data *after the fact*, it is important to be able to register hypotheses and predictions before data collection and, if that is not possible, at least before analysis.

R

Why Programming?

Things can go wrong with spreadsheets

https://eusprig.org/research-info/horror-stories/ (via Winter)

We need to **integrate our work with others**.

This leads to a situation where science is 'amateur software development' (in the words of Richard McElreath — see: https://www.youtube.com/watch?v=8qzVV7eEiaI).

continuous integration. teams of programmers. (This is why we will look with *git*).

testing as the majority of code.

(Also uses kitchen metaphor)

We have to get more *professional* for the sake of science.

(There's also a discussion of p-hacking here — some good examples of bad unintentional mistakes)

- Need to replicate *your own work* as a minimum

professional standards as the underlying thought.

Links 'data carpentry'.

— vs EXCEL (not a tool designed for science and especially for quality control.)

famous case of Gene names being changed (theverge.com/2020/8/6/21355674 (from mcelreath vid))

Not taught: organising data, curating data, testing data, manage distributed contributions, logically connect hypotheses to data analysis.

Also: https://www.youtube.com/watch?v=cpbtcsGE0OA

Why R?

Here's some reasons to use R, from better to worse.

- **
- **Nationalism:** R was created here in Aotearoa New Zealand. Don't get too excited though, it was made in Tāmaki Makaurau.

There are other tools out there. A lot of work in computational linguistics uses the Python programming language (). For projects interacting with contemporary Artificial Intelligence it may be more sensible to use Python than R. Some phoneticians use Matlab.

Resources

The content of these workshops have been strongly influenced by:

(Winter 2019)

Part I

Foundations

1 Getting Started

1.1 Install R and RStudio

R is a programming language. RStudio is a piece of software for *interacting* with R.

You don't *have* to use RStudio in order to use R, but we will assume you are using it in these workshops.[1].

To install R and RStudio on your own device follow the steps at https://posit.co/download/rstudio-desktop/.

To install R and RStudio on a University of Canterbury device:

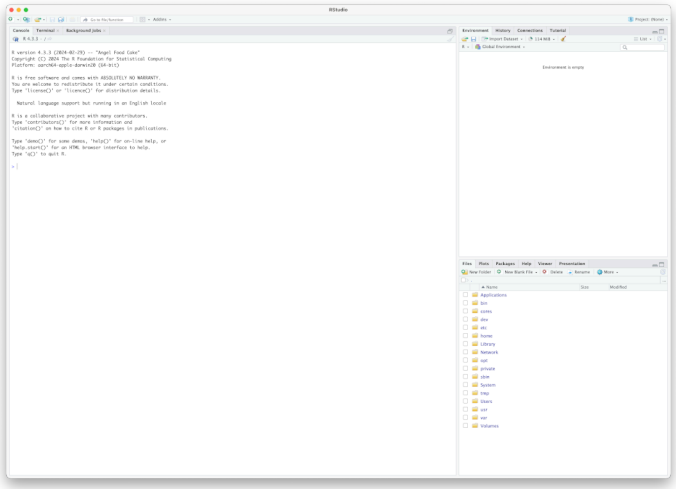
- Windows: open the "Software Center", search for RStudio, and press the install button.
- Mac: open "UC Self Service", search for RStudio, and press the install button.

Installing RStudio on a University of Canterbury device will also install R.

1.2 Open RStudio

Open RStudio. You should see something which looks like this:

[1]For alternatives which you might explore see Section 1.7



The RStudio interface has four primary 'panes'. Only three of these will be visible when you first open RStudio. The largest pane is the **console** pane. It is usually on the bottom left of the RStudio window, but currently takes up the entire left side. We also see the **environment** pane at the top right and the **output** pane at the bottom right.

The **console** pane should have a message telling you what version of R you are using and the platform you are on (i.e. your hardware and operating system). This is what you would see if you opened R by itself.[2]

The **environment** pane should be empty. You will see multiple tabs across the top of this pane. The environment tab will allow us to see the data which we are working with at a given time. At this stage, you *may* see a tab labelled 'Tutorial'. I'll tell you how to use this later (Section 1.6).

The **output** tab will start by showing you a list of files on your computer. This is useful for finding and manipulating files as you would in any file browser. In future, it is also where plots we make will appear and any help files that we look up.

1.3 Interact with R in the Console

Let's start to interact with R in the **console** pane. You will now see code to enter in to the console after the `>` and then press 'Enter' or 'Return'. The result of running this code will appear below the block. Make sure you get the same output.

[2]Try this. You should find a shortcut to open R in the Start menu on Windows or the Launchpad in macOS. On Linux or macOS you can also open a terminal window, type 'r', and press enter.

1.3.1 Basic arithmetic

We'll start with some basic arithmetic. We add two numbers together by writing the first number, the `+` sign, and the second number.

```
1 + 1
```

```
[1] 2
```

The other basic arithmetic operators work in the same way.

```
500 - 49
```

```
[1] 451
```

We use `*` for multiplication. We enter real numbers by using a decimal point.

```
43 * 6.4
```

```
[1] 275.2
```

For exponentiation we use `^` (usually, shift + 6).

```
924^5
```

```
[1] 6.735345e+14
```

The output given here is in scientific notation. This is a way to make very very small and very very large numbers easier to write. To convert from scientific notation to regular digits, multiple the number which appears *before* the `e` by 10 to the power of the number *after* the `e`. In this case, we take the number 6.735345 and multiply it by 10^{14} to get $673,534,500,000,000$. That is, six hundred seventy-three trillion and a bit. According to Wikipedia, this is something like the total number of cells in six and a half adult humans and a bit fewer than the number of ants on Earth.

There are a few different operators associated with division. Usually, you will want to use `/`. e.g.:

```
43 / 7
```

```
[1] 6.142857
```

Sometimes, it is useful to get the integer component on the answer or the remainder. If we want the integer, we use **%/%**:

```
43 %/% 7
```

```
[1] 6
```

If we want the remainder, we use:

```
43 %% 7
```

```
[1] 1
```

That is, if we divide 43 by 7, we get 6 groups of 7, with 1 remaining.

Computer programming requires attention to minor details of punctuation and spacing. Hours can be spent trying to discover why code is not working, only to discover a missing comma. This is especially true in the early stages of learning, where error messages are particularly confusing.[3]

It is worth knowing when you can add spaces and when you can't. The spaces in the code above between the numbers and the arithmetic operators are not necessary. So, for instance, you could write:

```
43/7
```

```
[1] 6.142857
```

In fact, you can add an arbitraty amount of spaces here.

```
34  /    2
```

```
[1] 17
```

[3] They never *stop* being confusing, of course. The words of Gred LeMond regarding cycling apply here: "it never gets easier; you just go faster".

The only reason to prefer one of the other is readability. This raises the issue of code *style*, which we will discuss in future chapters. You might want to look at this page: https://style.tidyverse.org/.[4] Note that, above, there wasn't a space in `924^5`—this is a style convention for `^` and some other ('high precedence') operators which we will encounter later.

1.3.2 Vectors and Variables

We work with large collections of experimental data or values derived from corpora. But the commands we've looked at above only deal with one or two numbers at a time. The simplest structure for dealing with more than one value is a vector.

We create vectors using the function `c()`. The `c()` function *combines* values in to a vector.

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

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

The `[1]` you see in the output is followed by the first element of the vector. If you print out a very long vector you will see numbers other than 1 inside the square brackets. For instance:

```
1:50
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

The `:` operator produces a vector from the first number to the second number (inclusive) in steps of 1. Note that there is a second number in square brackets. Exactly *which* number it is will vary according to the system you are using. For me, currently editing this text in RStudio, it is `[36]`.

The *elements* of a vector are of the same *type*. We'll talk about types more later. For now, just note that a number is a different kind of thing from a string of characters. So, what happens if we try to mix numbers and strings in a vector?

```
c(1, 2, 3, "dog")
```

```
[1] "1"   "2"   "3"   "dog"
```

[4] Beware, you may find intemperately held beliefs if you start engaging in discussions of code style on the internet!

R hasn't explicitly complained, but it has done something without telling you. The numbers we entered now have quotation marks around them. They have been turned in to strings. Keep an eye out for quotation marks — sometimes you might think you are dealing with numbers, but really you are dealing with strings. This is a common problem when loading up your own data.

Why worry? Your code likely won't work if you have strings rather than numbers. For instance, you can't apply arithmetic operators to strings.

```
"1" + "2"
```

```
Error in "1" + "2": non-numeric argument to binary operator
```

The above is the first *error message* you've seen in this course. You will see many more. The error message is telling you that what you are doing does not work on anything but numbers.

Vectors can also be used for arithmetic. Under the hood, statistics is mostly arithmetic with collections of vectors. How are these arithmetic operations implemented in R?

The simplist case is when we use a vector and a single number, as follows

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

```
[1] 2 4 6 8
```

Each element of the vector has been multiplied by 2. The same is true of addition, division, and subtraction. These are 'element-wise' operations.

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

```
[1] 3.00 1.50 1.00 0.75
```

This also works with two vectors. For instance:

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

```
[1]  1  4  9 16
```

Here we get the first elements multiplied together, then the second, then the third, and so on. If one vector is shorter than the other, is will be 'recycled' to match the longer vector:

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

```
[1] 1 4 3 8
```

You do not want to be entering the same vector over and over again. It's a good thing we can assign objects to *names*. These *names* are variables in the sense that they can take different values at different times (like a variable in algebra).[5]

To *assign* an object to a name, we use `<-`. For instance:

```
my_cool_vector <- c(6, 9, 4, 5, 2, 2)
```

Now the name `my_cool_vector` is associated with the vector `c(6, 9, 4, 5, 2, 2)`. If you look to the top right of the RStudio window you should now see this variable in your environment pane. The name will be on the right and the value on the left.

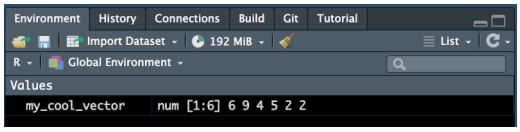


Figure 1.1: Our cool vector in the RStudio environment pane. Your screen may look a little different.

> **Warning**
>
> In most cases you can use `=` to assign an object to a name. This may seem more natural to you if you come from other programming languages. The convention is to use `<-`. Sometimes `=` takes on a different function, but `<-` is *always* assignment of an object to a name.

We can now manipulate the object *using* the name. For instance:

```
4 * my_cool_vector
```

```
[1] 24 36 16 20  8  8
```

To see what is associated with a vector

Naming variables is serious business. It is important to know what you *could* do and what you *should* do. First, anything placed within backticks (“`) can function as a variable name. If you have a chaotic temprament, you might decide to use a names like this:

[5] There's a more accurate computer science story here, but we'll avoid it for now.

```
# Eldritch variable
`ṭ ẹ̄ t̄ōw̄ēr̄s̄ ọf C̄ār̄cōs̄ā ŗōs b̄ẹh̄īn̄d̄ ţh̄ē, m̄ōọn̄.` <- 10

# Spooky variable
` ` <- 5
```

And you can even do some maths with them:

```
`ṭ ẹ̄ t̄ōw̄ēr̄s̄ ọf C̄ār̄cōs̄ā ŗōs b̄ẹh̄īn̄d̄ ţh̄ē, m̄ōọn̄.` + ` `
```

```
[1] 15
```

Do not do this.

If you try this without the backticks, you will get the following:

```
  <- 5
```

```
Error: <text>:1:1: unexpected input
1:
    ^
```

Setting aside the use of backticks, there are strict rules for names:

- A name must consist of letters, digits, `.`, and `_`.
- It **cannot** begin with digits or `_` ().
- It cannot come from a list of reserved words (e.g. `TRUE` — these names have important roles in R and can't be overridden.)

> **Note**
>
> What counts as a 'letter' varies by operating system and local settings (your 'locale'). The recommendation from Hadley Wickham is that you only use ASCII letters (i.e., avoid use of accents).[6]
> One local reason you might want to use non-ASCII characters is if you want to use te reo Māori with macrons for your variable names. This might be appropriate for a particular project (the question is always who you want to share your code with). Pretty much anyone using a modern opperating system should be able to use your code. You may decide that the small risk of incompatibility is worth it.

[6] See (https://adv-r.hadley.nz/names-values.html)

Why did I even telling you about the backticks? They often appear in practice as a result of importing data from a spreadsheet. Usually they appear because the column names in the spreadsheet have spaces in them. One of the first things to do when tidying up data loading from a spreadsheet is to change the names.

1.3.3 Exercises

(**winterStatisticsLinguistsIntroduction2019?**)

1.4 Start an R Script

If we just used R in the console, we would be in no better position than if we just used Excel or another spreadsheet programme. We want to be able to retrace our steps *and* for other researchers to be able to retrace our steps.

Open a new R script by ….

COMMENTS

1.4.1 Matrices and Dataframes

1.4.2 Functions and Packages

1.4.3 Install and Use a Package

One of the great advantages of R is that it has a large community of developers making 'packages' to share their code. Packages allow us to cumulatively build on each others work and to do things *quickly* which might otherwise take a lot of time and statistical knowledge to achieve.

Well start with a silly package: `cowsay`.[7]

[7] I first became aware of this package through a tutorial produced by Kevin Watson.

1.5 Modify RStudio Defaults

There are many useful options which you might want to change to improve your RStudio experience.

First, there are a series of aesthetic options. …

Second, I'm going to assert that you should change some settings now without properly explaining myself. I trust that the reasons for these changes should become clear soon.

1.6 Additional Resources

Try the tutorials from `learnr`. RStudio contains inbuilt tutorials, which you may find useful!

These also provide you with an opportunity to

It's always a good idea to have a look at the RStudio documentation: https://docs.posit.co/ide/user/.

Chapter … winter.

If you want to be a bit more technical about the syntax of R, have a look the documentation here: https://www.stat.auckland.ac.nz/~paul/ItDT/HTML/node75.html; or in even more detail here: https://cran.r-project.org/doc/manuals/r-devel/R-lang.html

1.7 Alternatives to RStudio

You can write R code in any text editor which you like. Popular options with more or less integration of R include:

- Visual Studio Code
- ESS (i.e. *E*macs *S*peaks *S*tatistics).

We won't discuss these alternatives in these workshops. The most likely reason for you to use one of them is that you are already a keen programmer with strong preferences concerning your tools.

Part II

Additional Topics

2 RStudio Server

Academics at the University of Canterbury can use the RStudio Server.

As of this writing writing… [details]

Current details

References

Winter, Bodo. 2019. *Statistics for Linguists: An Introduction Using r*. New York: Routledge. https://doi.org/10.4324/9781315165547.