Coding togetheR

Alistair Bailey

May 01 2019
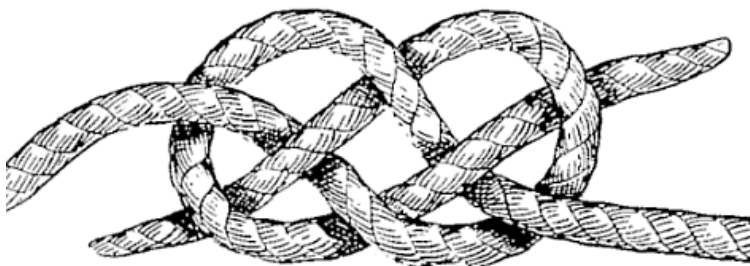
Contents

Welcome

A Carrick bend:[1] The Carrick bend is a type of sailor's knot used for joining two lines.

Coding togetheR is a series of collaborative workshops to teach foundational R coding and data science skills at the University of Southampton in 2019. This book contains the materials covered over eight, two hour sessions.

The workshops are for anyone at the University of Southampton with data to analyse and who is struggling with their current tools. This series of eight weekly two hour workshops provides an introduction to working with data using R in a supported environment. Unlike traditional lessons, we all code together with the emphasis on participants learning by doing and by helping each other.

These materials are a mash-up of my own and many others. I've endeavoured to credit everyone appropriately, but please message me[2] if I've messed up and I'll correct it. The main sources used here are: R for data science (R4DS)[3], the R4DS community[4], the Carpentries[5], Hands on Programming in R[6], swiRlstats[7] and Teaching Tech togetheR[8].

It was written using R (R Core Team, 2019) in RStudio (RStudio Team, 2018) using the bookdown package (Xie, 2018).

To follow these materials you will need an up to date version of R (R Core Team, 2019) and RStudio (RStudio Team, 2018). This may require requesting permission to install software from Isolutions if you have a University laptop.

---

[1]https://en.wikipedia.org/wiki/File:Carrick-bend-Guten-Verrill-modified.png
[2]https://ab604.uk/
[3]https://r4ds.had.co.nz/
[4]https://www.rfordatasci.com/
[5]https://carpentries.org/
[6]https://rstudio-education.github.io/hopr/
[7]https://swirlstats.com/
[8]http://teachtogether.tech/en/

R and RStudio

If you are new to R, then the first thing to know is that R is a programming language and RStudio is a program for working with R called an integrated development environment (IDE). You can use R without RStudio, but not the other way around. Further details in Chapter 1.

Download R here[9] and Download RStudio Desktop here[10].

These materials were generated using R version 3.6.

Once you've installed R and RStudio, you'll also need a few R packages. Packages are collections of functions.

Open RStudio and put the code below into the `Console` window and press `Enter` to install the `tidyverse,dslabs,janitor` and `here` packages.

```r
install.packages(c("tidyverse","dslabs","janitor","here"))
```

Who is coding togetheR for?

Following the lesson design process of (Wilson, 2018):

Arshad

As a PhD student in ecology Arshad doesn't have any formal coding training, but is gathering field data about bird populations. He is daunted by the prospect of learning to code. These lessons will introduce Arshad to coding by showing him how to organise and automate analysis of his data.

Jenny

As a post doctoral researcher in gerontology Jenny has experience of research, but is unsatisfied with her current spreadsheet tools for analysing data. These lessons will show her how to write code to analyse spreadsheets.

Lin

As a principal investigator Lin has experience using MATLAB, but has not used R and would like to know more about it. These lessons will introduce Lin to R syntax and RStudio workflows.

Code of conduct

Coding togetheR is for everyone, and in order to make it a supportive and inclusive environment we subscribe to the Carpentries Code of Conduct[11]. Please follow the link for details.

In a nutshell, expected behaviour is as follows:

- Use welcoming and inclusive language
- Be respectful of different viewpoints and experiences
- Gracefully accept constructive criticism
- Focus on what is best for the community
- Show courtesy and respect towards other community members

Participants who violate the code of conduct, will be asked to stop immediately and if necessary asked to leave the workshop and incidents reported as per University guidance on inappropriate behaviour[12].

---

[9] https://cran.r-project.org/
[10] https://www.rstudio.com/products/rstudio/download/
[11] https://docs.carpentries.org/topic_folders/policies/code-of-conduct.html
[12] https://www.southampton.ac.uk/studentservices/need-help/student-discipline/staff-information.page

Chapter 1

Getting started in R and RStudio

By the end of this chapter are you will:

- understand how to install packages in RStudio.
- know how to get help when you are stuck.
- have set-up your first R project.
- understand the atoms of R and how to use them to build data frames.
- understand how to assign objects in R.
- have created a plot using the `ggplot2` package.
- have written outputs from R to files.

1.1 Coding is for everyone

If, when faced with the thought of starting to learning to code you feel like this:

Then hopefully by the end of these materials, you'll feel a bit more like this:

And if you like that, there is more at R for cats[3].

1.2 A little background and philosophy

"There are only two kinds of languages: the ones people complain about and the ones nobody uses"

Bjarne Stroustrup, the inventor C++

1.2.1 What is R?

R is a programming language that follows the philosophy laid down by it's predecessor S. The philosophy being that users begin in an interactive environment where they don't consciously think of themselves as programming. It was created in 1993, and documented in (Ihaka and Gentleman, 1996).

Reasons R has become popular include that it is both open source and cross platform, and that it has broad functionality, from the analysis of data and creating powerful graphical visualisations and web apps.

Like all languages though it has limitations, for example the syntax is initially confusing.

---

[3]https://www.rforcats.net/

Figure 1.1: Imposter syndrome[1].
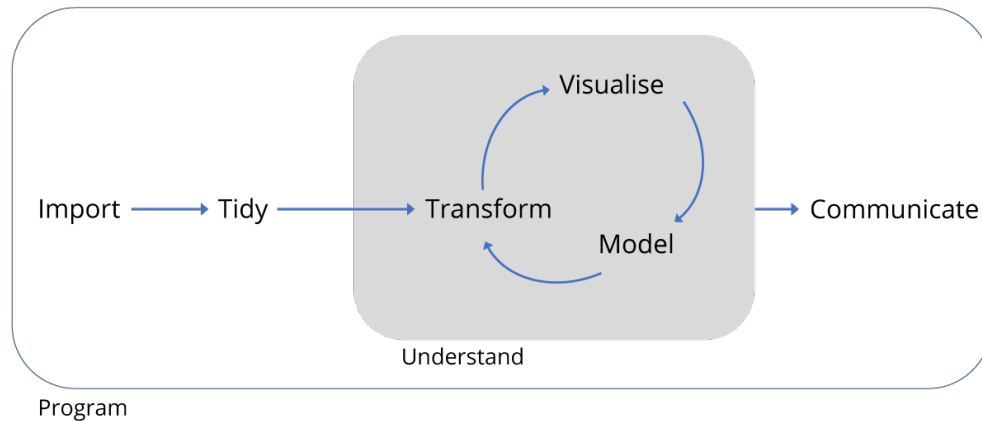

Figure 1.2: R cat[2]

Figure 1.3: Data project workflow.

Users and developers of R have in recent years sought to develop an inclusive and welcoming community which can found on twitter `#rstats` or through RStudio Community[4]. There are many useR groups, including groups seeking to promote diversity such as R-Ladies[5]: Jumping Rivers maintains a list[6].

1.2.2 Why learn to code at all?

In terms of the philosophy of learning to code:

1. The primary motivation for using tools such as R is to get more done, in less time and with less pain.

2. And the overall aim is to *understand and communicate* findings from our data.

3. Additionally, as per Greg Wilson's description of his motivation for teaching[7], if we're going to help make the world a better place, a bit of coding is likely to be key tool in your kit.

As shown in Figure 1.3 of typical data analysis workflow, to acheive this aim we need to learn tools that enable us to perform the fundamental tasks of tasks of importing, tidying and often transforming the data. Transformation means for example, selecting a subset of the data to work with, or calculating the mean of a set of observations.

1.2.3 A little goes a long way

Returning to our cat friend in Figure 1.2, one doesn't need to be an expert programmer to find coding useful. As illustrated in Figure 1.4 there is a whole spectrum of code users from practioners who are focused on applying some R to their specific problems, to those programmers who develop the R language itself. In reality one may move around on that spectrum as ones interests change over time.

1.3 RStudio

Let's begin by learning about RStudio[8], the Integrated Development Environment (IDE).

R is the language and RStudio is software created to facilitate our use of R. They are installed separately. You don't need RStudio to use R, but you do need R to used RStudio.

---

[4]https://community.rstudio.com/
[5]https://rladies.org/about-us/
[6]https://jumpingrivers.github.io/meetingsR/
[7]http://third-bit.com/2019/01/30/why-i-teach.html
[8]https://www.rstudio.com/

Practitioner   **Programmer**

Analyzes Data   **Me**   Explicitly programs R

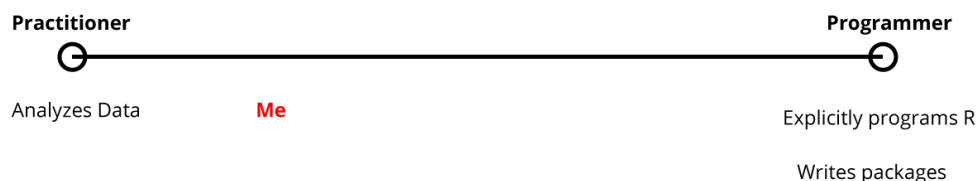   Writes packages

Figure 1.4: The Practioner-Programmer spectrum

We will use R Studio IDE to write code, navigate the files found on our computer, inspect the variables we are going to create, and visualize the plots we will generate. R Studio can also be used for other things (e.g., version control, developing packages, writing Shiny apps) that we don't have time to cover during this workshop.

R Studio is divided into "Panes", see Figure 1.5.

When you first open it, there are three panes, the console where you type commands, your environment/history (top-right), and your files/plots/packages/help/viewer (bottom-right).

The enivronment shows all the R objects you have created or are using, such as data you have imported.

The output pane can be used to view any plots you have created.

Not opened at first start up is the fourth default pane: the script editor pane, but this will open as soon as we create/edit a R script (or many other document types). *The script editor is where will be typing much of the time.*

The placement of these panes and their content can be customized (see menu, R Studio -> Tools -> Global Options -> Pane Layout). One of the advantages of using R Studio is that all the information you need to write code is available ina single window. Additionally, with many shortcuts, auto-completion, and highlighting for the major file types you use while developing in R, R Studio will make typing easier and less error-prone.

> RStudio has lots of keyboard shortcuts to make coding quicker and easier. Try to find the menu listing all the keyboard shortcuts, including the shortcut to find the menu!

Time for another philosphical diversion. . .

1.3.1   What is real?

At the start, we might consider our environment "real" - that is to say the objects we've created/loaded and are using are "real". But it's much better in the long run to consider our scripts as "real" - our scripts are where we write down the code that creates our objects that we'll be using in our environment.

As a script is a document, it is reproducible

Or to put it another way: we can easily recreate an environment from our scripts, but not so easily create a script from an enivronment.

To support this notion of thinking in terms of our scripts as real, we recommend turning off the preservation of workspaces between sessions by setting the Tools > Global Options menu in R studio as shown in Figure 1.6:

1.3.2   Where am I?

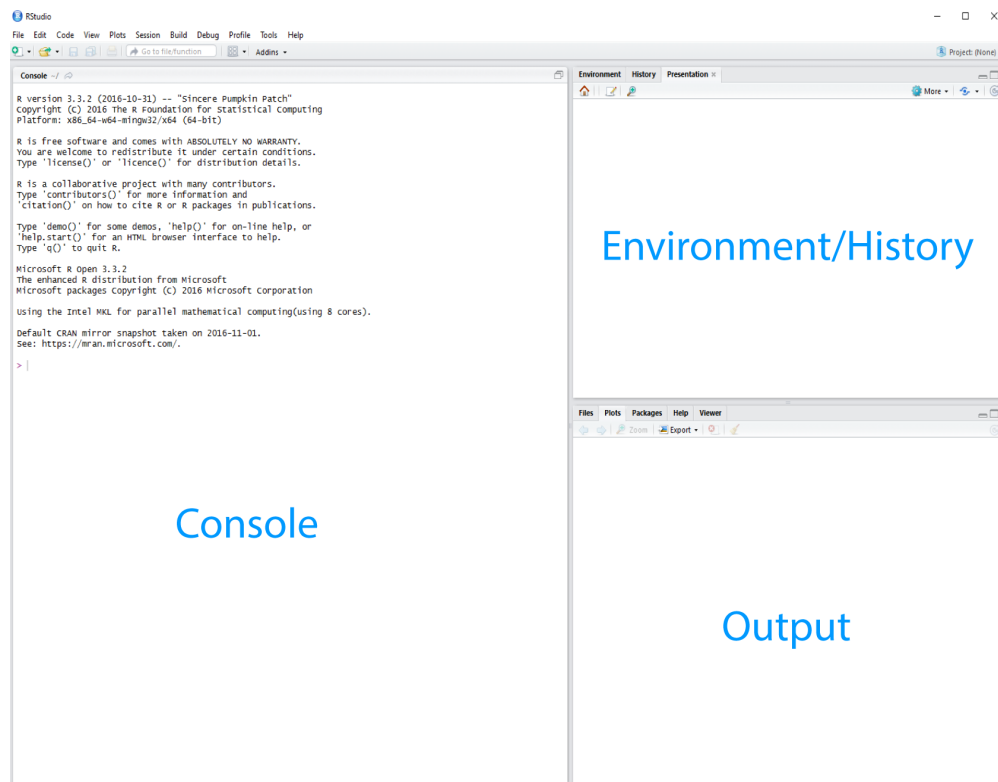R studio tells you where you are in terms of directory address like so:

Figure 1.5: The Rstudio Integrated Development Environment (IDE).

If you are unfamiliar with how computers structure folders and files, then consider a tree with a root from which the trunk extends and branches divide. In the image above, the ~ symbol represents a contraction of the path from the root to the 'home' directory (in Windows this is 'Documents') and then the forward slashes are the branches. (Note: Windows uses backslashes, Unix type systems and R use forwardslashes).

It is good practice to keep a set of related data, analyses, and text self-contained in a single folder, called the **working directory**. All of the scripts within this folder can then use *relative paths* to files that indicate where inside the project a file is located (as opposed to absolute paths, which point to where a file is on a specific computer). Working this way makes it a lot easier to move your project around on your computer and share it with others without worrying about whether or not the underlying scripts will still work.

1.3.3 Getting help

If you need help with a specific R function, let's say `barplot()`, you can type:

```
?barplot
```

A Google or internet search "R <task>" will often either send you to the appropriate package documentation or a helpful forum question that someone else already asked, such as the RStudio Community[9] or Stack Overflow[10].

As well as knowing where to ask[11], the key to get help from someone is for them to grasp your problem rapidly. You should make it as easy as possible to pinpoint where the issue might be.

Try to use the correct words to describe your problem. For instance, a package is not the same thing as

---

[9]https://community.rstudio.com/
[10]http://stackoverflow.com/questions/tagged/r
[11]https://www.tidyverse.org/help/#where-to-ask
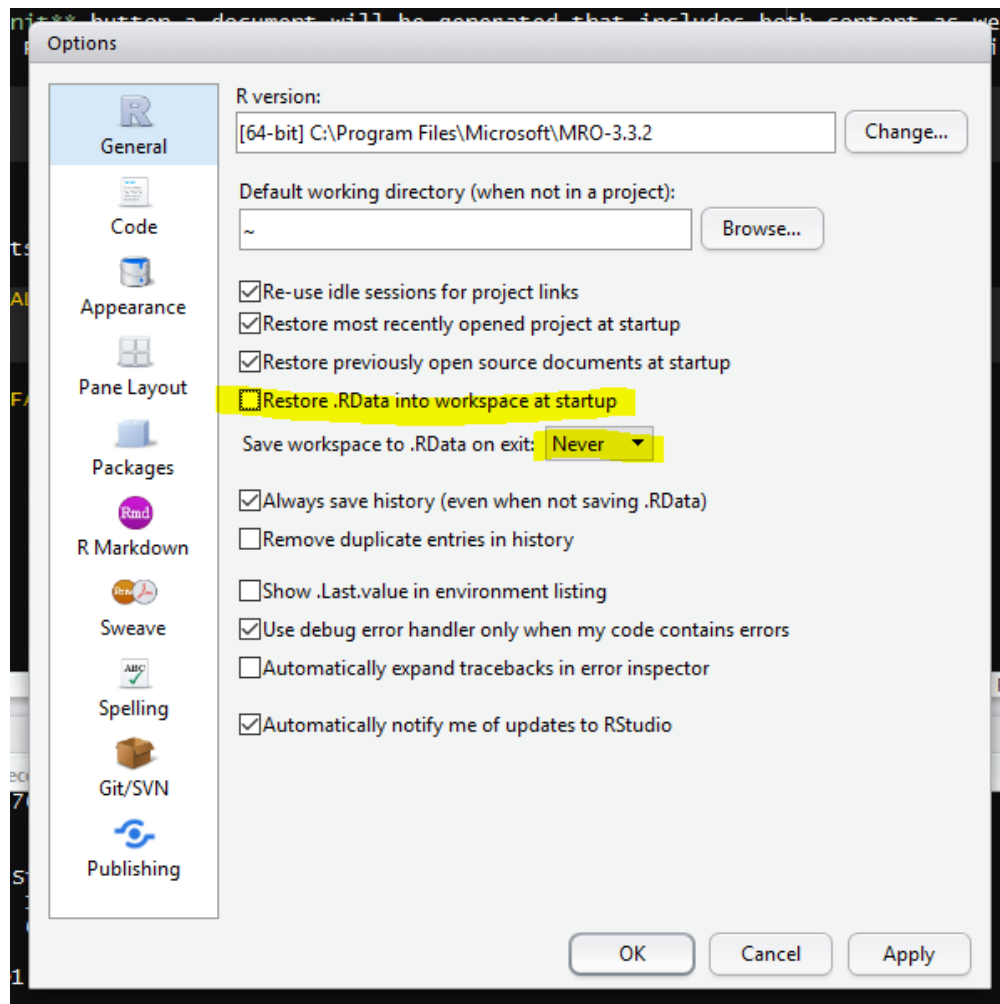
Figure 1.6: Don't save your workspace, save your script instead.
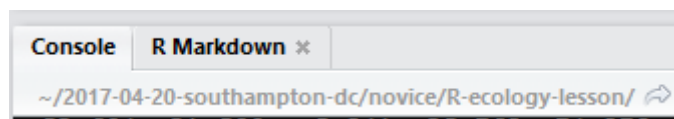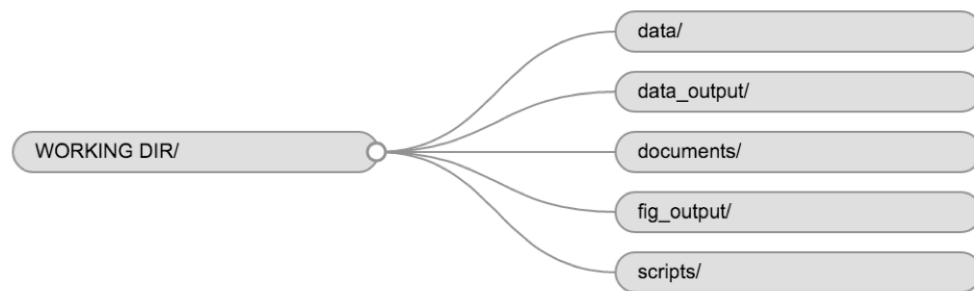

Figure 1.7: Your working directory

Figure 1.8: A typical directory structure

a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. Be as precise as possible when describing your problem.

If possible, try to reduce what doesn't work to a simple *reproducible example* otherwise known as a *reprex*.

For more information on how to write a reproducible example see this article[12] using the `reprex` package.

1.3.4 Installing and loading packages

Packages are collections of functions, and a function is a piece of code written to perform a specific task, such as installing a package.

Therefore, the function `install.packages()` is a piece of code written to perform the task of installing packages. We use it by typing `install.packages("tidyverse")` with the name of the package in quotes inside the round brackets. Here the package is `tidyverse`. Using the console panel to type this and pressing `Enter` will run the function.

We of course need to know the name of the packages we are interested in.

Once a package is installed we need to load it into our environment to use it. Loading packages is performed using the `library()` function. As with installation, we put the name of the package we want to load in between the round brackets like so `library(tidyverse)`. As before this can be done on the console, but we will usually load packages as part of script. *Note that we don't need the quotes for the library function.*

> Try installing the `cowsay` package and loading it. It has one function called `say()` that you can use to create messages with animals.

1.3.5 Using functions

As stated in 1.3.4 a function is a piece of code written to perform a specific task. Functions in R have the syntax of the name of the function followed by round brackets. The round brackets are where we

---

[12]https://www.tidyverse.org/help/#reprex

type the arguments that the function requires to carry out its task. For example, in 1.3.4 the function `install.packages()` requires the name of the package we want to install as arguements.

Many, if not most, functions can take more than one arguement. The creators of the function should have given these defaults for the situation where the user provides only one or some arguements. RStudio should prompt you for the arguements as you type, but if you need to see what they are, use the help function `?` with the function name in the Console and it will open the help panel or type the function name into the help panel search box.

For example, to find out all the arguements for `install.packages()` we'd type `?install.packages` and press `Enter`.

> Try using `say()` to say "I are programmer" in the `cowsay` package and then find out what the arguments you can provide to make it produce different types of message.

1.4   A project orientated workflow

This section is all about how to use R and RStudio to **"maximize effectiveness and reduce frustration."**

The above quote is from Jenny Bryan's article[13] about a project orientated workflow.

The main point here is that how you do things, **the workflow**, should not be mixed up with the **product of the workflow** itself.

The product being:

  • the raw data.
  • the code needed to produce the results from the raw data.

Ways in which you can mix workflow and product include having lines in your script that set your working directory, or using RStudio to save your environment when you are working.

But why is this a problem?

It's because **your computer isn't my computer** or **my laptop isn't my desktop** or **I'm now using a Windows machine and I wrote the code two years ago on a Mac**.

By hard coding the directory into a script I have ensured my code will only run on the machine in which it was written. Chances are you will want to share your code with someone, either for publication or for them to check your work, or because you are working collaboratively and therefore we need to avoid mixing workflow with product.

Likewise we can't share environments directly, but we can share the code that creates the environment.

If we organise our analysis into self-contained projects that hold everything needed to perform the analysis. These projects can be shared across machines and the analysis recreated, and thus the workflow is kept separate from the product.

What does this look like in practice?

1.4.1   RStudio Projects

Step one is to use an interactive development environment such as RStudio rather than using R on its own for your analysis.

RStudio contains a facility to keep all files associated with a particular analysis together called, as you might expect from 1.4, a Project.

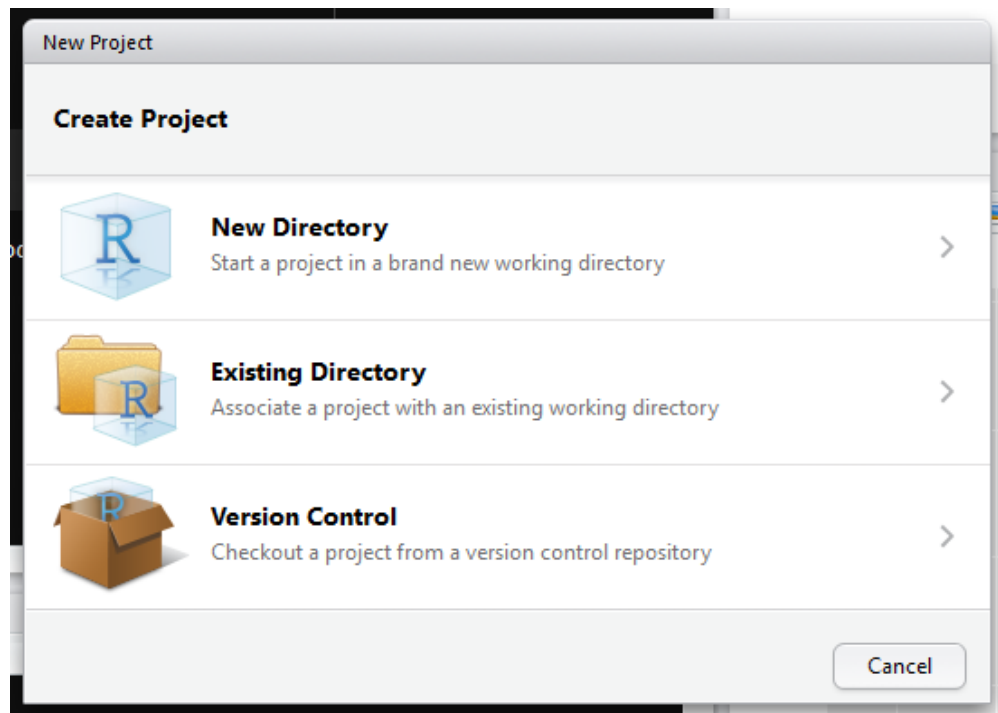[13]https://www.tidyverse.org/articles/2017/12/workflow-vs-script/

Figure 1.9: Creating a R project

Creating a Project creates a file `.Rproj` containing all the information associated with your analysis including the Project location (allowing you to quickly navigate to it), and optionally preserves custom settings and open files to make it easier to resume work after a break. This is also super helpful if you are working on multiple projects as you can switch between them at a click.

Below, we will go through the steps for creating an `Project`:

- Start R Studio (presentation of R Studio -below- should happen here)
- Under the `File` menu, click on `New project`, choose `New directory`, then `Empty project`
- Enter a name for this new folder (or "directory", in computer science), and choose a convenient location for it. This will be your **working directory** for the rest of the day (e.g., `~/coding-together`)
- Click on "Create project"
- Under the `Files` tab on the right of the screen, click on `New Folder` and create a folder named `data` within your newly created working directory. (e.g., `~/data`)
- Create a new R script (File > New File > R script) and save it in your working directory (e.g. `01-coding-together-worksho`

1.4.2 Level up with the `here` package

This is a bit more tricky, so you might like to come back to `here` later, but Jenny Bryan loves the `here`[14] package by Kirill Müller so much she wrote an ode to it[15].

In a nutshell, the `here()` function sets the path implicitly to the top level of the R project you are working in. **But what does that mean, and why should I care?**

Using the `here()` function like this:

```
library(here)

here("data", "file_i_want.csv")
```

----

[14]https://here.r-lib.org/index.html
[15]https://github.com/jennybc/here_here

where `"data"` is the folder containing `"file_i_want.csv"`, the function works out the rest of the path to the folder and file. This is useful if you open the project on different machines where the path is different. `here()` takes care of things, thus saving you some pain.

1.4.3   Naming things

Jenny Bryan[16] also has three principles for naming things[17] that are well worth remembering.

When you names something, a file or an object, ideally it should be:

1. Machine readable (no whitespace, punctuation, upper AND lowercase. . .)
2. Human readable (makes sense in 6 months or 2 years time)
3. Plays well with default ordering (numerical or date order)

We'll see examples of this as we go along.

1.5   The tidyverse and tidy data

The tidyverse[18] (Wickham, 2017) is *"an opinionated collection of R packages designed for data science"* .

Tidyverse packages contain functions that *"share an underlying design philosophy, grammar, and data structures."* It's this philiosophy that makes tidyverse functions and packages relatively easy to learn and use.

Tidy data follows three principals for tabular data as proposed in the Tidy Data paper http://www.jstatsoft.org/v59/i10/paper :

1. Every variable has its own column.
2. Every observation has its own row.
3. Each value has its own cell.

We'll be using the tidyverse and learning more about tidy data as we go along.

1.6   Atoms of R

Having set ourselves up in RStudio, let's turn our attention to the language of R itself.

The basic building blocks of how R stores data are called atomic vector types. It's from these atoms that more complex structures are built. Atomic vectors have one dimension, just like a single row or a single column in a spreadsheet.

The four main atoms of R are:

- Doubles: regular numbers, +ve or -ve and with or without decimal places. AKA numerics.
- Integers: whole numbers, specified with an upper-case L, e.g. `int <- 2L`
- Characters: Strings of text
- Logicals: these store `TRUE`'s and `FALSE`'s' which are useful for comparisons.

Let's make a character vector and check the atomic vector type, using the `typeof()`. This also introduces a very important R function `c()`. This lower case c stands for **combine**. So when we have several objects e.g. words or numbers, we can combine them into a vector the length of the number of objects, as illustrated here for a pack of cards:

---

[16] https://ropensci.org/blog/2017/12/08/rprofile-jenny-bryan/
[17] http://www2.stat.duke.edu/~rcs46/lectures_2015/01-markdown-git/slides/naming-slides/naming-slides.pdf
[18] https://www.tidyverse.org/

Atomic Vectors
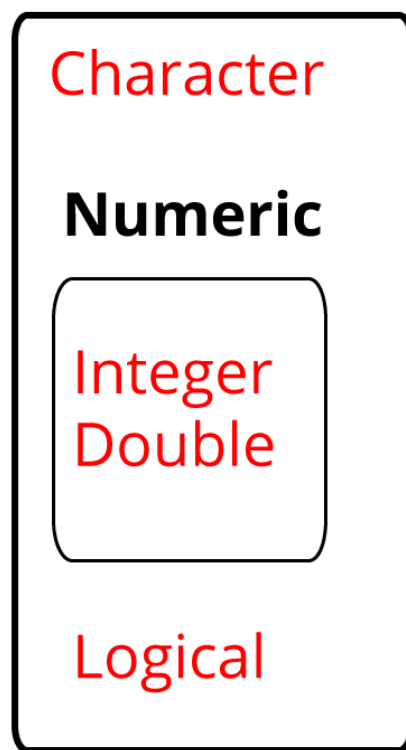

Figure 1.10: The four most used atomic vectors, the building blocks of R

Figure 1.11: Bibi remains a cat even if I call her Princess when she refuses to go out in the rain.

```
cards <- c("ace", "king", "queen", "jack", "ten")

cards
```

```
## [1] "ace"   "king"  "queen" "jack"  "ten"
```

```
typeof(cards)
```

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

Note here that we see the use of the assignment operator <- to assign our vector on the right as the object cards. We talk more about that in 1.7.

> Try creating a vector of numbers from 1 to 10 using the seq() function. Remember to use ?seq if you want to learn more about the function.

1.7   Assigning objects

Objects are just a way to store data inside the R environment. We assign labels to objects using the assignment operator <-:

```
mass_kg <- 55
```

Read this as *"mass_kg is assigned to value 55"* in your head. A subtle but important point here is that the object is 55 and the value remains 55 regardless of the label we assign to it. In fact we could assign more than one label to the same object. Another way to think about this is that Bibi is a cat, and remains a cat even if I call her Princess when she refuses to go out in the rain.

Using <- can be annoying to type, so use RStudio's keyboard short cut: Alt + - (the minus sign) to make life easier.

Many people ask why we use this assignment operator when we can use = instead?

Colin Fay had a Twitter thread on this subject[19], but the reason I favour most is that it provides clarity. The arrow points in the direction of the assigment (it is actually possible to assign in the other direction too) and

---

[19]https://twitter.com/_colinfay/status/1006139974377443328

it distinguishes between creating an object in the workspace and assigning a value inside a function.

Object name style is a matter of choice, but must start with a letter and can only contain letters, numbers, _ and .. We recommend using descriptive names and using _ between words. Some special symbols cannot be used in variable names, so watch out for those.

So here we've used the name to indicate its value represents a mass in kilograms. Look in your environment pane and you'll see the `mass_kg` object containing the (data) value 55.

We can inspect an object by typing it's name:

```
mass_kg
```

```
## [1] 55
```

What's wrong here?

```
mass_KG
```

```
Error: object 'mass_KG' not found
```

This error illustrates that typos matter, everything must be precise and `mass_KG` is not the same as `mass_kg`. `mass_KG` doesn't exist, hence the error.

Let's use `seq()` to create a **seq**uence of numbers, and at the same time practice tab completion.

Start typing `se` in the console and you should see a list of functions appear, add `q` to shorten the list, then use the up and down arrow to highlight the function of interest `seq()` and hit Tab to select. This is tab completion.

RStudio puts the cursor between the parentheses to prompt us to enter some arguments. Here we'll use `1` as the start and `10` as the end:

```
seq(1,10)
```

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

If we left off a parentheses to close the function, then when we hit enter we'll see a + indicating RStudio is expecting further code. We either add the missing part or press Escape to cancel the code.

Let's call a function and make an assignment at the same time. Here we'll use the base R function `seq()` which takes three arguments: `from`, `to` and `by`.

Read the following code as *"assign my_sequence to an object that stores a sequence of numbers from 2 to 20 by intervals of 2*.

```
my_sequence <- seq(2,20,2)
```

This time nothing was returned to the console, but we now have an object called `my_sequence` in our environment.

1.7.1 Indexing and subsetting

If we want to access and subset elements of `my_sequence` we use square brackets `[]` and the index number. Indexing in R starts at 1 such that 1 is the index of the first element in the sequence, element 1 having the the value of 2.

For example element five would be subset by:

```
my_sequence[5]
```

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

Here the number five is the index of the vector, not the value of the fifth element. The value of the fifth element is 10.

And returning multiple elements uses a colon `:`, like so

```
my_sequence[5:8]
```

```
## [1] 10 12 14 16
```

1.8   Lists, matrices and arrays

Lists also group data into one dimensional sets of data. The difference being that list group objects instead of individual values, such as several atomic vectors.

For example, let's make a list containing a vector of numbers and a character vector

```
list_1 <- list(1:110,"R")

list_1
```

```
## [[1]]
##   [1]    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17
##  [18]   18   19   20   21   22   23   24   25   26   27   28   29   30   31   32   33   34
##  [35]   35   36   37   38   39   40   41   42   43   44   45   46   47   48   49   50   51
##  [52]   52   53   54   55   56   57   58   59   60   61   62   63   64   65   66   67   68
##  [69]   69   70   71   72   73   74   75   76   77   78   79   80   81   82   83   84   85
##  [86]   86   87   88   89   90   91   92   93   94   95   96   97   98   99  100  101  102
## [103]  103  104  105  106  107  108  109  110
##
## [[2]]
## [1] "R"
```

Note the double brackets to indicate the list elements, i.e. element one is the vector of numbers and element two is a vector of a single character.

We won't be working with lists a great deal in these workshops, but they are a flexible way to store data of different types in R.

Accessing list elements uses double square brackets syntax, for example `list_1[[1]]` would return the first vector in our list.

And to access the first element in the first vector would combine double and single square brackets like so: `list_1[[1]][1]`.

Don't worry if you find this confusing, everyone does when they first start with R. Hadley Wickham tweeted an image to illustrate list indexing shown in 1.12.

Lists alongside `NULL` which indicates the absence of a vector, complete the set of base vectors in R as illustrated in 1.13.

1.8.1   Matrices and arrays

Matrices store values in a two dimensional array, whilst arrays can have n dimensions. We won't be using these either, but they are also valid R objects.

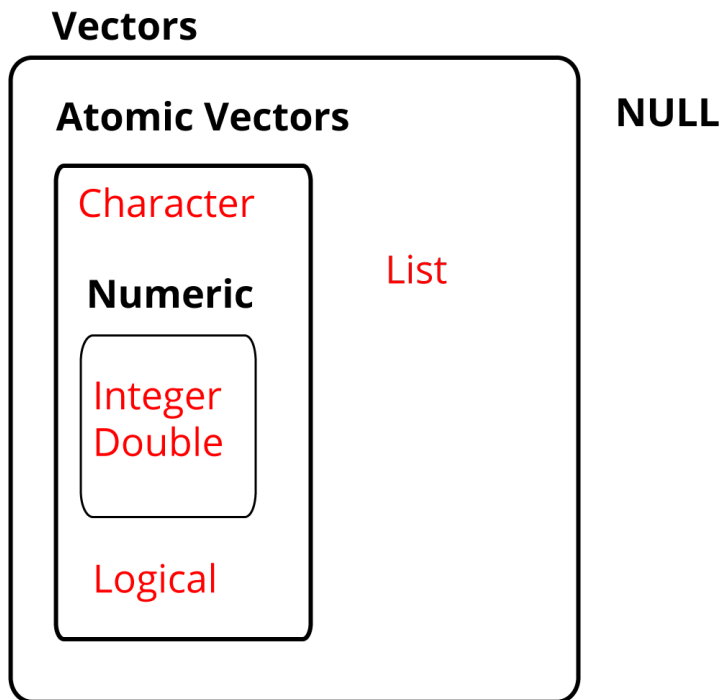Figure 1.12: List indexing by Hadley Wickham

Vectors

Figure 1.13: The base vectors in R.

1.9   Factors

Factors are Rs way of storing categorical information such as eye colour or car type. A factor is something that can only have certain values, and can be ordered (such as `low,medium,high`) or unordered such as types of fruit.

Factors are useful as they code string variables such as "red" or "blue" to integer values e.g. 1 and 2, which can be used in statistical models and when plotting, but they are confusing as they look like strings.

Factors look like strings, but behave like integers.

Historically R converts strings to factors when we load and create data, but it's often not what we want as a default. Fortunately, in the tidyverse strings are not treated as factors by default.

1.10   Data frames

For data analysis in R, we mostly be using data frames.

Data frames are two dimensional versions of lists, and this is form of storing data we are going to be using. In a data frame each atomic vector type becomes a column, and a data frame is formed by columns of vectors of the same length. Each column element must be of the same type, but the column types can vary.

Figure 1.14 shows an example data frame we'll refer to as saved as the object `df` consisting of three rows and three columns. Each column is a different atomic data type of the same length.

To create the data frame 1.14 we can use the `data.frame()` function in conjunction with the `c()` function to make the individual atomic vectors that comprise the data frame as follows. Note that I am naming the vectors as I make the data frame after the type of vector e.g. `numeric_vector = c(1,7,3)`. Also, as this is a base R function I need to tell the function not to treat the character strings as catergorical data using `stringsAsFactors = FALSE`

Figure 1.14: An example data frame `df`.

```r
df <- data.frame(numeric_vector = c(1,7,3),
                 character_vector = c("S","A","U"),
                 logical_vector = c(TRUE,FALSE,TRUE),
                 stringsAsFactors = FALSE)

df

##   numeric_vector character_vector logical_vector
## 1              1                S           TRUE
## 2              7                A          FALSE
## 3              3                U           TRUE
```

Packages in the tidyverse create a modified form of data frame called a tibble. You can read about tibbles here[20]. One advantage of tibbles is that they don't default to treating strings as factors. We deal with modifying data frames in later chapters.

Here's what the code to make the same data frame as before as a tibble looks like. Note how we get more information from a tibble when it is returned to the Console, it tells us what the dimensions are, and what type of vectors it contains.

```r
df <- tibble(numeric_vector = c(1,7,3),
             character_vector = c("S","A","U"),
             logical_vector = c(TRUE,FALSE,TRUE))

df

## # A tibble: 3 x 3
##   numeric_vector character_vector logical_vector
##            <dbl> <chr>            <lgl>
## 1              1 S                TRUE
## 2              7 A                FALSE
## 3              3 U                TRUE
```

---
[20]http://r4ds.had.co.nz/tibbles.html

Sub-setting data frames can also be done with square bracket syntax, but as we have both rows and columns, we need to provide index values for both row and column.

For example `df[1,2]` means **return the value of** `df` **row 1, column 2**. This corresponds with the value `A`.

We can also use the colon operator to choose several rows or columns, and by leaving the row or column blank we return all rows or all columns.

```r
# Subset rows 1 and 2 of column 1
df[1:2,1]

# Subset all rows of column 3
df[,3]
```

Don't worry too much about this for now, we won't be doing to much of this in these lessons, but it's worth being aware of this syntax.

1.10.1   Attributes

An attribute is a piece of information you can attach to an object, such as names or dimensions. Attributes such as dimensions are added when we create an object, but others such as names can be added.

Let's look at the `mpg` data frame dimensions:

```r
# mpg has 234 rows (observations) and 11 columns (variables)
dim(mpg)
```

```
## [1] 234  11
```

1.11   Plotting data

1.12   Exporting data

Chapter 2

Data wrangling I

Chapter 3

Data wrangling II

Chapter 4

Functions

Chapter 5

Exploratory modelling

Chapter 6

Visualistion

Chapter 7

Reports

Using R for report writing and presentations.

References

Ihaka, R. and Gentleman, R. (1996). R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

RStudio Team (2018). *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1.

Wilson, G., editor (2018). *Teaching Tech Together. 2018„ http://teachtogether.tech/*. Lulu.com.

Xie, Y. (2018). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.9.