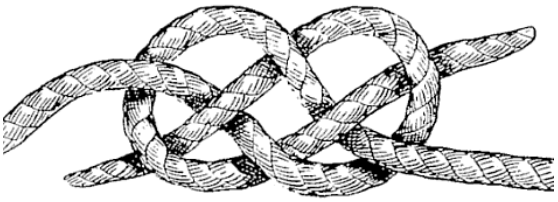
Coding togetheR

Alistair Bailey

November 21 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

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

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].

[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

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"

Bjarne Stroustrup, the inventor C++

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

Figure 1.1: Imposter syndrome[1].

Figure 1.2: R cat[2]

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

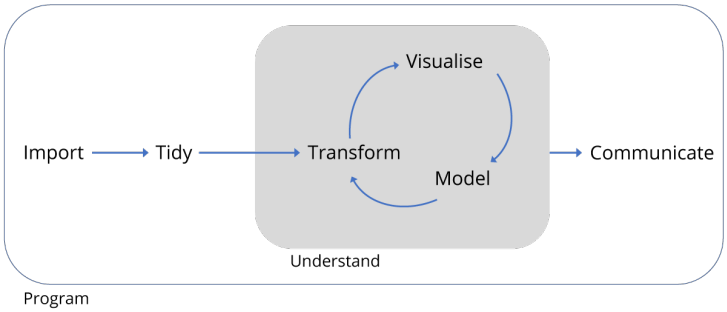


Figure 1.3: Data project workflow.

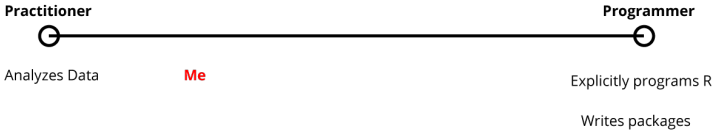


Figure 1.4: The Practitioner-Programmer spectrum

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

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

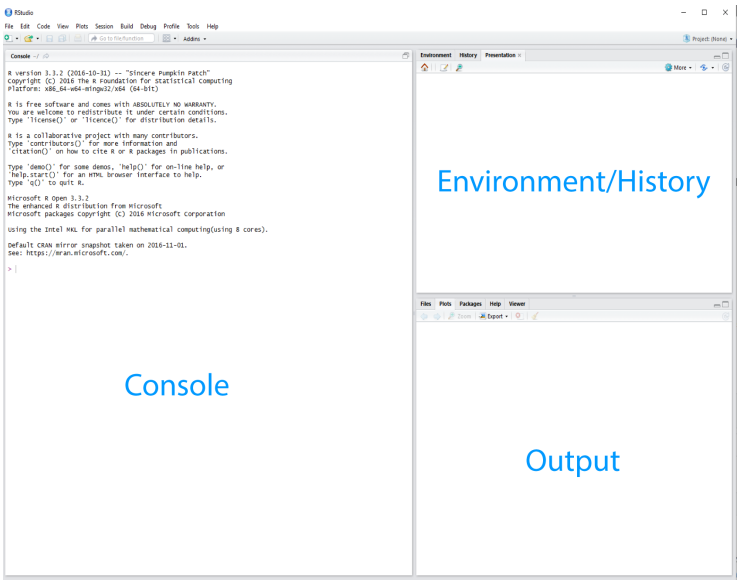


Figure 1.5: The RStudio Integrated Development Environment (IDE).

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 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, 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.

1.3.2 Where am I?

The part of your computer operating system that manages files and directories (aka folders) is called the file system. This dates back to 1969 and the Unix filesystem[9].

The idea is that we have a rooted tree, as with phylogenetic rooted trees[10] in biology. From the root all other directories and files exist along paths going back to the root as shown in Figure 1.7.

On Unix based systems such as Apple or Android, the root is denoted with /. On Windows the root is a back slash \. The / or \ is used to to separate directories along the path, denoting a change in the level of the tree

Note: in RStudio the path separator and root is always / regardless of the operating system.

1.3.2.1 Absolute path from the root /

In Figure 1.7 the absolute path from the root of `surveys.csv` is shown. In text this would be `/users/alistair/Documents/coding-together-week-02/data/surveys.csv`.

This is just a made-up example and the path on your machine will be different.

[9] https://en.wikipedia.org/wiki/Unix_filesystem
[10] https://en.wikipedia.org/wiki/Phylogenetic_tree#Rooted_tree

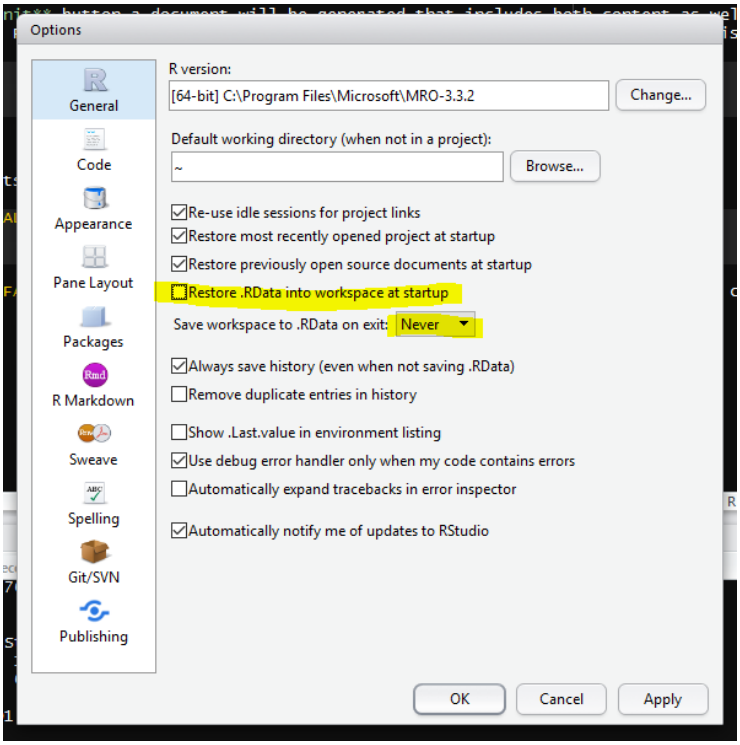


Figure 1.6: Don't save your workspace, save your script instead.

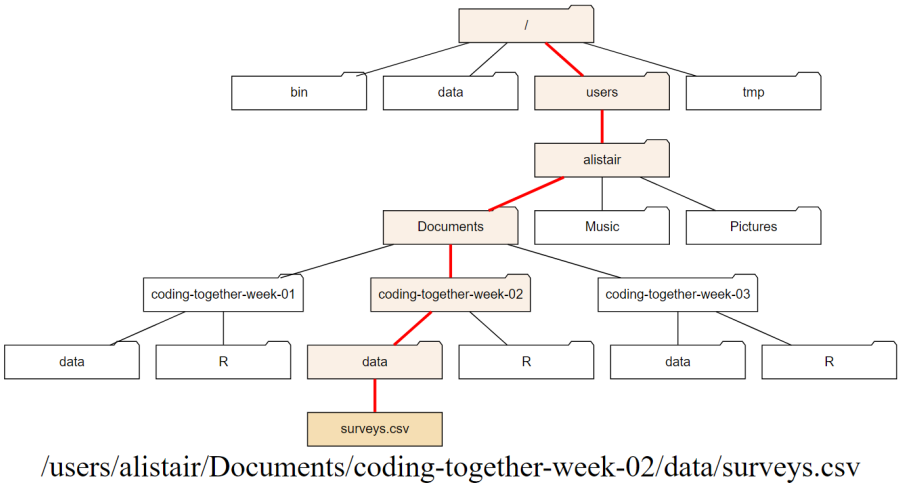


Figure 1.7: The red line shows the absolute path from the root `/` to `surveys.csv`.

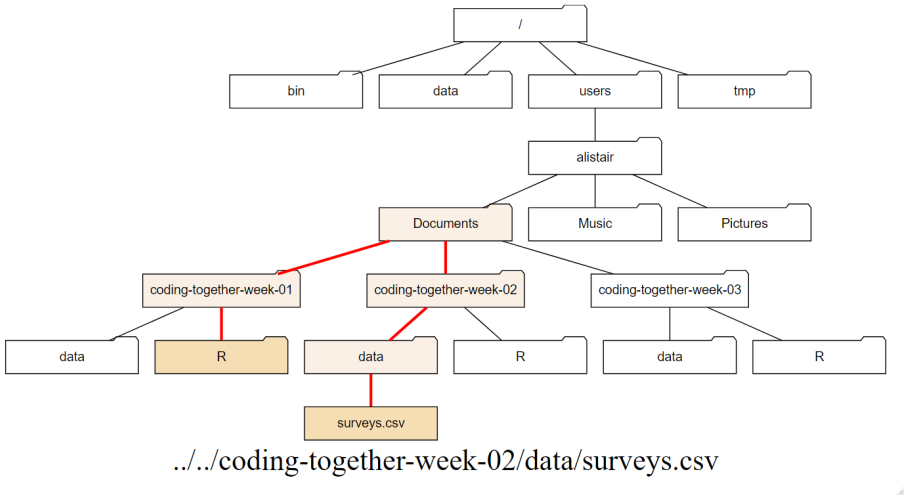


Figure 1.8: The red line represents the relative path from `R` to `surveys.csv`

1.3.2.2 Relative path from where you are `../`

However, we can also consider relative paths, that is relative to where we are working at present.

In Figure 1.8 I imagine that I am working in the `R` directory in `coding-together-week-01` folder. To access the `surveys.csv` file I have to go up two levels to `Documents` and then back down three levels. I could use the absolute path, but *if* I know where `surveys.csv` is then I can use the short-hand of `../` to go up one level, and then again to go up another, and then back down.

In text from the `R` directory this is: `../../coding-together-week-02/data/surveys.csv`

I don't reccomend this, it was just to illustrate what a relative path looks like.

1.3.2.3 Home directory path `~`

The home directory is the part of the file system dedicated to the user - that's us! - and our data. Usually when we get a new computer we'll have a home directory with our user name. On Windows machines this will contain `My Documents` etc. and on an Apple machine it will be `Documents` etc. On a shared machine each user will get a home directory associated with their user name.

Because we are usually working in our home directory and it's tiresome to keep typing out paths from the root and it takes up room on the screen, the tilde `~` denotes the path from the root to our home directory.

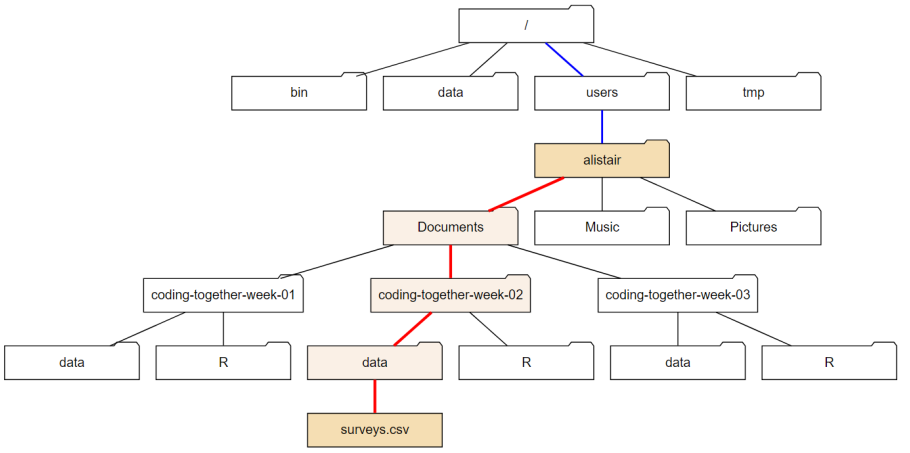


Figure 1.9: The home directory path to the `surveys.csv` with the `~` part in blue and the remaining path in red.

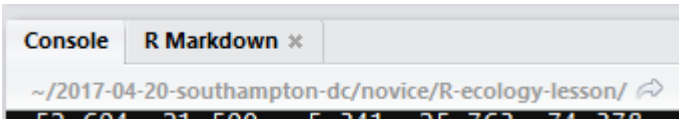


Figure 1.10: Your working directory

We can use the `~` to precede the rest of the path to shortcut an absolute path to the root.

Figure 1.9 indicates a home directory path to the `surveys.csv` with the `~` part in blue and the remaining path in red.

In text this would be: `~/Documents/coding-together-week-02/data/surveys.csv`

1.3.2.4 Working directory

R studio tells you where you are in terms of directory address as shown in Figure 1.10. Here this is a home directory path.

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 1.11. Working this way makes it a lot easier

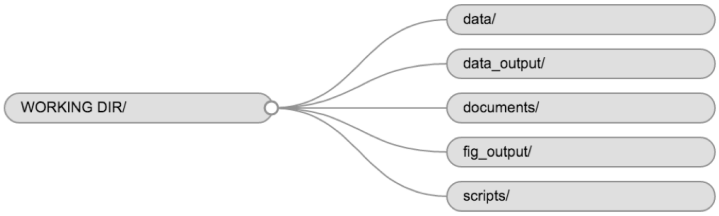


Figure 1.11: A typical directory structure

to move your project around on your computer. Section 1.7 builds upon this to create a robust workflow for data analysis.

1.4 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.

1.5 Using functions

As stated in 1.4 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 1.4 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.

1.6 Getting help

1.6.1 Using `?` to access R function help pages

If you need help with a specific R function, let's say `barplot()`, you can type the function name without round brackets, with a question mark at the start:

```
?barplot
```

1.6.2 Using Google to find R answers

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[11] or Stack Overflow[12].

[11] https://community.rstudio.com/
[12] http://stackoverflow.com/questions/tagged/r

1.6.3 Asking questions

As well as knowing where to ask[13], 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[14] using the `reprex` package.

1.7 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[15] 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**.

[13] https://www.tidyverse.org/help/#where-to-ask
[14] https://www.tidyverse.org/help/#reprex
[15] https://www.tidyverse.org/articles/2017/12/workflow-vs-script/

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?

1.7.1 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 1.7, 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"
- 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`)

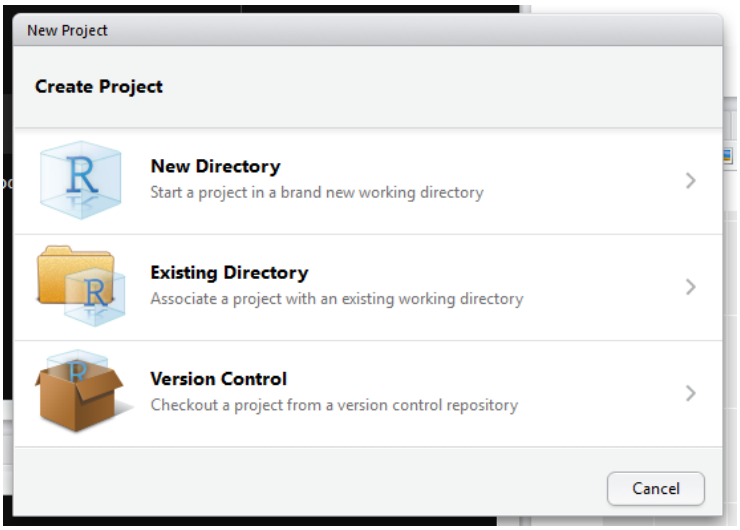


Figure 1.12: Creating a R project

1.7.2 R notebooks and R scripts

R notebooks[16] 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[17]. 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.

1.7.3 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`[18] package by Kirill Müller so much she wrote an ode to it[19].

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?**

[16] https://bookdown.org/yihui/rmarkdown/notebook.html
[17] https://bookdown.org/yihui/rmarkdown/markdown-syntax.html
[18] https://here.r-lib.org/index.html
[19] https://github.com/jennybc/here_here

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.

1.7.4 Naming things

Jenny Bryan[20] also has three principles for naming things[21] 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.

1.8 The tidyverse and tidy data

The tidyverse[22] (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.

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.

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

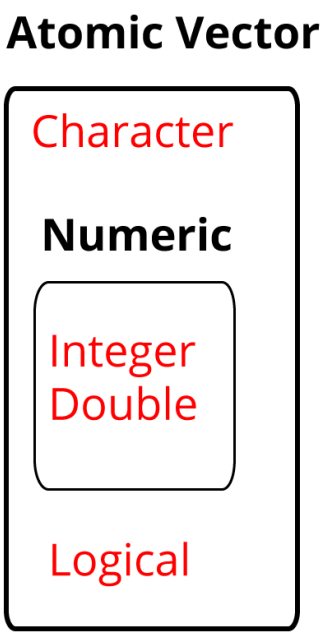


Figure 1.13: The four most used atomic vectors, the building blocks of R

1.9 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 1.10.

> 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.

1.10 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[23], but the reason I favour most is that it provides clarity. The arrow points in the direction of the assignment

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

Figure 1.14: Bibi remains a cat even if I call her Princess when she refuses to go out in the rain.

(it is actually possible to assign in the other direction too) and 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.

1.10.1 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
```

1.11 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 1.15.

Lists alongside `NULL` which indicates the absence of a vector, complete the set of base vectors in R as illustrated in 1.16.

Figure 1.15: List indexing by Hadley Wickham

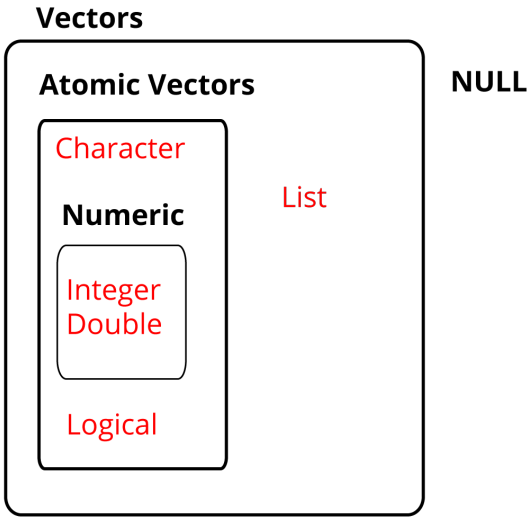


Figure 1.16: The base vectors in R.

1.11.1 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.

1.12 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.

1.13 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 1.17 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 1.17 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`

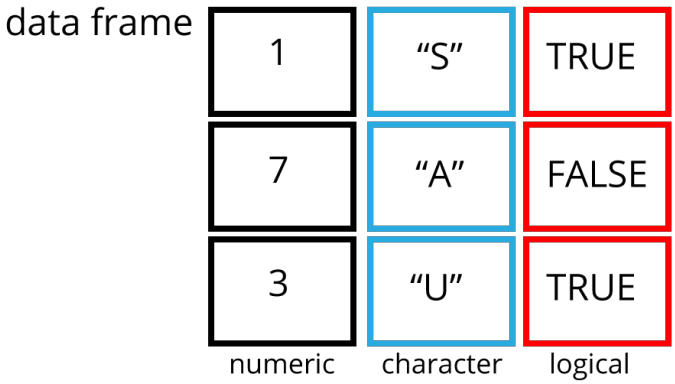


Figure 1.17: An example data frame `df`.

```
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[24]. 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
```

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

```
## # 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
```

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.

```
# 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.

1.13.1 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:

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

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

1.14 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 4, 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?*

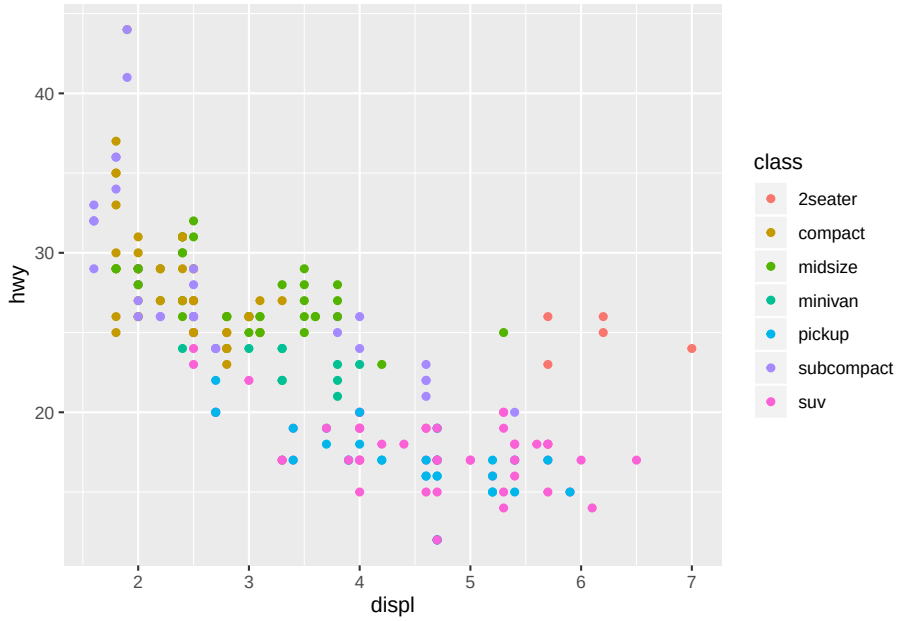
Try `?mpg` to learn more about the data.

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.

```
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.

1.15 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.

1.15.1 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`.

1.15.2 ggsave

If we want to save the last plot we made in `ggplot2` we can use the `ggsave()` function[25]

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")
```

1.15.3 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()

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

7. Save the plot as pdf and the tibble as csv file to the output folder.

Chapter 2

Data wrangling I

By the end of this chapter you will:

- have learnt to load csv files
- 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 and well worth reading for helping you think about best practices for organising your data for yourself, and when working with others.

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

[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

contain only the 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 `.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 from the Portal Project[4], which is a long running survey of rodents and other species in the Chihuahuan Desert, as 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.

A paper with details about the data and the project[5] is also available.

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.

We'll also use some of their data to explore this question: **What is the effect of the exclusion of kangaroo rats from a plot of land on the granivore population?**

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

[4] https://portal.weecology.org/
[5] https://www.biorxiv.org/content/10.1101/332783v1.full

Column	Description	Type
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

The rodents species surveyed are:

Kangaroo 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 e.g `coding-together-week-2`

The dataset is stored on-line, 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 (categorical variables) which is usually what we want.

As we read the data into our environment we need to assign a label to

Figure 2.1: Merriam's kangaroo rat, Dipodomys merriami[6]

Figure 2.2: Kangaroo Rat exclusion

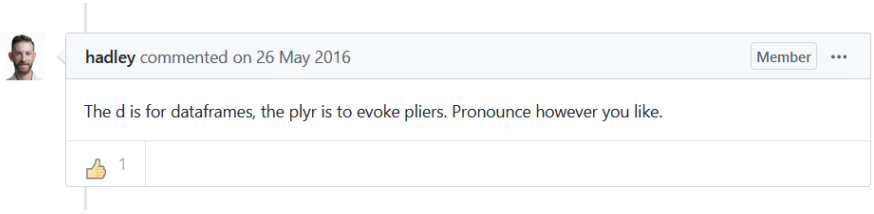


Figure 2.3: [Why is it called 'dplyr'?](https://github.com/tidyverse/dplyr/issues/1857)

the object we are creating. Here we assign the dataset to an object called `surveys` using the <- assignment operator.

```
# 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")
```

2.3 dplyr

`dplyr`[7] *"is a grammar of data manipulation"*. Concretely, it's a package of functions from the tidyverse that have been created for tasks that require manipulation of data stored in data frames[8].

We're going to use the most common verbs in `dplyr` to examine the Portal Project surveys data.

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 2.4 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.

[7] https://dplyr.tidyverse.org/
[8] data-frames

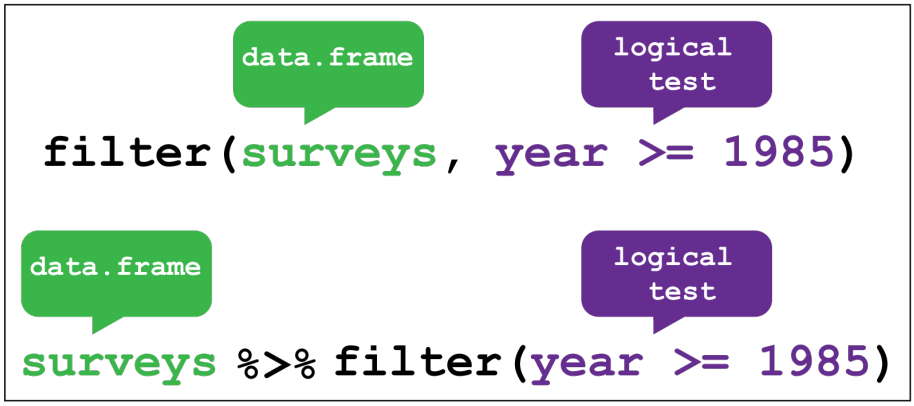


Figure 2.4: `dplyr::filter()`

An alternative way to use `filter()` is to "pipe" the function using pipe %>% from the `magrittr` package which you can think of as using the word *"then"*. We take our data set *then* filter it. Using the pipe makes more sense when combining several operations.

> Ctl+Shift+M is a keyboard short cut to create a pipe.

For the filter itself, from R4DS:

"To use filtering effectively, you have to know how to select the observations t hat you want using the comparison operators. R provides the standard suite: >, >=, <, <=, != (not equal), and == (equal). . . .For other types of combinations, you'll need to use Boolean operators yourself: & is "and", | is "or", and ! is "not"."

See Figure 5.1[9] in R4DS for to see how these operators work.

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

```
# 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
```

[9] https://r4ds.had.co.nz/transform.html#fig:bool-ops

```
##  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
##  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>
```

An example using Boolean logic, would be to use the "or" operator | to filter for the observations only occurring on `plot_type`'s control or long term kangaroo rat exclusion. This time we assign the output to a new data frame called `surveys_filtered`.

Note: as `plot_type` is a character vector we put the terms in quotes, and also the double equals sign == "for equal to".

```
# Keep only the rows corresponding with the Control and Long-term Krat Exclosure
surveys_filtered <- surveys %>%
        filter(plot_type == "Control" | plot_type == "Long-term Krat Exclosure")
```

Note: `filter()` only includes rows where the condition is TRUE; it excludes both `FALSE` and missing `NA` values. We have to explicitly ask to keep `NA` values using `is.na()` as an additional filter.

> How did I know which `plot_type`'s to filter? I used the `dplyr::distinct()` function, passing the `surveys` data frame and the `plot_type` variable to obtain a list of the unique plot types, from which I determined the `Control` and `Long-term Krat Exclosure` plots were the ones I was after. Note that you need to add the argument `.keep_all = TRUE` to `distinct()` to return all the columns in the data frame

```
distinct(surveys, plot_type)
```

```
## # A tibble: 5 x 1
##   plot_type
##   <chr>
## 1 Control
## 2 Long-term Krat Exclosure
## 3 Short-term Krat Exclosure
## 4 Rodent Exclosure
## 5 Spectab exclosure
```

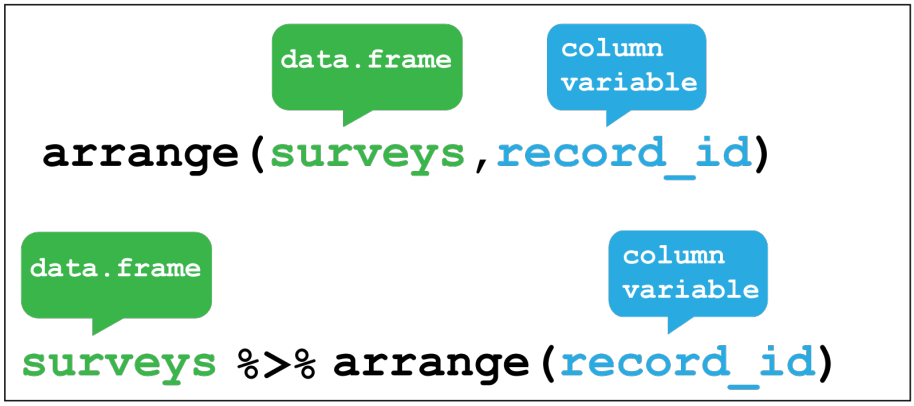


Figure 2.5: `dplyr::arrange()`

2.3.2 Arrange rows with `arrange()`

The next verb is `arrange()` which also operates on the rows, and enables you to arrange the observations in a data frame according to one or more variables.

As with `filter()` we supply the variable or variables of interest as the arguments to `arrange()`.

From R4DS[10]:

"If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns... Missing values are always sorted at the end."

Figure 2.5 shows how to arrange the observations according to the `record_id` variable.

```
surveys %>% arrange(record_id)
```

```
## # 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         1     7    16  1977       2 NL         M                  32
##  2         2     7    16  1977       3 NL         M                  33
##  3         3     7    16  1977       2 DM         F                  37
##  4         4     7    16  1977       7 DM         M                  36
##  5         5     7    16  1977       3 DM         M                  35
##  6         6     7    16  1977       1 PF         M                  14
```

[10] https://r4ds.had.co.nz/transform.html#arrange-rows-with-arrange

```
##  7         7     7    16  1977       2 PE         F                  NA
##  8         8     7    16  1977       1 DM         M                  37
##  9         9     7    16  1977       1 DM         F                  34
## 10        10     7    16  1977       6 PF         F                  20
## # ... with 34,776 more rows, and 5 more variables: weight <dbl>,
## #   genus <chr>, species <chr>, taxa <chr>, plot_type <chr>
```

Or we could use `arrange()` to find the record with the shortest hindfoot. *Note:* `arrange()` defaults to ascending order.

```
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>
```

To find the Cactus Mouse, (`species_id == "PE"`) with the longest hindfoot we combine `filter()` with `arrangee()` using the pipe:

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
```

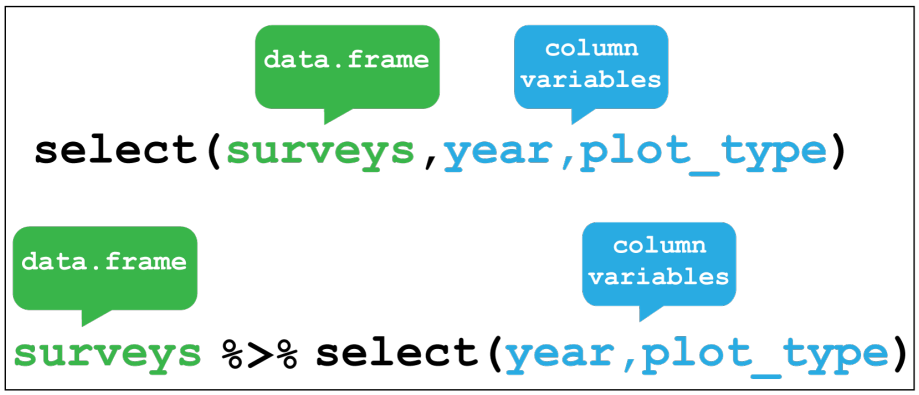


Figure 2.6: `dplyr::select()`

```
##  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()`

Often your data contains variables you don't need for the analysis you are performing, or you want to subset them to share with others. To select only the ones you need, or explore subsets of the variables, the `select()` verb enables you to keep only the columns of interest.

Figure 2.6 shows the use of `select()` to choose only the `year` and `plot_type` columns, with or without the pipe.

Selecting the variables contained in the columns can be done in various ways. For example, by the column number, the variable name or by range. Check the help function `?select` for more options.

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

We can also use negative selection by adding a minus sign - to variables we wish to discard. Here we discard `sex`,`hindfoot_length` and `weight` from the `surveys_filtered` object and keep everything else:

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

```
## # A tibble: 20,729 x 10
##    record_id month   day  year plot_id species_id genus species taxa
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr> <chr>   <chr>
##  1         1     7    16  1977       2 NL         Neot~ albigu~ Rode~
##  2        72     8    19  1977       2 NL         Neot~ albigu~ Rode~
##  3       224     9    13  1977       2 NL         Neot~ albigu~ Rode~
##  4       266    10    16  1977       2 NL         Neot~ albigu~ Rode~
##  5       349    11    12  1977       2 NL         Neot~ albigu~ Rode~
##  6       363    11    12  1977       2 NL         Neot~ albigu~ Rode~
##  7       435    12    10  1977       2 NL         Neot~ albigu~ Rode~
##  8       506     1     8  1978       2 NL         Neot~ albigu~ Rode~
##  9       588     2    18  1978       2 NL         Neot~ albigu~ Rode~
## 10       661     3    11  1978       2 NL         Neot~ albigu~ Rode~
## # ... with 20,719 more rows, and 1 more variable: plot_type <chr>
```

2.3.4 Create new variables with `mutate()`

Another common task is to create a new variable or variables, often from existing data within the data frame. For this we use the `mutate()` verb. It follows the same syntax as for `filter()`, `arrange()` and `select()` in that the first argument is the dataset, and the subsequent arguments are the new variables we wish to create.

Figure 2.7 shows how to create a new variable `weight_kg` by dividing the existing `weight` variable in grams by 1000.

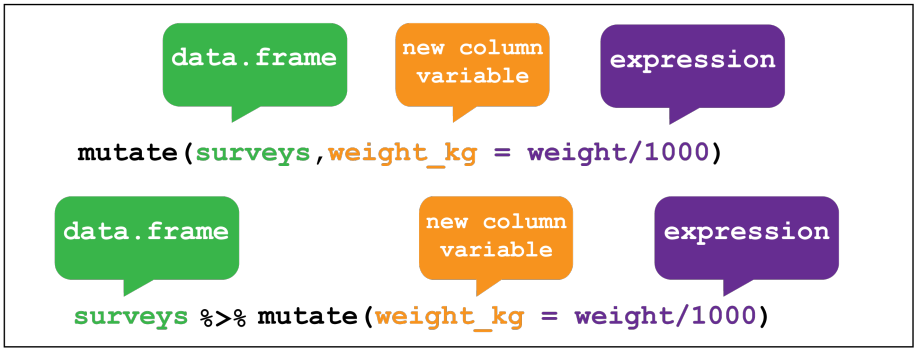


Figure 2.7: `dplyr::mutate()`

A more complicated mutation, and key to to our analysis exploring the question as to whether Kangaroo rats effect the size of the granivore population would be to create a variable that indicates which type of rodent an observation is recording.

To do this we can make use of another `dplyr` function called `case_when()`. This allows us to pass different values to our new `rodent_type` variable **if** they match either `species_id` values corresponding with Kangaroo rats or Granivores.

To remind us, the rodents species surveyed are:

Kangaroo 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

The first argument to `case_when()` is the variable and value we want to match, just like `filter()`, for example `species_id == "DM"`, and then we use the tilde operator `~` followed by the value we want give our new variable **if** a we match this condition. Here we want our new variable `rodent_type` to be "Kangaroo Rat".

We do this for every case we want to match. There are other species than rodents in this data, and we have choice to either provide values for each one, ignore them which will lead to the value `NA` for those rows or we can supply a single value to the rest by giving the argument `TRUE` followed by the value. This means if there are other values in `species_id` - if this is true - then give them all the same value. Here we supply the value "Other" for the remaining species: `TRUE ~ "Other"`.

```
# Mutate surveys_filtered
surveys_mutated <- surveys_filtered %>%
  # Create rodent type variable for K-rats and Granivores.Everything else, Other.
  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 output using distinct()
surveys_mutated %>% select(species_id,rodent_type) %>%
        distinct(species_id, .keep_all = T)
```

```
## # A tibble: 42 x 2
##    species_id rodent_type
##    <chr>      <chr>
##  1 NL         Other
##  2 DM         Kangaroo Rat
##  3 PF         Granivore
##  4 PE         Granivore
##  5 DS         Kangaroo Rat
##  6 PP         Granivore
##  7 SH         Other
##  8 OT         Other
##  9 DO         Kangaroo Rat
## 10 OX         Other
## # ... with 32 more rows
```

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

Finally we'll look at the verb `summarise()` and it's companion `group_by()`.

`summarise()` collapses a data frame into a single row. For example as shown in Figure 2.8, we could use it to find the average weight of all the animals surveyed in the original data frame using `mean()`. (Here the `na.rm = TRUE` argument is given to remove missing values from the data, otherwise R would return `NA` when trying to average.)

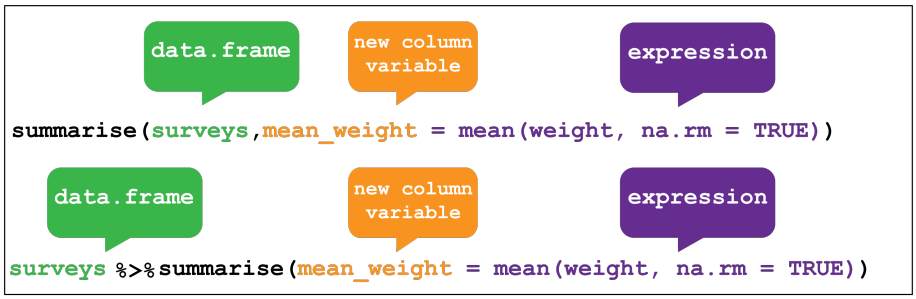


Figure 2.8: `dplyr::summarise()`

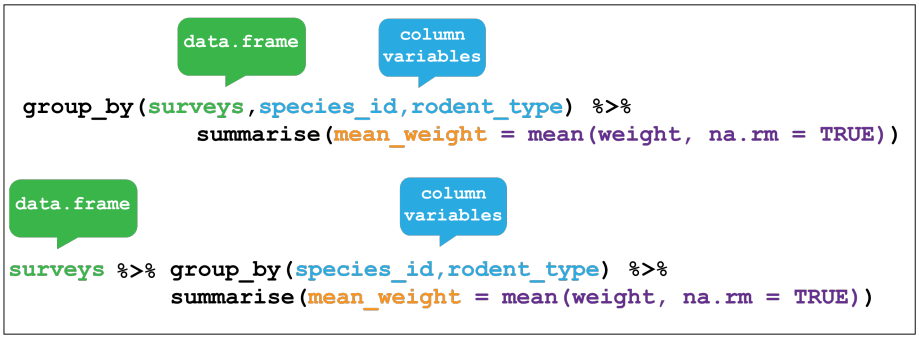


Figure 2.9: `dplyr::group_by()`

```
surveys %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   mean_weight
##         <dbl>
## 1        42.7
```

However `summarise()` is most useful when paired with `group_by()` which defines the variables upon which we operate upon.

Figure 2.9 shows how by grouping the observations according to the `sex` and `species_id` variables, we can then calculate the `mean_weight` for each of these groups.

Using `group_by()` with `summarise()` now returns a table with 23 rows instead of single row. Here I use `drop_na(weight)` to only keep rows with no missing values in `weight`. This ensures that no groups with only `NA` values are returned as `NaN` which is what would happen if we used `mean(weight, na.rm = TRUE)`.

```
surveys_mutated %>%
  group_by(species_id,rodent_type) %>% drop_na(weight) %>%
  summarize(mean_weight = mean(weight))
```

```
## # A tibble: 23 x 3
## # Groups:   species_id [23]
##    species_id rodent_type  mean_weight
##    <chr>      <chr>              <dbl>
##  1 BA         Other               8.76
##  2 DM         Kangaroo Rat       43.4
##  3 DO         Kangaroo Rat       48.7
##  4 DS         Kangaroo Rat      120.
##  5 NL         Other             159.
##  6 OL         Other              31.4
##  7 OT         Other              24.2
##  8 OX         Other              21.2
##  9 PB         Other              31.8
## 10 PE         Granivore          21.4
## # ... with 13 more rows
```

2.4 Using `dpylr` to explore the effect of Kangaroo Rat exclusion on Granivore populations

Let's use what we've learnt so far to explore the effect of Kangaroo Rats on Granivore populations for the entire time covered in the `surveys` dataset.

A line plot with time on the x-axis and number of rodents on the y-axis would be one way to visual this, comparing the observations between the control plots and the Kangaroo rat exclusion plots.

One way to do this is to:

1. `filter()` the observations for the control and exclusion plots.
2. Create a new `rodent_type` variable for Kangaroo Rats and Granivores.
3. Create a new variable for time from the existing `day`,`month` and `year` variables.
4. Group the data according to the `rodent_type`, `plot_type` our time variable, and use `summarise()` to calculate the number of observations for each group.

2.4.1 Re-cap of `filter()` and `mutate()`

Let's re-cap steps one and two:

```
# Keep only the rows corresponding with the Control and Long-term Krat Exclosure
surveys_filtered <- surveys %>%
        filter(plot_type == "Control" | plot_type == "Long-term Krat Exclosure")

# Mutate surveys_filtered
surveys_mutated <- surveys_filtered %>%
  # Create rodent type variable for K-rats and Granivores.Everything else, Other.
  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"))
```

2.4.2 Use `lubridate` to create new time variables

Step three introduces the tidyverse `lubridate` package[11]. As the name suggests, this is a package for wrangling dates and times.

It would be clearer to plot the data on three month (quarterly) basis rather than plotting every date in the dataset, so we need to create a variable that contains the quarter in which the observation was made, for each observation.

From `lubridate` we will use the function `make_date()` in combination with `mutate()` first to create a single column `date` variable from the `day`,`month` and `year` variables. We then use this `date` variable to create another new variable containing a value for the quarter of the year in which the observation was made `quarter` using the `quarter()` function.

We'll assign this output to a new data frame called `surveys_subset`.

```
library(lubridate)
surveys_subset <- surveys_mutated %>%
        mutate(date = make_date(day = day, month = month, year = year),
               quarter = quarter(date,with_year = TRUE))
```

[11] https://lubridate.tidyverse.org/

2.4.3 Group and summarise the data into quarterly observations

Step four is to group and summarise our quarterly observations.

We group by `rodent_type`, `plot_type` and `quarter` variables. In other words we've grouped the data according to Kangaroo Rat or Granivore, Control plot or Exclusion plot, and the quarter of the year in which the observation occurred.

Then we can calculate the number of captures for each of these groups by using `summarise()` to create a `mean_captures` variable which is equal to the number of rows for that group using the `n()` function. The value of `n` represents the total number of captures for each group per quarter. A quarter of a year is 3 months, so we divide by 3 to calculate the average captures per quarter.

Note: `n()` is a `dplyr` function that returns the number of observations in the current group.

```
by_quarter <- surveys_subset %>%
        group_by(rodent_type,plot_type,quarter) %>%
        summarise(mean_captures = n()/3)

by_quarter
```

```
## # A tibble: 567 x 4
## # Groups:   rodent_type, plot_type [6]
##    rodent_type plot_type quarter mean_captures
##    <chr>       <chr>       <dbl>         <dbl>
##  1 Granivore   Control     1977.          5
##  2 Granivore   Control     1977.          2.67
##  3 Granivore   Control     1978.          3.67
##  4 Granivore   Control     1978.          3
##  5 Granivore   Control     1978.          4.33
##  6 Granivore   Control     1978.          1.33
##  7 Granivore   Control     1979.          1.67
##  8 Granivore   Control     1979.          2
##  9 Granivore   Control     1979.          1.67
## 10 Granivore   Control     1980.          7.33
## # ... with 557 more rows
```

These steps have taken us from a table of 34,786 observations to a table of 567 observations.

2.4.4 Create a plot with `ggplot`

Now we can create a line and point plot, using the `by_quarter` data as our first `ggplot()` argument.

For the aesthetics we are plotting the time on the x-axis using the `quarter` variable, and the quarterly `mean_captures` on the y-axis, and we colour the data by `plot_type`.

Then we create line and point geometric mappings, and split the plot into two facets using `facet_wrap` according to `rodent_type`.

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

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

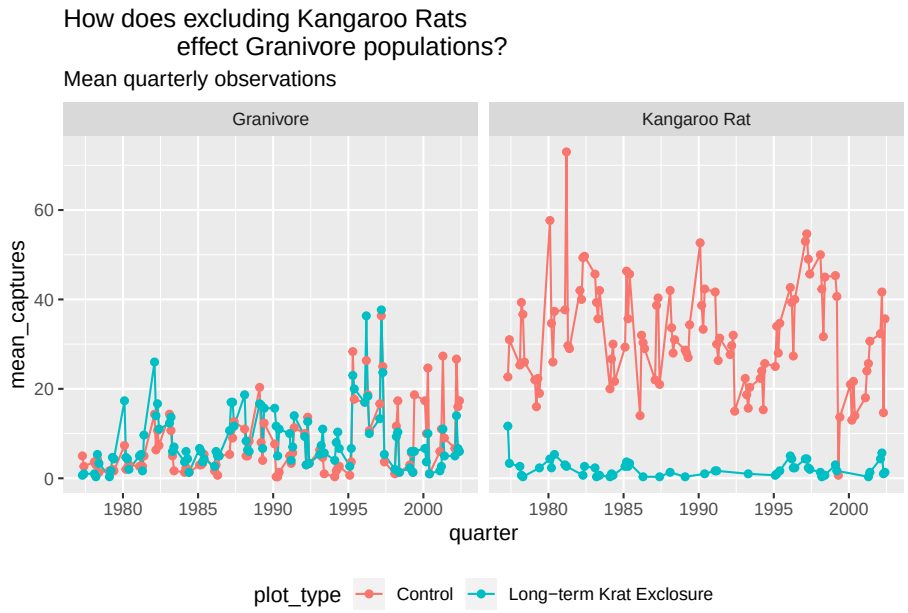
```
## Warning: Removed 5 rows containing missing values (geom_point).
```

But it would be more useful to only plot the Kangaroo Rats and Granivores data, so let's filter out the other species. And move the legend to the bottom of the plot and add a title.

```
by_quarter %>%
        filter(rodent_type != "Other") %>%
        ggplot(aes(x=quarter,y=mean_captures,colour=plot_type)) +
  geom_line() +
  geom_point() +
  facet_wrap(~ rodent_type) +
        theme(legend.position = "bottom") +
        ggtitle("How does excluding Kangaroo Rats
                effect Granivore populations?",
                subtitle = "Mean quarterly observations")
```

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

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



Challenge Can you do a similar analysis, but plotting the data only from 1980 to 2000 and by semester?

Chapter 3

Data wrangling II

Following on from chapter 2 this lesson deals with some common tidying problems with data using the `tidyr` package[1] . By the end of this chapter the learner will:

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

3.1 Reshaping data with pivots

As you will recall tidy data[2] means that in a table:

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

Two common problems making your data untidy are that:

1. A single variable is spread across multiple columnns.
2. A single observation is distributed across multiple rows.

In the first case the column names are actually the variable we are interested in, making our table **wide**.

[1] https://tidyr.tidyverse.org/
[2] tidy-data

In the second case, the observation is contained in several rows instead of a single row, making our table **long**.

Hence `tidyr` has functions that **pivot** (as in turning) a table from **wide-to-long** by reducing the number of columns and increasing the number of rows. Or from **long-to-wide** by reducing the number of rows and increasing the number of columns.

Usually, the hard part is identifying in a dataset our variables and our observations, as it is not always obvious which is which.

3.1.1 `tidyr::pivot_longer`

First we'll consider the case when our variable has been recorded as column names by returning to a version of the Portal surveys data[3].

A table containing the mean weight of 10 rodent species on each plot from the rodent survey data can be read directly into a `surveys_spread` object using the code below.

```
surveys_spread <- read_csv("https://raw.githubusercontent.com/ab604/ab604.github.io/mas
```

Feel free to explore the data, and snapshot is shown in Figure 3.1.

Imagine that we recorded the mean weight of each rodent species on a plot in the field. It makes sense to put the species as column headings, along with the plot id and then record the values in each cell.

However, really our variables of interest are the rodent species and our observational units, the rows, should contain the mean weight for a rodent species in a plot. Hence we need to reduce the number of columns and create a longer table.

To do this we `pivot_longer()` by using `names_to = "genus"` to create a new `genus` variable for the exisiting column heading, and `values_to = "mean_weight"` to create a variable `mean_weight` for the values. We put a minus sign before the variable `-plot_id` to tell the function not to use these values in the new variable column.

```
surveys_spread %>%
        pivot_longer(names_to = "genus", values_to = "mean_weight", -plot_id)
```

```
## # A tibble: 240 x 3
##    plot_id genus           mean_weight
##      <dbl> <chr>                 <dbl>
```

[3] portal-project

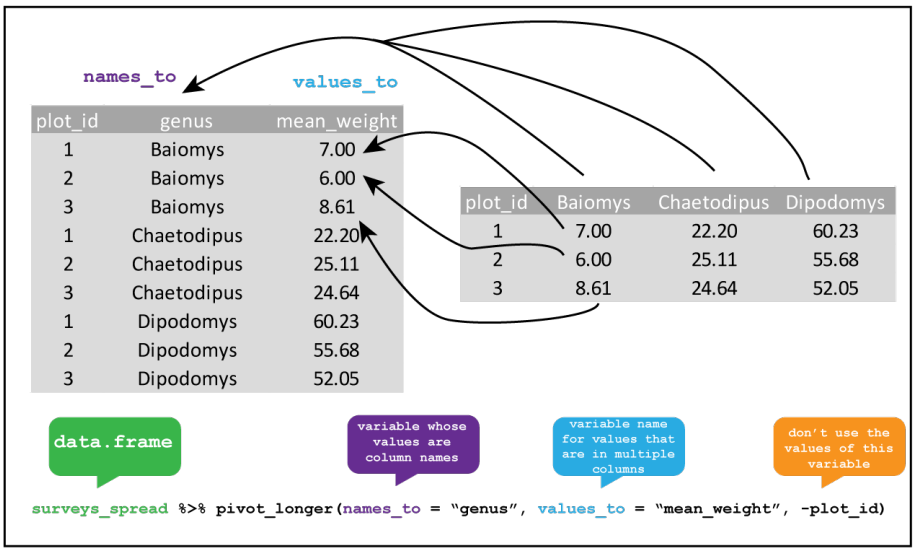


Figure 3.1: `tidyr::pivot_longer` The genus of the rodents have been used as column headings for the weights of each animal. By creating new variables `genus` which takes the column names, and another variable `mean_weight` which takes the cell values, and but not the `plot_id` column we pivot from a wide table to a long table.

```
##  1       1 Baiomys               7
##  2       1 Chaetodipus          22.2
##  3       1 Dipodomys            60.2
##  4       1 Neotoma             156.
##  5       1 Onychomys            27.7
##  6       1 Perognathus           9.62
##  7       1 Peromyscus           22.2
##  8       1 Reithrodontomys      11.4
##  9       1 Sigmodon             NA
## 10       1 Spermophilus         NA
## # ... with 230 more rows
```

3.1.2 tidyr::pivot_wider

If we wanted to go the other way, from wide to long, then we need to take values from a single column to become variables names of multiple columns, populated with values from an existing variable.

A version of the wide table in Figure 3.2 can be downloaded into our environment and assigned as `surveys_gw` as before.

```
surveys_gw <- read_csv("https://raw.githubusercontent.com/ab604/ab604.github.io/master/
```

This time we use `pivot_wider()` with the arguments `names_from` for the column we want to use to create variables, and `values_from` for the column whose values we want populate our new variable columns with.

```
surveys_gw %>%
        pivot_wider(names_from = genus, values_from = mean_weight)
```

```
## # A tibble: 24 x 11
##    plot_id Baiomys Chaetodipus Dipodomys Neotoma Onychomys Perognathus
##      <dbl>   <dbl>       <dbl>     <dbl>   <dbl>     <dbl>       <dbl>
##  1       1    7           22.2      60.2    156.      27.7        9.62
##  2       2    6           25.1      55.7    169.      26.9        6.95
##  3       3    8.61        24.6      52.0    158.      26.0        7.51
##  4       5    7.75        18.0      51.1    190.      27.0        8.66
##  5      18    9.5         26.8      61.4    149.      26.6        8.62
##  6      19    9.53        26.4      43.3    120       23.8        8.09
##  7      20    6           25.1      65.9    155.      25.2        8.14
##  8      21    6.67        28.2      42.7    138.      24.6        9.19
##  9       4   NA           23.0      57.5    164.      28.1        7.82
## 10       6   NA           24.9      58.6    180.      25.9        7.81
## # ... with 14 more rows, and 4 more variables: Peromyscus <dbl>,
## #   Reithrodontomys <dbl>, Sigmodon <dbl>, Spermophilus <dbl>
```

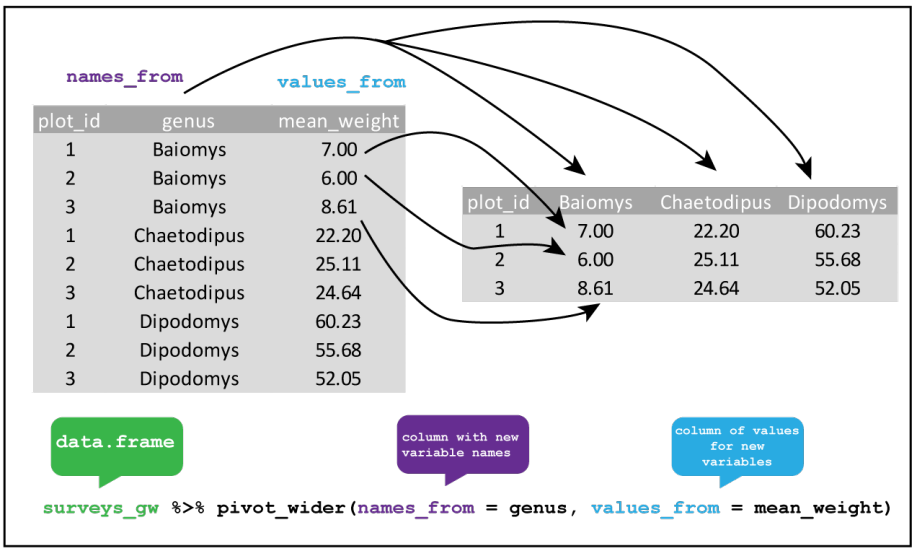


Figure 3.2: `tidyr::pivot_wider` From long table `surveys_gw` the `genus` column contains the values that become variables using `names_from` and the `mean_weight` column contains the values that fill the new columns using `values_from` in the pivot from long to wide.

3.2 Missing values

In R missing values are represented as `NA` (or `NaN` for an undefined mathematical operation such as dividing by zero).

As discussed in R4DS[4] values can be missing in two ways:

- **Explicitly**, i.e. flagged with NA.
- **Implicitly**, i.e. simply not present in the data.

If we had a subset of the surveys data that looked like this table:

```
# Create a table with missing values
surveys_ms <- tibble(year = c(1991,1991,1991,1991,1992,1992,1992),
                     qtr = c(1,2,3,4,1,2,4),
                     mean_weight = c(3.75,2.50,NA,8.50,7.50,2.25,2.50))

surveys_ms
```

```
## # A tibble: 7 x 3
```

[4] https://r4ds.had.co.nz/tidy-data.html#missing-values-3

```
##    year   qtr mean_weight
##   <dbl> <dbl>       <dbl>
## 1  1991     1        3.75
## 2  1991     2        2.5
## 3  1991     3       NA
## 4  1991     4        8.5
## 5  1992     1        7.5
## 6  1992     2        2.25
## 7  1992     4        2.5
```

Then we can see that in the third quarter of 1991, no data was recorded by the explicit `NA` value. However the third quarter of 1992 is missing altogether, hence it is *implicitly* missing.

Another issue is that explicit missing values are often recorded in real data in various ways e.g. an empty cell or as a dash. We can't cover all cases, but the general advice is to do some manual inspection of your data in addition to using R to understand how missing values have been recorded if there are any. In an ideal world data comes with a "code book" that explains the data, but this often doesn't happen. Functions such as `read_csv()` allow you to supply a vector of values representing `NA` which will convert these values in your input to `NA`s.

Once you have identified missing values, in general there two ways to deal with them:

1. Drop incomplete observations
2. Complete the missing observations: fill or impute values.

See also `tidyr` functions for missing values[5].

3.2.1 Checking for explicit missing values using R

In addition to manual inspection, one way to check for explicit missing values coded as `NA` in your data frame is to use the `is.na()` function which returns a logical `TRUE` or `FALSE` vector of values.

Using this to check a single variable could be done by combining it with `select()` for example using `surveys_ms`:

```
surveys_ms %>%
        select(mean_weight) %>%
        is.na()
```

[5] https://tidyr.tidyverse.org/reference/index.html#section-missing-values

```
##      mean_weight
## [1,]       FALSE
## [2,]       FALSE
## [3,]        TRUE
## [4,]       FALSE
## [5,]       FALSE
## [6,]       FALSE
## [7,]       FALSE
```

Or with `filter()` to return the row where there is a missing value for `mean_weight`.

```
surveys_ms %>%
        filter(is.na(mean_weight) == TRUE)
```

```
## # A tibble: 1 x 3
##    year   qtr mean_weight
##   <dbl> <dbl>       <dbl>
## 1  1991     3          NA
```

A more complicated use would be to combine `is.na()` with another function such `sum()` to provide the total number of missing values for a variable, or to combine `sum()` with `complete.cases()` to check each row for for missing values.

For example, to find the number of observations (rows) in `surveys` with no missing values:

```
sum(complete.cases(surveys))
```

```
## [1] 30676
```

And for a single variable, we could extend our use of `select`:

```
surveys_ms %>%
        select(mean_weight) %>%
        is.na() %>%
        sum()
```

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

Extending this find the missing values per variable in a whole data frame requires introducing more syntax and the use of a map function from the

`purr` package. (Base R can also do this, but we're staying in the tidyverse where possible).

The idea here is that we want to do the same calculation *mapped* across each variable. The calculation here being count the number of missing values in each column.

We pipe `surveys` to the `map_dfr()` function where `dfr` means return a data frame. The `~` represents formula, meaning *"depends upon"*, such that `sum(is.na(.))` where `.` represents the input variable, depends upon the variables in `surveys`. The map function then does this for each variable in the `surveys` dataframe and returns the output as a data frame. I've piped the output to `glimpse()` for readability.

In words what this syntax is saying is **"take the surveys dataframe and for each column check if each value is `NA`, then sum the number of `NA`s in each column and return the totals for all the columns as a dataframe."**

This is pretty complicated if you've never seen this before, but hopefully you can follow the idea.

This could be combined with `filter()` or `select()` prior to the map function if you wanted to subset the data further for example.

See R4DS map functions[6] for more details.

```
surveys %>%
        map_dfr(~ sum(is.na(.))) %>%
        glimpse()
```

```
## Observations: 1
## Variables: 13
## $ record_id       <int> 0
## $ month           <int> 0
## $ day             <int> 0
## $ year            <int> 0
## $ plot_id         <int> 0
## $ species_id      <int> 0
## $ sex             <int> 1748
## $ hindfoot_length <int> 3348
## $ weight          <int> 2503
## $ genus           <int> 0
## $ species         <int> 0
## $ taxa            <int> 0
## $ plot_type       <int> 0
```

[6] https://r4ds.had.co.nz/iteration.html#the-map-functions

As we might expect the missing values are all for variables that record measurements for the rodents.

3.2.2 Dropping missing values

The simplest solution to observations with missing values is to drop those from the data set. If it makes sense to do this, then `tidyr::drop_na()` makes it easy to do this. **Note:** This will create implicit missing values.

In Section 3.2.1 we found 30,676 rows with no missing values by using `complete.cases()` on the `surveys` data of 34,786 observations.

Therefore we expect passing `surveys` to `drop_na()` will return a dataframe of 30,676 observations:

```
surveys %>%
        drop_na() %>%
        glimpse()
```

```
## Observations: 30,676
## Variables: 13
## $ record_id       <dbl> 845, 1164, 1261, 1756, 1818, 1882, 2133, 2184,...
## $ month           <dbl> 5, 8, 9, 4, 5, 7, 10, 11, 1, 5, 5, 7, 10, 11, ...
## $ day             <dbl> 6, 5, 4, 29, 30, 4, 25, 17, 16, 18, 18, 8, 1, ...
## $ year            <dbl> 1978, 1978, 1978, 1979, 1979, 1979, 1979, 1979...
## $ plot_id         <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2...
## $ species_id      <chr> "NL", "NL", "NL", "NL", "NL", "NL", "NL", "NL"...
## $ sex             <chr> "M", "M", "M", "M", "M", "M", "F", "F", "F", "...
## $ hindfoot_length <dbl> 32, 34, 32, 33, 32, 32, 33, 30, 33, 31, 33, 30...
## $ weight          <dbl> 204, 199, 197, 166, 184, 206, 274, 186, 184, 8...
## $ genus           <chr> "Neotoma", "Neotoma", "Neotoma", "Neotoma", "N...
## $ species         <chr> "albigula", "albigula", "albigula", "albigula"...
## $ taxa            <chr> "Rodent", "Rodent", "Rodent", "Rodent", "Roden...
## $ plot_type       <chr> "Control", "Control", "Control", "Control", "C...
```

`drop_na()` can accept variables arguments, meaning only the observations with missing values in those columns will be dropped.

For example, here we drop only missing `weight` observations, so the rows which have missing values for `hindfoot_length` or `sex` are kept.

```
surveys %>%
        drop_na(weight) %>%
        glimpse()
```

```
## Observations: 32,283
## Variables: 13
## $ record_id       <dbl> 588, 845, 990, 1164, 1261, 1453, 1756, 1818, 1...
## $ month           <dbl> 2, 5, 6, 8, 9, 11, 4, 5, 7, 10, 11, 1, 5, 5, 7...
## $ day             <dbl> 18, 6, 9, 5, 4, 5, 29, 30, 4, 25, 17, 16, 18, ...
## $ year            <dbl> 1978, 1978, 1978, 1978, 1978, 1978, 1979, 1979...
## $ plot_id         <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2...
## $ species_id      <chr> "NL", "NL", "NL", "NL", "NL", "NL", "NL", "NL"...
## $ sex             <chr> "M", "M", "M", "M", "M", "M", "M", "M", "M", "...
## $ hindfoot_length <dbl> NA, 32, NA, 34, 32, NA, 33, 32, 32, 33, 30, 33...
## $ weight          <dbl> 218, 204, 200, 199, 197, 218, 166, 184, 206, 2...
## $ genus           <chr> "Neotoma", "Neotoma", "Neotoma", "Neotoma", "N...
## $ species         <chr> "albigula", "albigula", "albigula", "albigula"...
## $ taxa            <chr> "Rodent", "Rodent", "Rodent", "Rodent", "Roden...
## $ plot_type       <chr> "Control", "Control", "Control", "Control", "C...
```

3.2.3 Completing missing values

There are numerous ways to complete missing values, but what they generally have in common is the approach of guessing the missing value based on existing information, otherwise known as imputation[7]. Here we'll cover a couple of ways to do it with `dplyr` and `tidyr` functions.

First, let's deal with implicit missing values using `tidyr::complete()`.

Imagine we have the table `surveys_ms` which is missing the value for the `mean_weight` for the third quarter of 1991 and all observations for the third quarter of 1992:

```
surveys_ms
```

```
## # A tibble: 7 x 3
##    year   qtr mean_weight
##   <dbl> <dbl>       <dbl>
## 1  1991     1        3.75
## 2  1991     2        2.5
## 3  1991     3       NA
## 4  1991     4        8.5
## 5  1992     1        7.5
## 6  1992     2        2.25
## 7  1992     4        2.5
```

To add the row for the third quarter of 1992 we use `complete()` with the `year` and `qtr` variables as arguments. The function finds all the unique

[7] https://en.wikipedia.org/wiki/Imputation_(statistics)

combinations of `year` and `qtr` and then adds any that are missing to create a complete set of observations for 1991 and 1992 with explicit missing values.

```
surveys_ms %>%
        complete(year,qtr)
```

```
## # A tibble: 8 x 3
##    year   qtr mean_weight
##   <dbl> <dbl>       <dbl>
## 1  1991     1        3.75
## 2  1991     2        2.5
## 3  1991     3       NA
## 4  1991     4        8.5
## 5  1992     1        7.5
## 6  1992     2        2.25
## 7  1992     3       NA
## 8  1992     4        2.5
```

Next let's consider a complete table `surveys_ic` with an explicit missing value for the `species_id`.

```
surveys_ic <- tibble(species_id = c("DM","DM",NA,"DS","DS","DS"),
                     mean_weight = c(3.75,2.50,8.50,7.50,5.50,2.50))
surveys_ic
```

```
## # A tibble: 6 x 2
##   species_id mean_weight
##   <chr>            <dbl>
## 1 DM                3.75
## 2 DM                2.5
## 3 <NA>              8.5
## 4 DS                7.5
## 5 DS                5.5
## 6 DS                2.5
```

In this case we might reasonably assume that the missing value is `DM` as we have six observations on two species.

We can use `tidyr::fill()` to replace `NA` with the last non-missing value for `species_id`. This is also known as last observation carried forward.

Passing `species_id` to `fill()` completes the table:

```
surveys_ic %>%
       fill(species_id)
```

```
## # A tibble: 6 x 2
##   species_id mean_weight
##   <chr>            <dbl>
## 1 DM                3.75
## 2 DM                2.5
## 3 DM                8.5
## 4 DS                7.5
## 5 DS               NA
## 6 DS                2.5
```

Another common strategy is to impute missing values by using the mean or the median of exisiting values for the same variable.

We can do that using `dplyr::coalesce()`[8] which is a function that finds the first non-missing value in a variable and then replaces it using the second argument.

For example, if we return to `surveys_ms` we used `complete()` to complete the observations, but we have two `NA`s in the `mean_weight` column.

Let's impute the missing values bt using the median value for `mean_weight` to replace the `NA` values.

We can `mutate()` to overwrite the `mean_weight` variable (be careful when you do this!) using `coalesce()` with `mean_weight` as the first argument to look for missing values, and then the `median()` function is used to replace the missing values with the median `mean_weight`, remembering to remove `NA`.

```
surveys_ms %>%
       complete(year,qtr) %>%
       mutate(mean_weight = coalesce(mean_weight,
                                     median(mean_weight, na.rm = TRUE)))
```

```
## # A tibble: 8 x 3
##    year   qtr mean_weight
##   <dbl> <dbl>       <dbl>
## 1  1991     1        3.75
## 2  1991     2        2.5
## 3  1991     3        3.12
## 4  1991     4        8.5
## 5  1992     1        7.5
```

[8] https://dplyr.tidyverse.org/reference/coalesce.html

```
## 6  1992     2        2.25
## 7  1992     3        3.12
## 8  1992     4        2.5
```

This gives us a complete table with . Whether it makes sense to do this is another question, and you should think carefully about your missing value strategy as it will influence your final output and conclusions.

Let's repeat part of the analysis we did to find the `mean_weight`, and now `mean_hindfoot`, for the full `surveys` dataset, but this time we'll complete the table by imputing values and compare the result to the result when we drop the observations with missing values.

For simplicity, let's just look at kangaroo rats. Here I introduce the `%in%` operator with filter and vector containing the `species_id` for the kangaroo rats.

This leaves 16,127 observations.

```
krats <- surveys %>%
        filter(species_id %in% c("DM","DS","DO"))

krats %>% glimpse()
```

```
## Observations: 16,127
## Variables: 13
## $ record_id       <dbl> 3, 226, 233, 245, 251, 257, 259, 268, 346, 350...
## $ month           <dbl> 7, 9, 9, 10, 10, 10, 10, 10, 11, 11, 11, 11, 1...
## $ day             <dbl> 16, 13, 13, 16, 16, 16, 16, 16, 12, 12, 12, 12...
## $ year            <dbl> 1977, 1977, 1977, 1977, 1977, 1977, 1977, 1977...
## $ plot_id         <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2...
## $ species_id      <chr> "DM", "DM", "DM", "DM", "DM", "DM", "DM", "DM"...
## $ sex             <chr> "F", "M", "M", "M", "M", "M", "M", "F", "F", "...
## $ hindfoot_length <dbl> 37, 37, 25, 37, 36, 37, 36, 36, 37, 37, 38, 36...
## $ weight          <dbl> NA, 51, 44, 39, 49, 47, 41, 55, 36, 47, 44, 40...
## $ genus           <chr> "Dipodomys", "Dipodomys", "Dipodomys", "Dipodo...
## $ species         <chr> "merriami", "merriami", "merriami", "merriami"...
## $ taxa            <chr> "Rodent", "Rodent", "Rodent", "Rodent", "Roden...
## $ plot_type       <chr> "Control", "Control", "Control", "Control", "C...
```

How many missing values are there in `krats` and for which variables?

```
# How many missing values
krats %>% map_dfr(~ sum(is.na(.))) %>% glimpse()
```

```
## Observations: 1
```

```
## Variables: 13
## $ record_id       <int> 0
## $ month           <int> 0
## $ day             <int> 0
## $ year            <int> 0
## $ plot_id         <int> 0
## $ species_id      <int> 0
## $ sex             <int> 131
## $ hindfoot_length <int> 1136
## $ weight          <int> 617
## $ genus           <int> 0
## $ species         <int> 0
## $ taxa            <int> 0
## $ plot_type       <int> 0
```

Grouping by `species_id` and `sex`, find the `mean_weight` and `mean_hindfoot` when observations with missing values are dropped:

```
krats %>% drop_na() %>% group_by(species_id,sex) %>%
        summarise(mean_weight = mean(weight),
                  mean_hindfoot = mean(hindfoot_length))
```

```
## # A tibble: 6 x 4
## # Groups:   species_id [3]
##   species_id sex   mean_weight mean_hindfoot
##   <chr>      <chr>       <dbl>         <dbl>
## 1 DM         F            41.6          35.7
## 2 DM         M            44.3          36.2
## 3 DO         F            48.5          35.5
## 4 DO         M            49.1          35.7
## 5 DS         F           117.           49.6
## 6 DS         M           123.           50.4
```

Now we complete the table, first using `fill()` to impute the missing `sex` of each kangaroo rat based on the last non-missing sex observation. Then we group the kangaroo rats according to species and sex. Then we impute the missing `weight` and `hindfoot_length` values with `mutate()` and `coalesce()` from each groups median value. Finally we summarise again to find the `mean_weight` and `mean_hindfoot` for each group.

```
krats %>%
        fill(sex) %>%
        group_by(species_id,sex) %>%
        mutate(weight = coalesce(weight, median(weight, na.rm = TRUE)),
```

```
               hindfoot_length = coalesce(hindfoot_length,
                                          median(hindfoot_length, na.rm = TRUE))) %>%
        summarise(mean_weight = mean(weight),
                  mean_hindfoot = mean(hindfoot_length))
```

```
## # A tibble: 6 x 4
## # Groups:   species_id [3]
##   species_id sex   mean_weight mean_hindfoot
##   <chr>      <chr>       <dbl>         <dbl>
## 1 DM         F            41.6          35.7
## 2 DM         M            44.4          36.2
## 3 DO         F            48.5          35.5
## 4 DO         M            49.2          35.7
## 5 DS         F           118.           49.6
## 6 DS         M           123.           50.3
```

The results are very similar.

3.3 Joining tables

See also R4DS Relational data chapter[9]

Previously we've used data from the Portal project[10] where everything we needed was already contained within a single table, but often we have *related* information spread across multiples tables that we want to analyse. In these situations we need to join pairs of tables to explore relationships of interest.

We'll use a dummy dataset here that recreates part of the data the Portal project collects. As well as the animal census they separately record weather information.

Imagine we had two small tables, one called `airtemp` that contains the air temperature in Celsius for three plots for one timepoint on one day. And another called `krat` that records the observations for three plots on the same day for kangaroo rat speciecs `DM`, `DS` and `RM`.

```
airtemp <- tibble(date = date(c('1990-01-07','1990-01-07','1990-01-07')),
               plot = c(4,8,1),
               airtemp = c(7.3,9.5,11))

krat <- tibble(date = date(c('1990-01-07','1990-01-07','1990-01-07')),
               plot = c(4,4,6),
               species = c("DM","DS","RM"))
```

[9] https://r4ds.had.co.nz/relational-data.html
[10] portal-project

```
airtemp %>% glimpse()
```

```
## Observations: 3
## Variables: 3
## $ date    <date> 1990-01-07, 1990-01-07, 1990-01-07
## $ plot    <dbl> 4, 8, 1
## $ airtemp <dbl> 7.3, 9.5, 11.0
```

```
krat %>% glimpse()
```

```
## Observations: 3
## Variables: 3
## $ date    <date> 1990-01-07, 1990-01-07, 1990-01-07
## $ plot    <dbl> 4, 4, 6
## $ species <chr> "DM", "DS", "RM"
```

We can think of three types of join, two of which correspond with `dplyr` verbs:

1. Joins that **mutate** the table. These are joins that create a new variable in one table using observations from another.
2. Joins that **filter** the table. These are joins that keep only a subset of observations from one table based on whether they match the observations in another.
3. Joins that perform **set operations**. These are joins corresponding with the mathematical operation of intersection $\cap$, union $\cup$, and difference $-$.

   Note Joins always use two tables. To add a third relationship it would require joining the table created from the first join to another table. And so on.

3.3.1 Keys

A key is a variable or set of variables that uniquely identifies an observation in a table. In a simple case only one variable is sufficient, but often several variables are required.

- Primary key: a key that uniquely identifies the observation in its own table e.g. the combination of `date` and `plot` is unique to each set of observations in the `airtemp` table.
- Foreign key: a key that uniquely identifies the observation in another table. e.g. the combination of `date`, `plot` and `species` is unique to each set of observations in the `krat` table.

```
airtemp %>% count(date,plot)
```

```
## # A tibble: 3 x 3
##   date        plot     n
##   <date>     <dbl> <int>
## 1 1990-01-07     1     1
## 2 1990-01-07     4     1
## 3 1990-01-07     8     1
```

```
krat %>% count(date,plot,species)
```

```
## # A tibble: 3 x 4
##   date        plot species     n
##   <date>     <dbl> <chr>   <int>
## 1 1990-01-07     4 DM          1
## 2 1990-01-07     4 DS          1
## 3 1990-01-07     6 RM          1
```

So joining `airtemp` and `krat` uses the `airtemp` primary key of `date` and `plot`.

Keys can be both primary and foreign, as they might be primary for one table and foreign in another or vice versa.

It's a good idea to verify a key is primary i.e. uniquely identifies an observation. See the examples in R4DS keys[11]

If you discover your table lacks a primary key you can add one with `mutate()`.

Keys are called explicity using `by =` followed by the keys as a character vector. For example, here we call `by = c("plot","date)`.

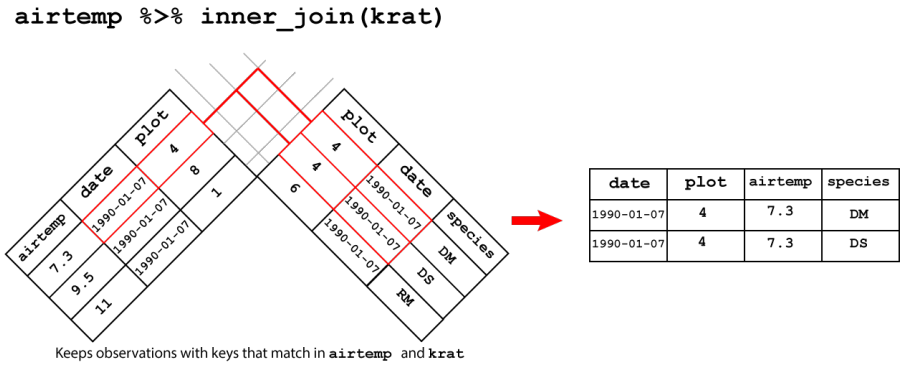
```
airtemp %>% inner_join(krat, by = c("plot","date"))
```

```
## # A tibble: 2 x 4
##   date        plot airtemp species
##   <date>     <dbl>   <dbl> <chr>
## 1 1990-01-07     4     7.3 DM
## 2 1990-01-07     4     7.3 DS
```

If we implicitly let the join function choose the keys, we get an output telling us what was used.

If we had a key that had a different name in both tables, we tell the join function that they are the same key by saying one variable is equal to another. For example if we had `plot` in one table and it was called `plot_id` in another, we could use `by = c("plot" = "plot_id")` to tell the join function that this is the key.

[11] https://r4ds.had.co.nz/relational-data.html#keys



date	plot	airtemp	species
1990-01-07	4	7.3	DM
1990-01-07	4	7.3	DS

Figure 3.3: Inner join

3.3.2 Mutating joins

These are joins that created new columns, hence they act like `dplyr::mutate`.

From `dplyr` we have:

- `inner_join()` : keeps only the observations with matching keys in both tables
- `left_join()` : keeps all the observations in the left hand table
- `right_join()` : keeps all the observations in the right hand tables
- `full_join()` : keeps all the observations in both tables

3.3.2.1 Inner join

> `inner_join()` : keeps only the observations with matching keys in both tables

An inner join between `airtemp` and `krat` keeps only the observations which have match in both tables using the primary key of `date` and `plot` as shown in Figure 3.3.

```
airtemp %>% inner_join(krat)
```

```
## Joining, by = c("date", "plot")
```

```
## # A tibble: 2 x 4
##   date        plot airtemp species
##   <date>     <dbl>   <dbl> <chr>
## 1 1990-01-07     4     7.3 DM
## 2 1990-01-07     4     7.3 DM
```

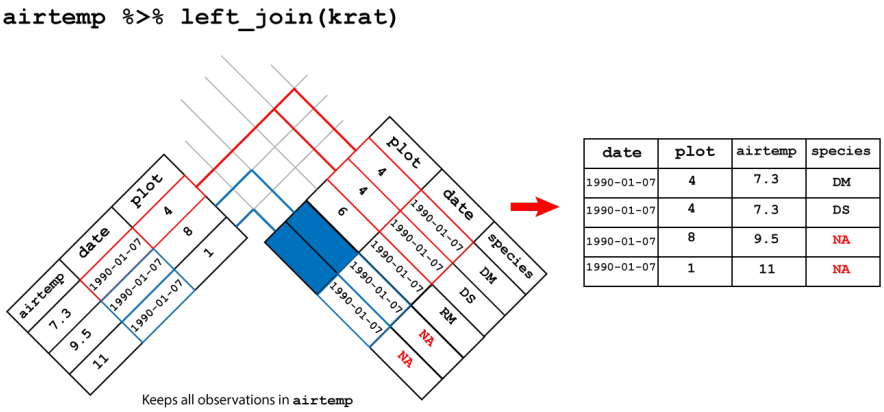


Figure 3.4: Left join

3.3.2.2 Left join

`left_join()` : keeps all the observations in the left hand table

An left join between `airtemp` and `krat` keeps all the observations in `airtemp` and fills the missing observations in `species` with `NA` as shown in Figure 3.4.

```
airtemp %>% left_join(krat)
```

```
## Joining, by = c("date", "plot")
```

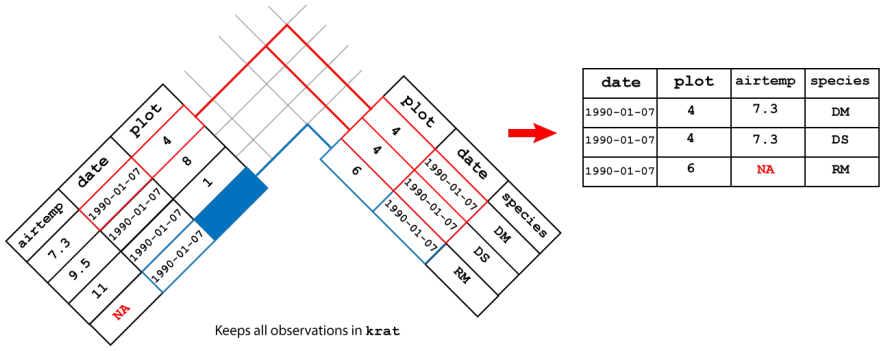
```
## # A tibble: 4 x 4
##   date        plot airtemp species
##   <date>     <dbl>   <dbl> <chr>
## 1 1990-01-07     4     7.3 DM
## 2 1990-01-07     4     7.3 DM
## 3 1990-01-07     8     9.5 <NA>
## 4 1990-01-07     1    11   <NA>
```

3.3.2.3 Right join

`right_join()` : keeps all the observations in the right hand tables

An right join between `airtemp` and `krat` keeps all the observations in `krat` and fills the missing observations in `airtemp` with `NA` as shown in Figure 3.5.

airtemp %>% right_join(krat)



date	plot	airtemp	species
1990-01-07	4	7.3	DM
1990-01-07	4	7.3	DS
1990-01-07	6	NA	RM

Figure 3.5: Right join

```
airtemp %>% right_join(krat)
```

```
## Joining, by = c("date", "plot")
```

```
## # A tibble: 3 x 4
##   date        plot airtemp species
##   <date>     <dbl>   <dbl> <chr>
## 1 1990-01-07     4     7.3 DM
## 2 1990-01-07     4     7.3 DM
## 3 1990-01-07     6    NA   RM
```

3.3.2.4 Full join

`full_join()` : keeps all the observations in both tables

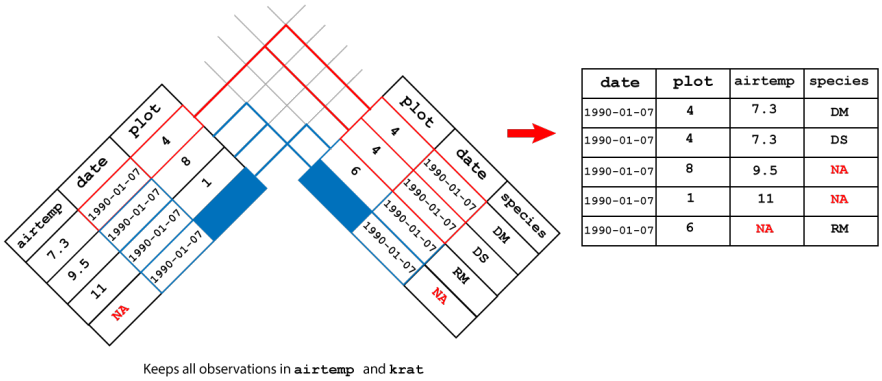
An full join between `airtemp` and `krat` keeps all the observations in both tables and fills the missing observations in `airtemp` and `species` with `NA` as shown in Figure 3.6.

```
airtemp %>% full_join(krat)
```

```
## Joining, by = c("date", "plot")
```

```
## # A tibble: 5 x 4
##   date        plot airtemp species
```

```
airtemp %>% full_join(krat)
```



date	plot	airtemp	species
1990-01-07	4	7.3	DM
1990-01-07	4	7.3	DS
1990-01-07	8	9.5	NA
1990-01-07	1	11	NA
1990-01-07	6	NA	RM

Figure 3.6: Full join

```
##   <date>     <dbl>   <dbl> <chr>
## 1 1990-01-07     4     7.3 DM
## 2 1990-01-07     4     7.3 DM
## 3 1990-01-07     8     9.5 <NA>
## 4 1990-01-07     1    11   <NA>
## 5 1990-01-07     6    NA   RM
```

3.3.3 Filtering joins

Filtering joins act like the `filter` function keeping only observations that match the join without adding or removing columns.

`semi_join()` and `anti_join()`

3.3.3.1 Semi join

> `semi_join()` keeps only the observations in the left hand table that have a match in the right hand table.

A semi-join between between `airtemp` and `krat` keeps only the observation in `airtemp` that matches in `krat` as showin in Figure 3.7.

```
airtemp %>% semi_join(krat)
```

```
## Joining, by = c("date", "plot")
```

```
airtemp %>% semi_join(krat)
```

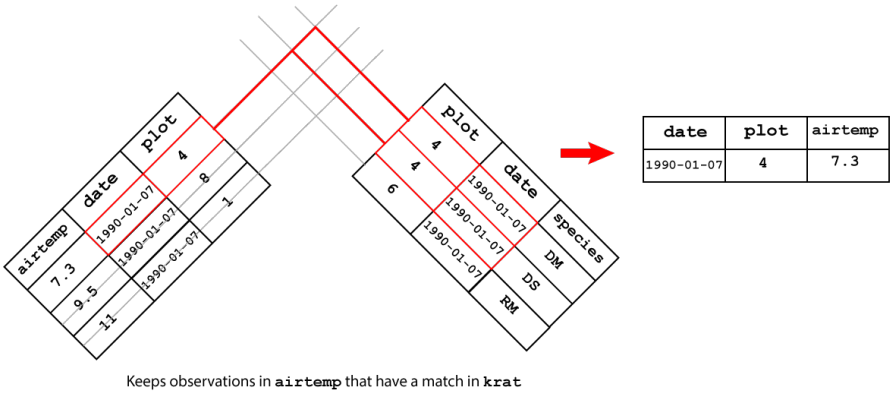


Figure 3.7: Semi join

```
## # A tibble: 1 x 3
##   date        plot airtemp
##   <date>     <dbl>   <dbl>
## 1 1990-01-07     4     7.3
```

3.3.3.2 Anti join

> `anti_join()` drops the observations in the left hand table that have a match in the right hand table.

A anti-join between between `airtemp` and `krat` drops the observations in `airtemp` that match in `krat` as shown in Figure 3.8.

```
airtemp %>% anti_join(krat)
```

```
## Joining, by = c("date", "plot")
```

```
## # A tibble: 2 x 3
##   date        plot airtemp
##   <date>     <dbl>   <dbl>
## 1 1990-01-07     8     9.5
## 2 1990-01-07     1    11
```

```
airtemp %>% anti_join(krat)
```

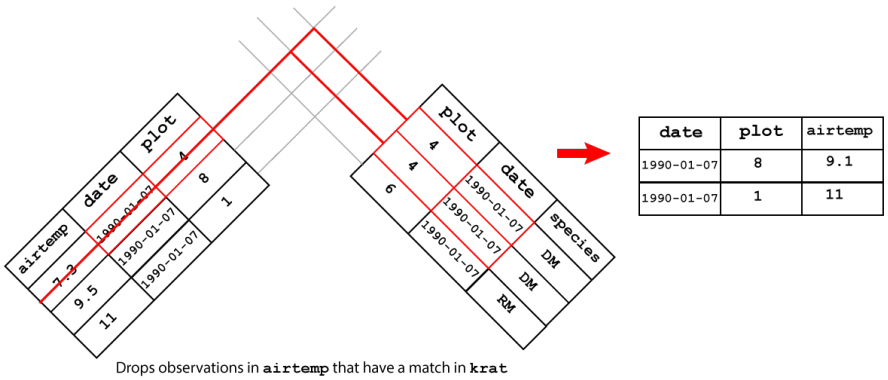


Figure 3.8: Anti join

3.3.4 Set operations

These are joins corresponding with the mathematical operation of intersection $\cap$, union $\cup$, and difference $-$ as shown in Figure 3.9. These are `intersect()`, `union()` and `setdiff()`.

See R4DS set operations[12] for examples of usage.

3.3.5 Duplicate keys

If we have duplicate keys in each tables which aren't used as part of the join, then the join will create columns for both varaibles. For example, if we join `airtemp` with `krat` only using `plot` as a key, but `date` is in both, we reuturn `date.x` and `date.y` for `airtemp` and `krat` respecitively:

```
airtemp %>% full_join(krat, by = "plot")
```

```
## # A tibble: 5 x 5
##   date.x      plot airtemp date.y     species
##   <date>     <dbl>   <dbl> <date>     <chr>
## 1 1990-01-07     4     7.3 1990-01-07 DM
## 2 1990-01-07     4     7.3 1990-01-07 DS
## 3 1990-01-07     8     9.5 NA         <NA>
## 4 1990-01-07     1    11   NA         <NA>
## 5 NA             6    NA   1990-01-07 RM
```

[12] https://r4ds.had.co.nz/relational-data.html#set-operations

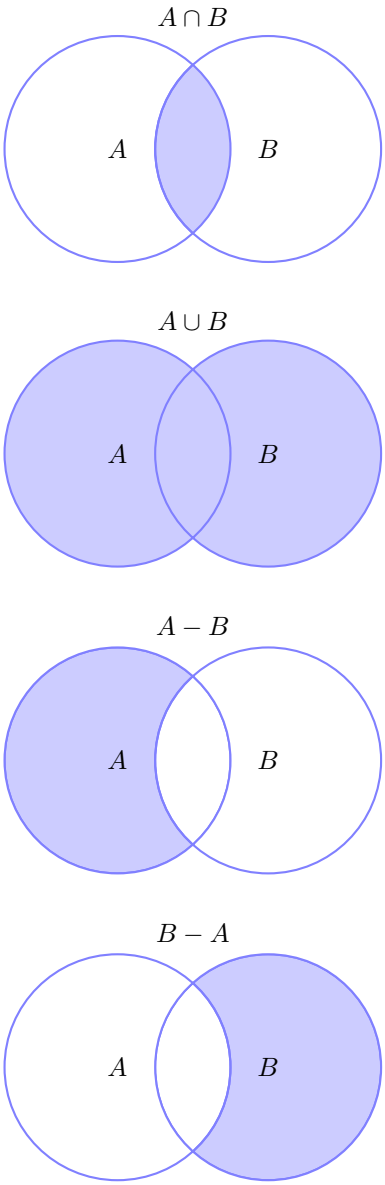


Figure 3.9: Set operations `intersect(A,B)`, `union(A,B)` and `setdiff(A,B)`.

3.3.6 Binding rows and columns

We can also join tables by binding rows or columns. Binding rows is useful when we have tables with different sets of observations for the same variables, such as when we have multiple replicates of an experiment. Binding columns can be done when we want to add a column from one table that corresponds with the observations in another.

See `dplyr bind`[13] :

"When row-binding, columns are matched by name, and any missing columns will be filled with NA."

When column-binding, rows are matched by position, so all data frames must have the same number of rows."

These functions are `bind_rows` and `bind_cols`.

[13] https://dplyr.tidyverse.org/reference/bind.html

Chapter 4

Visualisation and communication

Just as there are many types of data e.g. tables and audio files, correspondingly there are many types of data visualisation e.g. statistical plots and maps. Likewise visualisations we do for an exploration may be different to those we do for communication. Therefore we narrow the scope here to introduce some fundamentals of visualisation, presentations and reports.

By the end of this chapter you will be able to:

- Use the `ggplot2` package to create exploratory plots and customize their plots
- Use geometrical objects and aesthetic mappings to create box plots, bar plots and time series plots
- Use facets to split plots into sub-plots according to variables
- Transform plots using statistical mappings and categorical variables (factors).
- Transform data positions and coordinates
- Use `ggplot2` themes and the `scales` package to customize plots
- Export figures for use in documents and presentations
- Create reports and presentations directly from RStudio

These skills will enable to start creating visualisations as well as reports and presentations using R, but importantly it is up to you to determine what sort of visualisation is appropriate for the task at hand and for you to use your critical judgement in assessing a visualisation, presentation or report.

4.1 Visualisation overview

For a deeper understanding of the art of data visualisation, a good place to start is with the work of Albert Cairo[1], a journalist and academic specialising in data visualisation who has written and continues to write extensively on the subject.

Betsy Mason has written an article that summarises many of the key ideas: Why scientists need to be better at data visualization[2] including the issues around how we perceive shade and hue, that can make plots such as heatmaps problematic.

Rafeal Irizarry has also written about Why dynamite plots must die[3]. These are very common plots in biomedical science, but are far from optimal.

The point here is not to tell you explicitly what plots to make, or that heatmaps are bad etc., but to encourage you to think about what type of plot is best for the task in hand. Choosing to plot your data in a certain way because, *"that's what everyone else does"* is unlikely to be the best reason.

To begin with two questions one might immediately have are:

1. Why visualise our data in the first place?
2. What do we know about what makes an effective visualisation?

4.1.1 Making comparisons

Starting with the second question, a helpful set of principles for statistical visualisations is the work of Cleveland and McGill, namely: Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods by Cleveland and McGill[4].

What they established is a hierarchy of perception, that is to say we there is an order to which our brains find it easier or harder to make comparisons with visual information.

The order of comparisons from easiest to hardest is as follows, but this is best illustrated by some plotting some example data.

1. Positions on a common scale
2. Positions on the same but non-aligned scales
3. Lengths

[1] http://www.thefunctionalart.com/
[2] https://www.knowablemagazine.org/article/mind/2019/science-data-visualization
[3] https://simplystatistics.org/2019/02/21/dynamite-plots-must-die/
[4] https://www.jstor.org/stable/pdf/2288400.pdf?refreqid=excelsior%3Ab321aba183cce22a7f93335ca107eec6

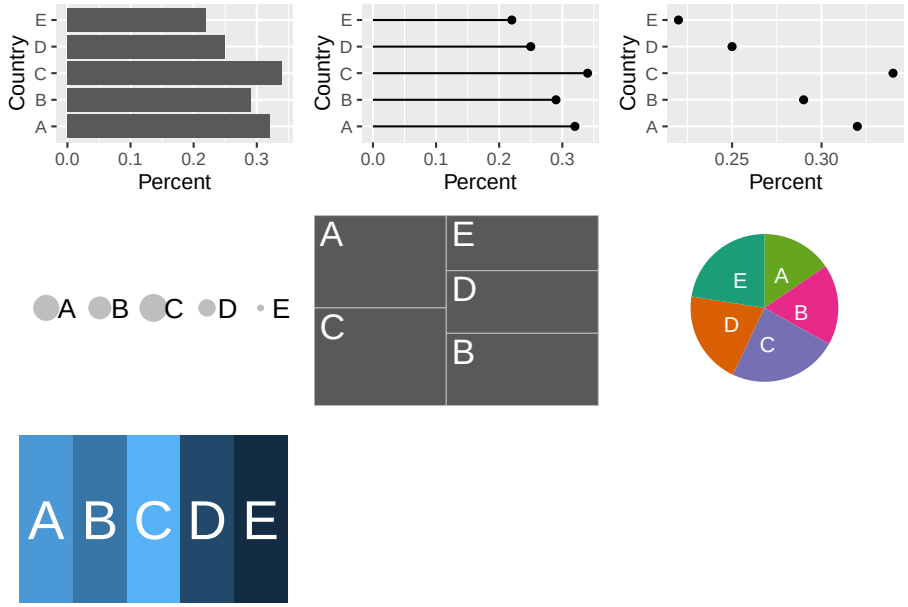


Figure 4.1: The same percentages for five Countries A-E is plotted seven ways to illustrate the differences in ease of making comparisons depending the plot type. The percentages are A:32%, B:29%, C:34%, D:25% and E:22% in all plots.

4. Angles, slopes
5. Area
6. Volume, colour saturation
7. Colour hue

Figure 4.1 illustrates plotting the same percentages for five Countries seven ways corresponding with Cleveland and McGill's hierarchy.

```
viz_dat <- tibble(Country =LETTERS[seq( from = 5, to = 1 )],
                  Percent = c(0.22,0.25,0.34,0.29,0.32))
```

The message here is that when choosing a visualisation for comparisons between sets of observations the best place to start is with plotting the observations on a common scale (but not necessarily finish).

4.2 ggplot2()

In his paper A layered grammar of graphics[5] Hadley Wickham discusses the theory behind the `ggplot2()` package. As with tidyverse the aim was to create a consistent system for building plots.

The key components of a plot are:

- data with aesthetic mappings e.g. position on an axis, colour or size.
- geometric objects e.g. points, lines.

To this other components or layers can be added, such as facets or statistical transformations to make increasingly complex plots.

As `ggplot2()` was first released in 2007 it preceded the `magrittr` package in 2014 from which we get the pipe %>% it has one syntax difference that is sure to catch you out. Whilst we can pipe into a `ggplot()` function, we build the plots using the plus sign + to add our next layer.

Likewise, forgetting to use the aesthetic function `aes()` within a layer is a common mistake I make. Aesthetics and geometries are related, so for example you can't set the shape aesthetic of a line. I usually make this mistake with colour and fill aesthetics, where I try to use one when I need the other.

The basic form for creating plots with the `ggplot2` package is as follows:

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

To recap:

- We provide a data frame as an argument to the `ggplot()` function
- Variables in the data frame are mapped to visual properties using the aesthetics function `aes()`
- The aesthetics are mapped to a geometric object such as points using a layer with a `geom()` function.
- If we were to plot several `geoms()` on the same plot wanted to map the same aesthetics to each `geom()` we can pass the aesthetics argument to `ggplot()` directly: `ggplot(data = <DATA>,mapping = aes(<MAPPINGS>))`.
- We can also pipe data into `ggplot()` as we would for other functions.

[5] http://vita.had.co.nz/papers/layered-grammar.pdf

4.2.1 Datasaurus dozen

Returning to the first question: Why visualise data in the first place?

Inspired by the Anscombe's Quartet[6] and Albert Cairo's Datasaurus[7], Justin Matejka and George Fitzmaurice created the Datasaurus dozen[8] to illustrate a problem with relying only on summary statistics to understand data.

```
# Download the Datasaurus Dozen
datasaurus <- read_tsv("https://raw.githubusercontent.com/ab604/coding-together/master/exercises/
```

```
## Parsed with column specification:
## cols(
##   dataset = col_character(),
##   x = col_double(),
##   y = col_double()
## )
```

```
# Quick look
glimpse(datasaurus)
```

```
## Observations: 1,846
## Variables: 3
## $ dataset <chr> "dino", "dino", "dino", "dino", "dino", "dino", "dino"...
## $ x       <dbl> 55.3846, 51.5385, 46.1538, 42.8205, 40.7692, 38.7179, ...
## $ y       <dbl> 97.1795, 96.0256, 94.4872, 91.4103, 88.3333, 84.8718, ...
```

```
# How many datasets
datasaurus %>% distinct(dataset)
```

```
## # A tibble: 13 x 1
##    dataset
##    <chr>
##  1 dino
##  2 away
##  3 h_lines
##  4 v_lines
##  5 x_shape
##  6 star
##  7 high_lines
##  8 dots
```

[6] https://en.wikipedia.org/wiki/Anscombe%27s_quartet
[7] http://www.thefunctionalart.com/2016/08/download-datasaurus-never-trust-summary.html
[8] https://www.autodeskresearch.com/publications/samestats

```
##  9 circle
## 10 bullseye
## 11 slant_up
## 12 slant_down
## 13 wide_lines
```

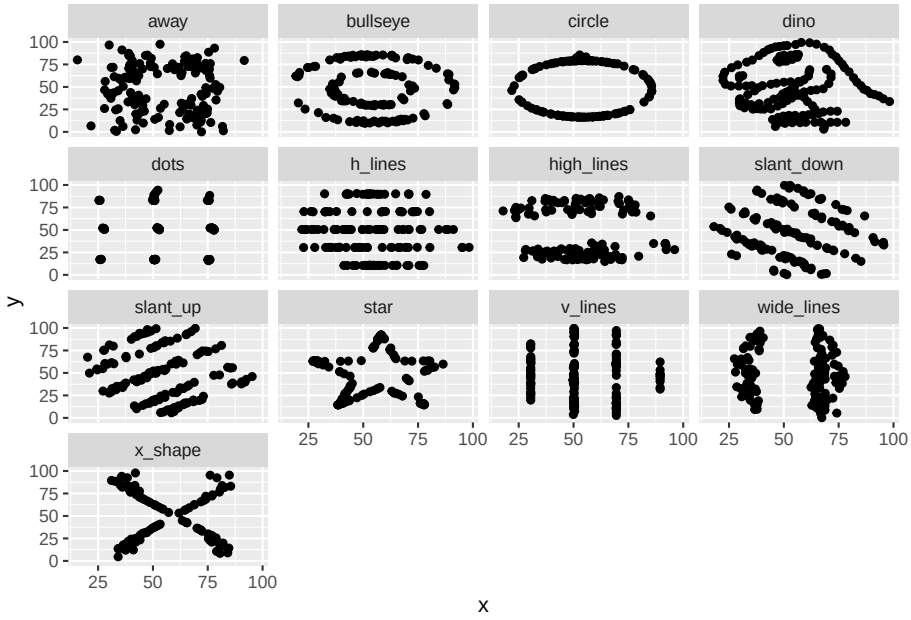
We have a bakers' dozen of datasets, each with 1,846 observations for variables `x` and `y`. Let's look at the mean and standard deviation for each variable using a grouped summary and the `mean()` and standard deviation `sd()` functions.

```
datasaurus %>% group_by(dataset) %>% summarise(mean_x = mean(x),
                                               mean_y = mean(y),
                                               sd_x = sd(x),
                                               sd_y = sd(y))
```

```
## # A tibble: 13 x 5
##    dataset    mean_x mean_y  sd_x  sd_y
##    <chr>       <dbl>  <dbl> <dbl> <dbl>
##  1 away         54.3   47.8  16.8  26.9
##  2 bullseye     54.3   47.8  16.8  26.9
##  3 circle       54.3   47.8  16.8  26.9
##  4 dino         54.3   47.8  16.8  26.9
##  5 dots         54.3   47.8  16.8  26.9
##  6 h_lines      54.3   47.8  16.8  26.9
##  7 high_lines   54.3   47.8  16.8  26.9
##  8 slant_down   54.3   47.8  16.8  26.9
##  9 slant_up     54.3   47.8  16.8  26.9
## 10 star         54.3   47.8  16.8  26.9
## 11 v_lines      54.3   47.8  16.8  26.9
## 12 wide_lines   54.3   47.8  16.8  26.9
## 13 x_shape      54.3   47.8  16.8  26.9
```

Here we use the basic form to plot the `datasaurus` but with the addition of `facet_wrap()` in which the first argument is `~` representing "formula" (a type of data structure) and the second argument is the variable we wish to create individual plots for. In words read this as *"facets depend upon the dataset"*:

```
ggplot(data = datasaurus) +
       geom_point(mapping = aes(x,y)) +
       facet_wrap(~ dataset)
```



The message here is that plotting the data reveals structure that is not apparent from summary statistics. Therefore if possible: **plot early, and plot often.**

4.2.2 Exploratory plots

As the name implies, the point of an exploratory plot is to explore our data. This can happen before, during or after tidying/transforming data, and as discussed in The Art of Data Science[9] serves two key purposes that we saw with the Datasaurus:

1. Creating expectations
2. Checking for deviations from our expectations.

In Chapter 2 having transformed the Portal rodent surveys data into a table summarising the rodent observations per three month period per plot type (control or exclosure). The original paper was exploring the hypothesis that kangaroo rat and granivore populations were in competition for resources in their habitat.

We plotted the average number of captures over time as a line and point plot, coloured according to plot type, and facetting the plot according to animal type.

[9] https://leanpub.com/artofdatascience

Here we are exploring the effect of the exlosure on the animal populations over time, and the variation in populations over time in both plot types.

```
ggplot(data = by_quarter,
       mapping = aes(x=quarter,y=mean_captures,colour=plot_type)) +
  geom_line() +
  geom_point() +
  facet_wrap(~ rodent_type)
```

Expectations created by this plot for example are that the populations fluctuate quite a lot over time, and the exclosure appears to work well with few kangaroo rats observed in those plots.

What is less clear is what the effect of the exclosure is having, if any, on the granivore population, suggesting we might wish to try a different type of plot.

4.2.3 Statistical transformations

Often we want to transform the data as part of the plotting process, for example to create plots that reveal statistical information such as distributions or averages. For these we can use geoms which are statistical in nature or to which we provide statistical arguments.

4.2.3.1 Barplots

These simple plots actually reveal subtleties in plots. In the following code we take the `by_month_species` subset and create a bar chart of `species_id`, filling the bars with colour according to `rodent_type`.

```
# Create a bar plot of genus from by_month_species
by_month_species <- surveys_subset %>%
        filter(rodent_type != "Other") %>%
        group_by(year,month,species_id,rodent_type,plot_type) %>%
        summarise(captures = n()/31)

by_month_species %>%
  ggplot() +
  geom_bar(mapping = aes(x = species_id, fill=rodent_type)) +
  facet_wrap(~ plot_type)
```

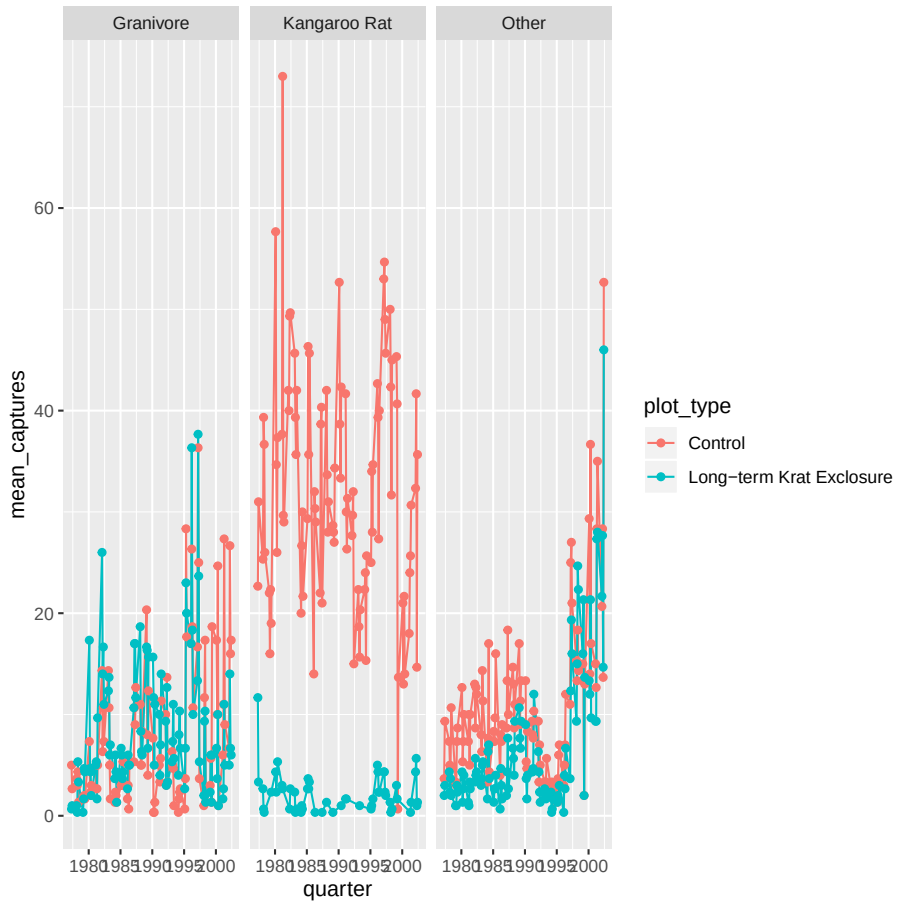
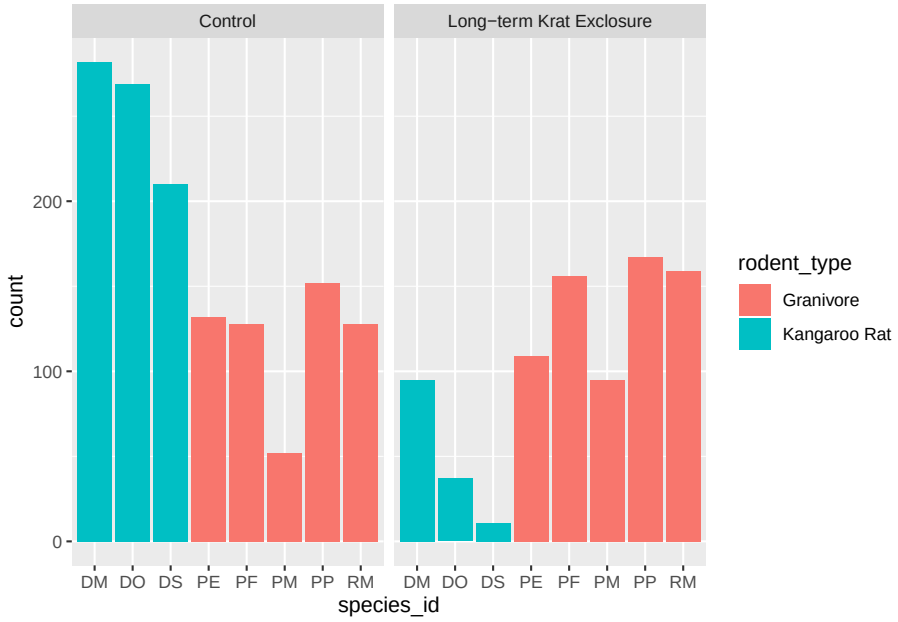


Figure 4.2: The average number of captures over time as a line and point plot, coloured according to plot type, and facetting the plot according to animal type.



We see that the code has automatically created a count variable on the y-axis for each species_id plotted as a bar on the x-axis.

This is because the `geom_bar()` algorithm automatically performs a statistical transformation of the mapped variable. That is to say it performs a count and bins the data according to the genus. This means we have a chart where the bar height corresponds to the number of rows for each species_id.

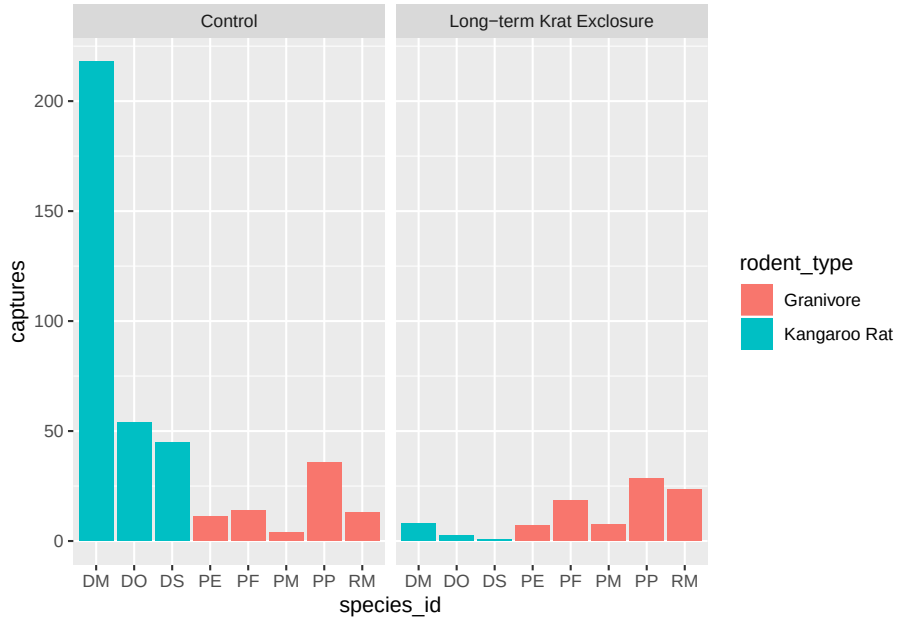
But this isn't a count of the captures, it's how many rows each species_id appears in of `by_month_species`. Remember how we summarised this data to calculate captures per month.

Often when people talk about bar charts they are referring to when the height of the bar is a variable present in the data, here that would be the `captures` column. In this kind of situation we need to map the y-axis too, and change the default `geom_bar()` statistical transformation to `stat = "identity"` so that it takes the values provided in the `captures` column to determine the height of the bar.

To reiterate: **if you are trying to create a bar chart using x and y variables contained in your data set, you need to set `geom_bar(stat = "identity")`.**

```
# Create a bar plot of genus from by_month_species using y = captures and
# stat = "identity"
by_month_species %>%
  ggplot() +
```

```
  geom_bar(mapping = aes(x = species_id, y = captures, fill=rodent_type), stat = "identity") +
  facet_wrap(~ plot_type)
```



Let's check this by looking at PE in the control plots and counting the rows and comparing that to summing the number of captures. Note we need to ungroup() before doing another summary. There are 132 rows, but only 11 captures.

```
# How many rows for PE
by_month_species %>%
  filter(species_id == "PE", plot_type == "Control") %>% ungroup() %>% tally()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1   132
```

```
# How many captures for PE
by_month_species %>%
  filter(species_id == "PE", plot_type == "Control") %>% ungroup() %>%
  summarise(total = sum(captures))
```

```
## # A tibble: 1 x 1
```

```
##   total
##   <dbl>
## 1  11.1
```

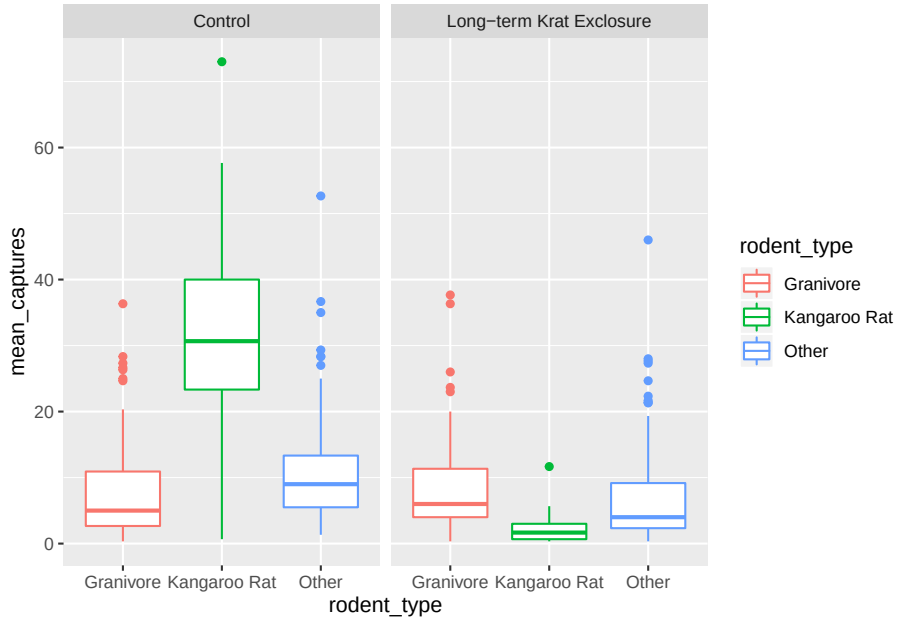
4.2.3.2 Boxplots

The previous exploratory plot showed us changes with time, but averages and spread or other statistical summaries of the surveys data might be more informative for understanding the effect of exlosure on granivore populations.

Box-plots are a standard way to plot the distribution of a data set. Plotting the data this way means that we drop the dynamic information, the time dimension, but gain a compact summarised view of the variability of the number of captures, and hopefully evidence of any changes in granivore population following exclosure.

In the following code we use the `boxplot()` geom, and this time swap the mapping such that colour maps to the rodent type, and plots are faceted according to the plot type.

```
ggplot(data= by_quarter,
       mapping = aes(x = rodent_type,
                     y = mean_captures,
                     colour = rodent_type)) +
  geom_boxplot() +
  facet_wrap(~ plot_type)
```



Comparing the median bar and the interquartile range, we can see evidence suggesting a modest increase in the granivore population when kangaroo rats are excluded.

4.2.4 Position adjustments

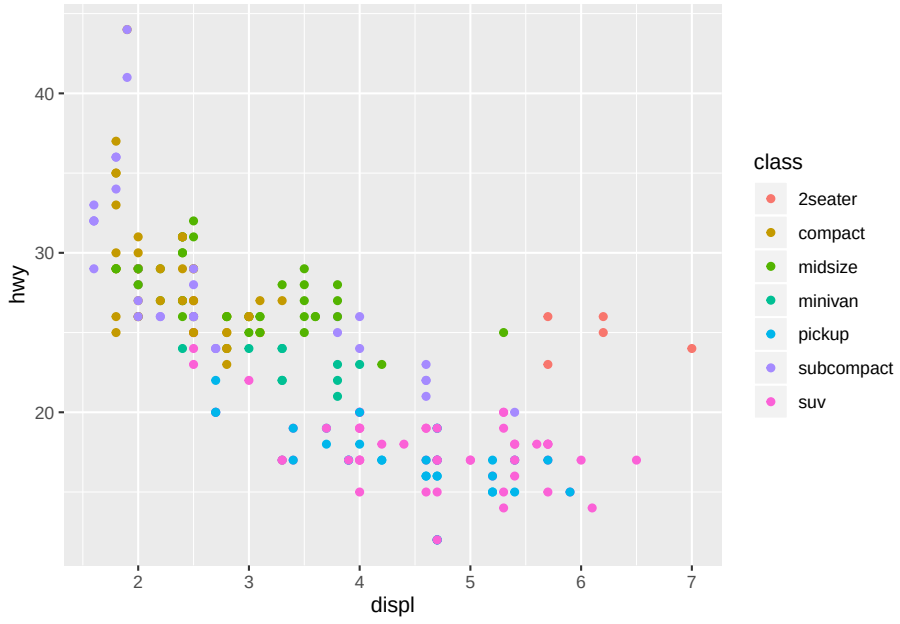
Another common adjustment we might make to our plots is to adjust the position of the data.

Recall the `mpg` data set for cars and our plot showing the fuel efficiency versus the size of the engine. Many of the points are actually over plotted such that we can't see them.

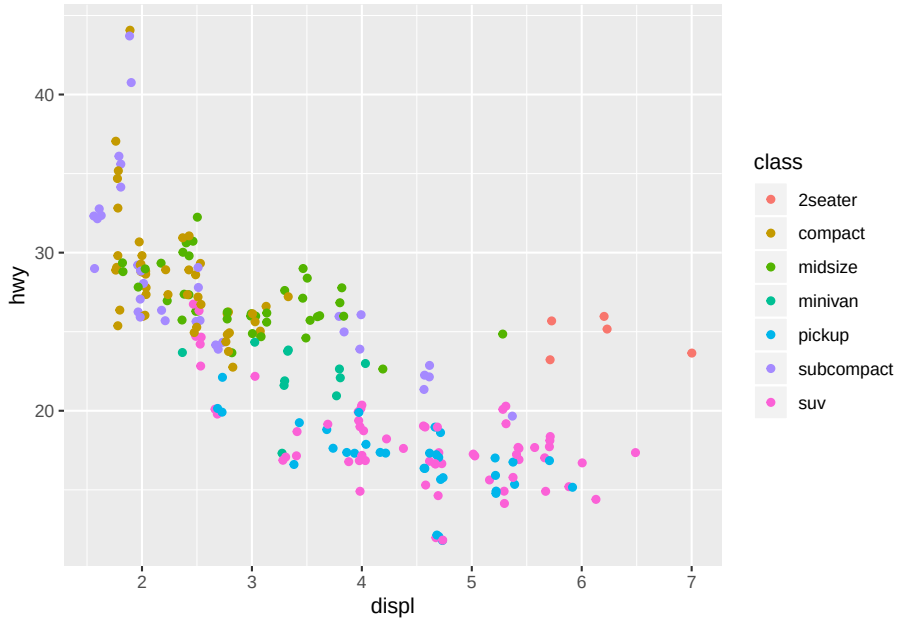
By adding a bit of random noise to the position each data point, called `jitter` we can spread the points out and see them more clearly. This is seen in the second plot below, where the `position = "jitter"` argument has been provided to the `geom_point()` function.

It's a little counter intuitive to add noise to a plot, but when we're trying to explore data for patterns it can be very useful.

```
# Without jitter
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, colour = class))
```

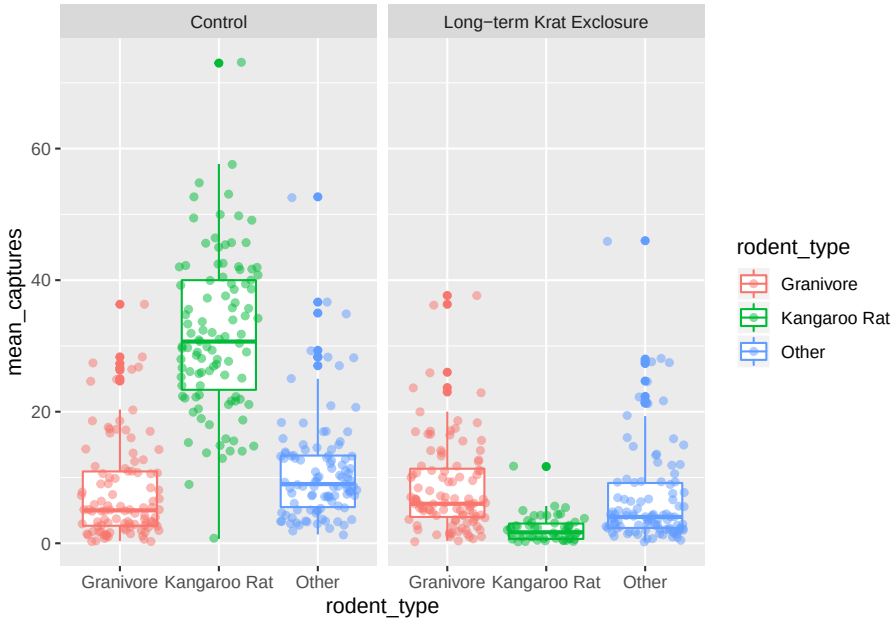



```
# With jitter
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, colour = class),
             position = "jitter")
```



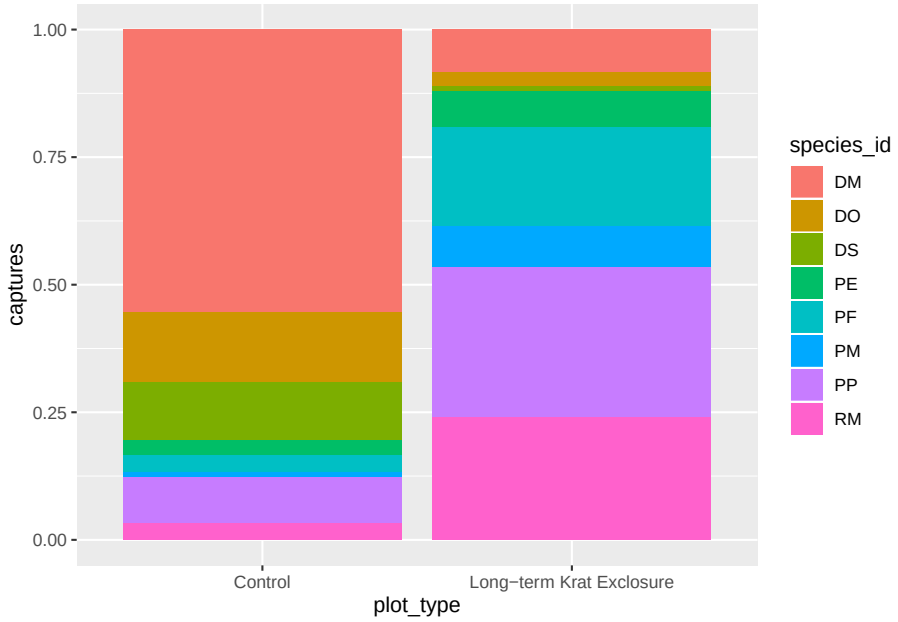
Let's add points to our box-plot and try comparing the plot to one where the points are jittered.

```
# Boxplot with points and jitter
ggplot(data= by_quarter,
       mapping = aes(x = rodent_type, y = mean_captures,
                     colour = rodent_type)) +
  geom_boxplot() +
  geom_point(position = "jitter", alpha = 0.5) +
  facet_wrap(~ plot_type)
```



We can also apply positional adjustments to other plots such as barplots. We can combine aesthetic `fill = species_id` with `position = "fill"` in `geom_bar()` to create a separate bar for each species and put plot_type on the x axis for `by_month_species` plots bars of length 1, but filled with the proportion of captures for that species.

```
# Create a bar plot of genus from by_month_species using x = plot_id and
# stat = "identity"
by_month_species %>%
  ggplot() +
  geom_bar(mapping = aes(x = plot_type, y = captures,  fill=species_id),
           position = "fill" , stat = "identity")
```



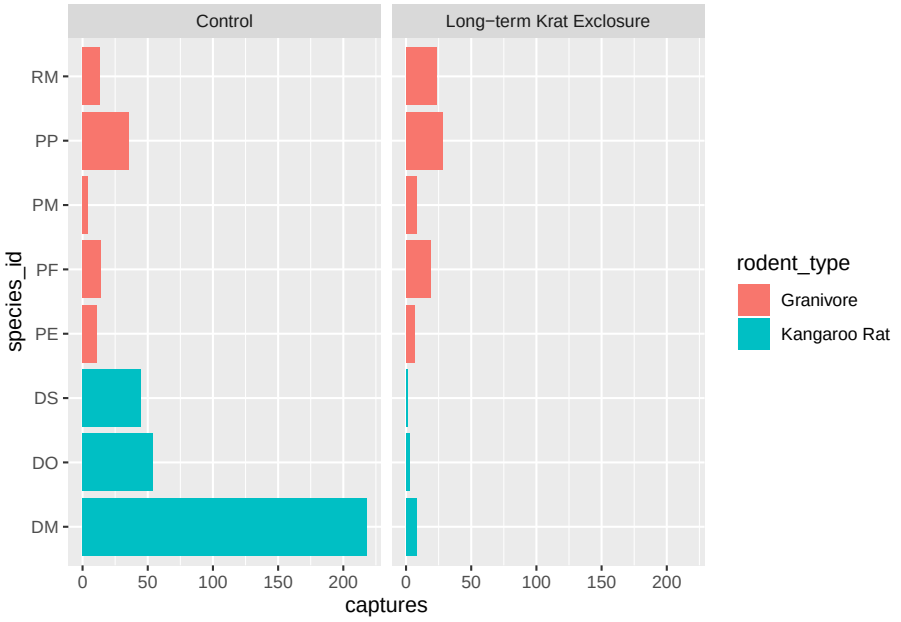
Perhaps not that useful given what we know about making comparisons.

For more positional adjustments, check out the help files `?postion_jitter`, `?position_dodge`, `?position_fill`, `?position_identity`, and `?position_stack`.

4.2.5 Coordinate adjustments

Previously, we plotted `by_month_species` and sometimes it's tricky when there are lots of labels on the x-axis, so let's flip the plot round using `coord_flip()` to improve things:

```
# Flip the coordinates to make the labels more readable
ggplot(by_month_species) +
  geom_bar(mapping = aes(x = species_id, y = captures, fill = rodent_type),
           stat = "identity") +
  facet_grid(~ plot_type) +
  coord_flip()
```



4.2.5.1 Factors

Finally for this plot, it would be nice to put the bars in order of size. We can do this by converting the genus to a factor.

Recall that factors are how R represents categorical variables, variables with a limited number of values, such as genus.

Also recall **factors look like strings, but behave like integers**. That is to say they are strings of characters with associated values called levels that can be used to order catergories.

This is why factors are useful, especially when we want to place things in non-alphabetical order. We can take advantage of factor levels to order things as we wish.

Check out the forcats[10] tidyverse package to explore the power of factors, but here to give you a taste of what is possible we're going to use the `forcats` package `fct_reorder()` function to convert `species_id` into a factor and in doing so set the level according to to the number of captures. In this way we can order the bars in our plot according to the number of captures and see the pattern more clearly.

In `fct_reorder()`, the first argument is the variable we wish to make into a factor `species_id`, and the second argument is variable we wish to use to

[10] http://forcats.tidyverse.org/

create the order. Here we want to order according to the number of captures, so `captures` is the second argument.

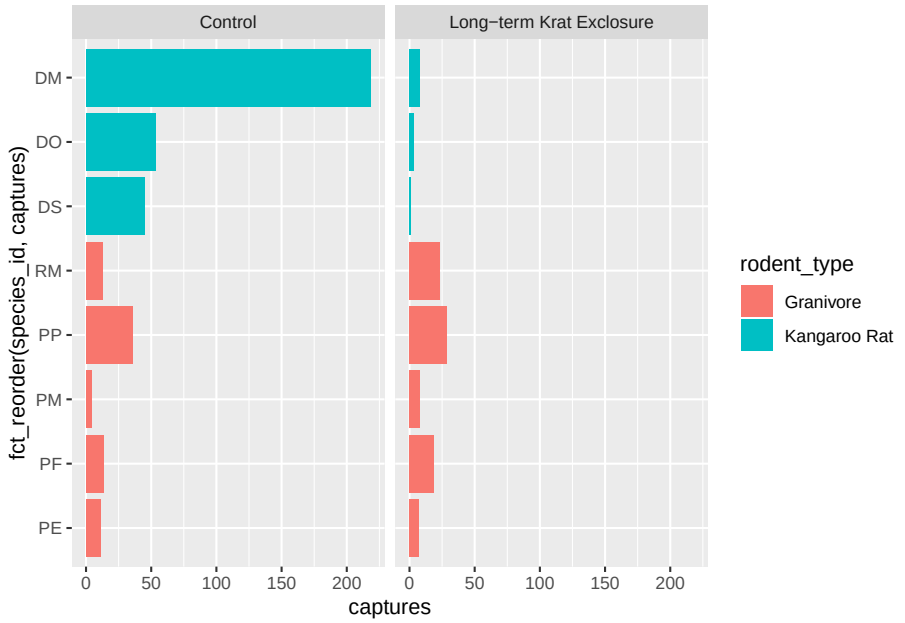
Note we still keep x and y mappings, but now x is now a function of both the genus and the capture columns.

This gives us a nice ordered bar chart by genus and number of captures during the experimental period.

We can clearly see that kangaroo rat exclosure works, and that this corresponds with an increase in granivore captures these plots.

```
# Load the forcats tidyverse package
library(forcats)

# Order the rodents according to number of captures
ggplot(by_month_species) +
  geom_bar(mapping = aes(x = fct_reorder(species_id, captures), y = captures,
                         fill = rodent_type), stat = "identity") +
  facet_grid(~ plot_type) +
  coord_flip()
```



4.3 Themes and customisations

When we're exploring our data we generally don't mind if the labels and colours etc... are a bit messy. But when we want to publish or communicate our findings to others we want everything to be just so.

This is where the code can become quite complicated and it's hard to provide a general case. However, the underlying template remains the same:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>),
     stat = <STAT>,
     position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

It's unlikely that you'll need to statistical transformations, positional adjustments, coordinate adjustments, and facets all in the same plot, but this provides an idea of what is possible. Remember the aim is clarity, not to make something complicated just because you can.

This template we can use to add all the additional functions and arguments that will change the plot to make it exactly how we want. It takes time, but remember code is reusable. Solve it once and you've solved it forever.

For all the possible customisations check out the ggplot2 documentation[11], and this chapter of R for data science: Graphics for communication[12]

But here's a few common changes.

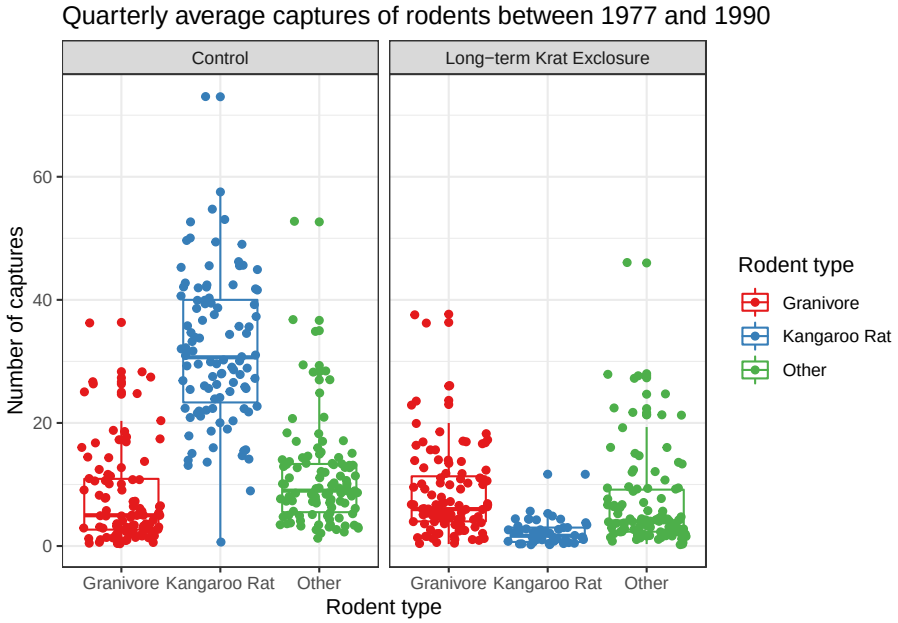
4.3.1 Changing colours, labels, themes and adding titles

Recall our box-plot with points overlaid. Following the facet function, we're going add a succession of functions to change the colours `scale_colour_brewer()`, change the axis labels, add a title using `labs()` and change the theme.

Themes deal with non-data elements e.g. the grid, the axes etc. There are a range of built-in themes and packages of themes such as `ggthemes`. Here we'll use the built-in theme, `theme_bw()` that gives a simple layout.

[11] http://ggplot2.tidyverse.org/reference/index.html
[12] http://r4ds.had.co.nz/graphics-for-communication.html#themes

```
ggplot(data= by_quarter,
       mapping = aes(x = rodent_type, y = mean_captures,
                     colour = rodent_type)) +
  geom_boxplot() +
  geom_point(position = "jitter") +
  facet_wrap(~ plot_type) +
  scale_colour_brewer(palette = "Set1") +
  labs(title = "Quarterly average captures of rodents between 1977 and 1990",
       x = "Rodent type",
       y = "Number of captures",
       colour = "Rodent type") +
  theme_bw()
```



Check out `?scale_colour_brewer` and `?scale_fill_brewer` for more information on colour schemes for discrete data. And `?scale_color_continuous` for creating colour gradients for continuous data.

4.4 Reports

Using R for report writing and presentations.

4.5 Presentations

My personal go-to for slide design guidance is Melinda Seckington[13]. She has written a fantastic step-by-step guide to creating effective slides.

Condensing her advice down she suggests:

- Focus on one point per slide
- Don't distract
- Make important information stand out using what we know about graphical perception
- Be consistent

I created a short example presentation about cleaning your teeth[14] following these ideas.

[13] https://missgeeky.com/2017/08/04/the-art-of-slide-design/

[14] https://docs.google.com/presentation/d/e/2PACX-1vTa8VfB_jpB6fG4i157eNJb8sAyyKjJW06iZ8NNL1LPcB40w4NLLpK3TetbCuWSckX3wc8Xu
pub?start=false&loop=false&delayms=30000

Chapter 5

Exploratory data analysis

Putting everything together to perform exploratory data analysis.

Art of data science checklist

Ihaka, Ross, and Robert Gentleman. 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*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

RStudio Team. 2018. *RStudio: Integrated Development Environment for R*. Boston, MA: RStudio, Inc. http://www.rstudio.com/.

Wickham, Hadley. 2017. *Tidyverse: Easily Install and Load the ’Tidyverse’*. https://CRAN.R-project.org/package=tidyverse.

Wilson, Greg, ed. 2018. *Teaching Tech Together. 2018„ Http://Teachtogether.tech/.* Lulu.com. http://teachtogether.tech/.

Xie, Yihui. 2019. *Bookdown: Authoring Books and Technical Documents with R Markdown*. https://CRAN.R-project.org/package=bookdown.