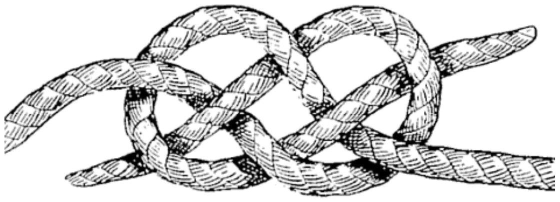
Coding togetheR

Alistair Bailey

October 15 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].

[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/

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

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.

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.

```
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.

[9]https://cran.r-project.org/
[10]https://www.rstudio.com/products/rstudio/download/

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].

[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 the cat in Figure 1.1.

Then hopefully by the end of these materials, you'll feel a bit more like the cat in Figure 1.2.

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"

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

Figure 1.1: Imposter syndrome[1].

Figure 1.2: R cat[2]

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.

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 achieve 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.

[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

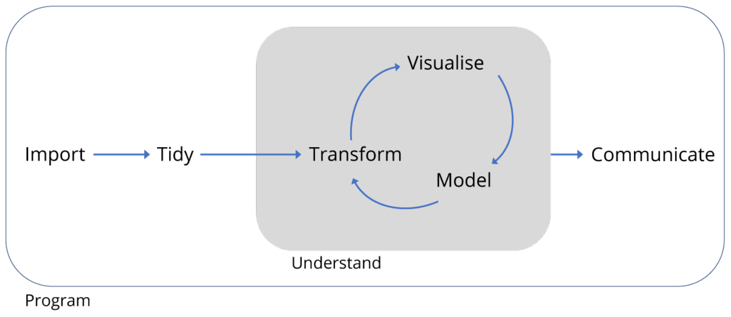


Figure 1.3: Data project workflow.

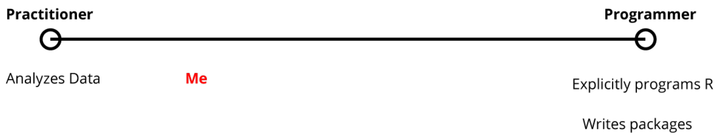


Figure 1.4: The Practitioner-Programmer spectrum

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 practitioners 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.

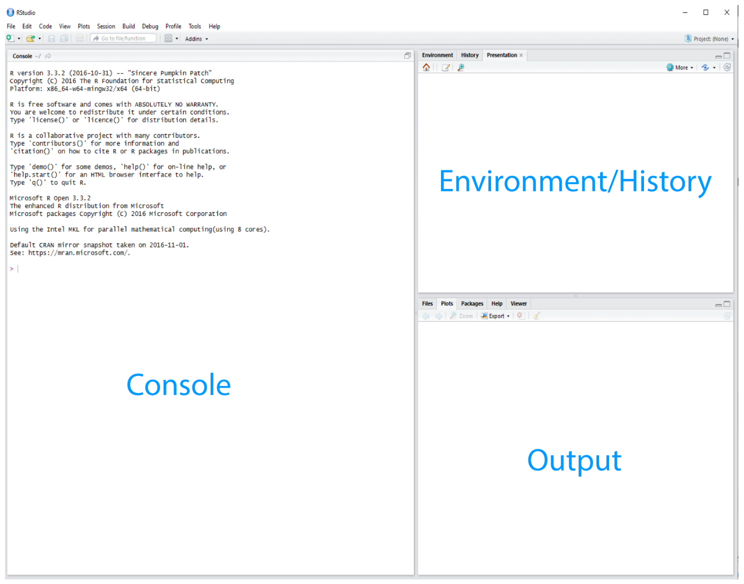
1.3 RStudio

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.

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

[8] https://www.rstudio.com/


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
Console ~/
Environment History Presentation
More
Files Plots Packages Help Viewer
Zoom Export
Environment/History
Console
Output

```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Microsoft R Open 3.3.2
The enhanced R distribution from Microsoft
Microsoft packages Copyright (C) 2016 Microsoft Corporation

Using the Intel MKL for parallel mathematical computing(using 8 cores).

Default CRAN mirror snapshot taken on 2016-11-01.
See: https://mran.microsoft.com/.

> 
```

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

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).

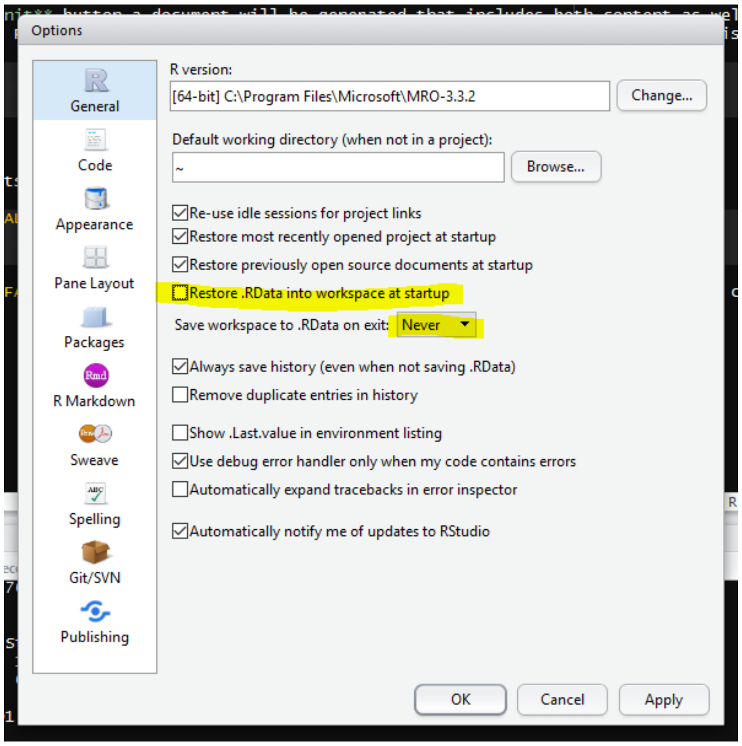
The environment shows all the R objects you have created or are using, such as data you have imported.

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 in a single window. Additionally, with many short-cuts, 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 short-cuts to make coding quicker and easier. Try to find the menu listing all the keyboard short-cuts,



R version:
[64-bit] C:\Program Files\Microsoft\MRO-3.3.2

Default working directory (when not in a project):
~

☑ Re-use idle sessions for project links
☑ Restore most recently opened project at startup
☑ Restore previously open source documents at startup
☐ Restore .RData into workspace at startup
Save workspace to .RData on exit: Never
☑ Always save history (even when not saving .RData)
☐ Remove duplicate entries in history
☐ Show .Last.value in environment listing
☑ Use debug error handler only when my code contains errors
☐ Automatically expand tracebacks in error inspector
☑ Automatically notify me of updates to RStudio

Figure 1.6: Don't save your workspace, save your script instead.

including the short-cut to find the menu!

Time for another philosophical 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 environment.

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.

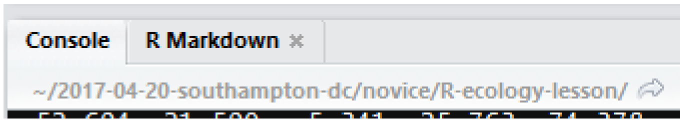


Figure 1.7: Your working directory

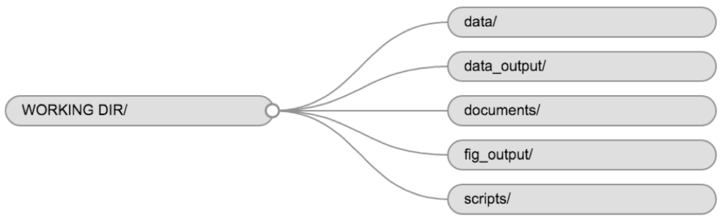


Figure 1.8: A typical directory structure

1.3.2 Where am I?

R studio tells you where you are in terms of directory address as shown in Figure 1.7.

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 forward slashes).

It is good practice to keep a set of related data, analyses, and text self-contained in a single folder, called **your 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). An example directory structure is illustrated in Figure 1.8. Working this way makes it a lot easier to move your project around on your computer. Section 1.4 builds upon this to create a robust workflow for data analysis.

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 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.*

[9] https://community.rstudio.com/
[10] http://stackoverflow.com/questions/tagged/r
[11] https://www.tidyverse.org/help/#where-to-ask
[12] https://www.tidyverse.org/help/#reprex

> 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 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 arguments.

Many, if not most, functions can take more than one argument. The creators of the function should have given these defaults for the situation where the user provides only one or some arguments. RStudio should prompt you for the arguments 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 arguments 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.

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

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.

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"

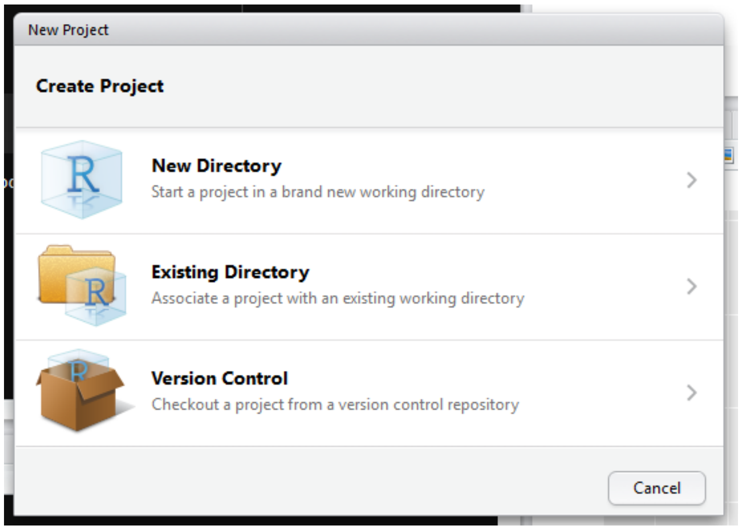

New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

Figure 1.9: Creating a R 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 R notebook (File > New File > R notebook) and save it in your working directory (e.g. `01-coding-together-workshop-02-05-2019.Rmd`)
- Or create a new R script (File > New File > R script) and save it in your working directory (e.g. `01-coding-together-workshop-02-05-2019.R`)

1.4.2 R notebooks and R scripts

R notebooks[14] combine writing text with R code in chunks. The R code chunks are indicated by three backticks and a lowercase r in brackets: ` ```{r} ``` `. Text can be formatted using markdown syntax[15]. These are great for doing analysis and report wrting at the same time.

R scripts are text files containing the commands that you would enter into the R console. They are great for containing code you wish to call into another script such as code for a function, or if you are submitting a script as job to run on another computer without the need for RStudio.

[14] https://bookdown.org/yihui/rmarkdown/notebook.html
[15] https://bookdown.org/yihui/rmarkdown/markdown-syntax.html

1.4.3 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`[16] package by Kirill Müller so much she wrote an ode to it[17].

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")
```

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.

1.4.4 Naming things

Jenny Bryan[18] also has three principles for naming things[19] that are well worth remembering.

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

1. Machine readable (no white space, punctuation, upper AND lower-case…)
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[20] (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 philosophy that makes tidyverse functions and packages relatively easy to learn and use.

[16] https://here.r-lib.org/index.html
[17] https://github.com/jennybc/here_here
[18] https://ropensci.org/blog/2017/12/08/rprofile-jenny-bryan/
[19] http://www2.stat.duke.edu/~rcs46/lectures_2015/01-markdown-git/slides/naming-slides/naming-slides.pdf
[20] https://www.tidyverse.org/

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:

```
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.

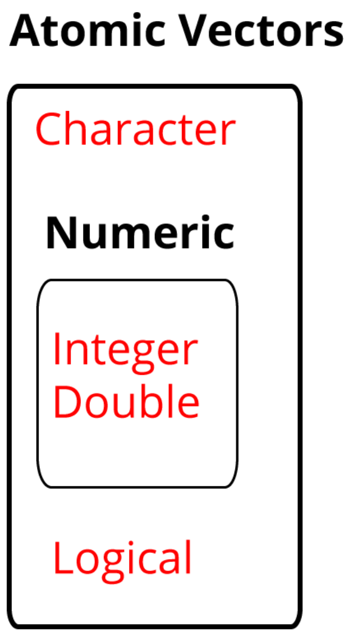


Figure 1.10: The four most used atomic vectors, the building blocks of R

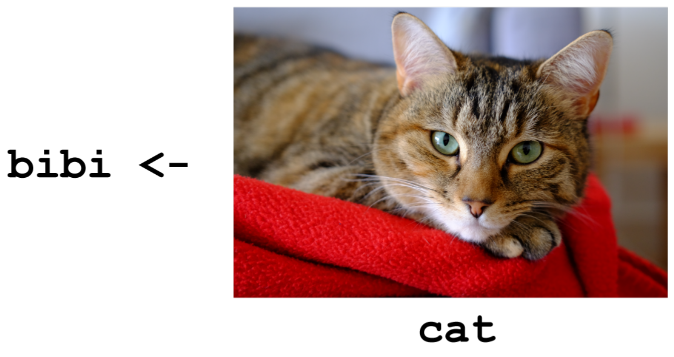


Figure 1.11: Bibi remains a cat even if I call her Princess when she refuses to go out in the rain.

> 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[21], but the reason I favour most is that it provides clarity. The arrow points in the direction of the assignment (it is actually possible to assign in the other direction too) and it distinguishes

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

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.

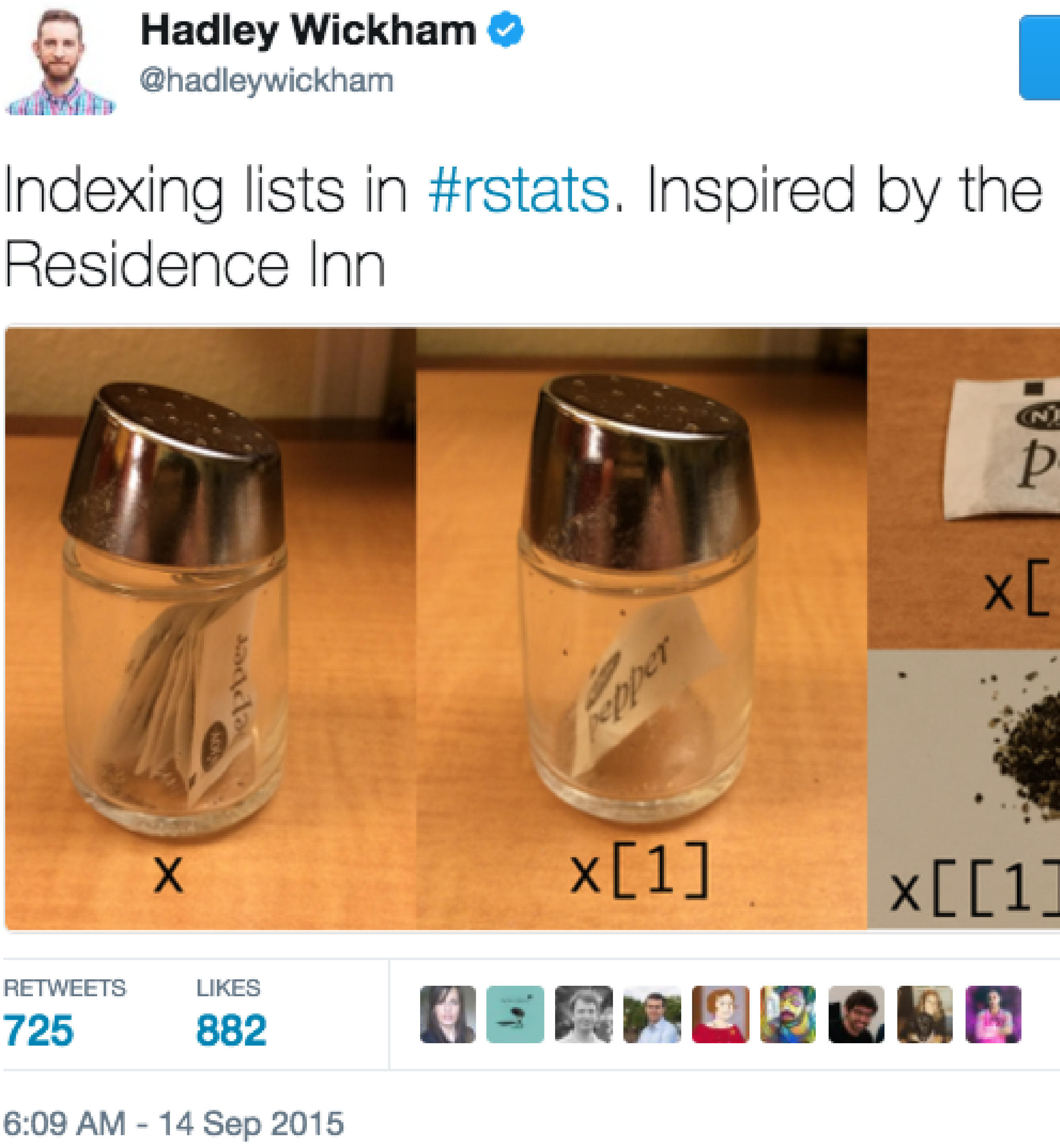
1.9 Factors

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.


Hadley Wickham
@hadleywickham

Indexing lists in #rstats. Inspired by the Residence Inn

RETWEETS 725 LIKES 882

6:09 AM - 14 Sep 2015

Figure 1.12: List indexing by Hadley Wickham

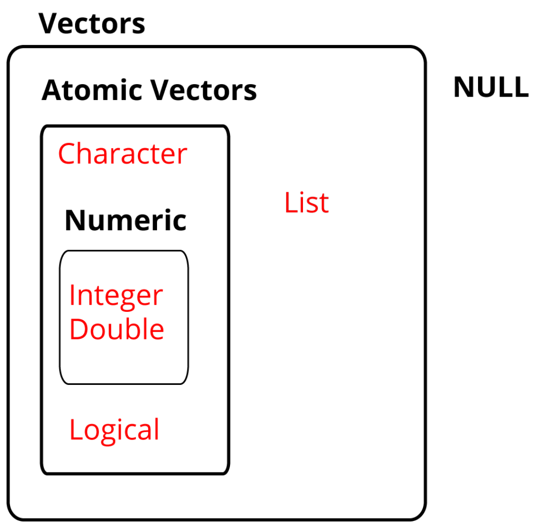


Figure 1.13: The base vectors in R.

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 categorical data using `stringsAsFactors = FALSE`

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

df
```

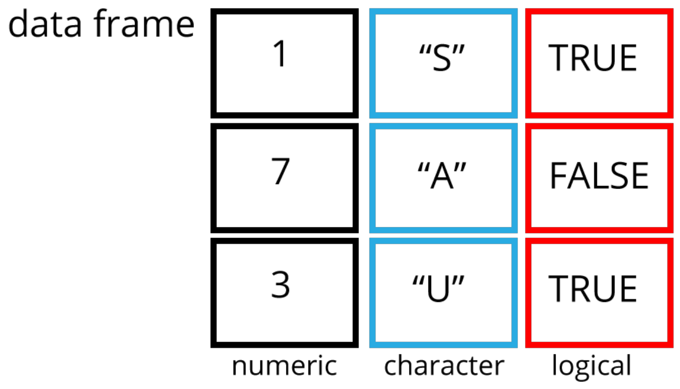


Figure 1.14: An example data frame `df`.

```
##   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[22]. One advantage of tibbles is that they don't default to treating strings as factors. We deal with transforming data frames in chapters 2 and 3.

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.

```
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
```

[22] 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

One of the most useful and important parts of any data analysis is plotting data. We'll be spending a whole lesson on it in chapter 6, but to give you an example, we'll use the `ggplot2` package as an introduction to automating a task in code, and as a tool for understanding data.

`ggplot2` implements the *grammar of graphics*, for describing and building graphs. The idea being that we construct a plot in the following way:

1. Call the `ggplot()` function to create a graph.
2. Pass our data as the first argue to the `ggplot()` function.

3. Then pass some arguments to the aesthetics function `aes()` inside the `gpplot()` which tell ggplot how to plot the data e.g. which data goes on the x and y axis.
4. Then we follow the ggplot function with a `+` sign to indicate we are going to add more code, followed by a geometric object function, a `geom` which maps the data to type of plot we want to make e.g. a histogram or scatter plot.

Don't worry if this sounds confusing, it becomes clear with practice and all plots follow this grammar.

We'll use the `mpg` dataset that comes with the tidyverse to examine the question *do cars with big engines use more fuel than cars with small engines?*

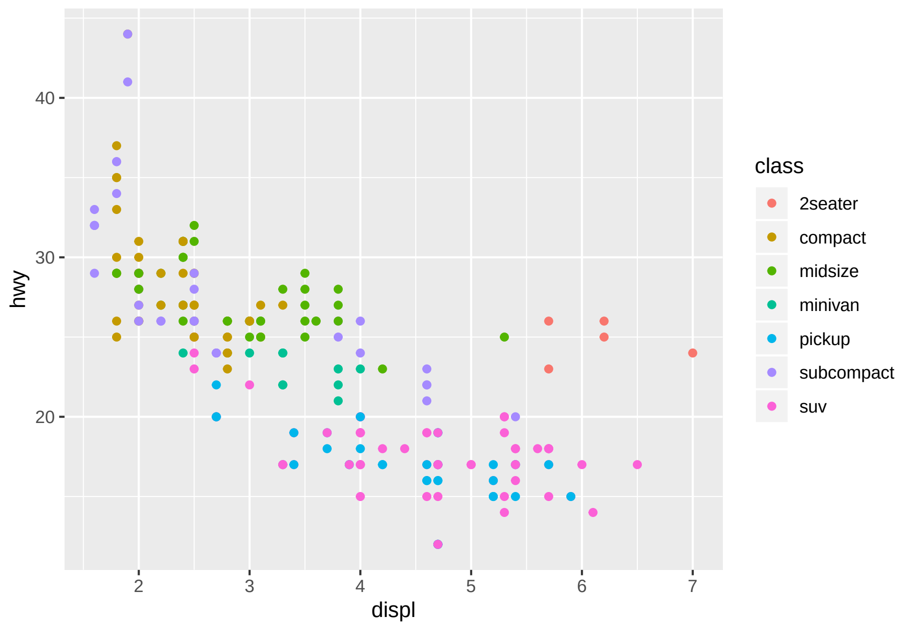
Try `?mpg` to learn more about the data.

1. Engine size in litres is in the `displ` column.
2. Fuel efficiency on the highway in miles per gallon is given in the `hwy` column.

To create a plot of engine size `displ` (x-axis) against fuel efficiency `hwy` (y-axis) we do the following:

Now try extending this code to include to add a `colour` aesthetic to the the `aes()` function, let `colour = class`, `class` being the vehicle type. This should create a plot with as before but with the points coloured according to the vehicle type to expand our understanding.

```r
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, colour = class))
```



Now we can see that as we might expect, bigger cars such as SUVs tend to have bigger engines and are also less fuel efficient, but some smaller cars such as 2-seaters also have big engines and greater fuel efficiency. Hence we have a more nuanced view with this additional aesthetic.

Check out the ggplot2 documentation for all the aesthetic possibilities (and Google for examples): http://ggplot2.tidyverse.org/reference/

So now we have re-usable code snippet for generating plots in R:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

Concretely, in our first example `<DATA>` was `mpg`, the `<GEOM_FUNCTION>` was `geom_point()` and the arguments we supplies to map our aesthetics `<MAPPINGS>` were `x = displ, y = hwy`.

As we can use this code for any tidy data set, hopefully you are beginning to see how a small amount of code can do a lot.

1.12 Exporting data

We'll spend more time on getting data in and out of our R environment in the next chapter 3, but just to wrap this lesson up let's imagine we wanted to export our plot and data for a colleague or presentation.

1.12.1 readr

To export tibbles and data frames, we'll use the `readr` package, and the `write_excel_csv()` function. This creates a table in comma separated variable format that can opened by spreadsheet software such as excel.

As it is a function is has round brackets and the main arguments we pass are the object containing the data we want to output and the name of the file and the location we want to write the file to.

```
write_excel_csv(df, "outputs/example-data-02-05-2019.csv")
```

Here we are writing the `df` data frame as a csv file to the `outputs` folder and a file called `example-data-02-05-2019.csv`.

1.12.2 ggsave

If we want to save the last plot we made in `ggplot2` we can use the `ggsave()` function[23]

We tell `ggsave()` the filename, and it will save it as that type depending on how we name the file. For example if we use `file.pdf` it will save a PDF and if we use `file.jpeg` it will save a `jpeg`.

Check out `?ggsave` or the line above for more options.

To save our last plot for example:

```
ggsave("outputs/example-plot-02-05-2019.pdf")
```

1.12.3 Exercise

1. Create a new project called 'coding-assessment-01'
2. Create two folders in this project: R and outputs
3. Create a R script using best naming practices i.e. name-date.R
4. In the script, write some comments at the top e.g. name, date, description
5. Create a tibble comprising of a character vector, a numeric vector
6. Install and load the `dslabs` package and create a density plot with ggplot2 using the `heights` dataset, using the `x = height` variable and `fill = sex` to create a density plot ggplot(data = heights, aes(x =height, fill = sex)) + geom_density()
7. Save the plot as pdf and the tibble as csv file to the output folder.

[23]https://ggplot2.tidyverse.org/reference/ggsave.html

Chapter 2

Data wrangling I

By the end of this chapter you will:

- have learnt to load and export csv and excel files
- know what tidy data is and how to organise data such that it is tidy
- have used the key verbs of the `dplyr` package for transforming data to arrange and filter observations, select variables, create new variables, and create summaries.
- have learnt how to combine functions with the pipe from the `magrittr` package to combine tasks

The following sections are based upon the data transformation chapter[1] in R4DS and the Data Carpentry ecology lesson[2].

2.1 Data organisation in spreadsheets

Karl Broman and Kara Woo wrote as paper all about Data Organization in Spreadsheets[3].

It's full of practical advice and context.

2.1.1 Flat formats and Excel files

File formats like `.csv` and `.tsv`, comma separated variables and tab separated variables respectively are plain text files. That is to say they contain only the

[1] https://r4ds.had.co.nz/transform.html
[2] https://datacarpentry.org/R-ecology-lesson/index.html
[3] https://www.tandfonline.com/doi/full/10.1080/00031305.2017.1375989

data, as text information, and are the simplest and most convenient way to share data as most software can read and interpret them.

Excel files saves files into its own proprietary format viz xls or xlsx that holds information in addition to the data itself. For reading and writing excel files in R, the tidyverse `readxl` package is very useful.

2.2 The Portal Project data

In this chapter we are going to focus on data analysed in the 1994 paper by Heske et. al:

Long-Term Experimental Study of a Chihuahuan Desert Rodent Community: 13 Years of Competition, DOI: 10.2307/1939547.

Specifically they explored the effect on the populations of small seed eating rodents as a result of the exclusion of larger competitor kangaroo rats over a period from 1977 to 1991.

Figure 2.1 shows an image of one of the species of kangaroo rats excluded during the study.

Figure 2.2 indicates how the exclusion works, where a for number of fenced plots the kangaroo rats were either able to enter by a hole or kept out.

The plots are 50 metres by 50 metres, and a survey of the species within each plot has been ongoing once a month for many years.

The dataset is stored as a comma separated value (CSV) file. Each row holds information for a single animal, and the columns represent:

Column	Description	Type
record_id	Unique id for the observation	numeric
month	month of observation	numeric
day	day of observation	numeric
year	year of observation	numeric
plot_id	ID of a particular plot	numeric
species_id	2-letter code	character
sex	sex of animal (“M”, “F”)	character
hindfoot_length	length of the hindfoot in mm	numeric
weight	weight of the animal in grams	numeric
genus	genus of animal	character
species	species of animal	character
taxa	e.g. Rodent, Reptile, Bird, Rabbit	character
plot_type	type of plot	character

Figure 2.1: Merriam's kangaroo rat, Dipodomys merriami[4]

Figure 2.2: Kangeroo Rat exclusion

The rodents species surveyed are:

Kangeroo Rats

species_id	Scientific name	Common name
DM	Dipodomys merriami	Merriam's kangaroo rat
DO	Dipodomys ordii	Ord's kangaroo rat
DS	Dipodomys spectabilis	Banner-tailed kangaroo rat

Granivores

species_id	Scientific name	Common name
PP	Chaetodipus penicillatus	Desert pocket mouse
PF	Perognathus flavus	Silky pocket mouse
PE	Peromyscus eremicus	Cactus mouse
PM	Peromyscus maniculatus	Deer Mouse
RM	Reithrodontomys megalotis	Western harvest mouse

2.2.1 Downloading and importing the data

First create a R project for this analysis

The dataset is stored online, so we use the utility function `download.file()` to download the csv file to our data folder. (Did you create a data folder in the project directory?)

Here we pass the `url =` and `destfile =` arguments to `download.file()`.

As we have the `tidyverse` packages we can use the `readr` package it contains, which has many functions for reading files, including `read_csv()`. The advantage of `read_csv()` over base R `read.csv()` is that it defaults to reading strings as character vectors rather than factors (catergorical variables) which is usually what we want.

As we read the data into our environment we need to assign a label to the object we are creating. Here we assign the dataset to an object called `surveys` using the `<-` assignment operator.

```r
# Download the data
download.file(url="https://ndownloader.figshare.com/files/2292169",
              destfile = "data/portal_data_joined.csv")

# Read into R as an object called surveys
surveys <- read_csv("data/portal_data_joined.csv")
```

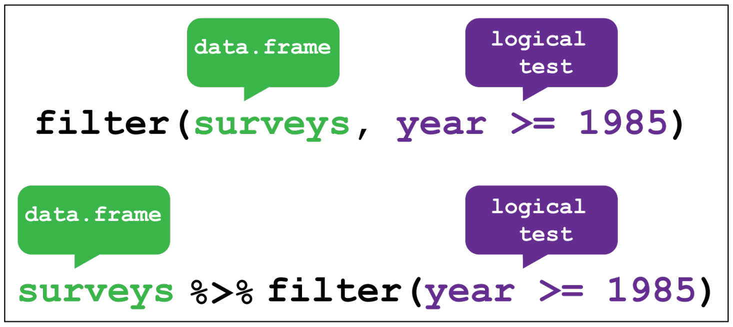

data.frame
logical test
filter(surveys, year >= 1985)
data.frame
logical test
surveys %>% filter(year >= 1985)

Figure 2.3: `dplyr::filter()`

2.3 dplyr

2.3.1 Filter rows with `filter()`

The first verb to consider is the `filter()` function which enables us to subset observations based on their value.

Consider the surveys data and sub-setting observations that only occurred from 1985 onwards. It's fairly natural to say *"filter the survey where the year variable is equal or greater than 1985"*. And indeed this is how we use `filter()` as a verb.

Figure 2.3 shows how we give the filter function two arguments. The first is the data frame, the second is the variable and condition on which we wish to filter.

Alternatively we can pipe the `surveys` object to the function and it implicitly uses `surveys` as the first argument.

(Note that we aren't assigning the output to an object here, so we can see it.)

```r
# Filter observations that only occurred from 1985 onwards
 filter(surveys, year >= 1985)
```

```
## # A tibble: 25,290 x 13
##    record_id month   day  year plot_id species_id sex   hindfoot_length
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>
##  1     10606     7    24  1985       2 NL         F                  30
##  2     10617     7    24  1985       2 NL         M                  32
##  3     10627     7    24  1985       2 NL         F                  32
##  4     10720     8    20  1985       2 NL         F                  31
##  5     10923    10    13  1985       2 NL         F                  31
##  6     10949    10    13  1985       2 NL         F                  33
##  7     11215    12     8  1985       2 NL         F                  32
```

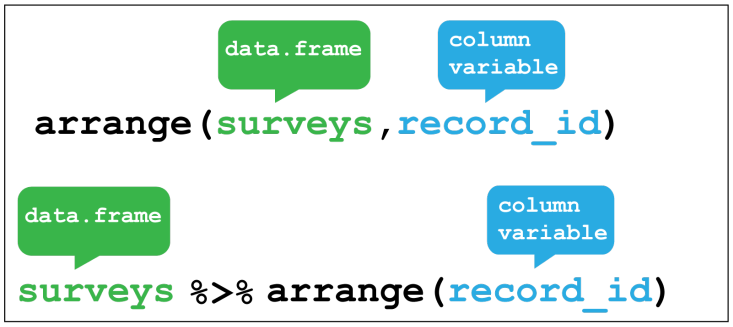


Figure 2.4: `dplyr::arrange()`

```
##  8     11329     3     9  1986       2 NL         M                    34
##  9     11496     5    11  1986       2 NL         F                    31
## 10     11498     5    11  1986       2 NL         F                    31
## # ... with 25,280 more rows, and 5 more variables: weight <dbl>,
## #   genus <chr>, species <chr>, taxa <chr>, plot_type <chr>
```

```
# Control Plots: 2,8,12,22
# Treatment Plots: 3,15,19,21

# Create a vector for the experimental plots
exp_plots <- c(8,11,12,14,3,15,19,21)

# Keep only the rows corresponding with the experimental plot_id's
surveys_filtered <- surveys %>% filter(plot_id %in% exp_plots)
```

2.3.2 Arrange rows with `arrange()`

Use arrange to find the shortest hindfoot

```
surveys %>% arrange(hindfoot_length)
```

```
## # A tibble: 34,786 x 13
##    record_id month   day  year plot_id species_id sex   hindfoot_length
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>
##  1     31400     9    30  2000      19 PB         M                   2
##  2     10067     3    16  1985      19 RM         M                   6
##  3     19567     1     8  1992      19 BA         M                   6
##  4     19015     9     9  1991      19 BA         F                   7
##  5     21036     8    19  1993      21 PF         F                   7
##  6     31457     9    31  2000       6 RM         M                   8
##  7      19191    10    11  1991       13 PF          F                       8
##  8       5801     4    29  1982        7 RM          <NA>                    8
##  9      33647     3    14  2002        3 PF          M                       9
## 10      20562    12    22  1992        5 RM          F                       9
## # ... with 34,776 more rows, and 5 more variables: weight <dbl>,
## #   genus <chr>, species <chr>, taxa <chr>, plot_type <chr>
```

Find the Cactus Mouse, (`species_id == "PE"`) with the longest hindfoot:

Hint Use the `desc()` function to arrange from biggest to smallest.

```
surveys %>%
        filter(species_id == "PE") %>%
        arrange(desc(hindfoot_length))
```

```
## # A tibble: 1,299 x 13
##    record_id month   day  year plot_id species_id sex   hindfoot_length
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>
##  1      1202     9     3  1978       7 PE         F                  30
##  2       517     1     8  1978       2 PE         M                  26
##  3     32443     8    25  2001      23 PE         F                  24
##  4      5080    12    30  1981      15 PE         F                  23
##  5      5090    12    30  1981      15 PE         F                  23
##  6     33700     3    14  2002       9 PE         F                  23
##  7       604     2    18  1978       2 PE         M                  22
##  8     12459     3     2  1987       2 PE         F                  22
##  9     13992     1    24  1988       2 PE         M                  22
## 10     14516     5    15  1988       2 PE         F                  22
## # ... with 1,289 more rows, and 5 more variables: weight <dbl>,
## #   genus <chr>, species <chr>, taxa <chr>, plot_type <chr>
```

2.3.3 Select columns with `select()`

```
# Select the year and plot type columns
surveys %>% select(year,plot_type)
```

```
## # A tibble: 34,786 x 2
##     year plot_type
##    <dbl> <chr>
##  1  1977 Control
##  2  1977 Control
##  3  1977 Control
##  4  1977 Control
```

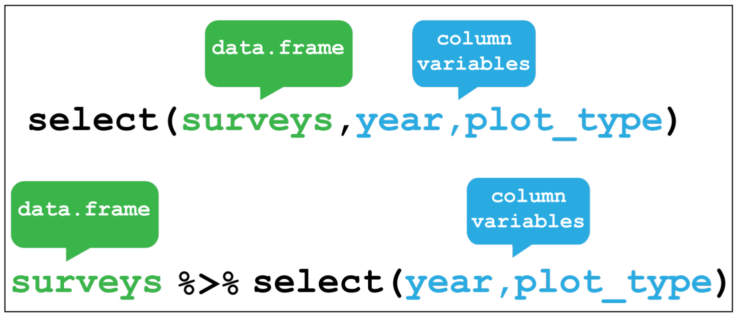

data.frame
column variables
select(surveys,year,plot_type)
data.frame
column variables
surveys %>% select(year,plot_type)

Figure 2.5: `dplyr::select()`

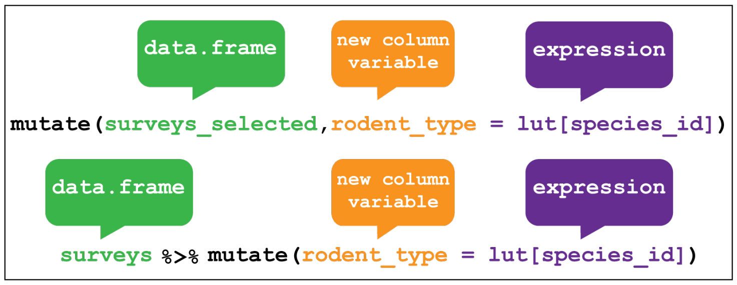

data.frame
new column variable
expression
mutate(surveys_selected,rodent_type = lut[species_id])
data.frame
new column variable
expression
surveys %>% mutate(rodent_type = lut[species_id])

Figure 2.6: `dplyr::mutate()`

```
##   5  1977 Control
##   6  1977 Control
##   7  1977 Control
##   8  1978 Control
##   9  1978 Control
##  10  1978 Control
## # ... with 34,776 more rows
```

```
# Select everything except sex, hindfoot and weight
surveys_selected <- surveys_filtered %>% select(-sex,-hindfoot_length,-weight)
```

2.3.4 Create new variables with mutate()

```
# Mutate surveys_selected
surveys_mutated <- surveys_selected %>%
  #filter(species_id %in% names(lut)) %>%
  mutate(rodent_type = case_when(
         species_id == "DM" ~ "Kangaroo Rat",
         species_id == "DO" ~ "Kangaroo Rat",
         species_id == "DS" ~ "Kangaroo Rat",
         species_id == "PP" ~ "Granivore",
         species_id == "PF" ~ "Granivore",
         species_id == "PE" ~ "Granivore",
         species_id == "PM" ~ "Granivore",
         species_id == "RM" ~ "Granivore",
         TRUE ~ "Other"))

# Check the output using a summary pipe, we should have 8 species of 2 types
surveys_mutated %>% group_by(species_id,rodent_type) %>% summarise()
```

```
## # A tibble: 42 x 2
## # Groups:   species_id [42]
##    species_id rodent_type
##    <chr>      <chr>
##  1 AB         Other
##  2 AH         Other
##  3 AS         Other
##  4 BA         Other
##  5 CB         Other
##  6 CM         Other
##  7 CQ         Other
##  8 CS         Other
##  9 CV         Other
## 10 DM         Kangaroo Rat
## # ... with 32 more rows
```

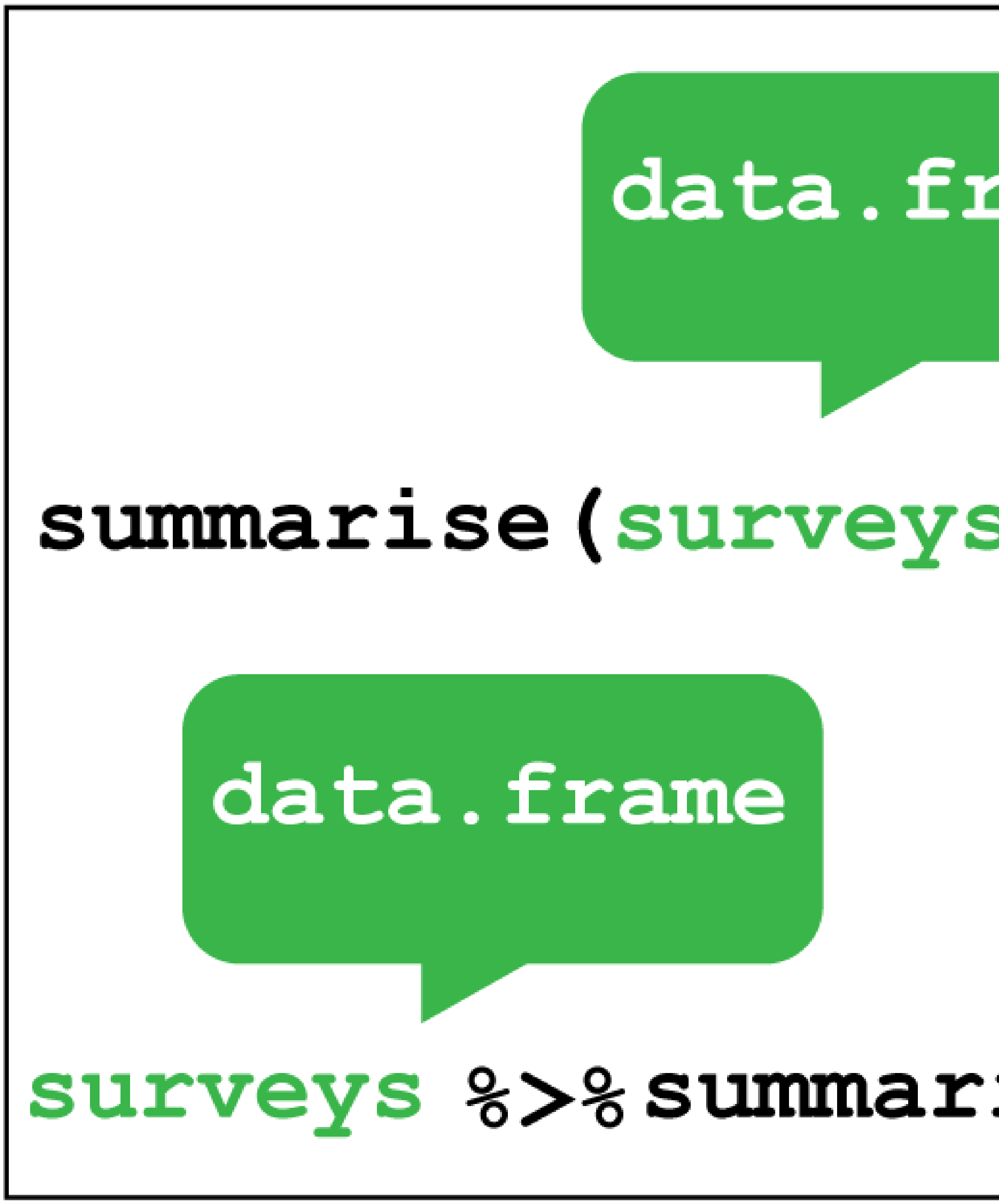
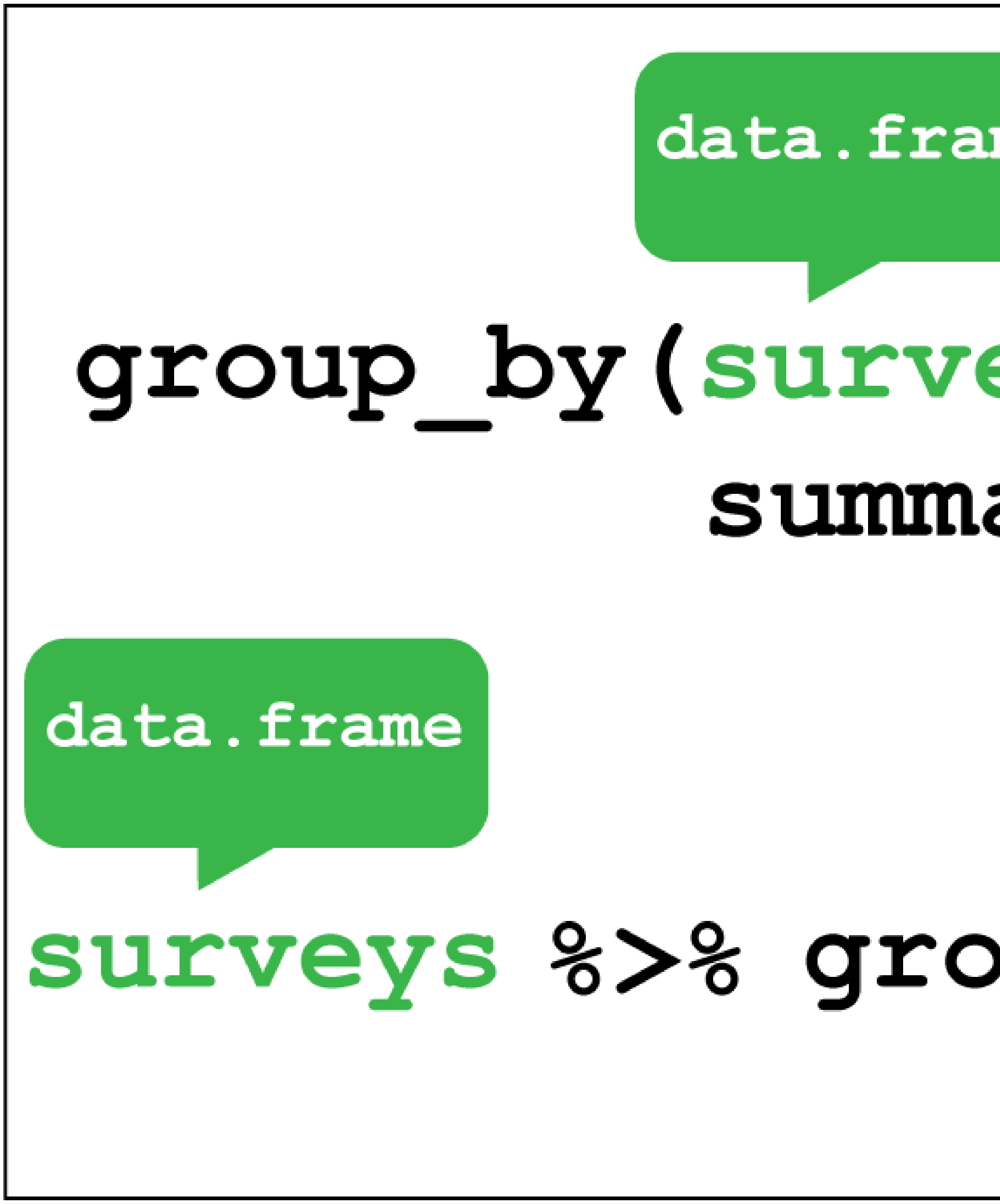


Figure 2.7: `dplyr::group_by()` and `dplyr::summarise()`

2.3.5 Grouped summaries with `group_by()` and `summarise()`

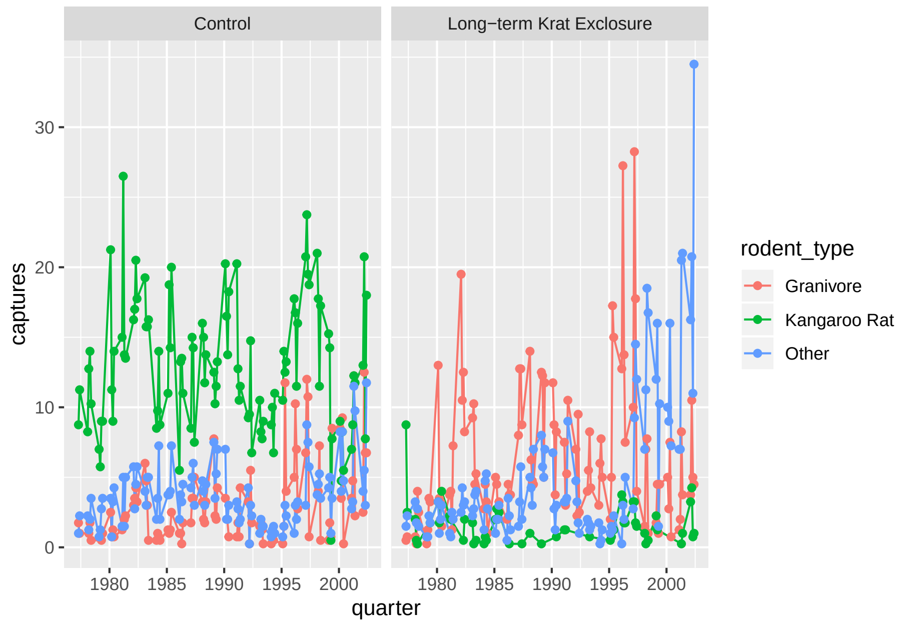


```r
library(lubridate)
surveys_subset <- surveys_mutated %>%
        mutate(date = make_date(day = day, month = month, year = year),
               quarter = quarter(date,with_year = TRUE)) %>%
        filter(!is.na(rodent_type))
```

```r
by_quarter <- surveys_subset %>%
        group_by(rodent_type,plot_type,quarter) %>%
        summarise(captures = n()/4)
```

```r
ggplot(by_quarter,
       aes(x=quarter,y=captures,colour=rodent_type)) +
  geom_line() +
  geom_point() +
  facet_wrap(~ plot_type)
```

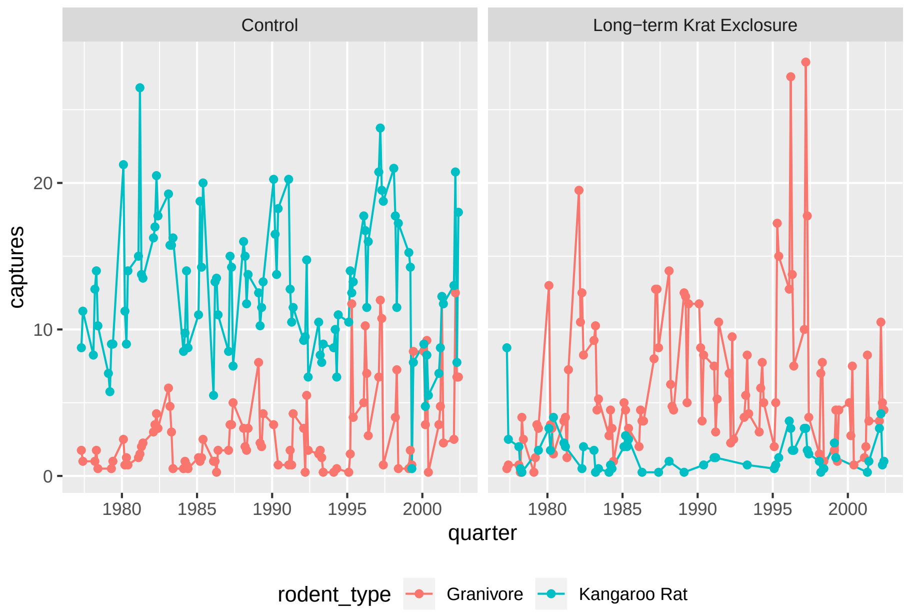
```
## Warning: Removed 2 rows containing missing values (geom_path).

## Warning: Removed 5 rows containing missing values (geom_point).
```



```r
by_quarter %>%
        filter(rodent_type != "Other") %>%
        ggplot(aes(x=quarter,y=captures,colour=rodent_type)) +
    geom_line() +
    geom_point() +
    facet_wrap(~ plot_type) +
        theme(legend.position = "bottom")
```

```
## Warning: Removed 1 rows containing missing values (geom_path).
```

```
## Warning: Removed 3 rows containing missing values (geom_point).
```



Chapter 3

Data wrangling II

Following on from chapter 2 this lesson deals with some common tidying problems with data. By the end of this chapter the learner will:

- have learnt some ways to deal with missing values
- have learnt how to transform rows and columns to reorganise data
- have learnt how to join data contained in separate tables into a single table

Chapter 4

Functions

Chapter 5

Exploratory data analysis

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. (2019). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.13.