Intro R Workshop: Data Manipulation, Analysis, and Visualisation

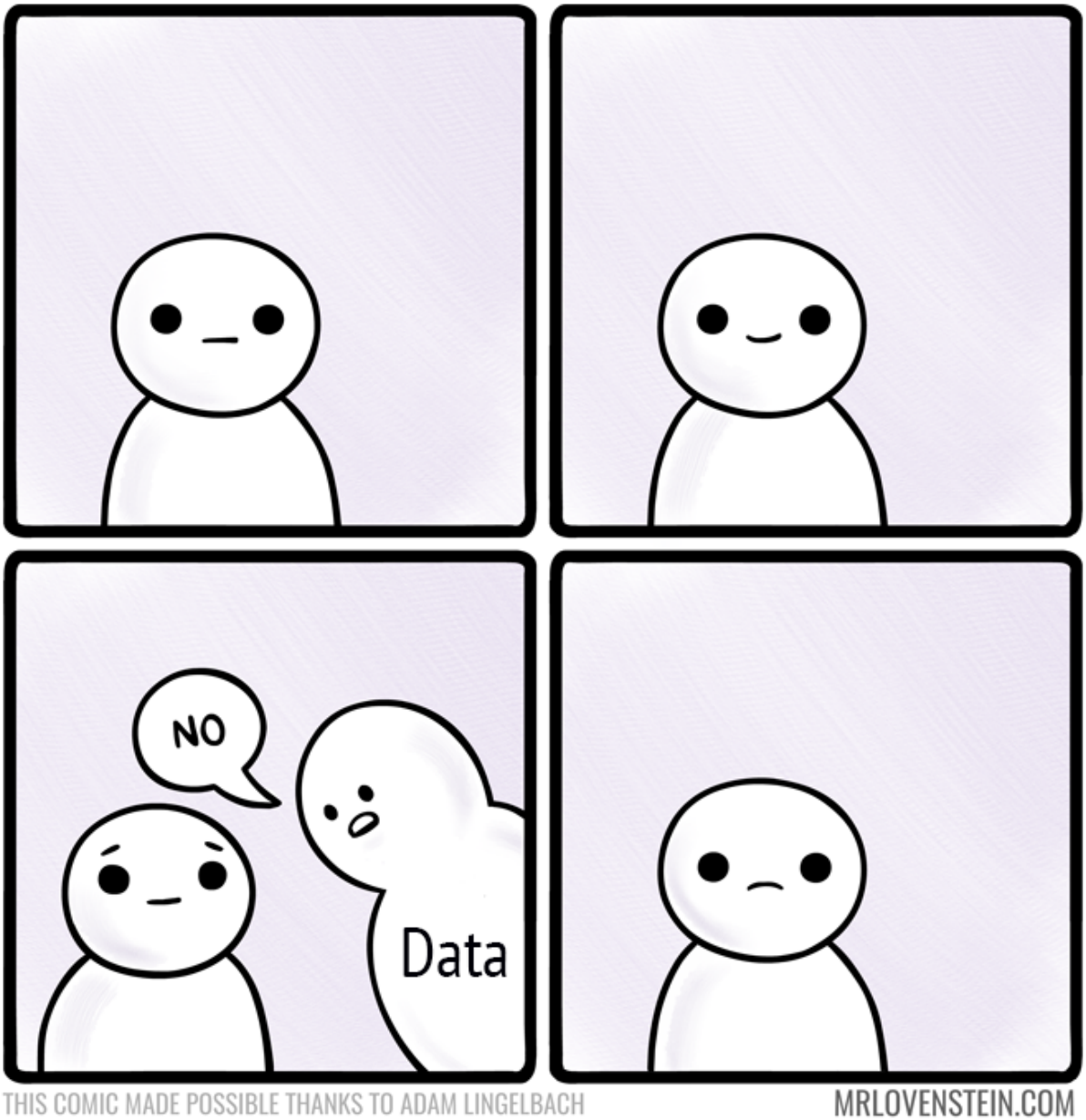
UWC BCB Honours 2020

AJ Smit, Amieroh Abrahams & Robert W Schlegel

2020-01-31

Contents

Preface



This online book contains all of the content to be covered in this Intro R Workshop.
The associated files may be downloaded at https://github.com/ajsmit/Intro_R_Official
Please use the table of contents (TOC) on the left to navigate the course content as desired.

Chapter 1

Preliminaries

> "In the beginning, the universe was created. This has made a lot of people very angry and been widely regarded as a bad move."
>
> — Douglas Adams
>
> "The history of life thus consists of long periods of boredom interrupted occasionally by panic."
>
> — Elizabeth Kolbert, The Sixth Extinction

1.1 Venue, date and time

This workshop will take place in the week of **28 January – 31 January 2020**, from **9:00–16:00** each day. We will meet in the BCB computer lab on the 5th floor.

1.2 Course outline

Day 1 – In the Beginning

- Presentation: Preliminaries
- Exercise: It which shall not be named
- – break –
- Demonstration: The New Age
- Interactive Session: Introduction to R and RStudio
- – lunch –
- Interactive Session: An R workflow
- – break –
- Interactive Session: An R workflow
- – end –

Day 2 – Show and tell

- Interactive Session: The basics of `ggplot2`
- – break –
- Interactive Session: Faceting figures in `ggplot2`
- – lunch –
- Interactive Session: Brewing colours in `ggplot2`
- – break –
- Assignment: DIY figures
- – end –

Day 3 – Going deeper

- Interactive Session: Mapping with `ggplot2`
- – break –
- Interactive Session: Mapping with style
- – lunch –
- Interactive Session: Mapping with Google
- – break –
- Assignment: DIY maps
- – end –

Day 4 – The Enlightened Researcher

- Interactive Session: Tidy data
- – break –
- Interactive Session: Tidier data
- – lunch –
- Interactive Session: Tidiest data
- – end –

Day 5 – The world is yours

- Presentation: Recap
- – break –
- Interactive Session: Open Floor
- – lunch –
- Optional Session: More Open Floor
- – end –

1.3 About this Workshop

The aim of this 4-day introductory workshop is to guide you through the basics of using R via RStudio for analysis of environmental and biological data. It is ideal for people new to R or who have limited experience. This workshop is not comprehensive, but is necessarily selective. We are not hardcore statisticians, but rather ecologists who have an interest in statistics, and use R frequently. Our emphasis is thus on the steps required to analyse and visualise data in R, rather than focusing on the statistical theory.

The workshop is laid out so it begins simply and slowly to impart the basics of using R. It then gathers pace, so that by the end we are doing intermediate level analyses. Day 1 is concerned with becoming familiar with getting data into R, doing some simple descriptive statistics, data manipulation and visualisation. Day 2 takes a more in depth look at manipulating and visualising data. Day 3 focuses on creating maps. Day 4 deals with the fundamentals of reproducible research. The 3-week self study/practice opportunity allows one to utilise all of the skills learned throughout the week by creating a final project. The workshop is case-study driven, using data and examples primarily from our background in the marine sciences and real life situations. There is homework and in class assignments.

Don't worry if you feel overwhelmed and do not follow everything at any time during the Workshop; that is totally natural with learning a new and powerful program. Remember that you have the notes and material to go through the exercises later at your own pace; we will also be walking the room during sessions and breaks so that we can answer questions one-on-one. We hope that this Workshop gives you the confidence to start incorporating R into your daily workflow, and if you are already a user, we hope that it will expose you to some new ways of doing things.

Finally, bear in mind that we are self-taught when it comes to R. Our methods will work, but you will learn as you gain more experience with programming that there are many ways to get the right answer or to accomplish the same task.

1.4 Why use R?

As scientists, we are increasingly driven to analyse and manipulate datasets. As these datasets grow in size our analyses are becoming more sophisticated. There are many statistical packages on the market that one can use, but R has become the global standard. There are several reasons for this trend in R gaining popularity:

1. It is **free**, which is nice if you despise commercial software such as Microsoft Office, as we do — in fact, this entire document was written in Rmarkdown and the files supporting this Workshop material can be edited on *any* computer using a variety of operating systems such as MacOS, Linux, and Microsoft Windows
2. It is powerful, flexible and robust; it is developed and used by leading academic statisticians
3. It contains advanced statistical routines not yet available in other software
4. The cutting-edge statistical routines open up scientific possibilities in creative new ways
5. It has state-of-the-art graphics
6. Users continually extend the functionality by updating existing packages and adding new ones and make these available for free

7. It does not depend on a pointy-and-clicky interface, such as SPSS, and requires one to write scripts — more on the advantages of scripts later
8. It is a coding langauge, so has tools to easily eliminate repetitive tasks
9. It is equally suited to simple calculations and unbelievably complex data analytical workflows
10. It permits and encourages the principles of **Reproducible Research** (more on this later)

It is truly amazing that such a powerful and comprehensive package is freely available and we are indebted to the developers of R for going down this path.

1.4.1 Some negatives of using R

Although there are many positives of using R, some people perceive some negatives:

1. It can have a steep learning curve for those whom do not like statistics or data manipulation, and it does require frequent use to remain familiar with it and to develop advanced skills
2. Error trapping can be confusing and frustrating
3. Rudimentary debugging, although there are some packages available to enhance the process
4. Handles large datasets (100 MB), but can have some trouble with massive datasets (GBs)
5. Some simple tasks can be tricky to do in R
6. There are multiple ways of doing the same thing

1.4.2 The challenge: learning to program in R

The big difference between R and many other statistical packages that you might have used is that it is not, and never will be, a menu-driven 'point and click' package. R requires you to write your own computer code to tell it exactly what you want to do. This means that there is a learning curve, but these are outweighed by numerous advantages:

1. To write new programs, you can modify your existing ones or those of others, saving you considerable time
2. You have a record of your statistical analyses and thus can re-run your previous analyses exactly at any time in the future, even if you can't remember what you did — this is central to reproducible research
3. The recorded code can include the liberal use of internal documentation, which is often overlooked by practising scientists
4. It is more flexible in being able to manipulate data and graphics than menu-driven software
5. You will develop and improve your programming, which is a valuable general skill
6. You will improve your statistical knowledge
7. You can automate large problems
8. You can provide and share code that underpins published analyses; journals are starting to request the code for analyses in papers, to increase transparency and repeatability
9. Integration with tools like git (*e.g.* GitHub and Bitbucket) enables online collaboration in large statistical research programmes and they allow one to rely on version control systems
10. Programming is simply heaps more fun than point-and-click!

    Code represents the ideas of humans.

1.5 Using your own computer?

1.5.1 Installing R

It is straightforward installing R on your machine. Follow these steps:

1. Go to the CRAN (Comprehensive R Archive Network) R website. If you type 'r' into Google it is the first entry
2. Choose to download R for Linux, Mac or Windows
3. For Windows users, just install 'base' and this will link you to the download file

4. For Mac users, choose the version relevant to your Operating System
5. If you are a Linux user, you know what to do!

1.5.2 Installing RStudio

Although R can run in its own console or in a terminal window (Mac and Linux; the Windows command line is a bit limiting), we will use RStudio in this Workshop. RStudio is a free front-end to R for Windows, Mac or Linux (*i.e.*, R is working in the background). It makes working with R easier, more productive, and organised, especially for new users. There are other front-ends, but RStudio is the most popular. To install:

1. Go to the RStudio website.
2. Choose the 'Download RStudio' button
3. Choose run 'RStudio on your Desktop' and follow the prompts
4. Choose the relevant 'Installers for ALL Platforms' to download
5. Install RStudio as per the instructions.

See you on Monday, 28 January 2020.

— Cheers, AJ and Amieroh

1.6 Resources

Below you can find the source code to some books and other links to websites about R. With some of the technical skills you'll learn in this course you'll be able to download the source code, compile the book on your own computer and arrive at the fully formatted (typeset) copy of the books that you can purchase for lots of money:

- ggplot2. Elegant Graphics for Data Analysis — the R graphics bible
- R for Data Science — data analysis using tidy principles
- R Markdown — reproducible reports in R
- bookdown: Authoring Books and Technical Documents with R Markdown — writing books in R
- Shiny — interactive website driven by R

1.7 Style and code conventions

Early on, develop the habit of unambiguous and consistent style and formatting when writing your code, or anything else for that matter. Pay attention to detail and be pedantic. This will benefit your scientific writing in general. Although many R commands rely on precisely formatted statements (code blocks), style can nevertheless to *some extent* have a personal flavour to it. The key is *consistency*. In this book we use certain conventions to improve readability. We use a consistent set of conventions to refer to code, and in particular to typed commands and package names.

- Package names are shown in a bold font over a grey box, *e.g.* **`tidyr`**.
- Functions are shown in normal font followed by parentheses and also over a grey box , *e.g.* `plot()`, or `summary()`.
- Other R objects, such as data, function arguments or variable names are again in normal font over a grey box, but without parentheses, *e.g.* `x` and `apples`.
- Sometimes we might directly specify the package that contains the function by using two colons, *e.g.* `dplyr::filter()`.
- Commands entered onto the R command line (console) and the output that is returned will be shown in a code block, which is a light grey background with code font. The commands entered start at the beginning of a line and the output it produces is preceded by `R>`, like so:

```
rnorm(n = 10, mean = 0, sd = 13)
```

```
R>  [1] 10.4917451  1.0683589 -3.7437561 -3.2334131  7.9644976 -2.3130205
R>  [7]  5.0977896  4.3434540 -0.1304216  7.7361639
```

Consult these resources for more about R code style :

- Google's R style guide
- The tidyverse style guide
- Hadley Wickham's advanced R style guide

We can also insert maths expressions, like this $f(k) = \binom{n}{k} p^k (1-p)^{n-k}$ or this:

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

1.8 About this document

This document was written in **bookdown** and transformed into the 'GitBook' you see here by **knitr**, **pandoc** and LaTeX (Figure 1.1). All the source code and associated data are available at AJ Smit's GitHub page. You can download the source code and compile this document on your own computer. If you can compile the document yourself you are officially a geek – welcome to the club! Note that you will need to complete the exercises in the chapter, An R workflow, before this will be possible.

Figure 1.1: The Rmarkdown workflow.

You will notice that this repository uses GitHub, and you are advised to set up your own repository for R scripts and all your data. We will touch on GitHub and the principles of reproducible research later, and GitHub forms a core ingredient of such a workflow.

1.9 Exercise: It which shall not be named

Now that you have heard (and perhaps read) our argument about the merits of using R, let's double down and spend the next hour seeing first-hand why we think this. Please open the file 'data/SACTN_data.csv' in MS Excel. Gasp! Yes I know. After all of that and now we are using MS Excel? But trust us, there is method to this madness. Your mission, should you choose to accept it, is to spend the next hour creating monthly climatologies and plotting them as a line graph. The South African Coastal Temperature Network (SACTN, which will be used several times during this workshop) data are three monthly temperature time series, each about 30 years long. To complete this objective you will need to first split up the three different time series, and then figure out how to create a monthly climatology for each. A monthly climatology is the average temperature for a given month at a given place. So in this instance, because we have three time series, we will want 36 total values comprised of January - December monthly means for each site (if a time series is 30 years long, then a climatological December will be the mean temperature of all of the data within the 30 Decembers for which data are available). Once those values have been calculated, it should be a relatively easy task to plot them as a dot and line graph. Please keep an eye on the time, if you are not done within an hour please stop anyway. Less than a quarter of workshop attendees have completed this task in the past.

After an hour has passed we will take a break. When we return we will see how to complete this task via R as part of 'The New Age' demonstration.

1.10 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R> character(0)
```

Chapter 2

RStudio

> "Strange events permit themselves the luxury of occurring."
>
> — Charlie Chan
>
> "Without data you're just another person with an opinion."
>
> — W. Edwards Deming

2.1 Setting up the workspace

2.1.1 General settings

Before we start using RStudio (which is a code editor and environment that runs R) let's first set it up properly. Find the 'Tools' ('Preferences') menu item, navigate to 'Global Options' ('Code Editing') and select the tick boxes as shown in Figure 2.1.

2.1.2 Customising appearance

RStudio is highly customisable. Under the **Appearance** tab under 'Tools'/'Global Options' you can see all of the different themes that come with RStudio. We recommend choosing a theme with a black background (*e.g.* Chaos) as this will be easier on your eyes and your computer. It is also good to choose a theme with a sufficient amount of contrast between the different colours used to denote different types of objects/values in your code.

2.1.3 Configuring panes

You cannot rearrange panes (see below) in RStudio by dragging them, but you can alter their position via the **Pane Layout** tab in the 'Tools'/'Global Options' ('RStudio'/'Preferences' – for Mac). You may arrange the panes as you would prefer however we recommend that during the duration of this workshop you leave them in the default layout.

2.2 The Rproject

A very nifty way of managing workflow in RStudio is through the built-in functionality of the Rproject. We do not need to install any packages or change any settings to use these. Creating a new project is a very simple task, as well. For this course we will be using the `Intro_R_Workshop.Rproj` file you downloaded with the course material so that we are all running identical projects. This will prevent a lot of issues by ensuring we are doing things by the same standard. Better yet, an Rproject integrates seamlessly into version control software (*e.g.* GitHub) and allows for instant world class collaboration on any research project. To initialise the 'Intro_R_Workshop' project on your machine please find where you saved `Intro_R_Workshop.Rproj` file and click on it. We will cover the concepts and benefits of an Rproject more as we move through the course.

2.3 Installing packages

The most common functions used in R are contained within the **`base`** package; this makes R useful 'out of the box.' However, there is extensive additional functionality that is being expanded all the time through the use of packages. Packages are simply collections of code called functions that automate complex mathematical or statistical tasks. One of the most useful features of R is that users are continuously developing new packages and making them available for free. You can find a comprehensive list of available packages on the CRAN website. There are currently (`2020-01-31`) `15357` packages available for R!

If the thought of searching for and finding R packages is daunting, a good place to start is the R Task View page. This page curates collections of packages for general tasks you might encounter, such as Experimental Design,

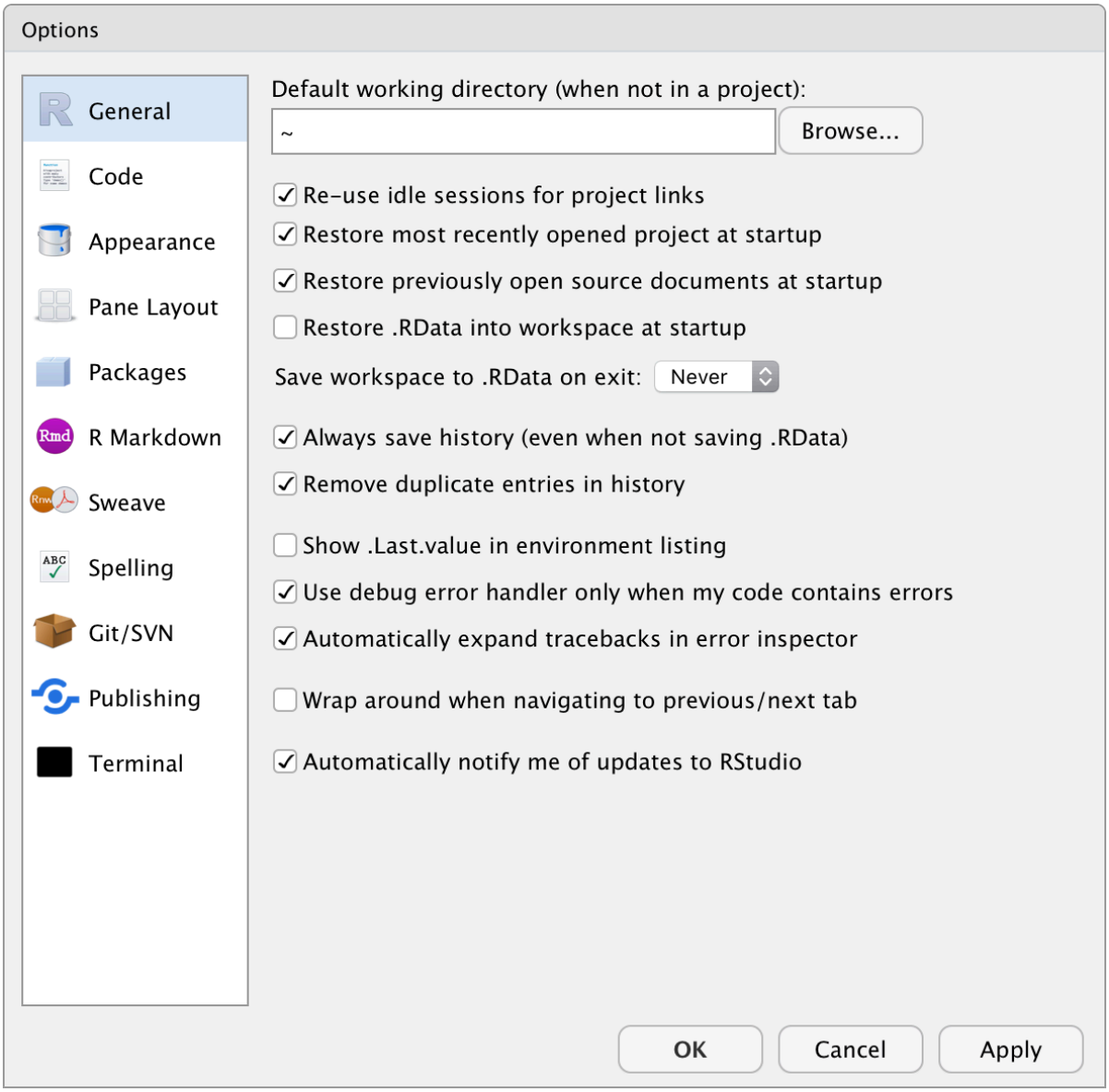


Figure 2.1: The RStudio Preferences menu.

Meta-Analysis, or Multivariate Analysis. Go and have a look for yourself, you might be surprised to find a good explanation of what you need.

After clicking 'Tools'/'Install Packages', type in the package name **tidyverse** in the 'Packages' text box (note that it is case sensitive) and select the Install button. The **Console** will run the code needed to install the package, and then provide some commentary on the installation of the package and any of its dependencies (*i.e.*, other R packages needed to run the required package).

The installation process makes sure that the functions within the packages contained within the **tidyverse** are now available on your computer, but to avoid potential conflicts in the names of functions, it will not load these automatically. To make R 'know' about these functions in a particular session, you need either to load the package via ticking the checkbox for that package in the **Packages** tab, or execute:

```
library(tidyverse)
```

To prepare ourselves for the week ahead, let us also install the following packages. Here I demonstate the command line approach to achieve the same thing that can be done via the menu:

```
# install.packages("rmarkdown")
# install.packages("tidyverse")
# install.packages("bindrcpp")
# install.packages("ggpubr")
# install.packages("magrittr")
# install.packages("boot")
# install.packages("ggsn")
# install.packages("scales")
# install.packages("maps")
# install.packages("ggmap")
# install.packages("lubridate")
# install.packages("bindrcpp")
```

Since we will develop the habit of doing all of our analyses from R scripts, it is best practice to simply list all of the libraries to be loaded right at the start of your script. Comments may be used to remind your future-self (to quote Hadley Wickham) what those packages are for.

> **Copying code from RStudio**
> Here you saw RStudio execute the R code needed to install (using `install.packages()`) and load (using `library()`) the package, so if you want to include these in one of your programs, just copy the text it executes. Note that you need only install the current version of a package once, but it needs to be loaded at the beginning of each R session.
>
> **Question** Why is it best practice to include packages you use in your R program explicitly?

2.4 The panes of RStudio

RStudio has four main panes each in a quadrant of your screen: **Source Editor**, **Console**, **Workspace Browser** (and **History**), and **Plots** (and **Files**, **Packages**, **Help**). These can also be adjusted under the 'Preferences' menu. Note that there might be subtle differences between RStudio installations on different operating systems. We will discuss each of the panes in turn.

2.4.1 Source Editor

Generally we will want to write programs longer than a few lines. The **Source Editor** can help you open, edit and execute these programs. Let us open a simple program:

1. Use Windows Explorer (Finder on Mac) and navigate to the file `BONUS/the_new_age.R`.
2. Now make RStudio the default application to open `.R` files (right click on the file Name and set RStudio to open it as the default if it isn't already)
3. Now double click on the file – this will open it in RStudio in the **Source Editor** in the top left pane.

Note `.R` files are simply standard text files and can be created in any text editor and saved with a `.R` (or `.r`) extension, but the Source editor in RStudio has the advantage of providing syntax highlighting, code completion, and smart indentation. You can see the different colours for numbers and there is also highlighting to help you count brackets (click your cursor next to a bracket and push the right arrow and you will see its partner bracket

highlighted). We can execute R code directly from the Source Editor. Try the following (for Windows machines; for Macs replace **Ctrl** with **Cmd**):

- Execute a single line (Run icon or **Ctrl+Enter**). Note that the cursor can be anywhere on the line and one does not need to highlight anything — do this for the code on line 2
- Execute multiple lines (Highlight lines with the cursor, then Run icon or **Ctrl+Enter**) — do this for line 3 to 6
- Execute the whole script (Source icon or **Ctrl+Shift+Enter**)

Now, try changing the x and/or y axis labels on line 18 and re-run the script.

Now let us save the program in the **Source Editor** by clicking on the file symbol (note that the file symbol is greyed out when the file has not been changed since it was last saved).

At this point, it might be worth thinking a bit about what the program is doing. R requires one to think about what you are doing, not simply clicking buttons like in some other software systems which shall remain nameless for now… Scripts execute sequentially from top to bottom. Try and work out what each line of the program is doing and discuss it with your neighbour. Note, if you get stuck, try using R's help system; accessing the help system is especially easy within RStudio — see if you can figure out how to use that too.

> **Comments**
> The hash (#) tells R not to run any of the text on that line to the right of the symbol. This is the standard way of commenting R code; it is VERY good practice to comment in detail so that you can understand later what you have done.

2.4.2 Console

This is where you can type code that executes immediately. This is also known as the command line. Throughout the notes, we will represent code for you to execute in R as a different font.

> **Type it in!** Although it may appear that one could copy code from this PDF into the **Console**, you really shouldn't. The first reason is that you might unwittingly copy invisible PDF formatting errors into R, which will make the code fail. But more importantly, typing code into the **Console** yourself gives you the practice you need, and allows you to make (and correct) your own errors. This is an invaluable way of learning and taking shortcuts now will only hurt you in the long run.

Entering code in the command line is intuitive and easy. For example, we can use R as a calculator by typing into the Console (and pressing **Enter** after each line):

```
6 * 3
```

```
R> [1] 18
```

```
5 + 4
```

```
R> [1] 9
```

```
2 ^ 3
```

```
R> [1] 8
```

Note that spaces are optional around simple calculations.

We can also use the assignment operator ← to assign any calculation to a variable so we can access it later (the = sign would work, too, but it's bad practice to use it… and we'll talk about this as we go):

```
a <- 2
b <- 7
a + b
```

```
R> [1] 9
```

To type the assignment operator (←) push the following two keys together: **alt -**. There are many keyboard shortcuts in R and we will introduce them as we go along.

Spaces are also optional around assignment operators. It is good practice to use single spaces in your R scripts, and the **alt -** shortcut will do this for you automagically. Spaces are not only there to make the code more readable to the human eye, but also to the machine. Try this:

```
d<-2
d < -2
```

Note that the first line of code assigns `d` a value of `2`, whereas the second statement asks R whether this variable has a value less than 2. When asked, it responds with FALSE. If we hadn't used spaces, how would R have known what we meant?

Another important question here is, is R case sensitive? Is `A` the same as `a`? Figure out a way to check for yourself.

2.5 Exercise

What are the values after each statement in the following?

```
mass <- 48
mass <- mass * 2.0 # mass?
age <- age - 17 # age? m
mass_index <- mass / age # mass_index?
```

Use R to calculate some simple mathematical expressions entered.

Assign the value of 40 to `x` and Assign the value of 23 to `y`. Make `z` the value of `x-y` Display `z` in the console

We can create a vector in R by using the combine `c()` function:

```
apples <- c(5.3, 3.8, 4.5)
```

A vector is a one-dimensional array (*i.e.*, a list of numbers), and this is the simplest form of data used in R (you can think of a single value in R as just a very short vector). We'll talk about more complex (and therefore more powerful) types of data structures as we go along.

If you want to display the value of apples type:

```
apples
```

```
R> [1] 5.3 3.8 4.5
```

Finally, there are default functions in R for nearly all basic statistical analyses, including `mean()` and `sd()` (standard deviation):

```
mean(apples)
```

```
R> [1] 4.533333
```

```
sd(apples)
```

```
R> [1] 0.7505553
```

> **Variable names**
> It is best not to use `c` as the name of a value or array. Why? What other words might not be good to use?

Or try this:

```
round(sd(apples), 2)
```

```
R> [1] 0.75
```

> **Question**
> What did we do above? What can you conclude from those functions?

RStudio supports the automatic completion of code using the **Tab** key. For example, type the three letters `app` and then the **Tab** key. What happens?

The code completion feature also provides brief inline help for functions whenever possible. For example, type `mean()` and press the **Tab** key.

The RStudio **Console** automagically maintains a 'history' so that you can retrieve previous commands, a bit like your Internet browser or Google (*see the code in: BONUS/mapping_yourself.Rmd*). On a blank line in the **Console**, press the up arrow, and see what happens.

If you wish to review a list of your recent commands and then select a command from this list you can use **Ctrl+Up** to review the list (**Cmd+Up** on the Mac). If you prefer a 'bird's eye' overview of the R command history, you may also use the RStudio History pane (see below).

The **Console** title bar has a few useful features:

1. It displays the current R working directory (more on this later)
2. It provides the ability to interrupt R during a long computation (a stop sign will appear whilst code is running)
3. It allows you to minimise and maximise the **Console** in relation to the **Source pane** using the buttons at the top-right or by double-clicking the title bar)

2.5.1 Environment and History panes

The **Environment** pane is very useful as it shows you what objects (*i.e.*, dataframes, arrays, values and functions) you have in your environment (workspace). You can see the values for objects with a single value and for those that are longer R will tell you their class. When you have data in your environment that have two dimensions (rows and columns) you may click on them and they will appear in the **Source Editor** pane like a spreadsheet.

You can then go back to your program in the **Source Editor** by clicking its tab or closing the tab for the object you opened. Also in the **Environment** is the History tab, where you can see all of the code executed for the session. If you double-click a line or highlight a block of lines and then double-click those, you can send it to the **Console** (*i.e.*, run them).

Typing the following into the **Console** will list everything you've loaded into the Environment:

```
ls()
```

```
R> [1] "a"        "apples"   "b"        "pkgs_lst" "url"
```

What do we have loaded into our environment? Did all of these objects come from one script, or more than one? How can we tell where an object was generated?

2.5.2 Files, Plots, Packages, Help, and Viewer panes

The last pane has a number of different tabs. The Files tab has a navigable file manager, just like the file system on your operating system. The **Plot** tab is where graphics you create will appear. The **Packages** tab shows you the packages that are installed and those that can be installed (more on this just now). The **Help** tab allows you to search the R documentation for help and is where the help appears when you ask for it from the **Console**.

Methods of getting help from the **Console** include…

```
?mean
```

…or:

```
help(mean)
```

We will go into this in more detail in the next session.

To reproduced Figure 2.2 in the **Plot** tab, simply copy and paste the following code into the **Console**:

```
library(tidyverse)
x <- seq(0, 2, by = 0.01)
y <- 2 * sin(2 * pi * (x - 1/4))
ggplot() +
  geom_point(aes(x = x, y = y), shape = 21, col = "salmon", fill = "white")
```

2.6 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>   forcats   stringr     purrr     readr    tibble   ggplot2 tidyverse     tidyr
R>   "0.4.0"   "1.4.0"   "0.3.3"   "1.3.1"   "2.1.3"   "3.2.1"   "1.3.0"   "1.0.0"
R>     dplyr     rvest      xml2
R>   "0.8.3"   "0.3.5"   "1.2.2"
```

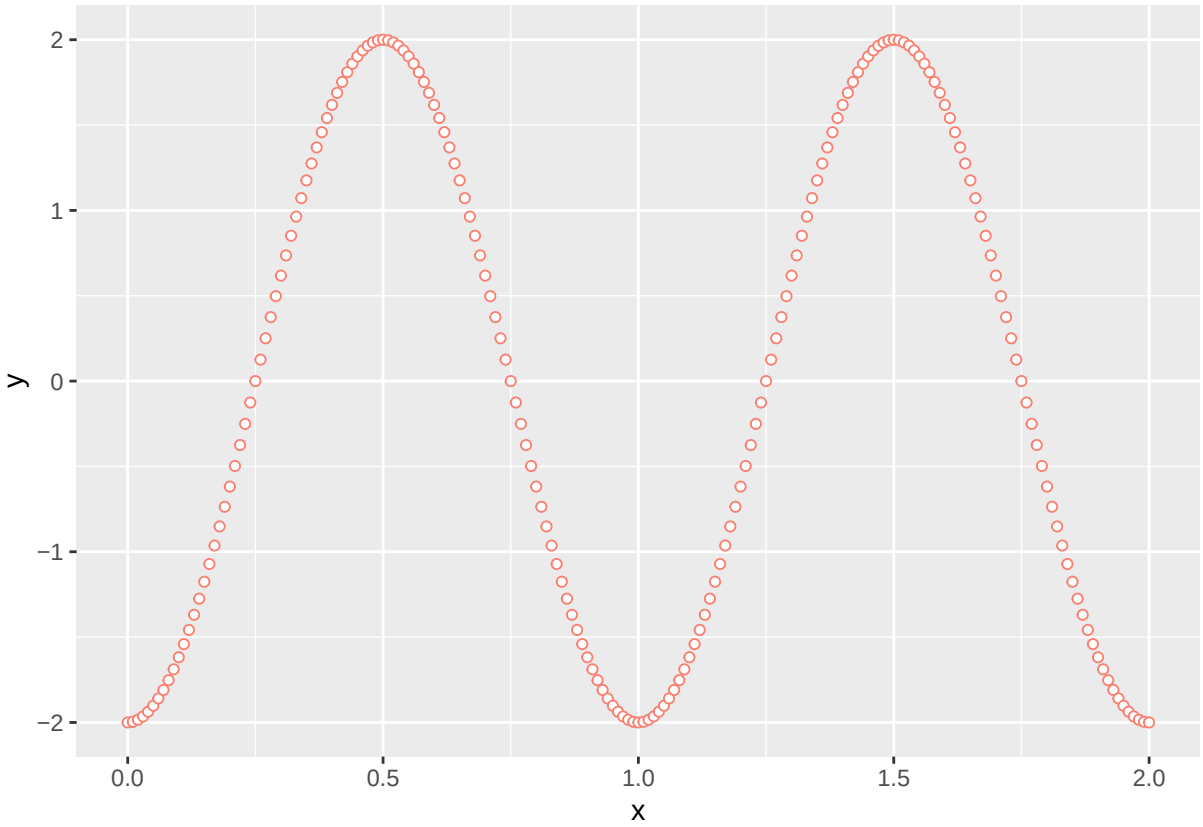


Figure 2.2: The same plot as above, but assembled with **ggplot2**.

Chapter 3

GitHub

3.1 What is GitHub?

GitHub is a code hosting platform for version control and collaboration. It uses the software Git to let you and others work together on projects from anywhere. The following teaches you the GitHub essentials like **Repositories**, **Branches**, **Commits**, and **Pull Requests**. You will first create your own account and repository, and learn GitHub's Pull and Push Request workflow, a popular way to create and review code.

3.2 Installing Git on Windows

By default, Git is installed on Linux and macOS computers as a command line option. However, Microsoft Windows does not include a Git command. To install GitHub on Windows open the Git website (https://git-scm.com/). Click the 'Download' link to download Git. The download should automatically start. Once downloaded, start the installation from the browser or the download folder. In the 'Select Components' window, leave all default options checked and check any other additional components you want installed. Click the 'Install' button. Once completed, you can check the option to launch Git Bash if you want to open a Bash command line.

3.3 Create and account and a repository

Create a GitHub account (https://github.com/join), complete the application steps. This should take less than five minutes. Now lets create a repository. A repository is usually used to organise a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets – anything your project needs. We recommend including a README, or a file with information about your project. GitHub makes it easy to add one at the same time you create your new repository.

Create a new repository — in the upper right corner, next to your avatar or identicon, click and then select 'New repository'. Name your repository (Intro_R) and write a short description.

3.4 Git configuring (activation) and connecting to a remote repository

Launch Git Bash and type the following command to configure your Git username, where will be your GitHub username:

```
git config --global user.name "<your name>"
```

After entering the above command, you should be returned to the command prompt. Next, enter your e-mail address by typing the following command, where is your e-mail address.

```
git config --global user.email "<your e-mail>"
```

3.5 Clone the new GitHub repository to your computer via RStudio.

Select the repository URL from GitHub. It will be something like https://github.com/AmierohAbrahams/Intro_R.

Enter RStudio. On the top left of the screen select the tab 'New Project'. Then select the 'Version control' tab followed by the 'Git' tab. Complete each option by placing the repository URL into the first box and complete the rest.

3.6 Commits

Bravo! Now, on GitHub, saved changes are called *Commits*. Each commit has an associated commit message, which is a description explaining why a particular change was made. Commit messages capture the history of your changes, so other contributors can understand what you've done and why. Make and commit changes.

3.7 Open a Pull Request

Nice edits! Now that you have changes in a branch off of master, you can open a Pull Request.

Pull Requests are the heart of collaboration on GitHub. When you open a pull request, you're proposing your changes and requesting that someone review and pull in your contribution and merge them into their branch. Pull requests show diffs, or differences, of the content from both branches. The changes, additions, and subtractions are shown in green and red.

As soon as you make a commit, you can open a pull request and start a discussion, even before the code is finished.

Chapter 4

An R workflow

> "A dream doesn't become reality through magic; it takes sweat, determination and hard work."
>
> — Colin Powell
>
> "Choose a job you love, and you will never have to work a day in your life."
>
> — Confucius

4.1 R Scripts

The first step for any project in R is to create a new script. We do this by clicking on the 'New Document' button (in the top left and selecting 'R Script'). This creates an unnamed file in the **Source Editor** pane. Best to save it first of all so we do not lose what we do. 'File'/'Save As'/ and the Working Directory should come up. Type in `Day_1` as the file name and click 'save.' R will automatically add a `.R` extension.

It is recommended to start a script with some basic information for you to refer back to later. Start with a comment line (the line begins with a `#`) that tells you the name of the script, something about the script, who created it, and the date it was created. In the source editor enter to following lines and save the file again:

```
# Day_1.R
# Reads in some data about Laminaria collected along the Cape Peninsula
# Do various data manipulations, analyses and graphs
# <your_name>
# <curent_date>
```

Remember that anything appearing after the `#` is not executed by R as script and is a comment.

It is recommend that for each workshop session you start a new script (in the **Source Editor**), type in the code as we go along, and only execute the required lines. That way you will have a record of what you have done.

Below we will learn how to import the file `laminaria.csv` into R, assign it to a dataframe named `laminaria`, and spend a while looking it over. These data reflect results of a sampling campaign on one of the species of kelps (*Laminaria pallida*) in the Western Cape designed to find the morphometric properties of populations at different sites. We visited 13 different locations along the Cape Peninsula (`site`), and at each site, collected *ca.* 13 specimens of the largest kelps we could find. We then brought the kelps back to the shore and measured/calculated nine morphometric properties of the plants (*e.g.* the mass of the fronds (`blade_weight`), the frond length (`blade_length`), *etc.*).

4.2 Reading data into R

We will now see how easy it is to read data into R. R will read in many types of data, including spreadsheets, text files, binary files and files from other statistical packages and software.

Full stops
Unfortunately in South Africa we are taught from a young age to use commas (`,`) instead of full stops (`.`) for decimal places. This simply will not do when we are working with a computer. You must always use a full stop for a decimal place and never insert commas anywhere into any numbers.

Commas
R generally thinks that commas mean the user is telling the computer to separate values. So if you

think you are typing a big number like 2,300 you may actually end up with two numbers. Never use commas with numbers.

4.2.1 Preparing data for R

Importing data can actually take longer than the statistical analysis itself! In order to avoid as much frustration as possible it is important to remember that for R to be able to analyse your data they need to be in a consistent format, with each variable in a column and each sample in a row. The format within each variable (column) needs to be consistent and is commonly one of the following types: a continuous numeric variable (*e.g.*, fish length (m): `0.133`, `0.145`); a factor or categorical variable (*e.g.*, Month: `Jan`, `Feb` or `1`, `2`, …, `12`); a nominal variable (*e.g.*, algal colour: `red`, `green`, `brown`); or a logical variable (*i.e.*, `TRUE` or `FALSE`). You can also use other more specific formats such as dates and times, and more general text formats.

We will learn more about working with data in R — specifically, we will teach you about the *tidyverse* principles and the distinction between *long* and *wide* format data in more detail on Day 4. For most of our work in R we require our data to be in the long format, but Excel users (poor things!) are more familiar with data stored in the wide format. For now let's bring some data into R and not worry too much about the data being tidy.

4.2.2 Converting data

Before we can read in the *Laminaria* dataset provided for the following exercises, we need to convert the Excel file supplied into a `.csv` file. Open 'laminaria.xlsx' in Excel, then select 'Save As' from the File menu. In the 'Format' drop-down menu, select the option called 'Comma Separated Values', then hit 'Save'. You'll get a warning that formatting will be removed and that only one sheet will be exported; simply 'Continue'. Your working directory should now contain a file called `laminaria.csv`.

4.2.3 Importing data

The easiest way to import data into R is by changing your working directory to be the same as the file path where the file(s) are you want to load. A file path is effectively an address. In most operating systems, if you open the folder where your files are you may click on the navigation bar and it will show you the complete file path. Many people develop the nasty habit of squirling away their files within folders within folders within folders… within folders within folders. Please don't do that.

The concept of file paths is either one that you are familiar with, or you've never heard of before. There tends to be little middle ground. Happily, RStudio allows us to circumvent this issue. We do this by using the `Intro_R_Workshop.Rproj` that you may find in the files downloaded for this workshop. If you have not already switched to the `Intro_R_Workshop.Rproj` as outlined in Chapter 2, click on the project button in the top right corner your RStudio window. Then navigate to where you saved `Intro_R_Workshop.Rproj` and select it. Notice that your RStudio has changed a bit and all of the objects you may have previously created in your environment have been removed and any tabs in the source editor pane have been closed. That is fine for now, but it may mean you need to re-open the `Day_1.R` script you just created.

Once we have the working directory set, either by doing it manually with `setwd()` or by loading a project, R will now know where to look for the files we want to read. The function `read_csv()` is the most convenient way to read in raw data. There are several other ways to read in data, but for the purposes of this workshop we'll stick to this one, for now. To find out what it does, we will go to its help entry in the usual way (*i.e.* `?read_csv`).

All R Help items are in the same format. A short *Description* (of what it does), *Usage*, *Arguments* (the different inputs it requires), *Details* (of what it does), *Value* (what it returns) and *Examples*. Arguments (the parameters that are passed to the function) are the lifeblood of any function, as this is how you provide information to R. You do not need to specify all arguments, as most have appropriate default values for your requirements, and others might not be needed for your particular case.

Data formats
R has pedantic requirements for naming variables. It is safest to not use spaces, special characters (*e.g.*, commas, semicolons, any of the shift characters above the numbers), or function names (*e.g.*, mean). One can use 'camelCase', such as `myFirstVariable`, or simply separate the 'parts' of the variable name using an underscore such as in `my_first_variable`. Always make sure to use meaningful names; eventually you will learn to find a balance between meaningfulness and something short that's easy enough to retype repeatedly (although R's ability to use tab completion helps with not having to type long names to often).

Import
`read_csv()` is simply a 'wrapper' (*i.e.*, a command that modifies) a more basic command called

`read_delim()`, which itself allows you to read in many types of files besides `.csv`. To find out more, type `?read_delim()`.

4.2.4 Loading a file

To load the `laminaria.csv` file we created, and assign it to an object name in R, we will use the `read_csv()` function from the **`tidyverse`** package, so let's make sure it is activated.

```
library(tidyverse)
```

Depending on the version of Excel you are using, or perhaps the settings within it, the 'laminaria.csv' file you created may be corrupted in different ways. Generally Excel likes to replace the `,` between columns in our .csv files with `;`. This may seem like a triviality but sadly it is not. Lucky for use, the **`tidyverse`** knows about this problem and they have made a plan. Please open your 'laminaria.csv' file and look at which character is being used to separate columns. If it is `,` then we will load the data with `read_csv()`. If the columns are separated with `;` we will use `read_csv2()`.

```
# Run this if 'laminaria.csv` has columns separated by ','
laminaria <- read_csv("data/laminaria.csv")
# Run this if 'laminaria.csv` has columns separated by ';'
laminaria <- read_csv2("data/laminaria.csv")
```

If one clicks on the newly created `laminaria` object in the **Environment** pane it will open a new panel that shows the information as a spreadsheet. To go back to your script click the appropriate tab in the **Source Editor** pane. With these data loaded we may now perform analyses on them.

At any point when working in R, you can see exactly what objects are in memory in several ways. First, you can look at the **Environment** tab in RStudio, then **Workspace Browser**. Alternatively you can type either of the following:

```
ls()
# or
objects()
```

You can delete an object from memory by specifying the `rm()` function with the name of the object:

```
rm(laminaria)
```

This will of course delete our variable, so we will import it in again using whichever of the following two lines of code matched our Excel situation.

```
laminaria <- read_csv("data/laminaria.csv")
# OR
laminaria <- read_csv2("data/laminaria.csv")
```

Managing variables
It is good practice to remove variables from memory that you are not using, especially if they are large.

4.3 Working with data

4.3.1 Examine your data

Once the data are in R, you need to check there are no glaring errors. It is useful to call up the first few lines of the dataframe using the function `head()`. Try it yourself by typing:

```
head(laminaria)
```

This lists the first six lines of each of the variables in the dataframe as a table. You can similarly retrieve the last six lines of a dataframe by an identical call to the function `tail()`. Of course, this works better when you have fewer than 10 or so variables (columns); for larger data sets, things can get a little messy. If you want more or fewer rows in your head or tail, tell R how many rows it is you want by adding this information to your function call. Try typing:

```
head(laminaria, n = 3)
tail(laminaria, n = 2)
```

You can also check the structure of your data by using the `glimpse()` function:

```
glimpse(laminaria)
```

This very handy function lists the variables in your dataframe by name, tells you what sorts of data are contained in each variable (*e.g.*, continuous number, discrete factor) and provides an indication of the actual contents of each.

If we wanted only the names of the variables (columns) in the dataframe, we could use:

```
names(laminaria)
```

4.3.2 Tidyverse sneak peek

Before we begin to manipulate our data further we need to briefly introduce ourselves to the **`tidyverse`**. And no introduction can be complete within learning about the *pipe* command, `%>%`. We may type this by pushing the following keys together: **ctrl-shift-m**. The pipe (`%>%`) allows us to perform calculations sequentially, which helps us to avoid making errors.

The pipe works best in tandem with the following five common functions:

- Arrange observations (rows) with `arrange()`
- Filter observations (rows) with `filter()`
- Select variables (columns) with `select()`
- Create new variables (columns) with `mutate()`
- Summarise variables (columns) with `summarise()`

We will cover these functions in more detail on Day 4. For now we will ease ourselves into the code with some simple examples.

4.3.3 Subsetting

Now let's have a look at specific parts of the data. You will likely need to do this in almost every script you write. If we want to refer to a variable, we specify the dataframe then the column name within the `select()` function. In your script type:

```
laminaria %>% # Tell R which dataframe we are using
  select(site, total_length) # Select only specific columns
```

If we want to only select values from specific columns we insert one more line of code.

```
laminaria %>%
  select(site, total_length) %>% # Select specific columns first
  slice(56:78)
# what does the '56:78' do? Change some numbers and run the code again. What happens?
```

If we wanted to select only the rows of data belonging to the Kommetjie site, we could type:

```
laminaria %>%
  filter(site == "Kommetjie")
```

The function `filter()` has two arguments: the first is a dataframe (we specify `laminaria` in the previous line and the pipe supplies this for us) and the second is an expression that relates to which rows of a particular variable we want to include. Here we include all rows for Kommetjie and we find that in the variable site. It returns a subset that is actually a dataframe itself; it is in the same form as the original dataframe. We could assign that subset of the full dataframe to a new dataframe if we wanted to.

```
lam_kom <- laminaria %>%
  filter(site == "Kommetjie")
```

DIY: Subsetting
In the script you have started, create a new named dataframe containing only kelps from two of the

sites. Check that the new dataframe has the correct values in it. What purpose can the naming of a newly-created dataframe serve?

4.3.4 Basic stats

Straight out of the box it is possible in R to perform a broad range of statistical calculations on a dataframe. If we wanted to know how many samples we have at Kommetjie, we simply type the following:

```
laminaria %>% # Tell R which dataset to use
  filter(site == "Kommetjie") %>% # Filter out only records from Kommetjie
  nrow() # Count the number of remaining rows
```

Or, if we want to select only the row with the greatest total length:

```
laminaria %>% # Tell R which dataset to use
  filter(total_length == max(total_length)) # Select row with max total length
```

4.4 Exercise

Using pipes, subset the laminaria data to include regions where the blade thickness is thicker than 5 cm and retain only the columns site, region, blade weight and blade thickness.

Now exit RStudio. Pretend it is three days later and revisit your analysis. Calculate the number of entries at Kommetjie and find the row with the greatest length. Do this now.

Imagine doing this daily as our analysis grows in complexity. It will very soon become quite repetitive if each day you had to retype all these lines of code. And now, six weeks into the research and attendant statistical analysis, you discover that there were some mistakes and some of the raw data were incorrect. Now everything would have to be repeated by retyping it at the command prompt. Or worse still (and bad for repetitive strain injury) doing all of it in SPSS and remembering which buttons to click and then re-clicking them. A pain. Let's avoid that altogether and do it the right way by writing an R script to automate and annotate all of this.

> **Dealing with missing data**
> The `.csv` file format is usually the most robust for reading data into R. Where you have missing data (blanks), the `.csv` format separates these by commas. However, there can be problems with blanks if you read in a space-delimited format file. If you are having trouble reading in missing data as blanks, try replacing them in your spreadsheet with `NA`, the missing data code in R. In Excel, highlight the area of the spreadsheet that includes all the cells you need to fill with `NA`. Do an Edit/Replace… and leave the 'Find what:' textbox blank and in the 'Replace with:' textbox enter `NA`, the missing value code. Once imported into R, the `NA` values will be recognised as missing data.

So far we have calculated the mean and standard deviation of some data in the *Laminaria* data set. If you have not, please append those lines of code to the end of your script. You can run individual lines of code by highlighting them and pressing **ctrl-Enter** (**cmd-Enter** on a Mac). Do this.

Your file will now look similar to this one, but of course you will have added your own notes and comments as you went along:

```
# Day_1.R
# Reads in some data about Laminaria collected along the Cape Peninsula
# do various data manipulations, analyses and graphs
# AJ Smit
# 9 January 2020

# Find the current working directory (it will be correct if a project was
# created as instructed earlier)
getwd()

# If the directory is wrong because you chose not to use an Rworkspace (project),
# set your directory manually to where the script will be saved and where the data
# are located
# setwd("<insert_path_here>")

# Load libraries
```

```
library(tidyverse)

# Load the data
laminaria <- read_csv("data/laminaria.csv")

# Examine the data
head(laminaria, 5) # First five lines
tail(laminaria, 2) # Last two lines
glimpse(laminaria) # A more thorough summary
names(laminaria) # THe names of the columns

# Subsetting data
laminaria %>% # Tell R which dataframe to use
  select(site, total_length) %>% # Select specific columns
  slice(56:78) # Select specific rows

# How many data points do we have at Kommetjie?
laminaria %>%
  filter(site == "Kommetjie") %>%
  nrow()

# The row with the greatest length
laminaria %>% # Tell R which dataset to use
  filter(total_length == max(total_length)) # Select row with max total length
```

Making sure all the latest edits in your R script have been saved, close your R session. Pretend this is now 2019 and you need to revisit the analysis. Open the file you created in 2017 in RStudio. All you need to do now is highlight the file's entire contents and hit **ctrl-Enter**.

> **Stick with `.csv` files**
> There are packages in R to read in Excel spreadsheets (*e.g.*, .xlsx), but remember there are likely to be problems reading in formulae, graphs, macros and multiple worksheets. We recommend exporting data deliberately to `.csv` files (which are also commonly used in other programs). This not only avoids complications, but also allows you to unambiguously identify the data you based your analysis on. This last statement should give you the hint that it is good practice to name your `.csv` slightly differently each time you export it from Excel, perhaps by appending a reference to the date it was exported.

> **Remember…**
> Friends don't let friends use Excel.

4.4.1 Summary of all variables in a dataframe

Import the data into a dataframe called `laminaria` once more (if it isn't already in your **Environment**), and check that it is in order. Once we're happy that the data have imported correctly, and that we know what the variables are called and what sorts of data they contain, we can dig a little deeper. Try typing:

```
summary(laminaria)
```

The output is quite informative. It tabulates variables by name, and for each provides summary statistics. For continuous variables, the name, minimum, maximum, first, second (median) and third quartiles, and the mean are provided. For factors (categorical variables), a list of the levels of the factor and the count of each level are given. In either case, the last line of the table indicates how many NAs are contained in the variable. The function `summary()` is useful to remember as it can be applied to many different R objects (*e.g.*, variables, dataframes, models, arrays, *etc.*) and will give you a summary of that object. We will use it liberally throughout the workshop.

4.4.2 Summary statistics by variable

This is all very convenient, but we may want to ask R specifically for just the mean of a particular variable. In this case, we simply need to tell R which summary statistic we are interested in, and to specify the variable to apply it to using `summarise()`. Try typing:

```
laminaria %>% # Chose the dataframe
  summarise(avg_bld_wdt = mean(blade_length)) # Calculate mean blade length
```

Or, if we wanted to know the mean and standard deviation for the total lengths of all the plants across all sites, do:

```
laminaria %>% # Tell R that we want to use the 'laminaria' dataframe
  summarise(avg_stp_ln = mean(total_length), # Create a summary of the mean of the total lengths
            sd_stp_ln = sd(total_length)) # Create a summary of the sd of the total lengths
```

Of course, the mean and standard deviation are not the only summary statistic that R can calculate. Try `max()`, `min()`, `median()`, `range()`, `sd()` and `var()`. Do they return the values you expected? Now try:

```
laminaria %>%
  summarise(avg_stp_ms = mean(stipe_mass))
```

The answer probably isn't what you would expect. Why not? Sometimes, you need to tell R how you want it to deal with missing data. In this case, you have NAs in the named variable, and R takes the cautious approach of giving you the answer of `NA`, meaning that there are missing values here. This may not seem useful, but as the programmer, you can tell R to respond differently, and it will. Simply append an argument to your function call, and you will get a different response. Type:

```
laminaria %>%
  summarise(avg_stp_ms = mean(stipe_mass, na.rm = T))
```

The `na.rm` argument tells R to remove (or more correctly 'strip') NAs from the data string before calculating the mean. It now returns the correct answer. Although needing to deal explicitly with missing values in this way can be a bit painful, it does make you more aware of missing data, what the analyses in R are doing, and makes you decide explicitly how you will treat missing data.

4.4.3 More complex calculations

Let's say you want to calculate something that is not standard in R, say the standard error of the mean for a variable, rather than just the corresponding standard deviation. How can this be done?

The trick is to remember that R is a calculator, so we can use it to do maths, even complex maths (which we won't do). The formula for standard error is:

$$se = \sqrt{\frac{var}{n}}$$

We know that the variance is given by `var()`, so all we need to do is figure out how to get `n` and calculate a square root. The simplest way to determine the number of elements in a variable is a call to the function `nrow()`, as we saw previously. We may therefore calculate standard error with one chunk of code, step by step, using the pipe. Furthermore, by using `group_by()` we may calculate the standard error for all sites in one go.

```
laminaria %>% # Select 'laminaria'
  group_by(site) %>% # Group the dataframe by site
  summarise(var_bl = var(blade_length), # Calculate variance
            n_bl = n()) %>%  # Count number of values
  mutate(se_bl = sqrt(var_bl / n_bl)) # Calculate se
```

When calculating the mean, we specified that R should strip the NAs, using the argument `na.rm = TRUE`. In the example above, we didn't have NAs in the variable of interest. What happens if we *do*?

Unfortunately, the call to the function `nrow()` has no arguments telling R how to treat NAs; instead, they are simply treated as elements of the variable and are therefore counted. The easiest way to resolve this problem is to strip out NAs in advance of any calculations. Try typing:

```
laminaria %>%
  select(stipe_mass) %>%
  summarise(n = n())
```

then:

```
laminaria %>%
  select(stipe_mass) %>%
  na.omit() %>%
  summarise(n = n())
```

You will notice that the function `na.omit()` removes NAs from the variable that is specified as its argument.

> **DIY: Using `na.omit()`** Using this new information, *calculate* the mean stipe mass and the corresponding standard error.

4.5 Exercise

Create a new data frame from the laminaria dataset that meets the following criteria: contains only the site column and a new column called total_length_half containing values that are half of the total_length. In this total_length_half column, there are no NAs and all values are less than 100. Hint: think about how the commands should be ordered to produce this data frame!

Use `group_by()` and `summarise()` to find the mean, min, and max blade_length for each site. Also add the number of observations (hint: see `?n`).

What was the heaviest stipe measured in each site? Return the columns `site`, `region`, and `stipe_length`.

4.6 Saving data

A major advantage of R over many other statistics packages is that you can generate exactly the same answers time and time again by simply re-running saved code. However, there are times when you will want to output data to a file that can be read by a spreadsheet program such as Excel (but try not to… please). The simplest general format is .csv (comma-separated values). This format is easily read by Excel, and also by many other software programs. To output a .csv type:

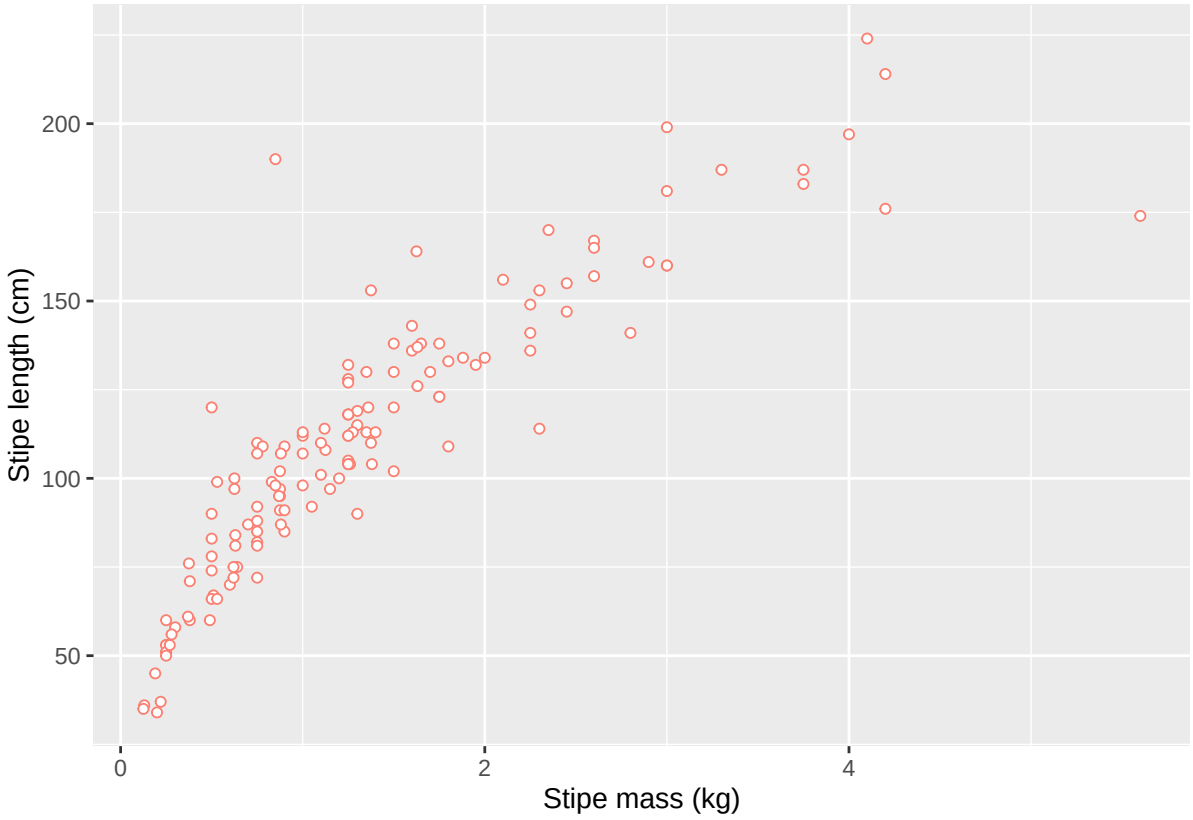
```
write_csv(site_by_region, path = "data/kelp_summary.csv")
```

The first argument is simply the name of an object in R, in this case our table (a data object of class *table*) of counts by region and site (other sorts of data are available, so play around to see what can be done). The second argument is the name of the file you want to write to. This file will always be written to your working directory, unless otherwise specified by including a different path in the file name. Remember that file names need to be within quotation marks. The resultant file can sadly be opened in Excel.

4.7 Visualisations

R has powerful and flexible graphics capabilities. In this Workshop we will not use the traditional graphics (*i.e. base graphics* in the graphics package automatically loaded in R). We will instead use a package called **`ggplot2`** that has the ability for extensive customisation (see the examples at the beginning of tomorrow's section), so it will cover most of the graphs that you will want to produce. We will spend the next two days working on our **`ggplot2`** skills. Here is a quick example of a **`ggplot2`** graphic made from two of the kelp variables to show the relationship between them:

```
ggplot(data = laminaria, aes(x = stipe_mass, y = stipe_length)) +
  geom_point(shape = 21, colour = "salmon", fill = "white") +
  labs(x = "Stipe mass (kg)", y = "Stipe length (cm)")
```



4.8 Clearing the memory

You will be left with many objects after working through these examples. Note that in RStudio when you quit it can save the Environment if you choose, and so it can retain the objects in memory when you start RStudio again. The choice to save the objects resulting from an R Session until next time can be selected in the Global Options menu ('Tools' > 'Global Options' > 'General' > 'Save workspace to .RData on exit'). Personally, we never save objects as it is preferable to start on a clean slate when one opens RStudio. Either way, to avoid long load times and clogged memory, it is good practice to clear the objects in memory every now and then unless you can think of a compelling reason not to. This may be done by clicking on the broom icon at the top of the **Environment** pane.

Of course, you could remove an individual object by placing only its name within the brackets of `rm()`. Do not use this line of code carelessly in the middle of your script; doing so will mean that you have to go back and regenerate the objects you accidentally removed – this is more of a nuisance than a train smash, especially for long, complicated scripts, as you will have (I hope!) saved the R script from which the objects in memory can be regenerated at any time.

4.9 Working directories

At the beginning of this session we glossed over this topic by setting the working directory via RStudio's project functionality. This concept is however critically important to understand so we must now cover it in more detail. The current working directory, where R will read and write files, is displayed by RStudio within the title region of the Console. There are a number of ways to change the current working directory:

1. Select 'Session'/'Set Working Directory' and then choose from the four options for how to set your working directory depending on your preference
2. From within the **Files** pane, navigate to the directory you want to set as the working directory and then select 'More'/'Set As Working Directory' menu item (navigation within the Files pane alone will not change the working directory)
3. Use `setwd()`, providing the name of your desired working directory as a character string

In the **Files** tab, use the directory structure to navigate to the Intro R Workshop directory… this will differ from person to person. Then under 'More', select the small upside down (drill-down) triangle and select 'Set As Working Directory'. This means that whenever you read or write a file it will always be working in that directory. This

gives us the code for setting the directory (below is the code that I would enter in the **Console** on my computer):

```
setwd("~/Intro_R_Workshop")
```

It will be different for you, but copy it into your script and make a note for future reference.

> **Working directories**
> For Windows users, if you copy from a file path the slashes will be the wrong way around and must be changed!

You can check that R got this right by typing into the **Console**:

```
getwd()
```

> **Organising R projects**
> For every R project, set up a separate directory that includes the scripts, data files and outputs.

4.10 Help

The help files in R are not readily clear. It requires a bit of work to understand them well. There is method however to what appears to be madness. The figure below shows the beginning of a help file for a function in R. Please type `?read.table()` in your console now to bring up this help file in your RStudio GUI.

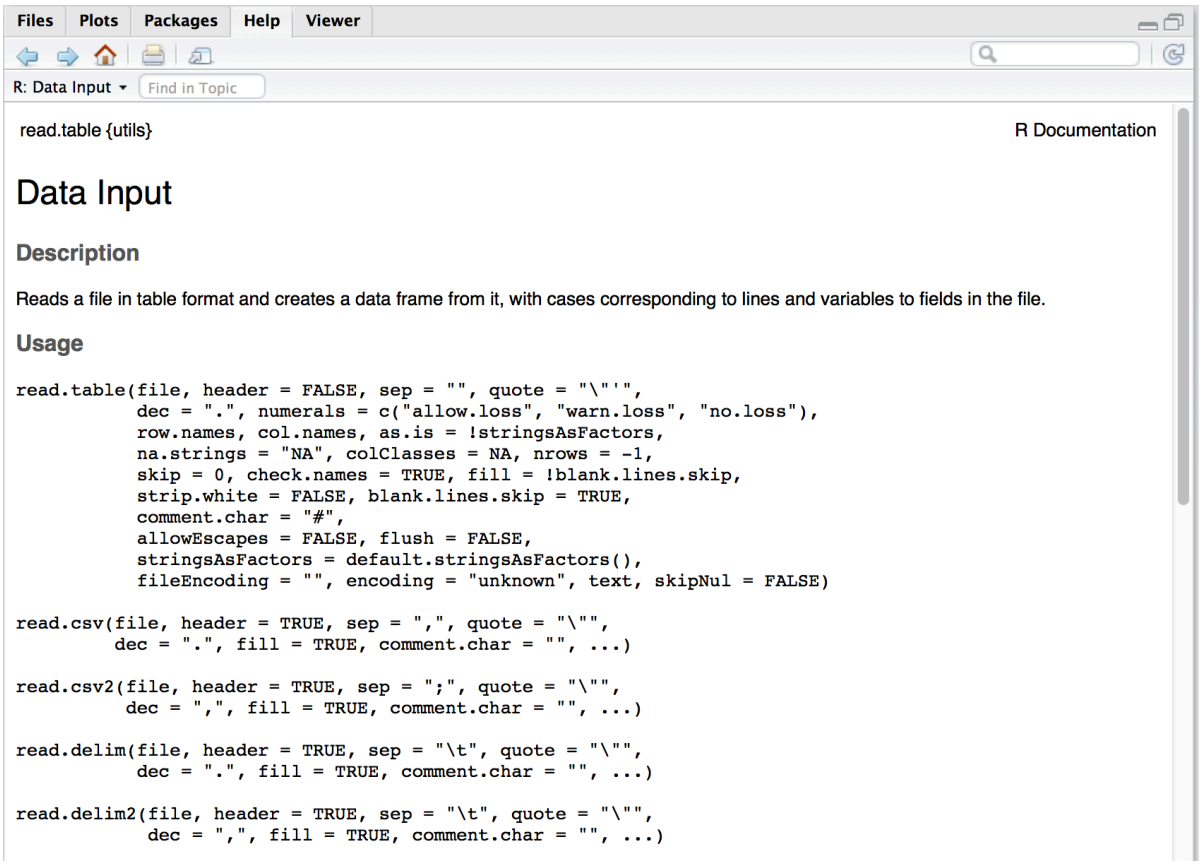


Figure 4.1: A portion of the help page produced by the above command.

The first thing we see at the top of the help file in small font is the name of the function, and the package it comes from in curly braces. After this, in very large text, is a very short description of what the function is used for. After this is the 'Description' section, which gives a sentence or two more fully explaining the use(s) of the function. The 'Usage' then shows all of the arguments that may be given to the function, and what their default settings are. When we write a function in our script we do *not* need to include all of the possible arguments. The help file shows us all of them so that we know what our options are. In some cases a help file will show the usage of several different functions together. This is done, as is the case here, if these functions forma a sort of 'family' and share many common purposes. The 'Arguments' section gives a long explanation for what each individual argument may do. The Arguments section here is particularly verbose. Up next is the 'Details'

section that gives a more in depth description of what the function does. The 'Value' section tells us what sort of output we may expect from the function. Some of the more well documented functions, such as this one, will have additional sections that are not a requirement for function documentation. In this case the 'Memory usage' and 'Note' sections are not things one should always expect to see in help files. Also not always present is a 'References' section. Should there be actual published documentation for the function, or the function has been used in a publication for some other purpose, these references tend to be listed here. There are many functions in the **vegan** package that have been used in dozens of publications. If there is additional reading relevant to the function in question, the authors may also have included a 'See also' section, but this is not standard. Lastly, any well documented function should end with an 'Examples' section. The code is this section is designed to be able to be copy-pasted directly from the help file into the users R script or console and run *as is*. It is perhaps a bad habit, but when I am looking up a help file for a function, I tend to look first at the Examples section. And only if I can't solve my problem with the examples do I actually read the documentation.

4.11 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>   forcats   stringr     dplyr     purrr     readr     tidyr    tibble   ggplot2
R>   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"   "2.1.3"   "3.2.1"
R> tidyverse
R>   "1.3.0"
```

Chapter 5

Graphics with `ggplot2`

> "The greatest value of a picture is when it forces us to notice what we never expected to see."
>
> — John Tukey
>
> "If I can't picture it, I can't understand it."
>
> — Albert Einstein

Though it may have started as statistical software, R has moved far beyond it's mundane origins. The language is now capable of a wide range of applications, some of which you have already seen, and some others you will see over the rest of this course. For the first half of Day 2 we are going to jump straight into data visualisation.

5.1 Example figures

Just to whet the appetite, below is provided a small selection of the figures that R and `ggplot2` are capable of producing. These are things that AJ and/or myself have produced for publication or in some cases just for personal interest. Remember, just because we are learning this for work, doesn't mean we can't use it for fun, too. The idea of using R for fun may seem bizarre, but perhaps by the end of Day 5 we will have been able to convince you otherwise!

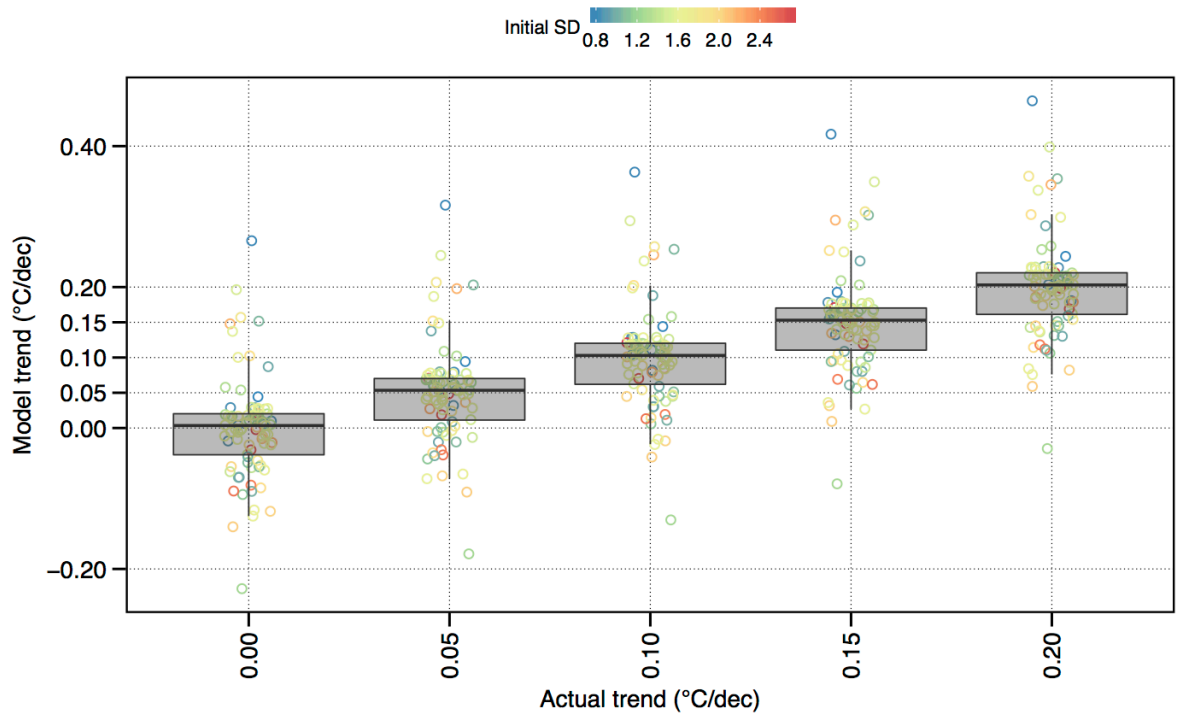


Figure 5.1: The effect of variance (SD) within a temperature time series on the accurate modelling of decadal trends.

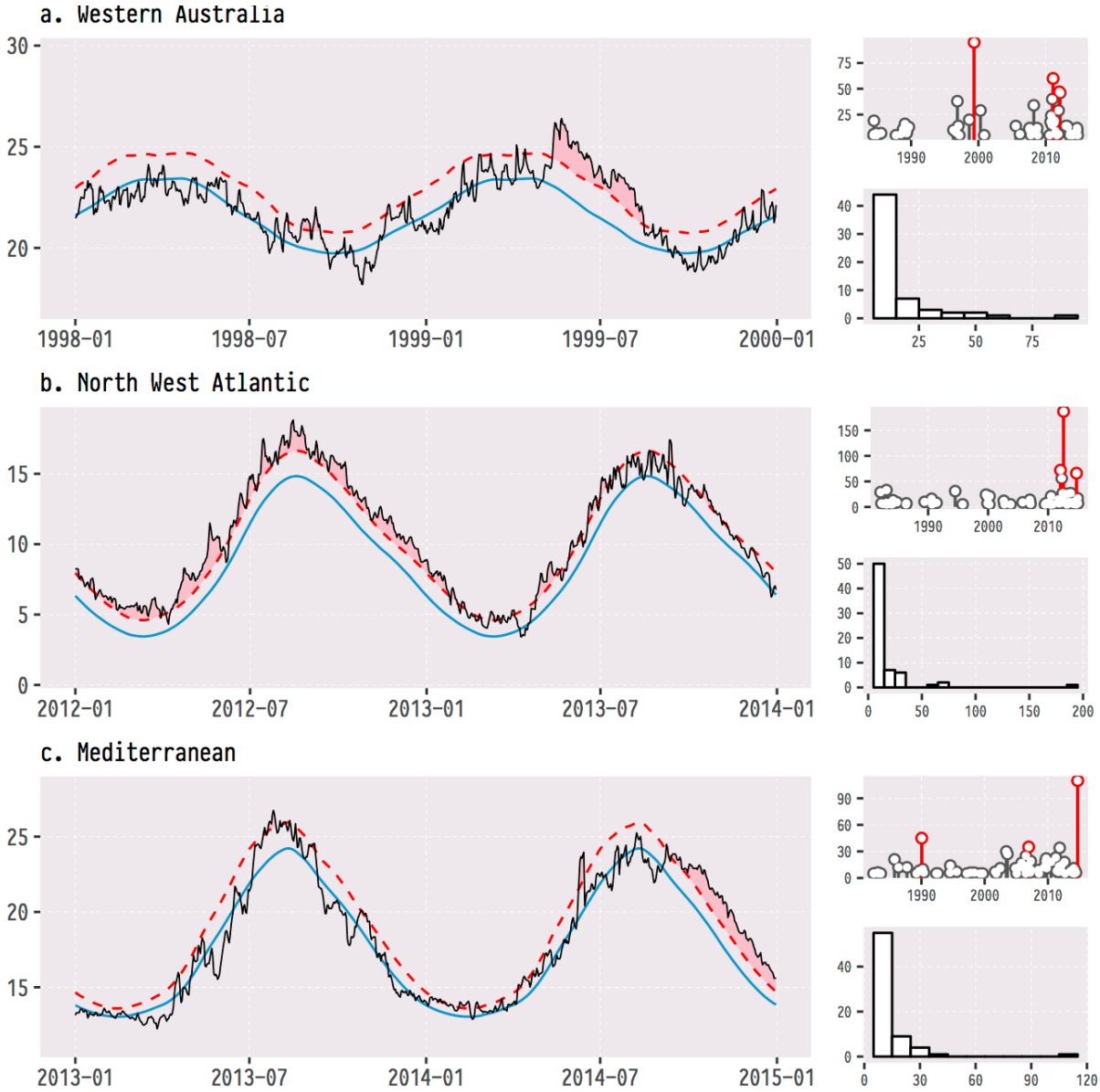


Figure 5.2: The (currently) three most infamous marine heatwaves (MHWs) around the world.

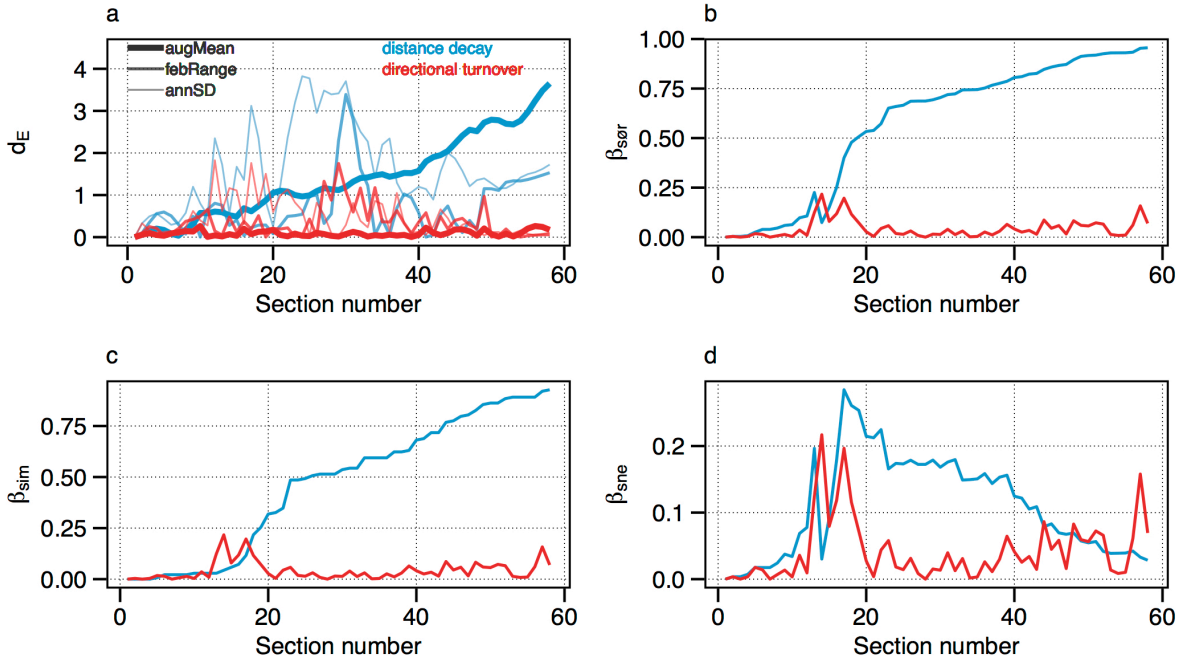


Figure 5.3: Changes in seaweed biodiversity along the South African coastline.

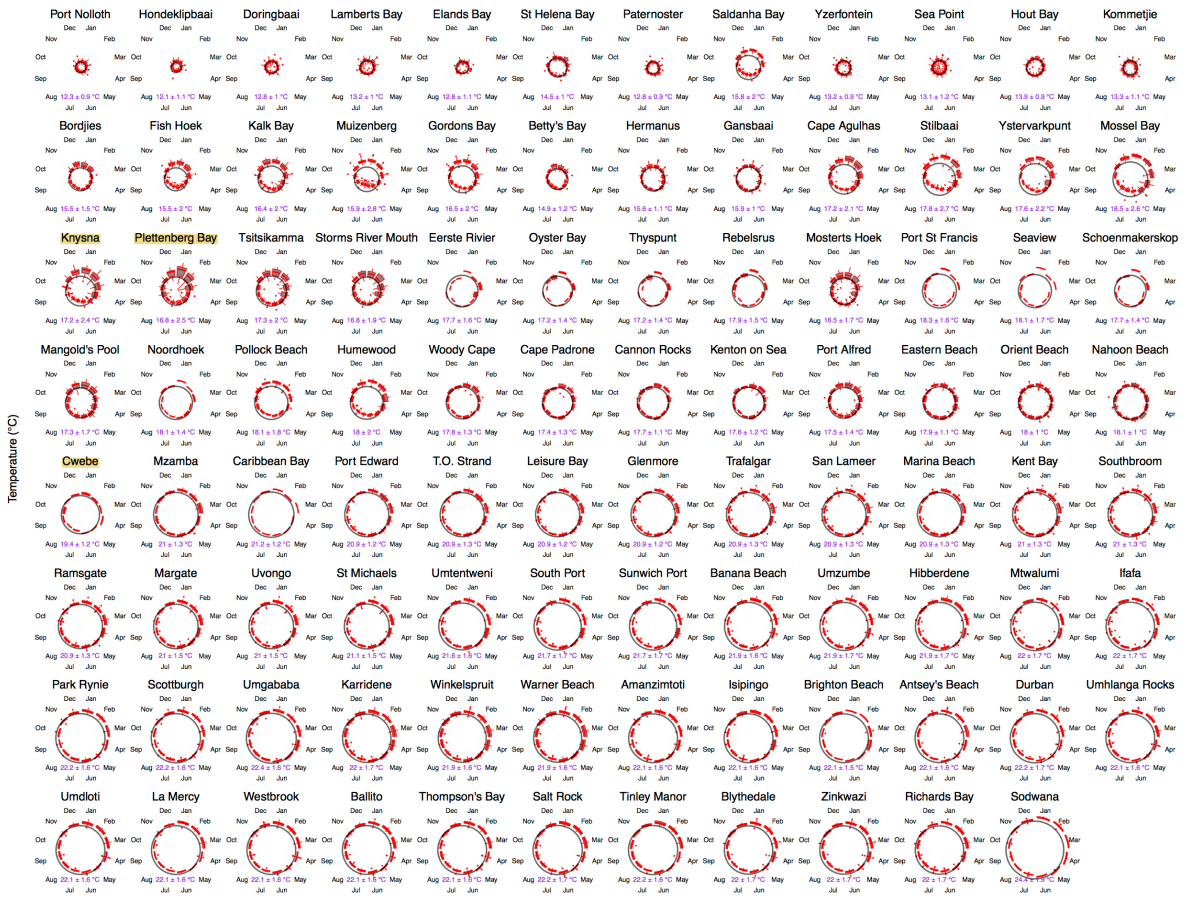


Figure 5.4: Polar plots of monthly temperatures.

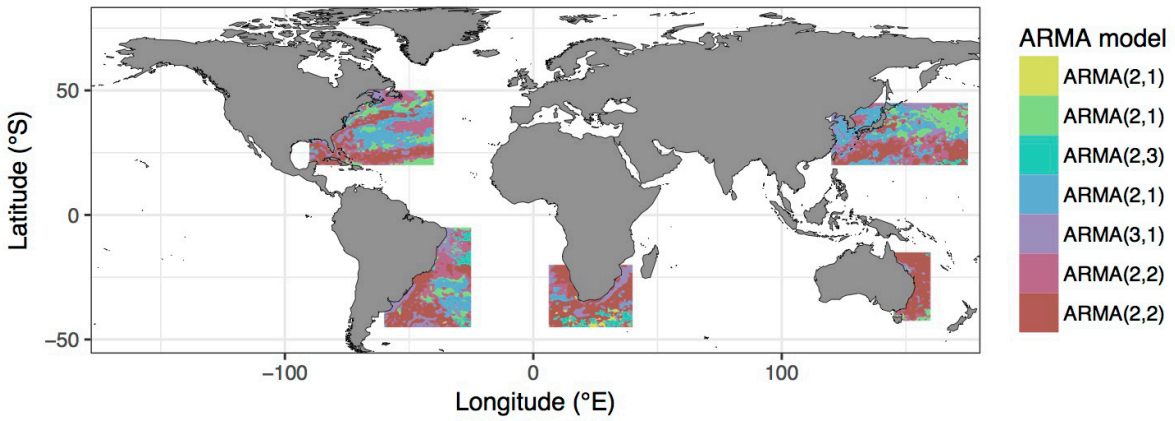


Figure 5.5: Most appropriate autoregressive correlation coefficients for areas around western boundary current.

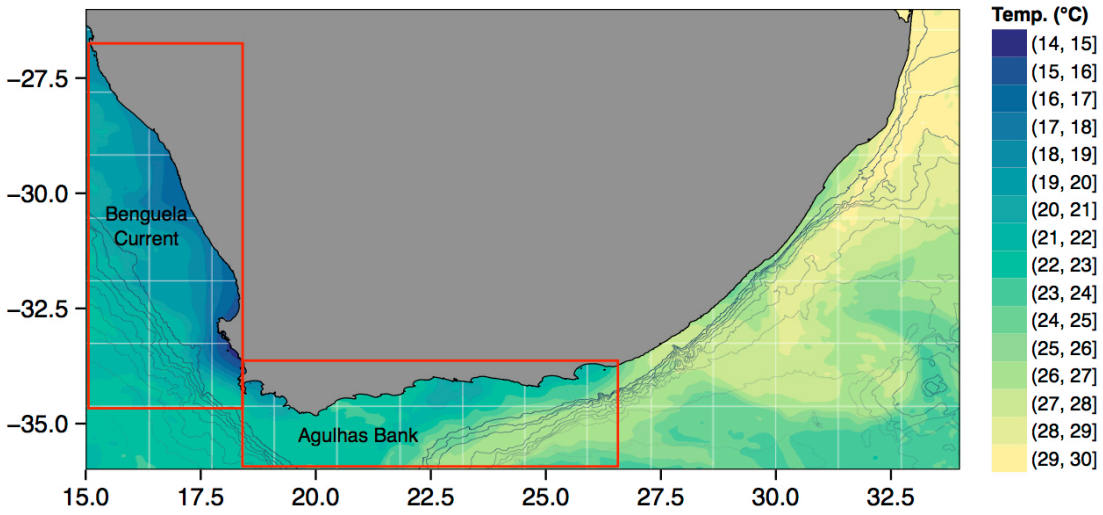


Figure 5.6: The bathymetry of South Africa with SSTs from the MUR product.

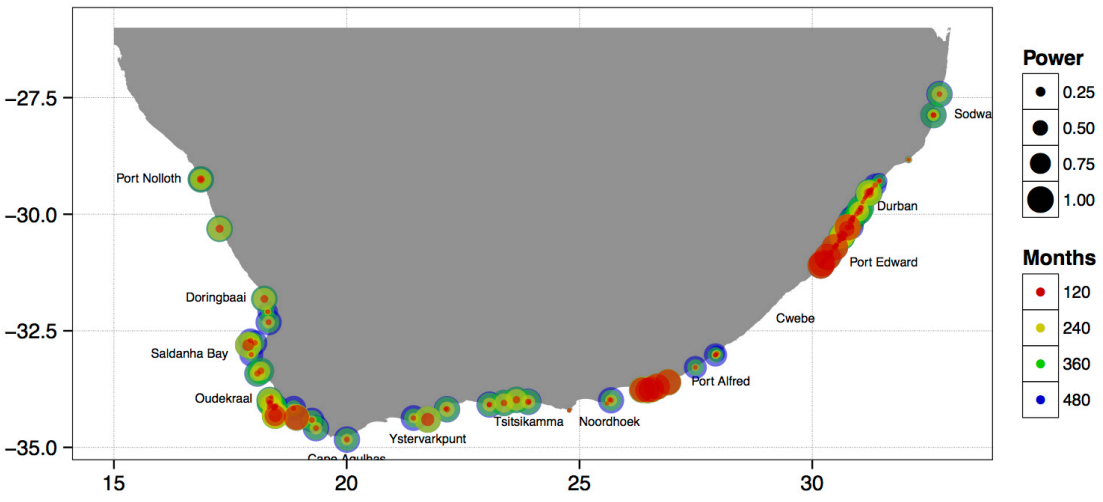


Figure 5.7: The power of the detected decadal trend at each coastal temperature collection site given a hypothetical number of months.

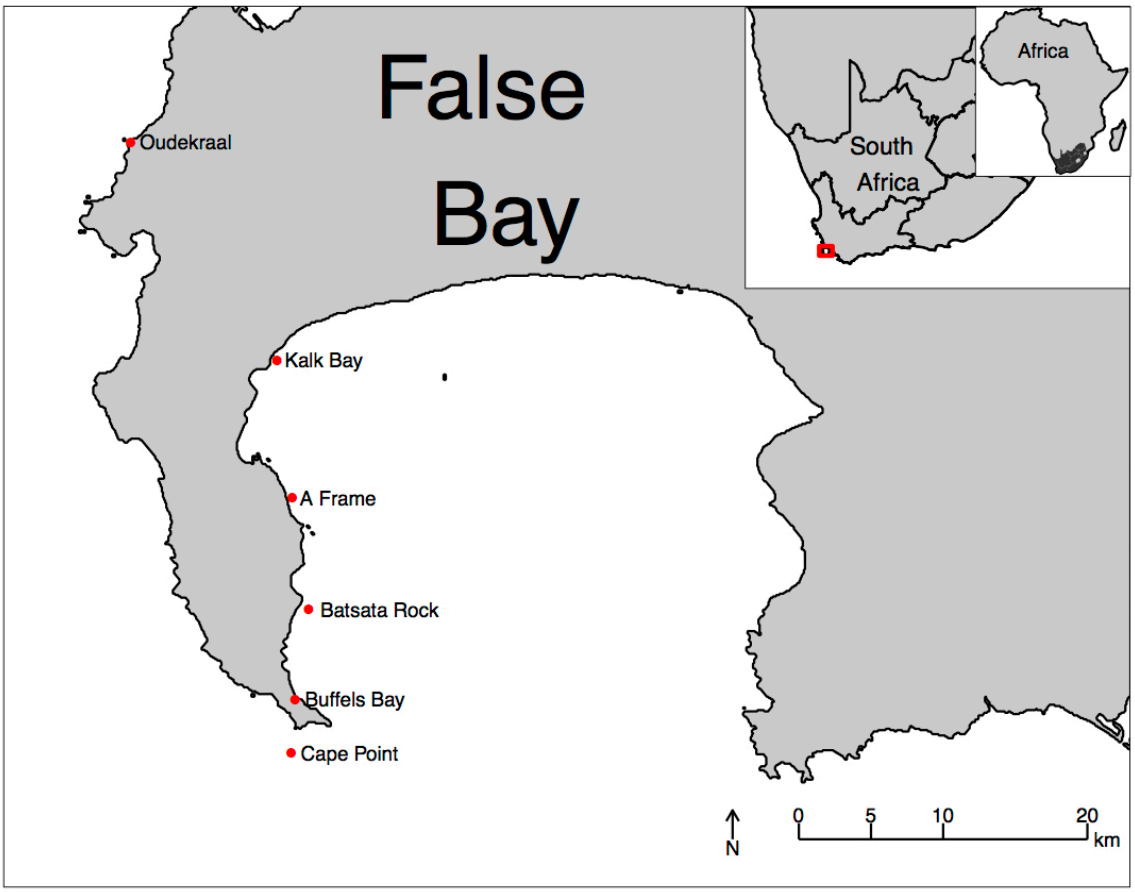


Figure 5.8: An inset map of False Bay.

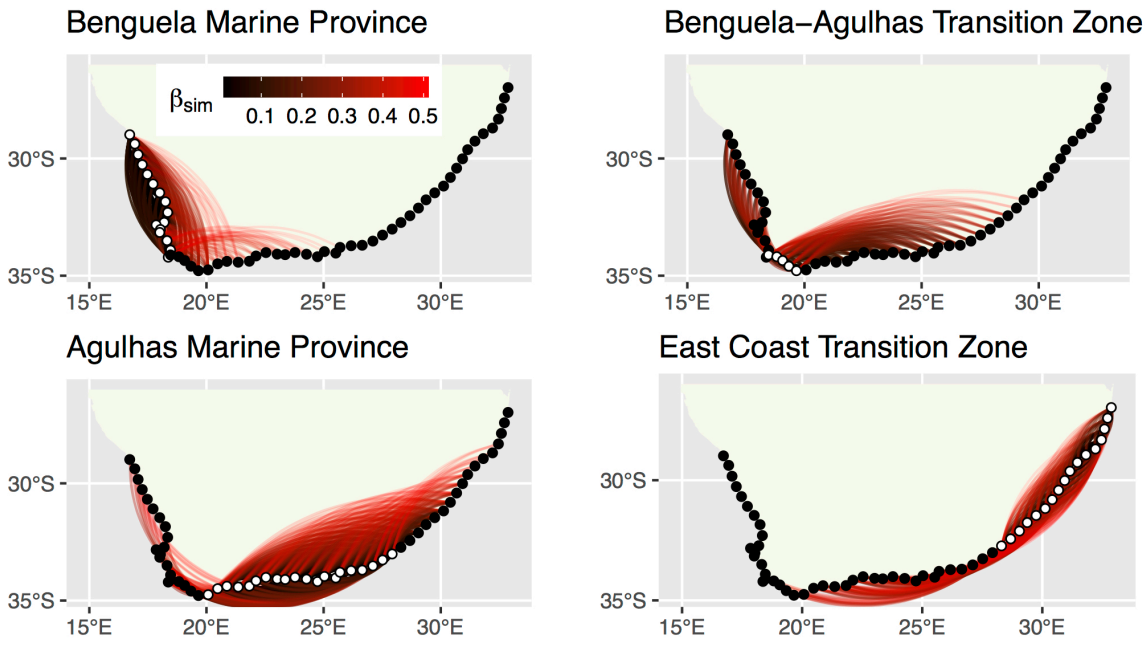


Figure 5.9: The strength of the relationship between each site based on their biodiversity.

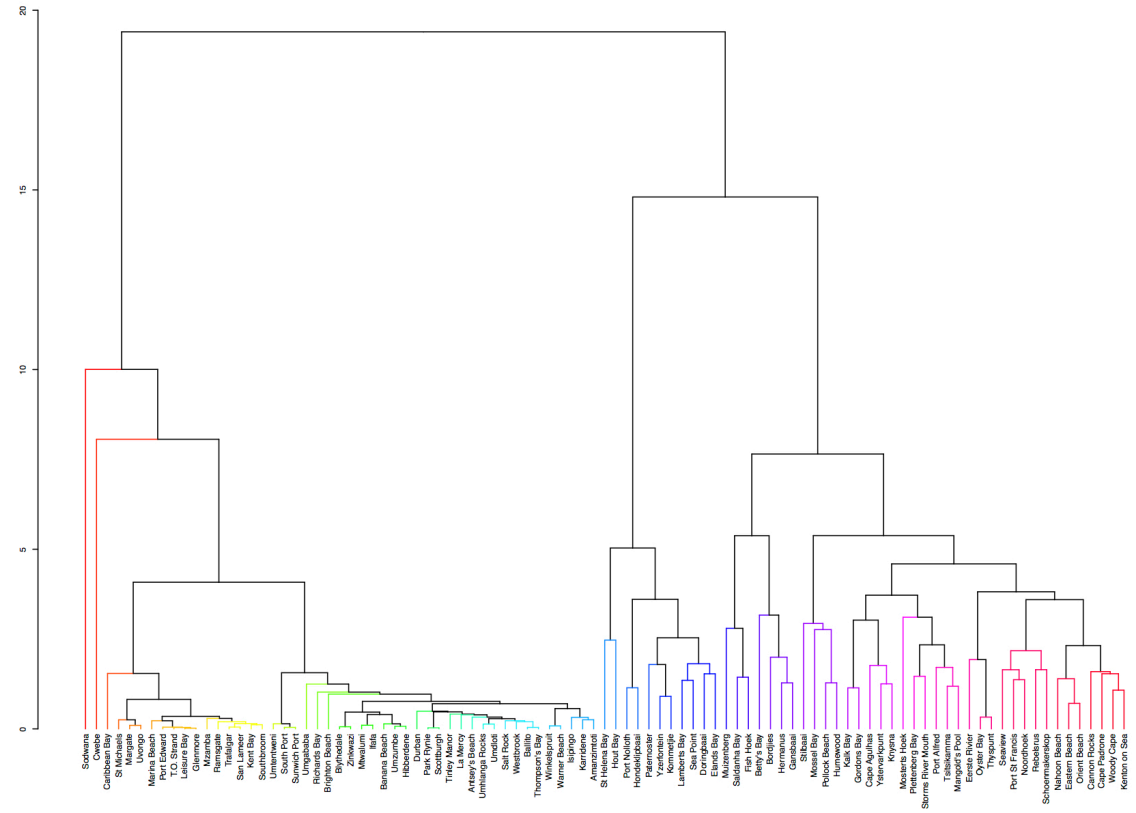


Figure 5.10: A hierarchical cluster analysis.

5.2 Basics of ggplot2

R comes with basic graphing capability, known colloquially (by nerds) as 'Base R'. The syntax used for this method of creating graphics is often difficult to interpret as there are few human words in the code. In addition to this issue, Base R also does not allow the user enough control over the look of the final product to satisfy the demands of many publishers. Meaning the figures tend not to look professional enough (but still much better than Excel). To solve both of these problems, and others, the **ggplot2** package was born.

As part of the **tidyverse** (as we saw briefly on Day 1, and will go into in depth on Day 4), the **ggplot2** package endeavours to use a clean, easy for humans to understand syntax that relies heavily on functions that do what they say. For example, the function `geom_point()` makes points on a figure. Need a line plot? `geom_line()` is the way to go! Need both at the same time? No problem. In **ggplot2** we may seamlessly merge a nearly limitless number of objects together to create startlingly sophisticated figures. Before we go over the code below, it is very important to note the use of the `+` signs. When we use **ggplot2** code we add different lines of code to one another. Each line of code represents one new geometric or aesthetic value of the figure. It is designed this way so as to make it easier for the human eye to read through the code. One may see below that the code naturally indents itself if the previous line ended with a `+` sign. This is because R knows that the top line is the parent line and the indented lines are it's children. This is a concept that will come up again when we learn about tidying data. What we need to know now is that a block of code that has `+` signs, like the one below, must be run together. As long as lines of code end in `+`, R will assume that you want to keep adding lines of code. If we are not mindful of what we are doing we may tell R to do something it cannot and we will see in the console that R keeps expecting more `+` signs. If this happens, click inside the console window and push the `esc` button to cancel the chain of code you are trying to enter.

```
# Load libraries
library(tidyverse)

# Load data
ChickWeight <- datasets::ChickWeight
```

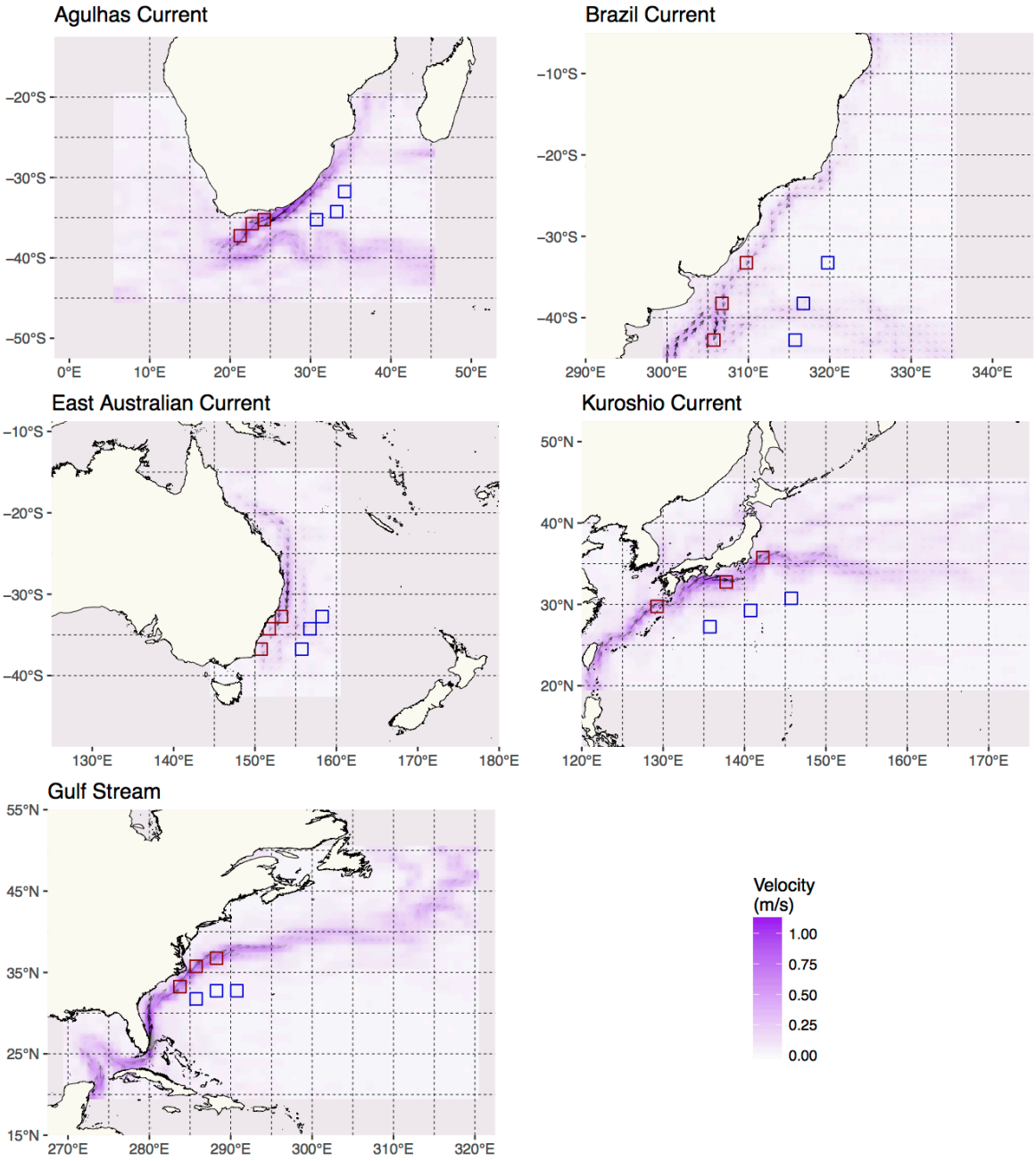
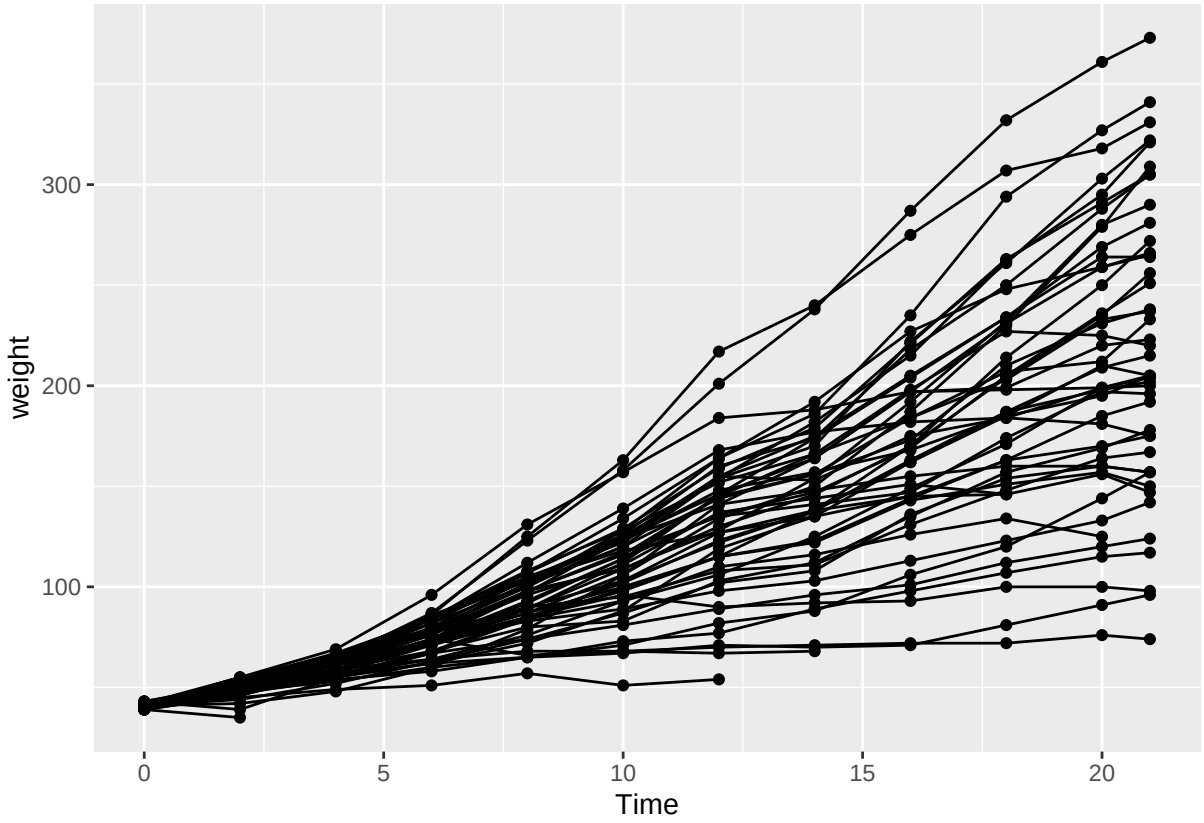


Figure 5.11: Current velcoties of Western Boundary Currents.

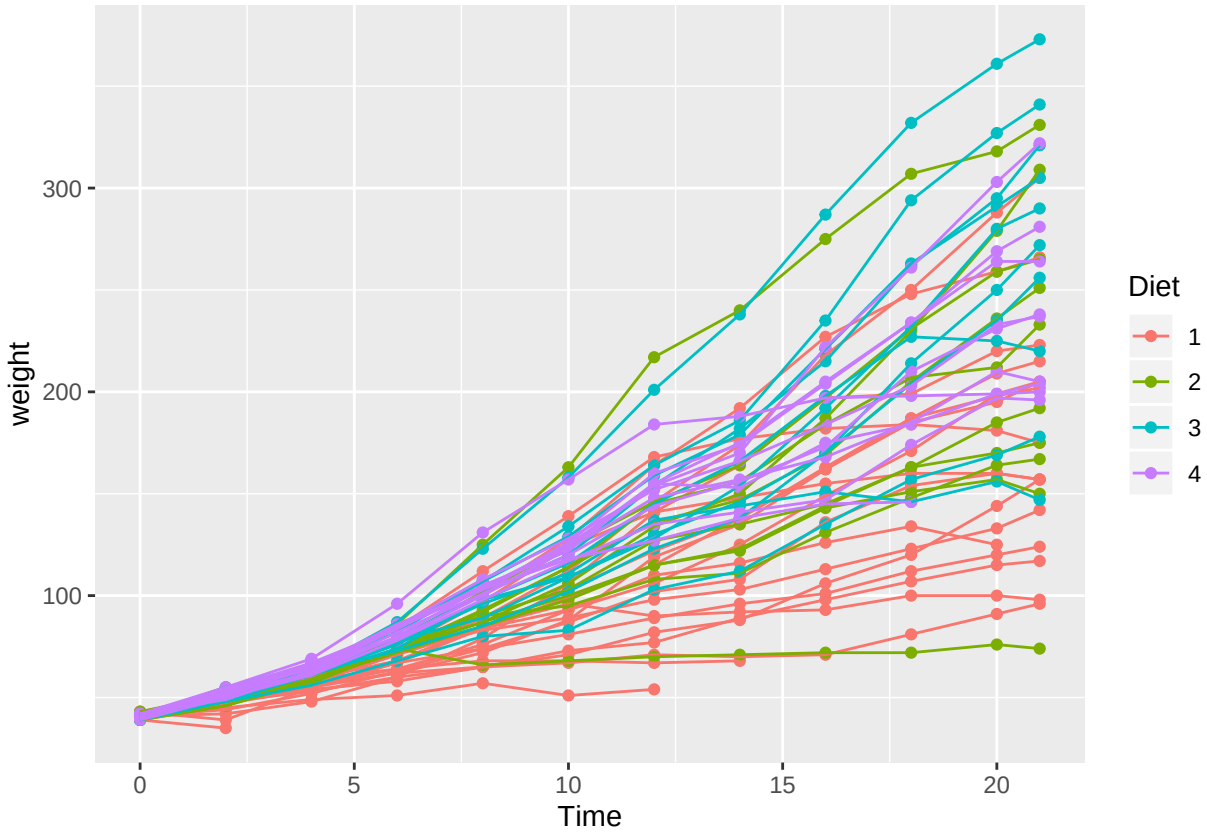
```
# Create a basic figure
ggplot(data = ChickWeight, aes(x = Time, y = weight)) +
  geom_point() +
  geom_line(aes(group = Chick))
```



So what is that code doing? We may see from the figure that it is creating a little black dot for every data point, with the `Time` of sampling on the x axis, and the `weight` of the chicken during that time on the y axis. It then connects the dots for each chicken in the dataset. Let's break this code down line for line to get a better idea of what it is doing. The first line of code is telling R that we want to create a ggplot figure. We know this because we are using the `ggplot()` function. Inside of that function we are telling R which dataframe (or tibble) we want to create a figure from. Lastly, with the `aes()` argument, which is short for 'aesthetic', we tell R what the necessary parts of the figure will be. This is also known as 'mapping'. The second line of code then takes all of that information and makes points (dots) out of it, added as a layer on the set of axes created by the `aes()` argument provided within `ggplot( ... )` — in other words, we add a 'geometry' layer, and hence the name of the kind of 'shape' we want to plot the data as is prefixed by `geom_`. The third line takes the same information and creates lines from it — it adds yet another layer on top of the pre-existing one. Notice in the third line that we have provided another mapping argument by telling R to group the data by `Chick`. This is how R knows to draw an individual line for each chicken, and not just one big messy jagged line. Try running this code without the group argument for `geom_line()` and see what happens.

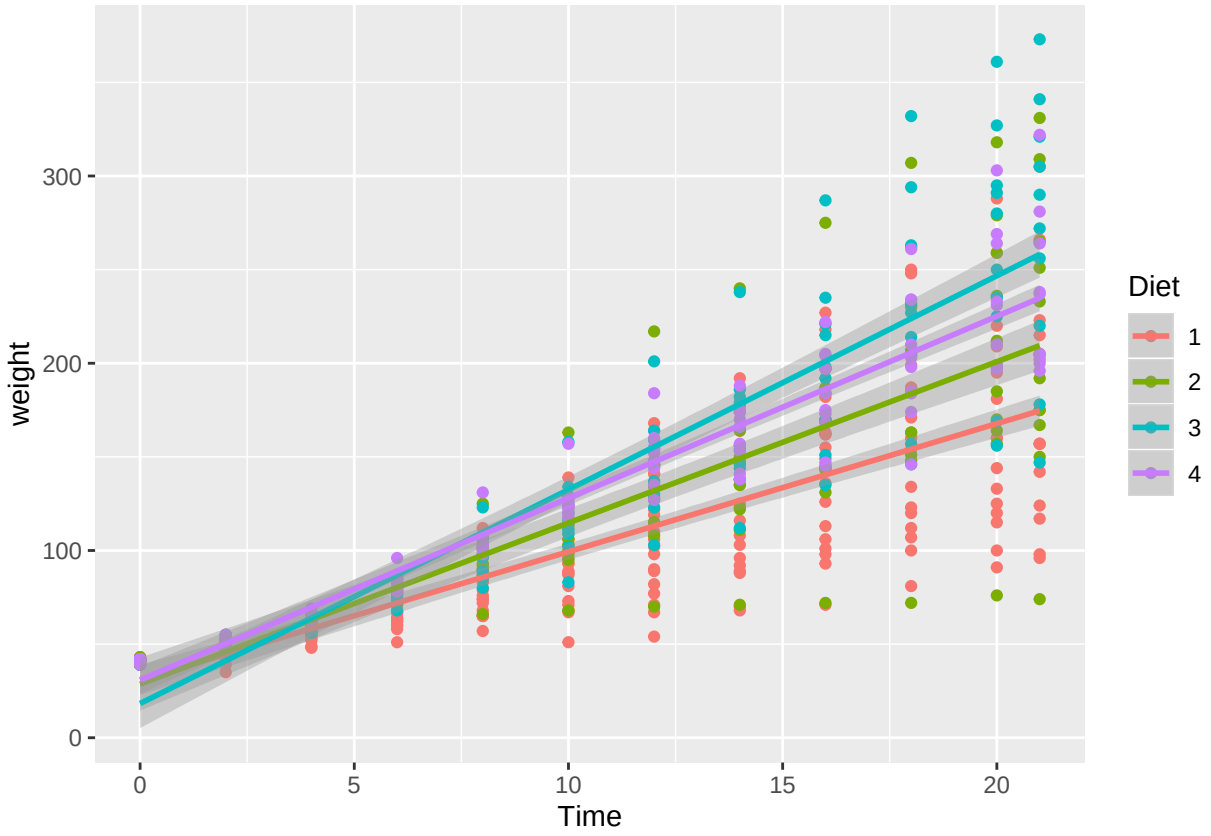
This figure doesn't look like much yet. We saw some examples above that show how sophisticated figures may become. This is a remarkably straight forward task. But don't take my word for it, let's see for ourselves. By adding one more aesthetic to the code above we will now show each `Diet` as a different colour.

```
ggplot(data = ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
  geom_point() +
  geom_line(aes(group = Chick))
```



Do any patterns appear to emerge from the data? Perhaps there is a better way to visualise them? With linear models for example.

```
ggplot(data = ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
  geom_point() +
  geom_smooth(method = "lm")
```

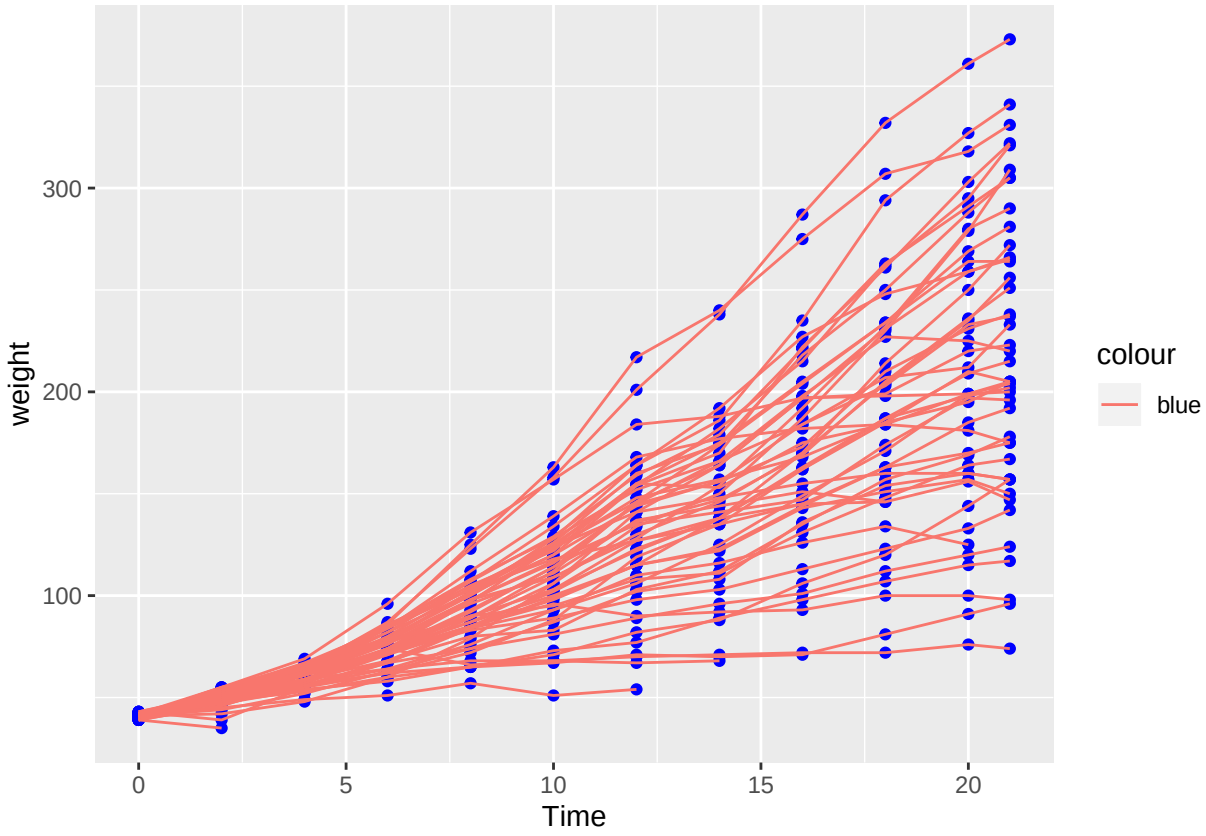


How is a linear model calculated? What patterns do we see in the data now? If you were a chicken, which feed would you want?

5.3 To `aes()` or not to `aes()`, that is the question

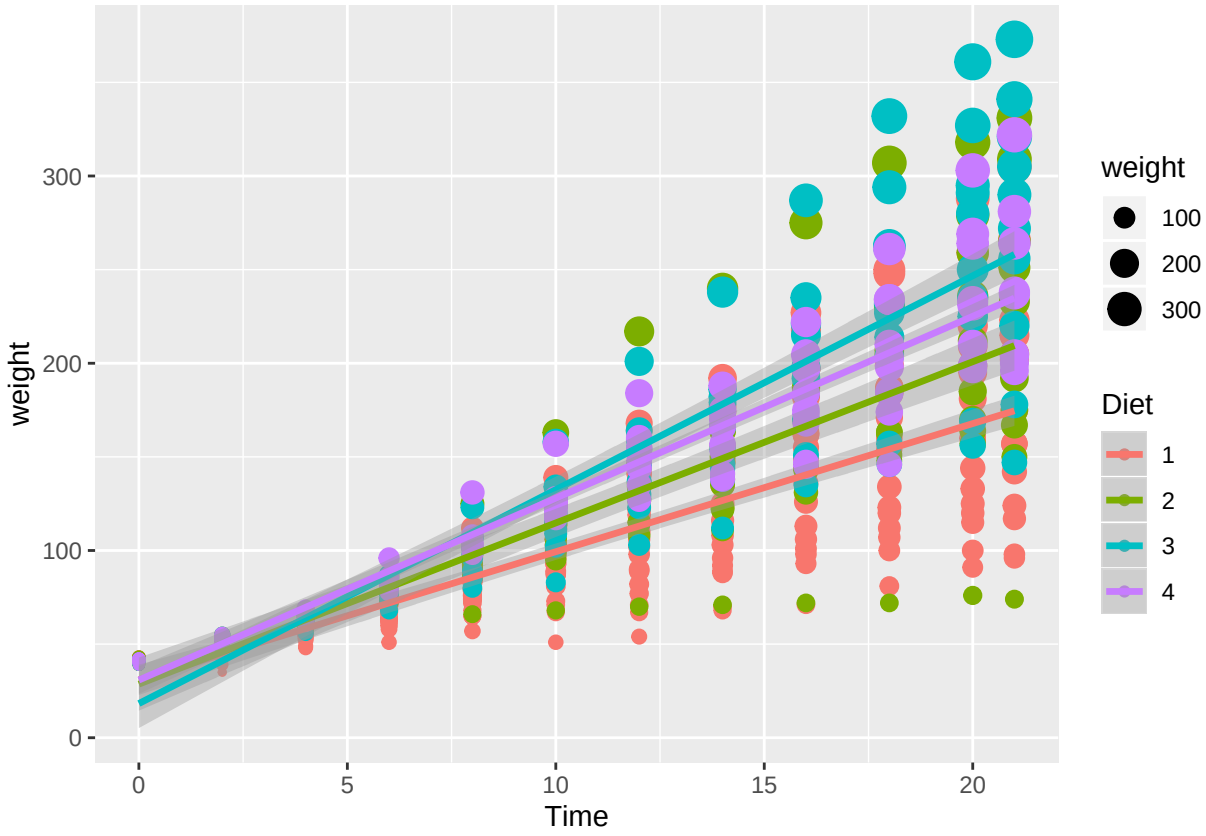
The astute eye will have noticed by now that most arguments we have added to the code have been inside of the `aes()` function. So what exactly is that `aes()` function doing sitting inside of the other functions? The reason for the `aes()` function is that it controls the look of the other functions dynamically based on the variables you provide it. If we want to change the look of the plot by some static value we would do this by passing the argument for that variable to the geom of our choosing *outside* of the `aes()` function. Let's see what this looks like by changing the colour of the dots.

```
ggplot(data = ChickWeight, aes(x = Time, y = weight)) +
  geom_point(colour = "blue") +
  geom_line(aes(group = Chick, colour = "blue"))
```



Why are the points blue, but the lines are salmon with a legend that says they are 'blue'? We may see that in the line responsible for the points (`geom_point()`) we did not put the colour argument inside of the `aes()` function, but for the lines (`geom_line()`) we did. If we know that we want some aspect of our figure to be a static value we set this value outside of the `aes()` function. If we want some aspect of our figure to reflect some part of the data in our dataframe, we must set that inside of `aes()`. Let's see an example where we set the size of the dots to equal the weight of the chicken and the thickness of the linear model lines to one static value.

```
ggplot(data = ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
  geom_point(aes(size = weight)) +
  geom_smooth(method = "lm", size = 1.2)
```

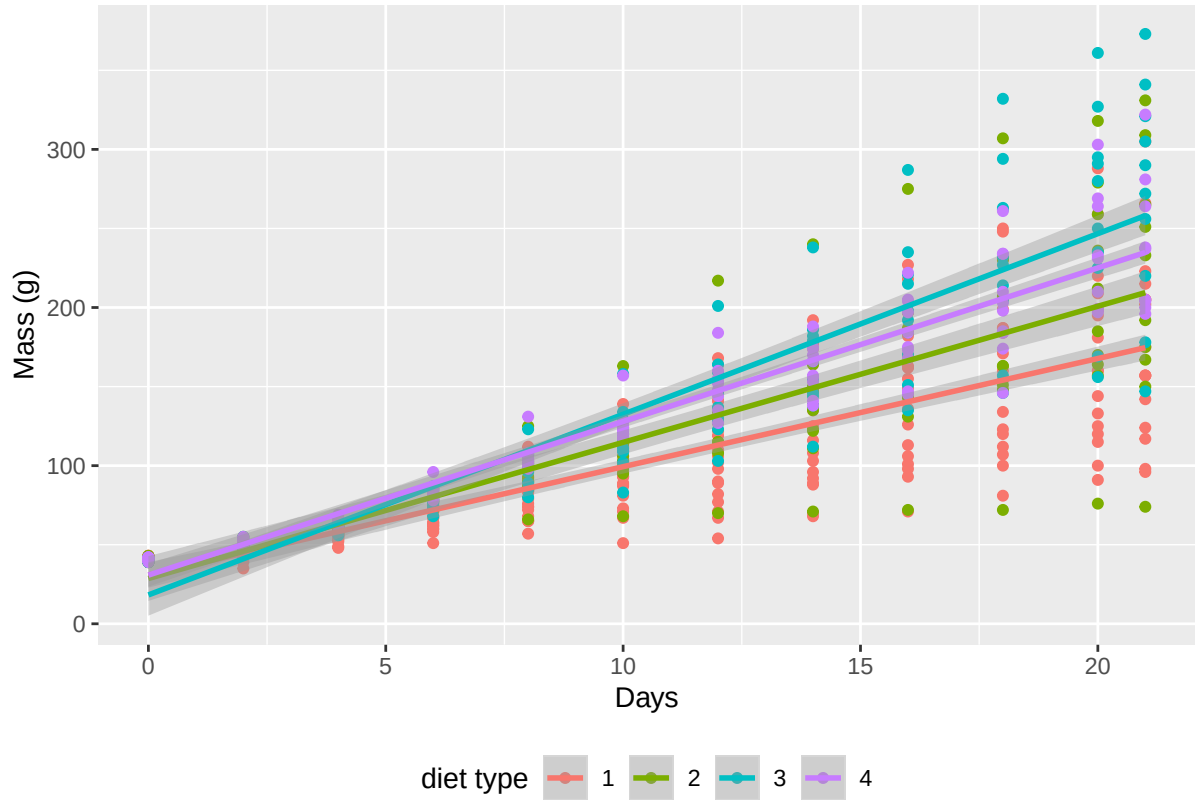


Notice that we have set the size of the points and the lines, but one is within `aes()` and the other not. Because the size of our points equals the weight of the chickens, the points become larger the heavier (juicier) the chickens become. But because we set the size of the lines to one static value, all of the lines are the same size and don't change because of any other variables.

5.4 Changing labels

When we use **ggplot2** we have control over every minute aspect of our figures if we so wish. What we want to do next is put the legend on the bottom of our figure with a horizontal orientation and change the axis labels so that they show the units of measurement. To change the labels we will need the `labs()` function. To change the position of the legend we need the `theme()` function as it is within this function that all of the little tweaks are performed. This is best placed at the end of your block of **ggplot2** code.

```
ggplot(data = ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Days", y = "Mass (g)", colour = "diet type") + # Change the labels
  theme(legend.position = "bottom") # Change the legend position
```



Notice that when we place the legend at the bottom of the figure ggplot automatically makes it horizontal for us. Why do we use 'colour' inside of `labs()` to change the legend title?

5.5 Exercise

With all of this information in hand, please take another five minutes to either improve one of the plots generated or create a beautiful graph of your own. Here are some ideas:

- See if you can change the thickness of the points/lines.
- Change the shape, colour, fill and size of each of the points.
- Can you find a way to change the name of the legend? What about its labels?
- Explore the different geom functions available. These include geom_boxplot, geom_density, etc.
- Try using a different color palette
- Use different themes

5.6 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>   forcats   stringr     dplyr     purrr     readr     tidyr    tibble   ggplot2
R>   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"   "2.1.3"   "3.2.1"
R> tidyverse
R>   "1.3.0"
```

Chapter 6

Faceting Figures

> "But let the mind beware, that though the flesh be bugged, the circumstances of existence are pretty glorious."
>
> — Kurt Vonnegut, Player Piano
>
> "You miss 100% of the shots you don't take."
>
> — Wayne Gretzky

So far we have only looked at single panel figures. But as you may have guessed by now, **ggplot2** is capable of creating any sort of data visualisation that a human mind could conceive. This may seem like a grandiose assertion, but we'll see if we can't convince you of it by the end of this course. For now however, let's just take our understanding of the usability of **ggplot2** two steps further by first learning how to facet a single figure, and then stitch different types of figures together into a grid. In order to aid us in this process we will make use of an additional package, **ggpubr**. The purpose of this package is to provide a bevy of additional tools that researchers commonly make use of in order to produce publication quality figures. Note that `library(ggpubr)` will not work on your computer if you have not yet installed the package.

```
# Load libraries
library(tidyverse)
library(ggpubr)
```

6.1 Faceting one figure

Faceting a single figure is built into **ggplot2** from the ground up and will work with virtually anything that could be passed to the `aes()` function. Here we see how to create an individual facet for each `Diet` within the `ChickWeight` dataset.

```
# Load data
ChickWeight <- datasets::ChickWeight

# Create faceted figure
ggplot(data = ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
  geom_point() +
  geom_smooth(method = "lm") + # Note the `+` sign here
  facet_wrap(~Diet, ncol = 2) + # This is the line that creates the facets
  labs(x = "Days", y = "Mass (g)")
```

6.2 New figure types

Before we can create a gridded figure of several smaller figures, we need to learn how to create a few new types of figures first. The code for these different types is shown below. Some of the figure types we will learn how to use now do not work well with the full `ChickWeight` dataset. Rather we will want only the weights from the final day of collection. To filter only these data we will need to use a bit of the 'tidy' code we saw on Day 1.

```
ChickLast <- ChickWeight %>%
  filter(Time == 21)
```

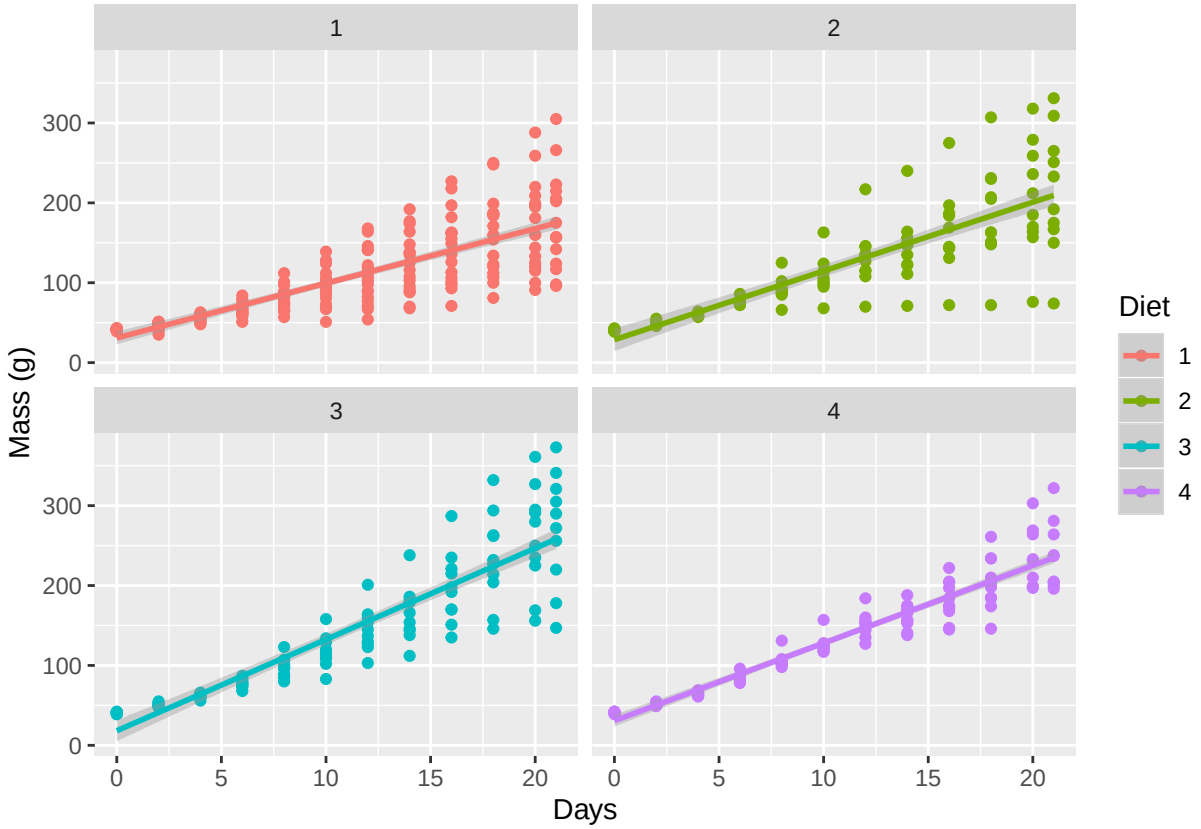


Figure 6.1: Simple faceted figure showing a linear model applied to each diet.

6.2.1 Line graph

```
line_1 <- ggplot(data = ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
  geom_point() +
  geom_line(aes(group = Chick)) +
  labs(x = "Days", y = "Mass (g)")
line_1
```

6.2.2 Linear model

```
lm_1 <- ggplot(data = ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
  geom_point() +
  geom_smooth(method = "gam") +
  labs(x = "Days", y = "Mass (g)")
lm_1
```

6.2.3 Histogram

```
# Note that we are using 'ChickLast', not 'ChickWeight'
histogram_1 <- ggplot(data = ChickLast, aes(x = weight)) +
  geom_histogram(aes(fill = Diet), position = "dodge", binwidth = 100) +
  labs(x = "Final Mass (g)", y = "Count")
histogram_1
```

6.2.4 Boxplot

```
# Note that we are using 'ChickLast', not 'ChickWeight'
box_1 <- ggplot(data = ChickLast, aes(x = Diet, y = weight)) +
  geom_boxplot(aes(fill = Diet)) +
  labs(x = "Diet", y = "Final Mass (g)")
box_1
```

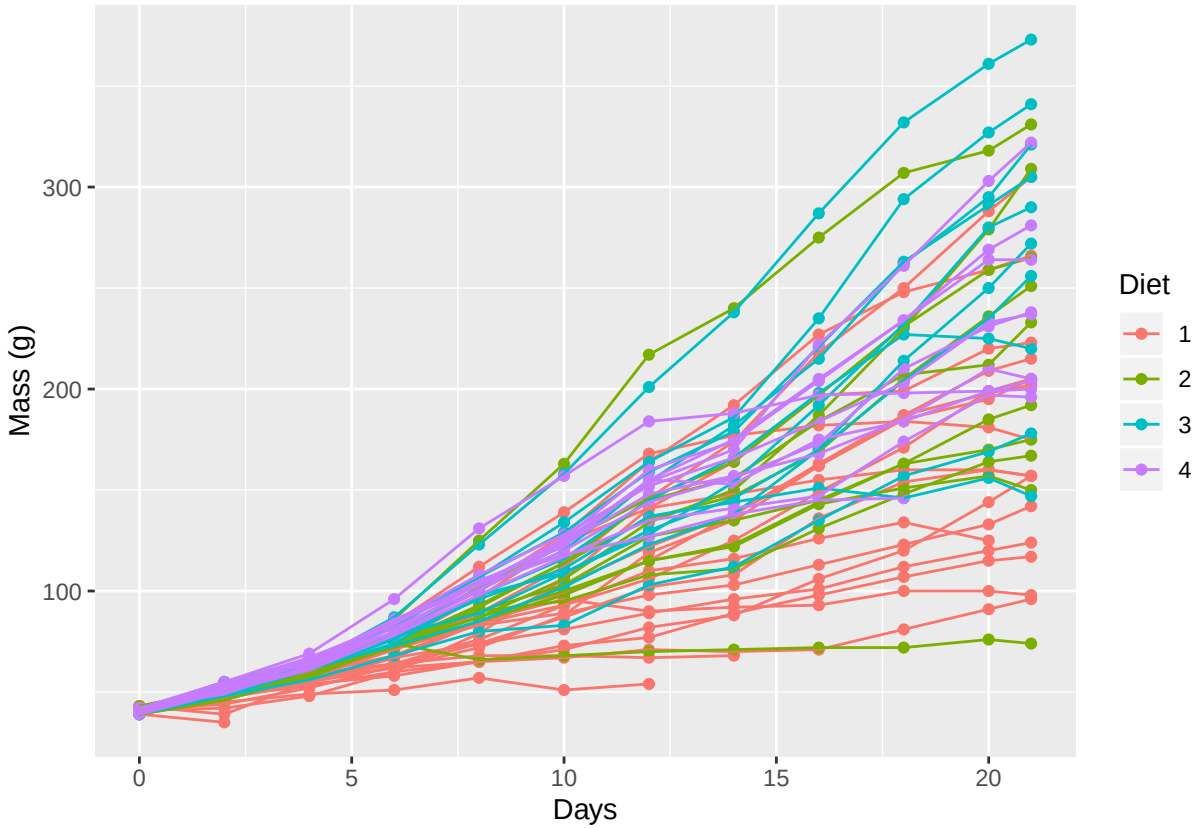


Figure 6.2: Line graph for the progression of chicken weights (g) over time (days) based on four different diets.

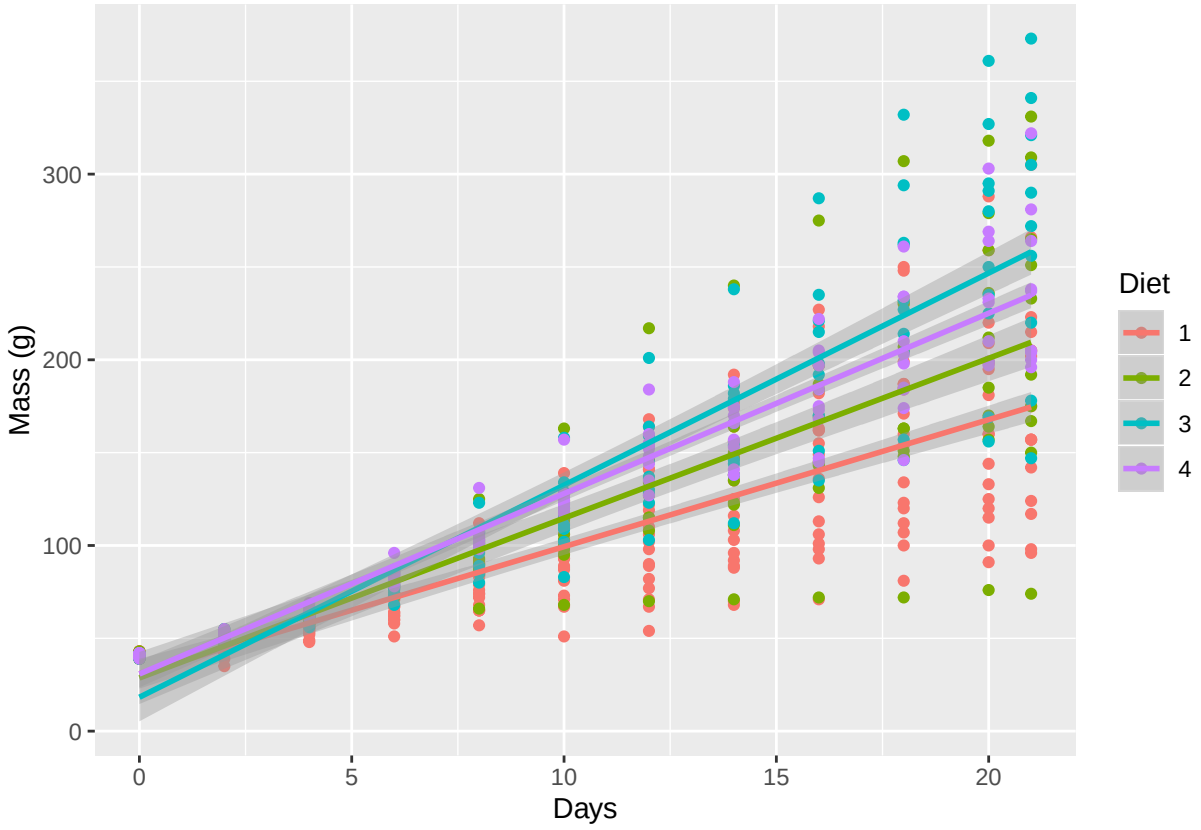


Figure 6.3: Linear models for the progression of chicken weights (g) over time (days) based on four different diets.

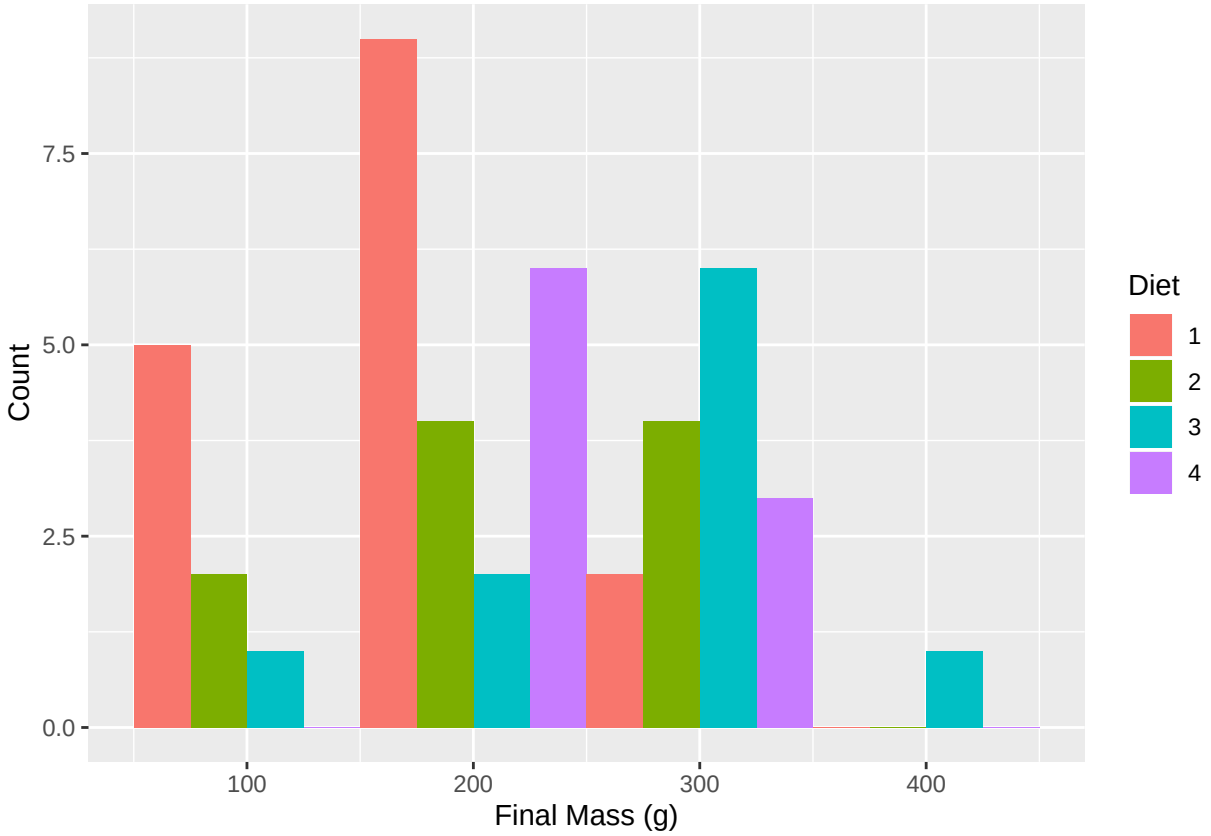


Figure 6.4: Histogram showing final chicken weights (g) by diet.

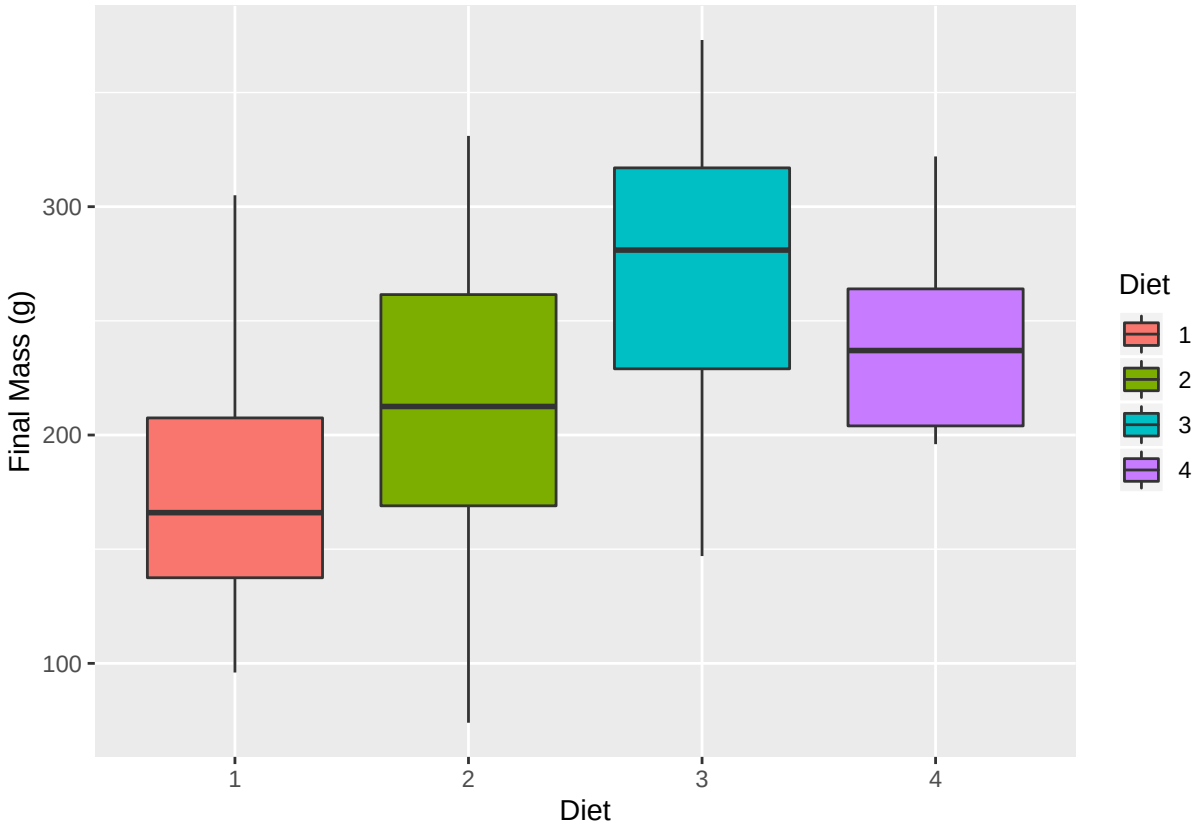


Figure 6.5: Violin plot showing the distribution of final chicken weights (g) by diet.

6.3 Gridding figures

With these four different figures created we may now look at how to combine them. By visualising the data in different ways they are able to tell us different parts of the same story. What do we see from the figures below that we may not have seen when looking at each figure individually?

```
ggarrange(line_1, lm_1, histogram_1, box_1,
          ncol = 2, nrow = 2, # Set number of rows and columns
          labels = c("A", "B", "C", "D"), # Label each figure
          common.legend = TRUE) # Create common legend
```

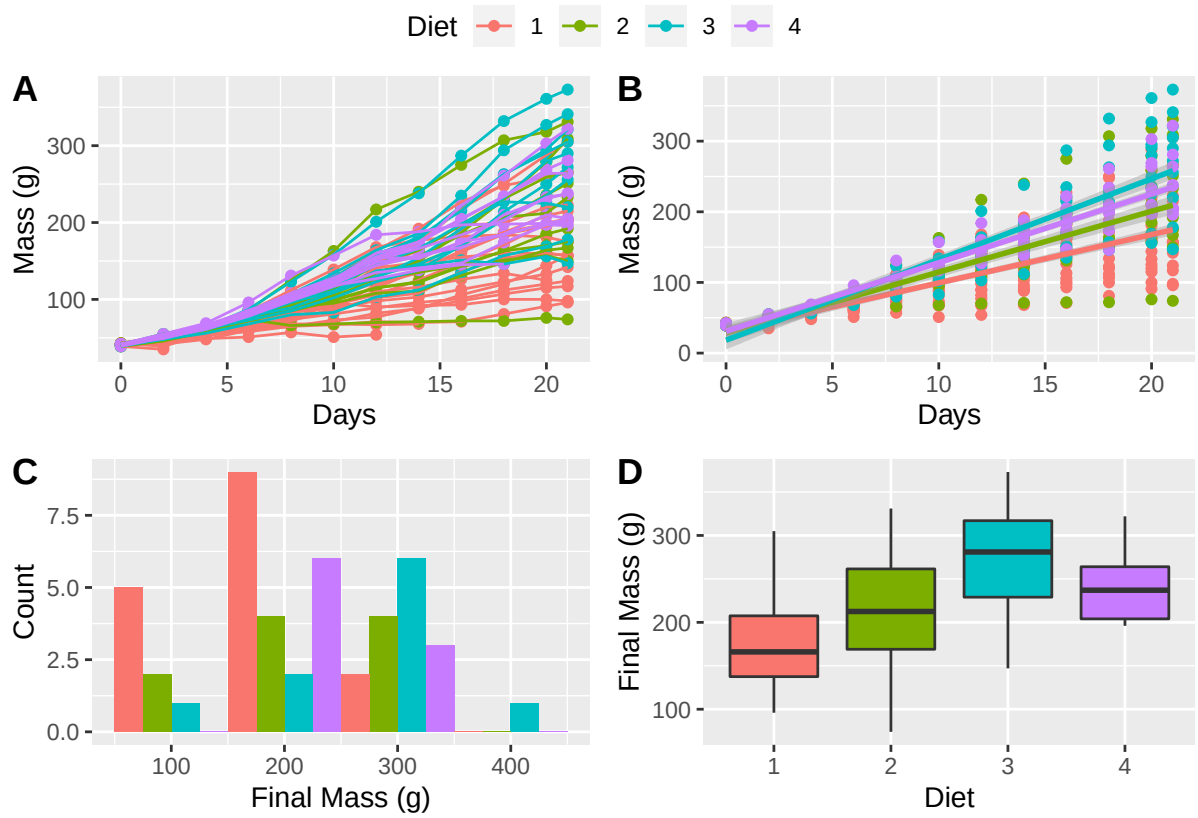


Figure 6.6: All four of our figures gridded together with an automagically created common legend.

The above figure looks great, so let's save a copy of it as a PDF to our computer. In order to do so we will need to assign our figure to an object and then use the `ggsave()` function on that object.

```
# First we must assign the code to an object name
grid_1 <- ggarrange(lm_1, histogram_1, density_1, violin_1,
                    ncol = 2, nrow = 2,
                    labels = c("A", "B", "C", "D"),
                    common.legend = TRUE)

# Then we save the object we created
ggsave(plot = grid_1, filename = "figures/grid_1.pdf")
```

6.4 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>    ggpubr  magrittr   forcats   stringr     dplyr     purrr     readr     tidyr
R>   "0.2.4"     "1.5"   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"
R>    tibble   ggplot2 tidyverse
R>   "2.1.3"   "3.2.1"   "1.3.0"
```

Chapter 7

Brewing colours in ggplot2

> "Every portrait that is painted with feeling is a portrait of the artist, not of the sitter."
>
> — Oscar Wilde
>
> "If you could say it in words, there would be no reason to paint."
>
> — Edward Hopper

Now that we have seen the basics of **ggplot2**, let's take a moment to delve further into the beauty of our figures. It may sound vain at first, but the colour palette of a figure is actually very important. This is for two main reasons. The first being that a consistent colour palette looks more professional. But most importantly it is necessary to have a good colour palette because it makes the information in our figures easier to understand. The communication of information to others is central to good science.

7.1 R Data

Before we get going on our figures, we first need to learn more about the built in data that R has. The base R program that we all have loaded on our computers already comes with heaps of example dataframes that we may use for practice. We don't need to load our own data. Additionally, whenever we install a new package (and by now we've already installed dozens) it usually comes with several new dataframes. There are many ways to look at the data that we have available from our packages. Below we show two of the many options.

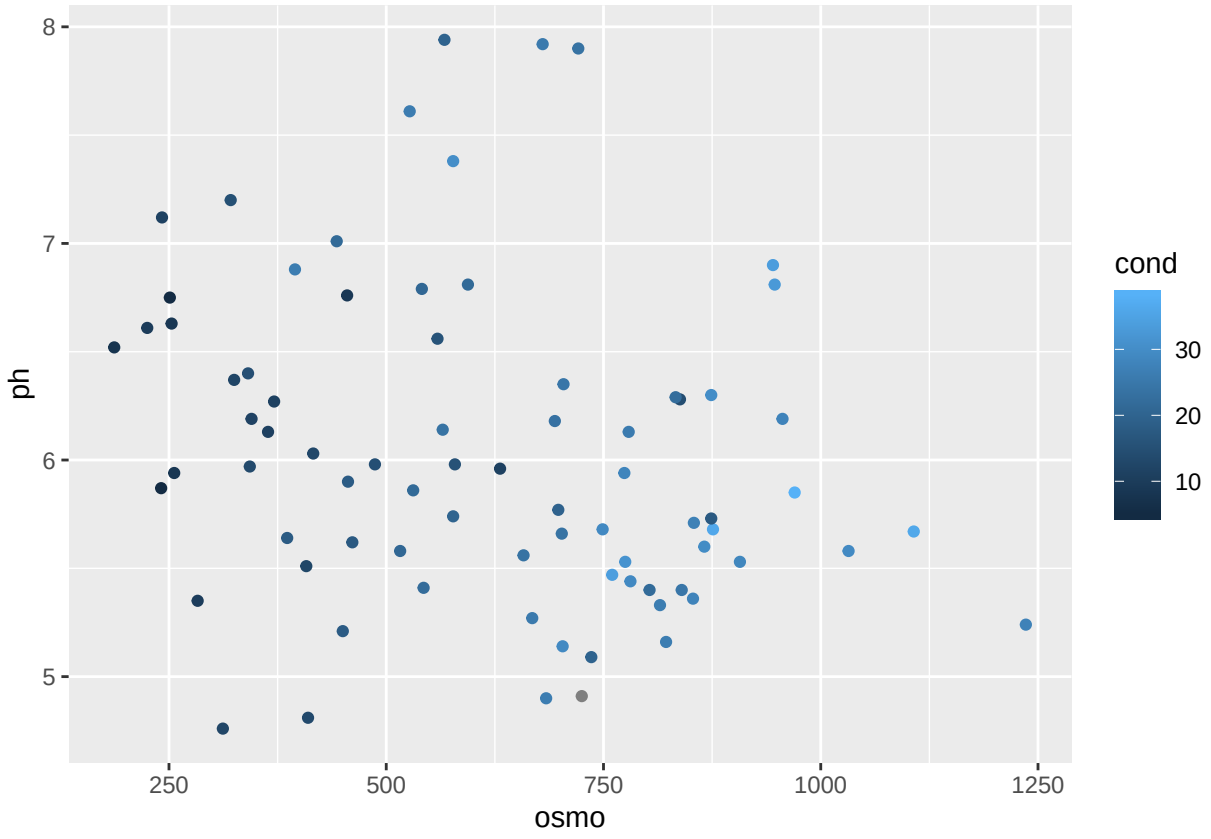
```
# To create a list of ALL available data
  # Not really recommended as the output is overwhelming
data(package = .packages(all.available = TRUE))

# To look for datasets within a single known package
  # type the name of the package followed by '::'
  # This tells R you want to look in the specified package
  # When the autocomplete bubble comes up you may scroll
  # through it with the up and down arrows
  # Look for objects that have a mini spreadsheet icon
  # These are the datasets

# Try typing the following code and see what happens ...
datasets::
```

We have an amazing amount of data available to us. So the challenge is not to find a dataframe that works for us, but to just decide on one. My preferred method is to read the short descriptions of the dataframes and pick the one that sounds the funniest. But please use whatever method makes the most sense to you. One note of caution, in R there are generally two different forms of data: wide OR long. We will see in detail what this means on Day 4, and what to do about it. For now we just need to know that **ggplot2** works much better with long data. To look at a dataframe of interest we use the same method we would use to look up a help file for a function.

Over the years I've installed so many packages on my computer that it is difficult to chose a dataframe. The package **boot** has some particularly interesting dataframes with a biological focus. Please install this now to access to these data. I have decided to load the `urine` dataframe here. Note that `library(boot)` will not work

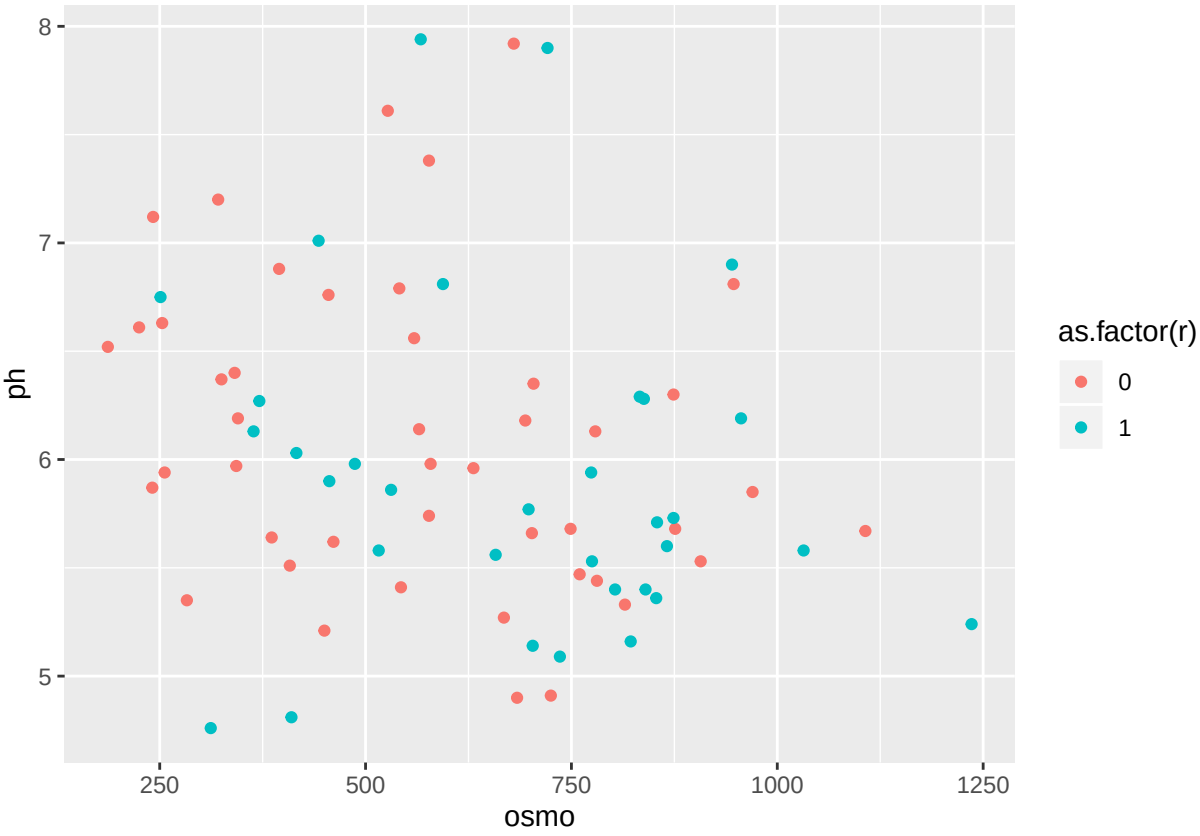
on your computer if you have not installed the package yet. With these data we will now make a scatterplot with two of the variables, while changing the colour of the dots with a third variable.

```
# Load libraries
library(tidyverse)
library(boot)

# Load data
urine <- boot::urine

# Look at help file for more info
# ?urine

# Create a quick scatterplot
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = cond))
```



And now we have a scatterplot that is showing the relationship between the osmolarity and pH of urine, with the conductivity of those urine samples shown in shades of blue. What is important to note here is that the colour scale is continuous. How can we now this by looking at the figure? Let's look at the same figure but use a discrete variable for colouring.

```
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = as.factor(r)))
```

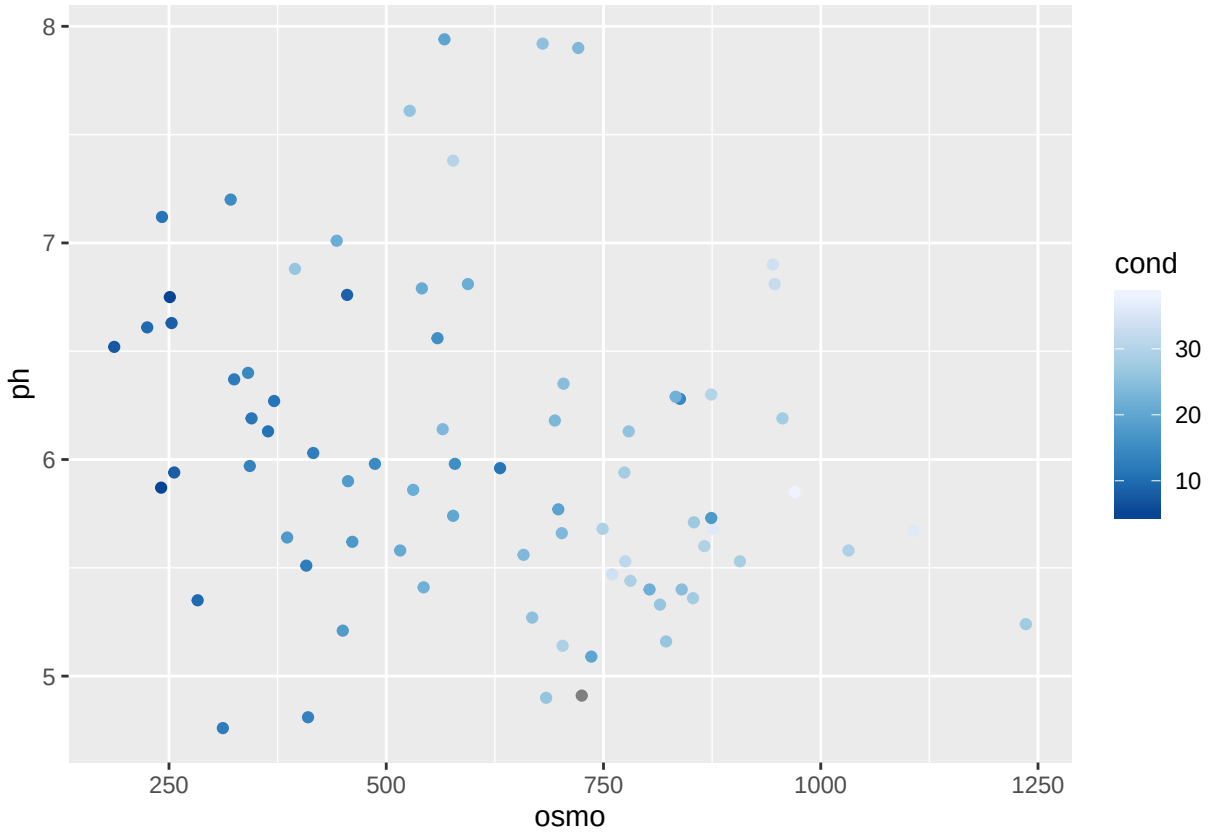


What is the first thing you notice about the difference in the colours? Why did we use `as.factor()` for the colour aesthetic for our points? What happens if we don't use this? Try it now.

7.2 RColorBrewer

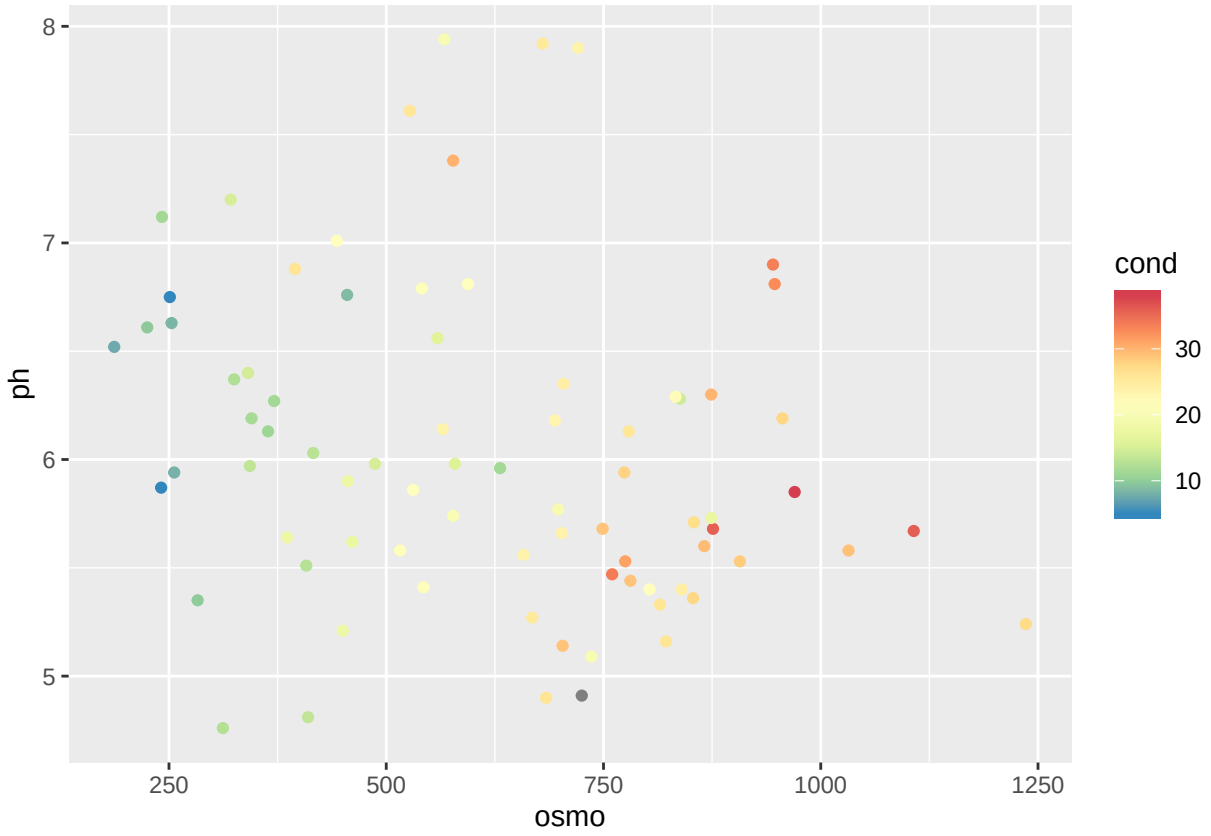
Central to the purpose of **ggplot2** is the creation of beautiful figures. For this reason there are many built in functions that we may use in order to have precise control over the colours we use, as well as additional packages that extend our options even further. The **RColorBrewer** package should have been installed on your computer and activated automatically when we installed and activated the **tidyverse**. We will use this package for its lovely colour palettes. Let's spruce up the previous continuous colour scale figure now.

```
# The continuous colour scale figure
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = cond)) +
  scale_colour_distiller() # Change the continuous variable colour palette
```



Does this look different? If so, how? The second page of the colour cheat sheet we included in the course material shows some different colour brewer palettes. Let's look at how to use those here.

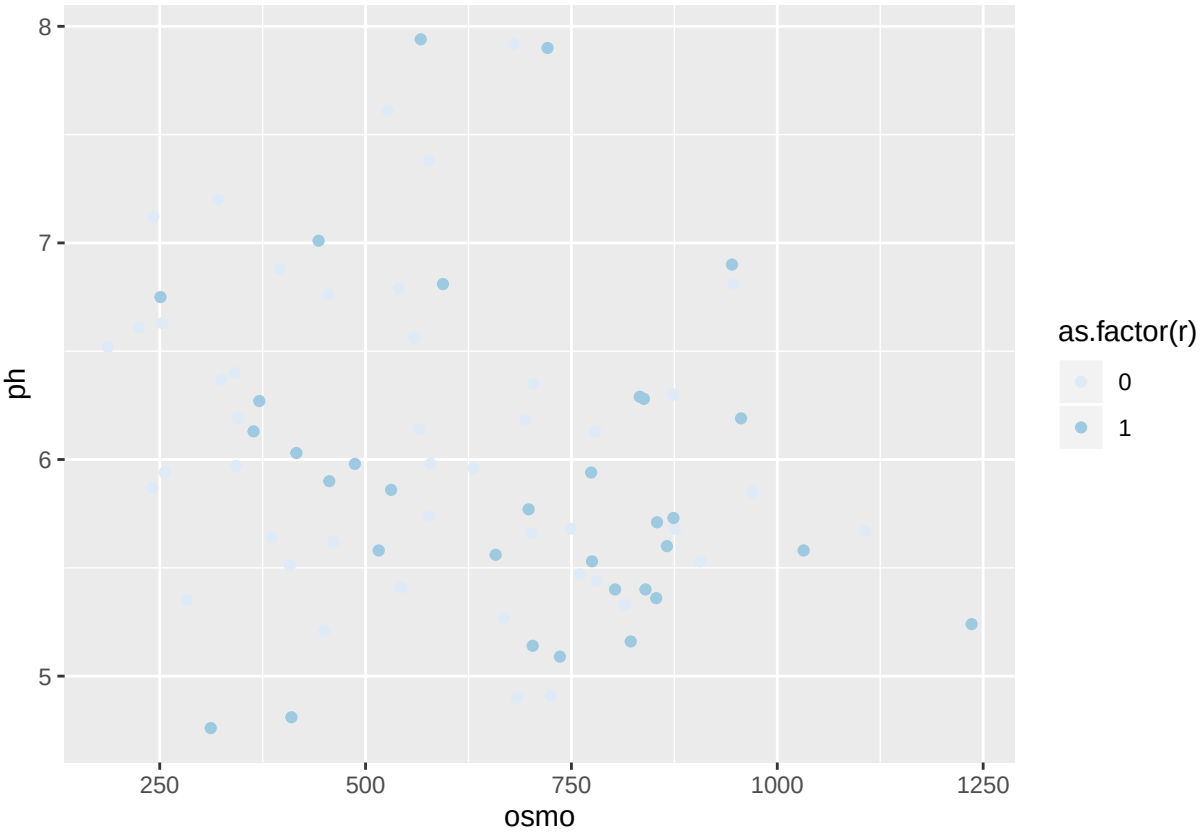
```
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = cond)) +
  scale_colour_distiller(palette = "Spectral")
```



Does that help us to see a pattern in the data? What do we see? Does it look like there are any significant relationships here? How would we test that?

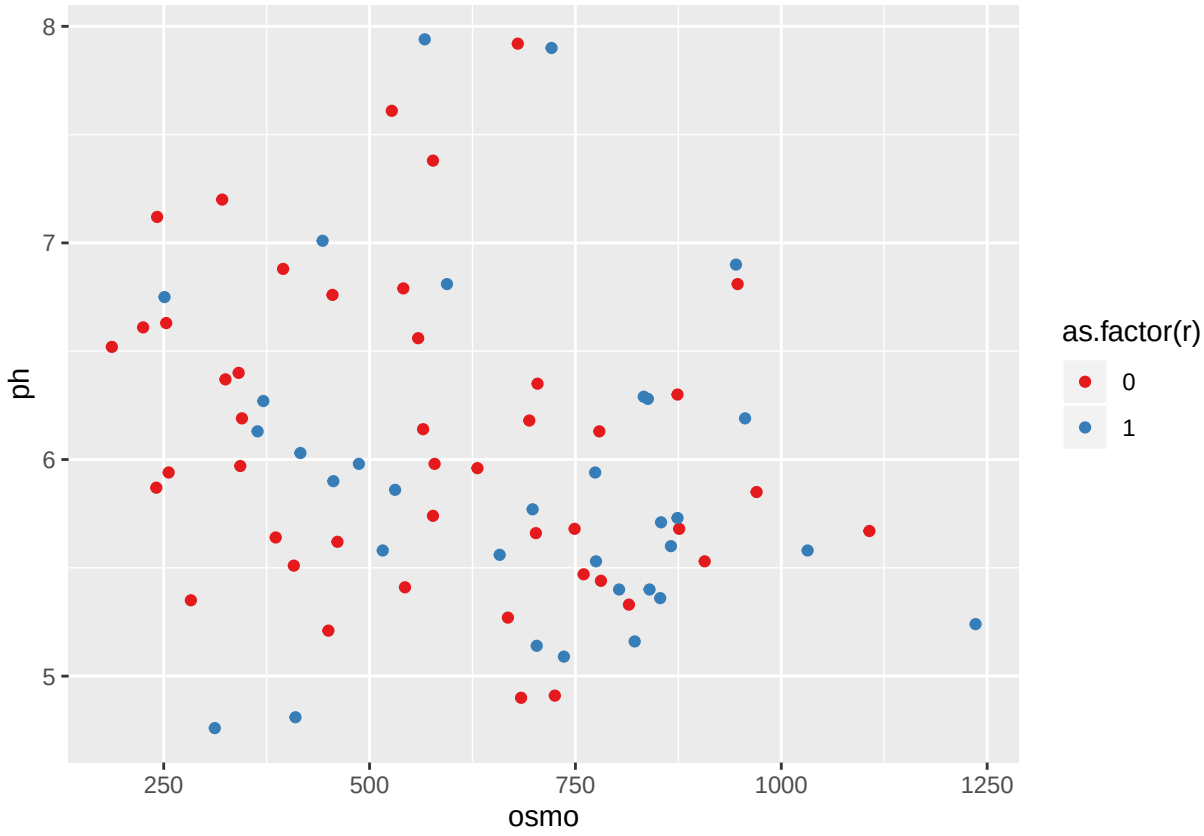
If we want to use colour brewer with a discrete variable we use a slightly different function.

```
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = as.factor(r))) +
  scale_colour_brewer() # This is the different function
```



The default colour scale here is not helpful at all. So let's pick a better one. If we look at our cheat sheet we will see a list of different continuous and discrete colour scales. All we need to do is copy and paste one of these names into our colour brewer function with inverted commas.

```
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = as.factor(r))) +
  scale_colour_brewer(palette = "Set1") # Here I used "Set1", but use what you like
```



7.3 Make your own palettes

This is all well and good. But didn't we claim that this should give us complete control over our colours? So far it looks like it has just given us a few more palettes to use. And that's nice, but it's not 'infinite choices'. That is where the Internet comes to our rescue. There are many places we may go to for support in this regard. The following links, in descending order, are very useful. And fun!

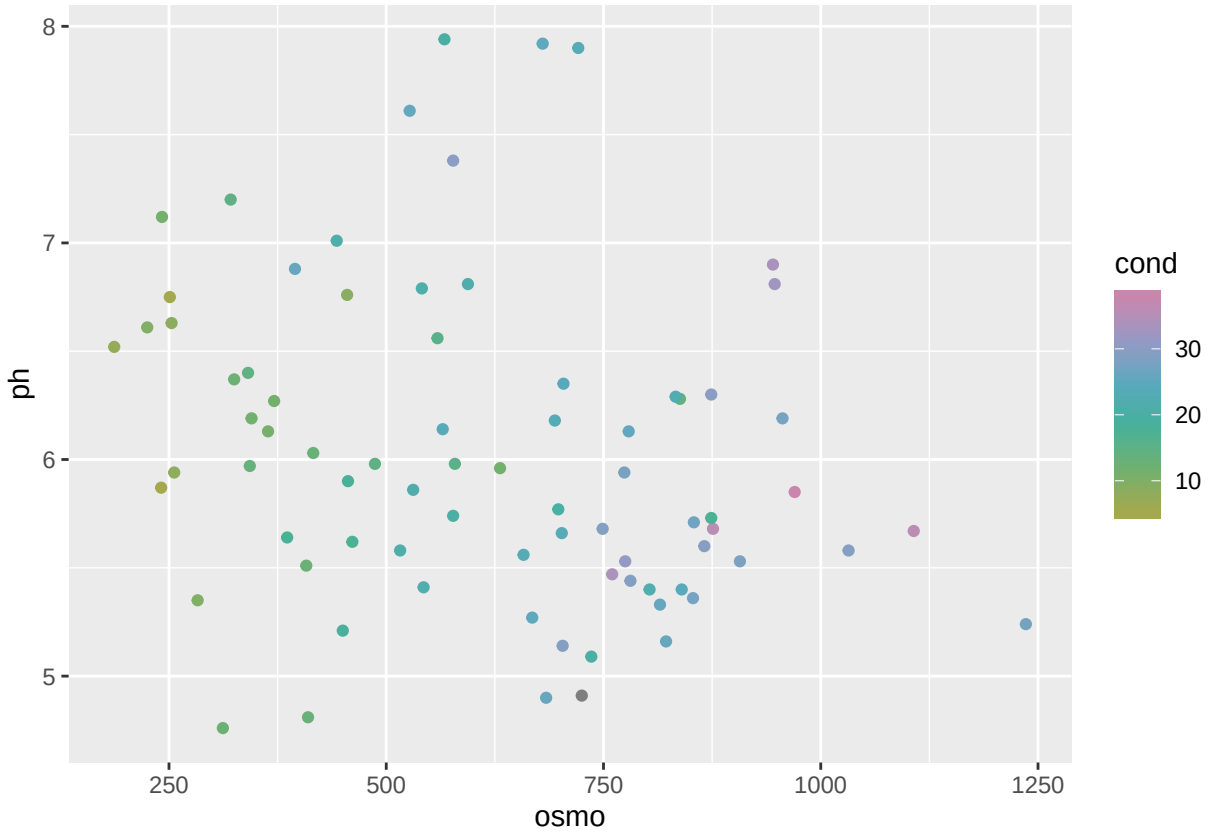
- http://tristen.ca/hcl-picker/#/hlc/6/0.95/48B4B6/345363
- http://tools.medialab.sciences-po.fr/iwanthue/index.php
- http://jsfiddle.net/d6wXV/6/embedded/result/

I find the first link the easiest to use. But the second and third links are better at generating discrete colour palettes. Take several minutes playing with the different websites and decide for yourself which one(s) you like.

7.4 Use your own palettes

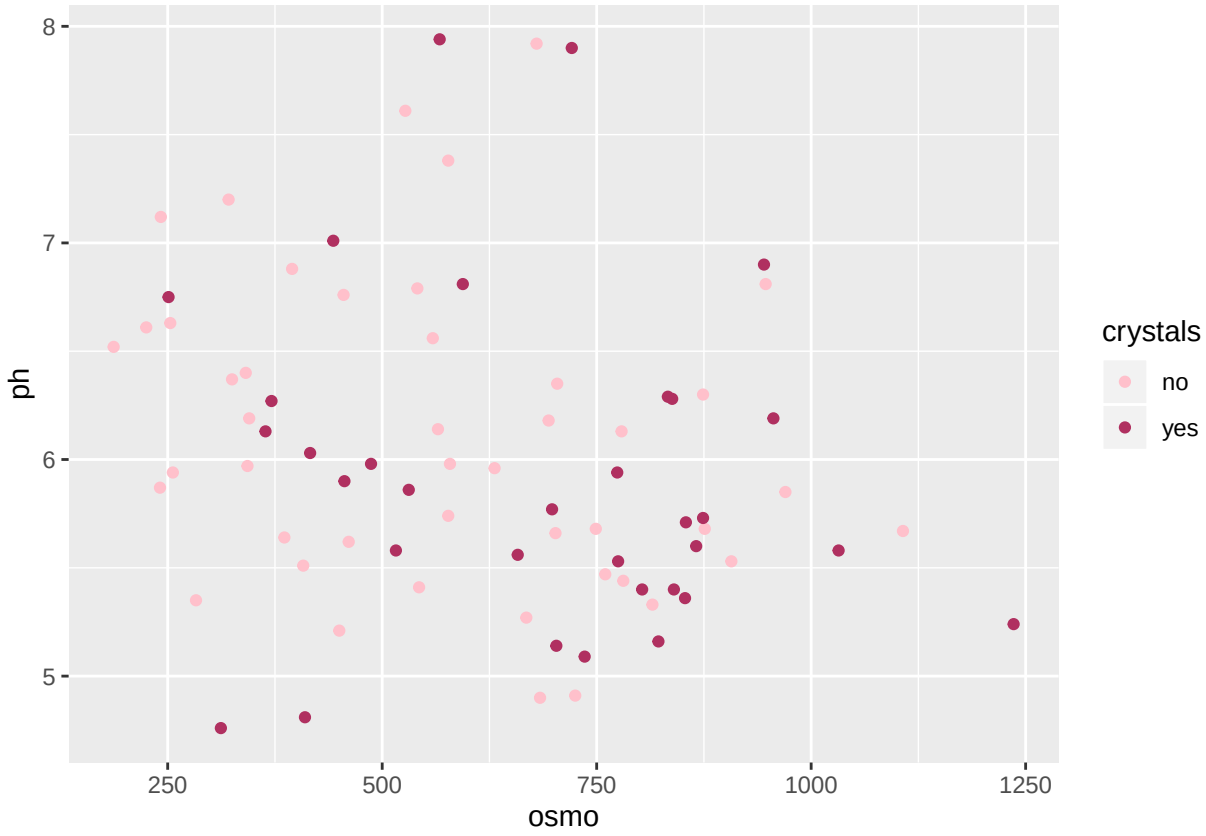
Now that we've had some time to play around with the colour generators let's look at how to use them with our figures. I've used the first web link to create a list of five colours. I then copy and pasted them into the code below, separating them with commas and placing them inside of `c()` and inverted commas. Be certain that you insert commas and inverted commas as necessary or you will get errors. Note also that we are using a new function to use our custom palette.

```
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = cond)) +
  scale_colour_gradientn(colours = c("#A5A94D", "#6FB16F", "#45B19B",
                                     "#59A9BE", "#9699C4", "#CA86AD"))
```



If we want to use our custom colour palettes with a discrete colour scale we use a different function as seen in the code below. While we are at it, let's also see how to correct the title of the legend and its text labels. Sometimes the default output is not what we want for our final figure. Especially if we are going to be publishing it. Also note in the following code chunk that rather than using hexadecimal character strings to represent colours in our custom palette, we are simply writing in the human name for the colours we want. This will work for the continuous colour palettes above, too.

```
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = as.factor(r))) +
  scale_colour_manual(values = c("pink", "maroon"), # How to use custom palette
                      labels = c("no", "yes")) + # How to change the legend text
  labs(colour = "crystals") # How to change the legend title
```



So now we have seen how to control the colours palettes in our figures. I know it is a but much. Four new functions just to change some colours! That's a bummer. Don't forget that one of the main benefits of R is that all of your code is written down, annotated and saved. You don't need to remember which button to click to change the colours, you just need to remember where you saved the code that you will need. And that's pretty great in my opinion.

7.5 DIY figures

Today we learned the basics of **ggplot2**, how to facet, how to brew colours, and how to plot stats. Sjog, that's a lot of stuff to remember! Which is why we will now spend the rest of Day 2 putting our new found skills to use. Please group up as you see fit to produce your very own **ggplot2** figures. We've not yet learned how to manipulate/tidy up our data so it may be challenging to grab any ol' dataset and make a plan with it. To that end we recommend using the `laminaria` or `ecklonia` datasets we saw on Day 1. You are of course free to use whatever dataset you would like, including your own. The goal by the end of today is to have created at least two figures (first prize for four figures) and join them together via faceting. We will be walking the room to help with any issues that may arise.

7.6 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>      boot   forcats   stringr     dplyr     purrr     readr     tidyr    tibble
R>   "1.3-24"   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"   "2.1.3"
R>   ggplot2 tidyverse
R>   "3.2.1"   "1.3.0"
```

Chapter 8

Mapping with `ggplot2`

> "There's no map to human behaviour."
>
> — Bjork
>
> "Here be dragons"
>
> — Unknown

Yesterday we learned how to create `ggplot2` figures, change their aesthetics, labels, colour palettes, and facet/arrange them. Now we are going to look at how to create maps.

Most of the work that we perform as environmental/biological scientists involves going out to a location and sampling information there. Sometimes only once, and sometimes over a period of time. All of these different sampling methods lend themselves to different types of figures. One of those, collection of data at different points, is best shown with maps. As we will see over the course of Day 3, creating maps in `ggplot2` is very straight forward and is extensively supported. For that reason we are going to have plenty of time to also learn how to do some more advanced things. Our goal in this chapter is to produce the figure below.

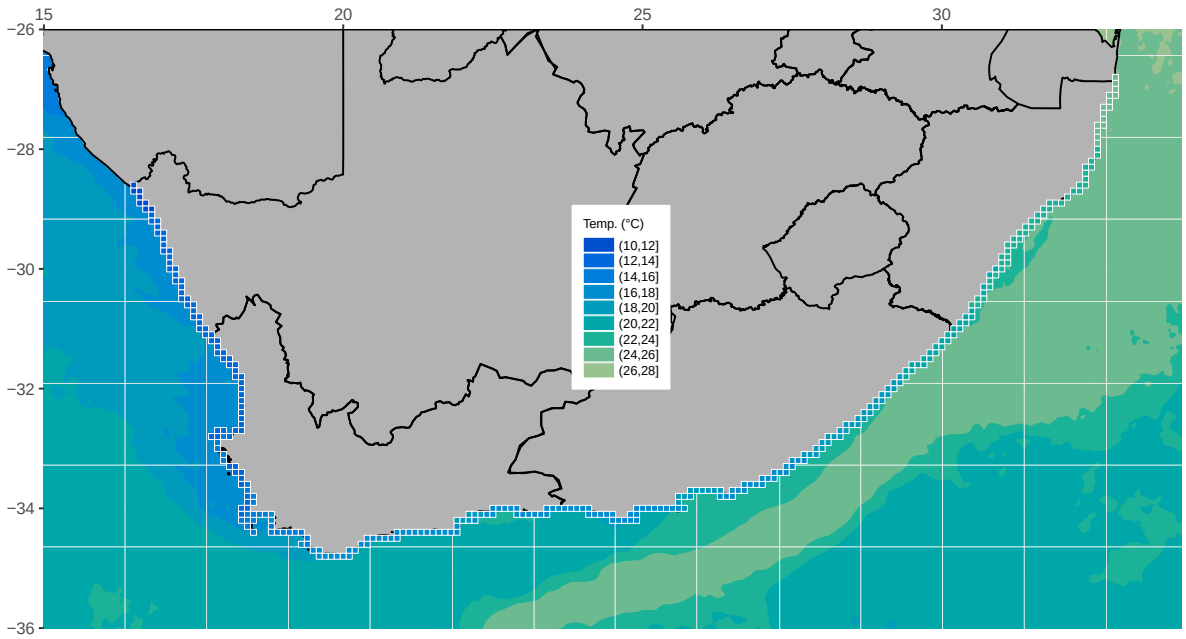


Figure 8.1: The goal for today.

Before we begin let's go ahead and load the packages we will need, as well as the several dataframes required to make the final product.

```
# Load libraries
library(tidyverse)
library(ggpubr)

# Load data
load("data/south_africa_coast.RData")
load("data/sa_provinces.RData")
load("data/rast_annual.RData")
load("data/MUR.RData")
load("data/MUR_low_res.RData")

# Choose which SST product you would like to use
sst <- MUR_low_res
# OR
sst <- MUR

# The colour pallette we will use for ocean temperature
cols11 <- c("#004dcd", "#0068db", "#007ddb", "#008dcf", "#009bbc",
            "#00a7a9", "#1bb298", "#6cba8f", "#9ac290", "#bec99a")
```

8.1 A new concept?

The idea of creating a map in R may be daunting to some, but remember that a basic map is nothing more than a simple figure with an x and y axis. We tend to think of maps as different from other scientific figures, whereas in reality they are created the exact same way. Let's compare a dot plot of the chicken weight data against a dot plot of the coastline of South Africa.

Chicken dots:

```
ggplot(data = ChickWeight, aes(x = Time, y = weight)) +
  geom_point()
```

South African coast dots:

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_point()
```

Does that look familiar? Notice how the x and y axis tick labels look the same as any map you would see in an atlas. This is because they are. But this isn't a great way to create a map. Rather it is better to represent the land mass with a polygon. With `ggplot2` this is a simple task.

8.2 Land mask

Now that we have seen that a map is nothing more than a bunch of dots and shapes on specific points along the x and y axes we are going to look at the steps we would take to build a more complex map. Don't worry if this seems daunting at first. We are going to take this step by step and ensure that each step is made clear along the way. The first step is to create a polygon. Note that we create an aesthetic argument inside of `geom_polygon()` and not `ggplot()` because some of the steps we will take later on will not accept the `group` aesthetic. Remember, whatever aesthetic arguments we put inside of `ggplot()` will be inserted for us into all of our other `geom_...()` lines of code.

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_polygon(colour = "black", fill = "grey70", aes(group = group)) # The land mask
```

8.3 Borders

The first thing we will add is the province borders as seen in Figure 8.1. Notice how we only add one more line of code to do this.

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_polygon(colour = "black", fill = "grey70", aes(group = group)) +
```

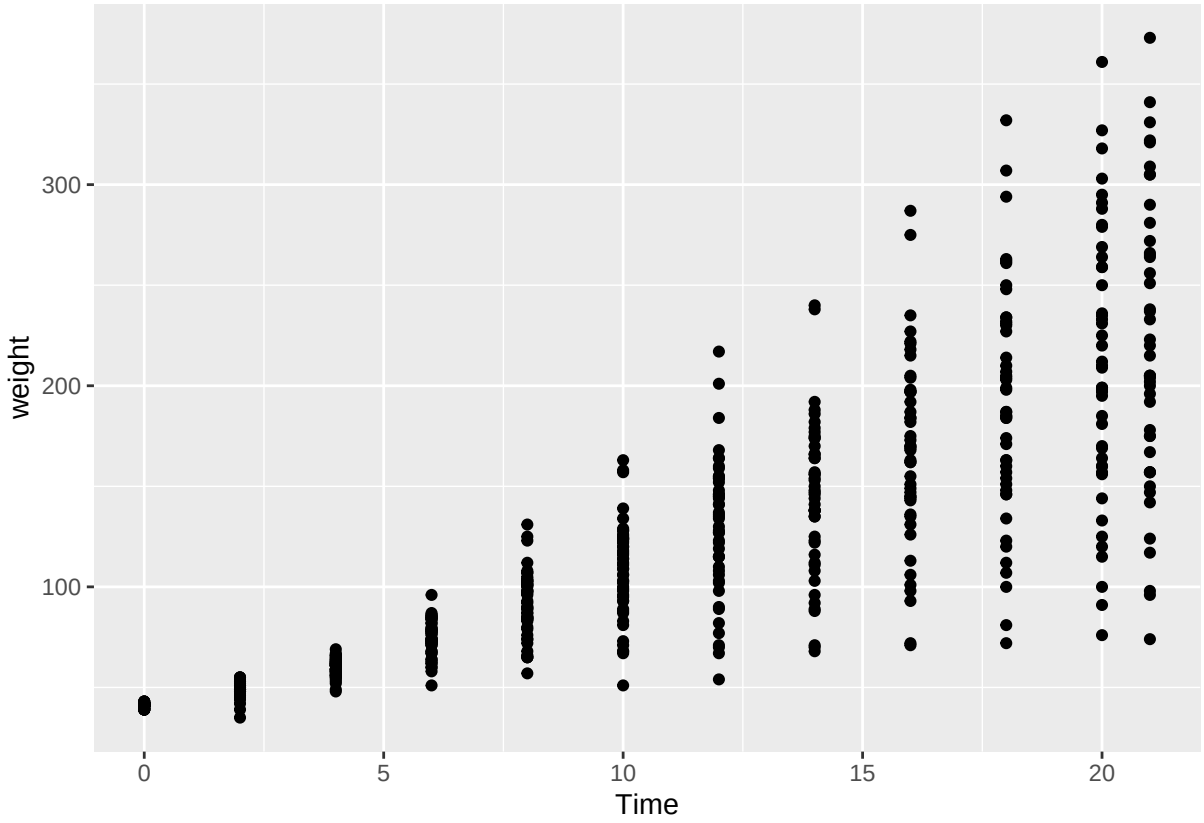


Figure 8.2: Dot plot of chicken weight data.

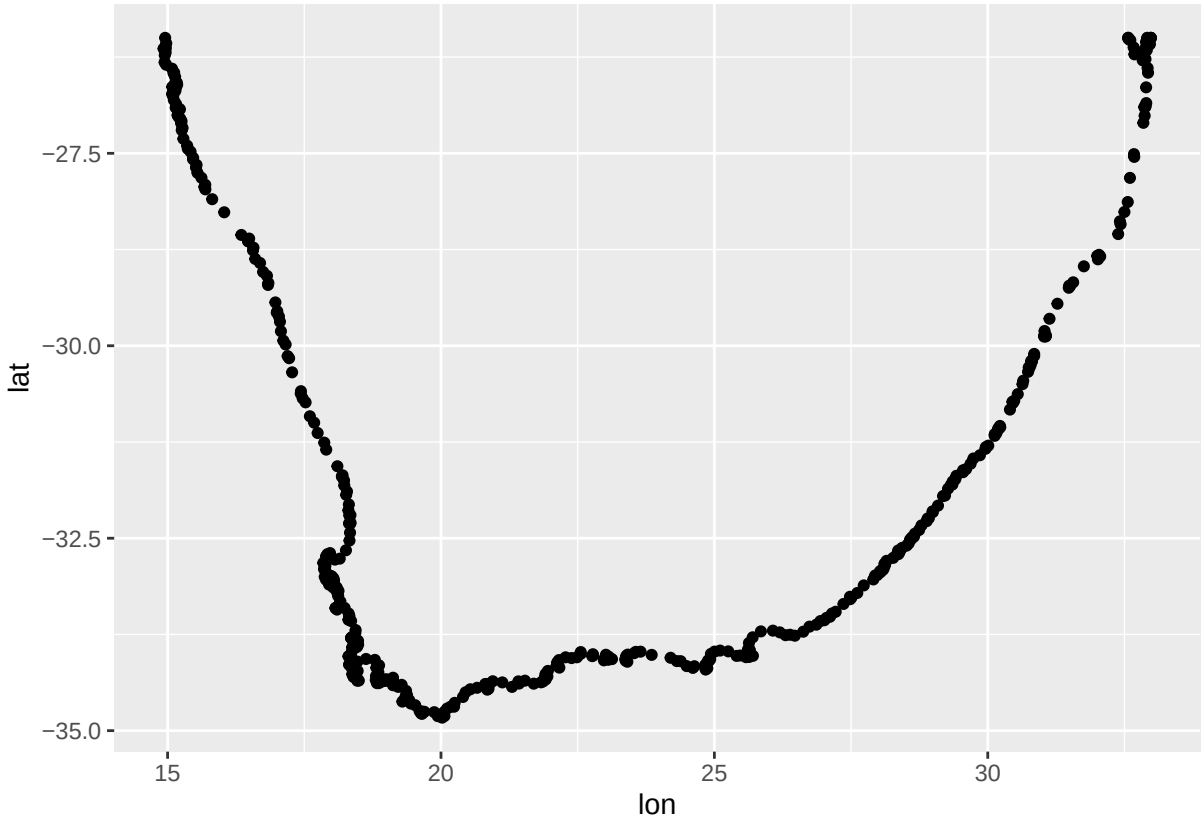


Figure 8.3: Dot plot off South African coast.

Figure 8.4: The map of South Africa. Now with province borders!

```
  geom_path(data = sa_provinces, aes(group = group)) # The province borders
```

8.4 Force lon/lat extent

Unfortunately when we added our borders it increased the plotting area of our map past what we would like. To correct that we will need to explicitly state the borders we want.

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_polygon(colour = "black", fill = "grey70", aes(group = group)) +
  geom_path(data = sa_provinces, aes(group = group)) +
  coord_equal(xlim = c(15, 34), ylim = c(-36, -26), expand = 0) # Force lon/lat extent
```

8.5 Ocean temperature

This is starting to look pretty fancy, but it would be nicer if there was some colour involved. So let's add the ocean temperature. Again, this will only require one more line of code. Starting to see a pattern? But what is different this time and why?

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_raster(data = sst, aes(fill = bins)) + # The ocean temperatures
  geom_polygon(colour = "black", fill = "grey70", aes(group = group)) +
  geom_path(data = sa_provinces, aes(group = group)) +
  coord_equal(xlim = c(15, 34), ylim = c(-36, -26), expand = 0)
```

That looks… odd. Why do the colours look like someone melted a big bucket of ice cream in the ocean? This is because the colours you see in this figure are the default colours for discrete values in **ggplot2**. If we want to change them we may do so easily by adding yet one more line of code.

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_raster(data = sst, aes(fill = bins)) +
  geom_polygon(colour = "black", fill = "grey70", aes(group = group)) +
  geom_path(data = sa_provinces, aes(group = group)) +
```

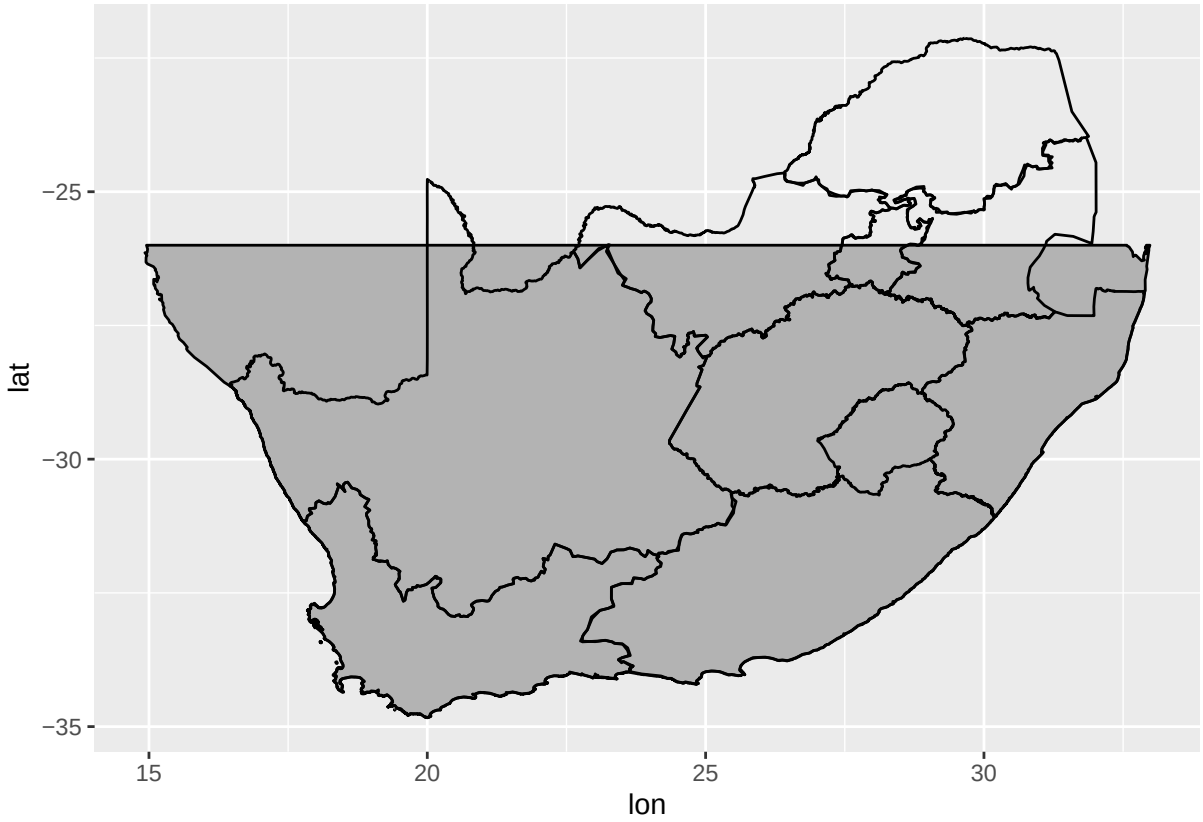


Figure 8.5: The map of South Africa. Now with province borders!

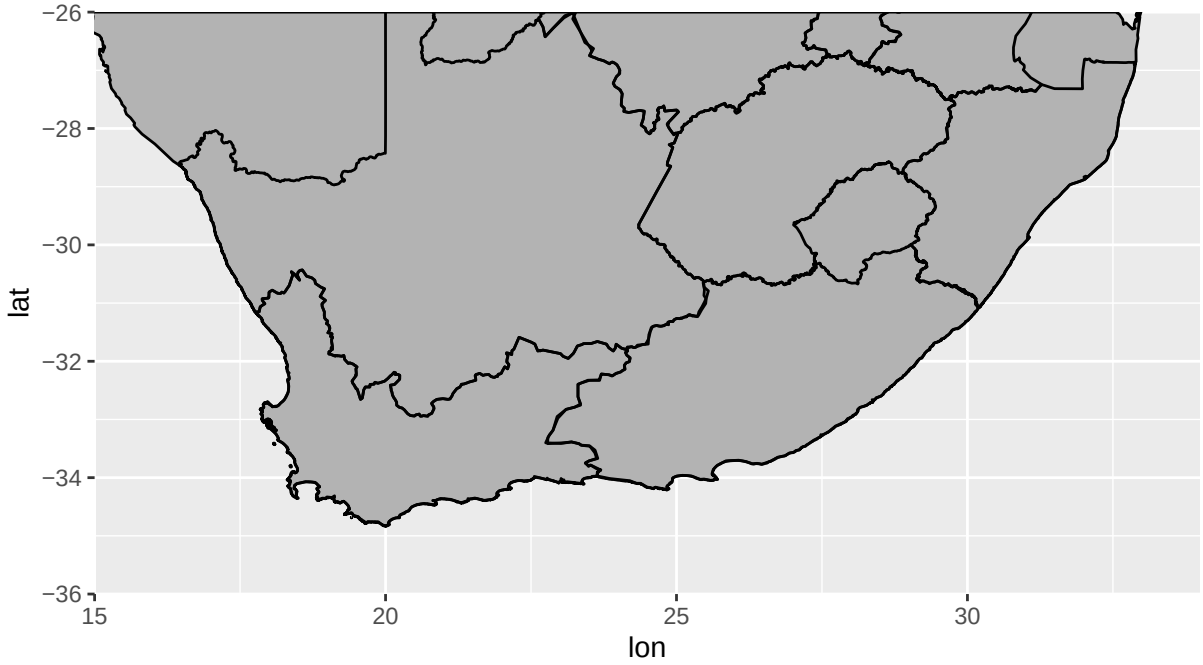


Figure 8.6: The map, but with the extra bits snipped off.

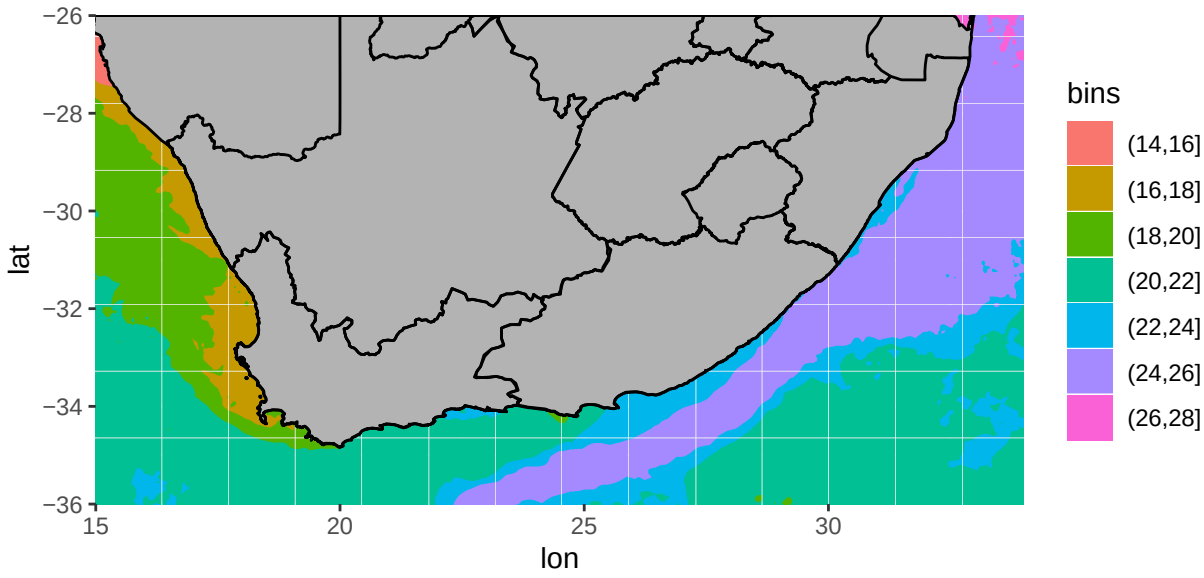


Figure 8.7: Ocean temperature (°C) visualised as an ice cream spill.

```
  scale_fill_manual("Temp. (°C)", values = cols11) + # Set the colour palette
  coord_equal(xlim = c(15, 34), ylim = c(-36, -26), expand = 0)
```

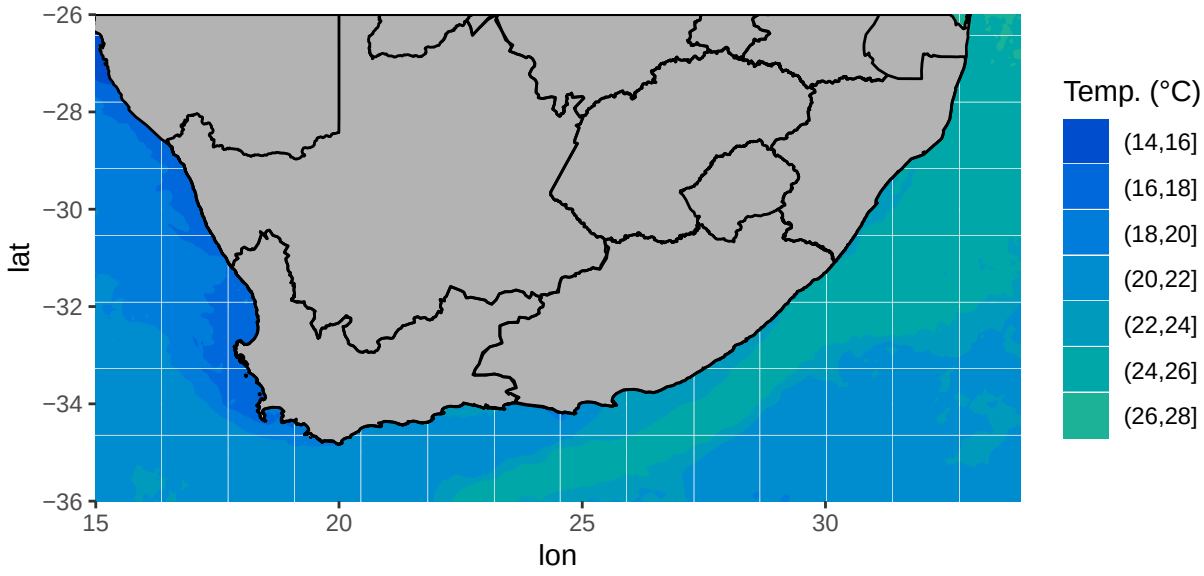


Figure 8.8: Ocean temperatures (°C) around South Africa.

There's a colour palette that would make Jacques Cousteau swoon. When we set the colour palette for a figure in **ggplot2** we must use that colour palette for all other instances of those types of values, too. What this means is that any other discrete values that will be filled in, like the ocean colour above, must use the same colour palette (there are some technical exceptions to this rule that we will not cover in this course). We normally want **ggplot2** to use consistent colour palettes anyway, but it is important to note that this constraint exists. Let's see what we mean. Next we will add the coastal pixels to our figure with one more line of code. We won't change anything else. Note how **ggplot2** changes the colour of the coastal pixels to match the ocean colour automatically.

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_raster(data = sst, aes(fill = bins)) +
  geom_polygon(colour = "black", fill = "grey70", aes(group = group)) +
  geom_path(data = sa_provinces, aes(group = group)) +
  geom_tile(data = rast_annual, aes(x = lon, y = lat, fill = bins),
```

```
            colour = "white", size = 0.1) + # The coastal temperature values
  scale_fill_manual("Temp. (°C)", values = cols11) +
  coord_equal(xlim = c(15, 34), ylim = c(-36, -26), expand = 0)
```

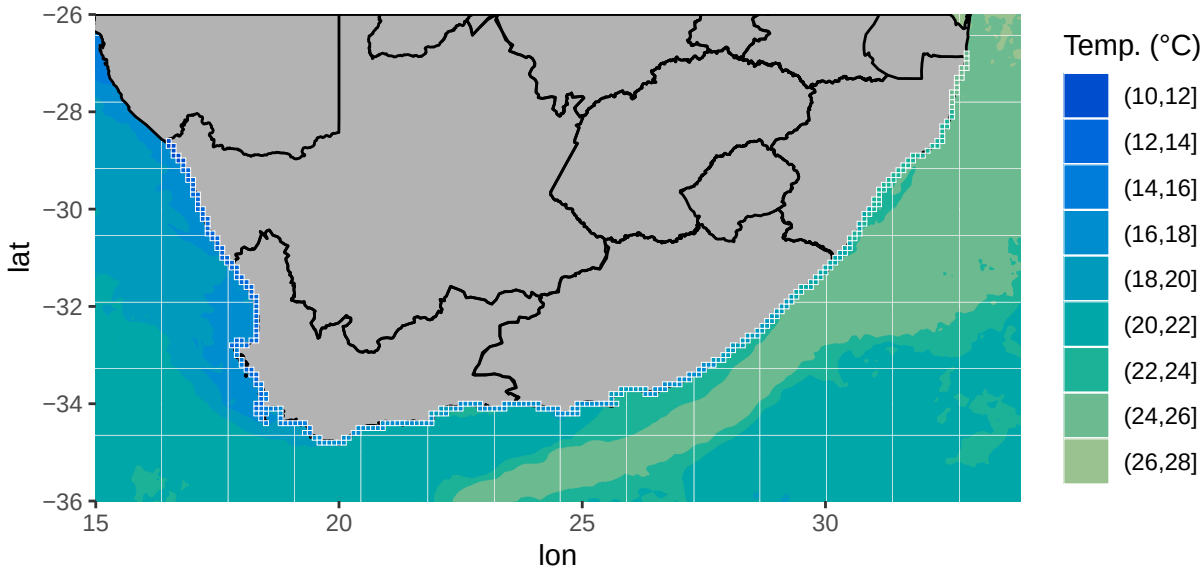


Figure 8.9: Map of SOuth Africa showing *in situ* temeperatures (°C) as pixels along the coast.

8.6 Final touches

We used `geom_tile()` instead of `geom_rast()` to add the coastal pixels above so that we could add those little white boxes around them. This figure is looking pretty great now. And it only took a few rows of code to put it all together! The last step is to add several more lines of code that will control for all of the little things we want to change about the appearance of the figure. Each little thing that is changed below is annotated for your convenience.

```
final_map <- ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_raster(data = sst, aes(fill = bins)) +
  geom_polygon(colour = "black", fill = "grey70", aes(group = group)) +
  geom_path(data = sa_provinces, aes(group = group)) +
  geom_tile(data = rast_annual, aes(x = lon, y = lat, fill = bins),
            colour = "white", size = 0.1) +
  scale_fill_manual("Temp. (°C)", values = cols11) +
  coord_equal(xlim = c(15, 34), ylim = c(-36, -26), expand = 0) +
  scale_x_continuous(position = "top") + # Put x axis labels on top of figure
  theme(axis.title = element_blank(), # Remove the axis labels
        legend.text = element_text(size = 7), # Change text size in legend
        legend.title = element_text(size = 7), # Change legend title text size
        legend.key.height = unit(0.3, "cm"), # Change size of legend
        legend.background = element_rect(colour = "white"), # Add legend background
        legend.justification = c(1, 0), # Change position of legend
        legend.position = c(0.55, 0.4) # Fine tune position of legend
        )
final_map
```

That is a very clean looking map so let's go ahead and save it on our computers

```
ggsave(plot = final_map, "figures/map_complete.pdf", height = 6, width = 9)
```

8.7 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>    ggpubr  magrittr   forcats   stringr     dplyr     purrr     readr     tidyr
```

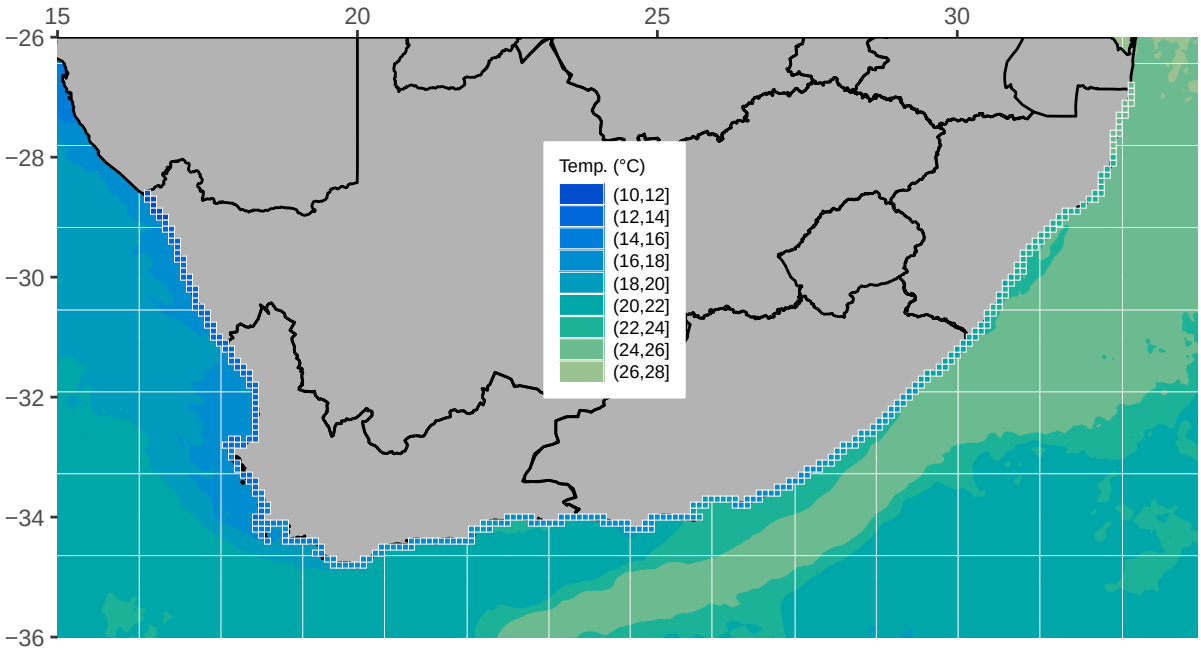


Figure 8.10: The cleaned up map of South Africa. Resplendent with coastal and ocean temperatures (°C).

```
R>   "0.2.4"     "1.5"   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"
R>    tibble   ggplot2 tidyverse
R>   "2.1.3"   "3.2.1"   "1.3.0"
```

Chapter 9

Mapping with style

> "How beautiful the world was when one looked at it without searching, just looked, simply and innocently."
>
> — Hermann Hesse, Siddartha
>
> "You can't judge a book by it's cover but you can sure sell a bunch of books if you have a good one."
>
> — Jayce O'Neal

Now that we have learned the basics of creating a beautiful map in **ggplot2** it is time to look at some of the more particular things we will need to make our maps extra stylish. There are also a few more things we need to learn how to do before our maps can be truly publication quality.

If we have not yet loaded the **tidyverse** let's do so.

```
# Load libraries
library(tidyverse)
library(scales)
library(ggsn)

# Load Africa map
load("data/africa_map.RData")
```

9.1 Default maps

In order to access the default maps included with the **tidyverse** we will use the function `borders()`.

```
ggplot() +
  borders() + # The global shape file
  coord_equal() # Equal sizing for lon/lat
```

Jikes! It's as simple as that to load a map of the whole planet. Usually we are not going to want to make a map of the entire planet, so let's see how to focus on just the area around South Africa.

```
sa_1 <- ggplot() +
  borders(fill = "grey70", colour = "black") +
  coord_equal(xlim = c(12, 36), ylim = c(-38, -22), expand = 0) # Force lon/lat extent
sa_1
```

That is a very tidy looking map of South(ern) Africa without needing to load any files.

9.2 Specific labels

A map is almost always going to need some labels and other visual cues. We saw in the previous section how to add site labels. The following code chunk shows how this differs if we want to add just one label at a time. This can be useful if each label needs to be different from all other labels for whatever reason. We may also see that the text labels we are creating have `\n` in them. When R sees these two characters together like this it reads this as an instruction to return down a line. Let's run the code to make sure we see what this means.

```
sa_2 <- sa_1 +
  annotate("text", label = "Atlantic\nOcean",
```

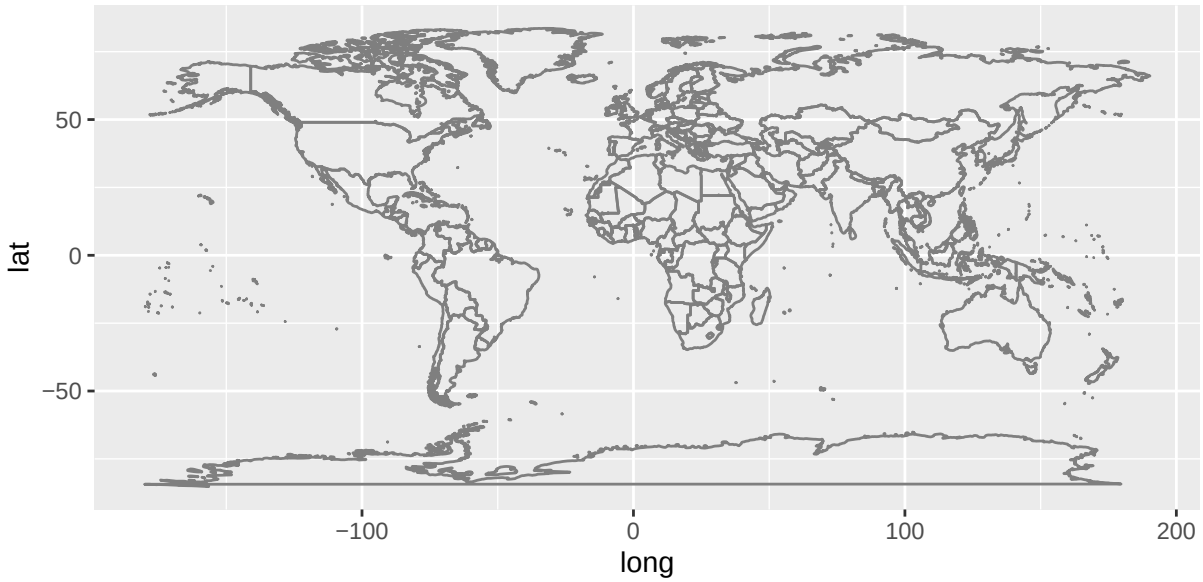


Figure 9.1: The built in global shape file.

Figure 9.2: A better way to get the map of South Africa.

```
           x = 15.1, y = -32.0,
           size = 5.0,
           angle = 30,
           colour = "navy") +
  annotate("text", label = "Indian\nOcean",
           x = 33.2, y = -34.2,
           size = 5.0,
           angle = 330,
           colour = "springgreen")
sa_2
```

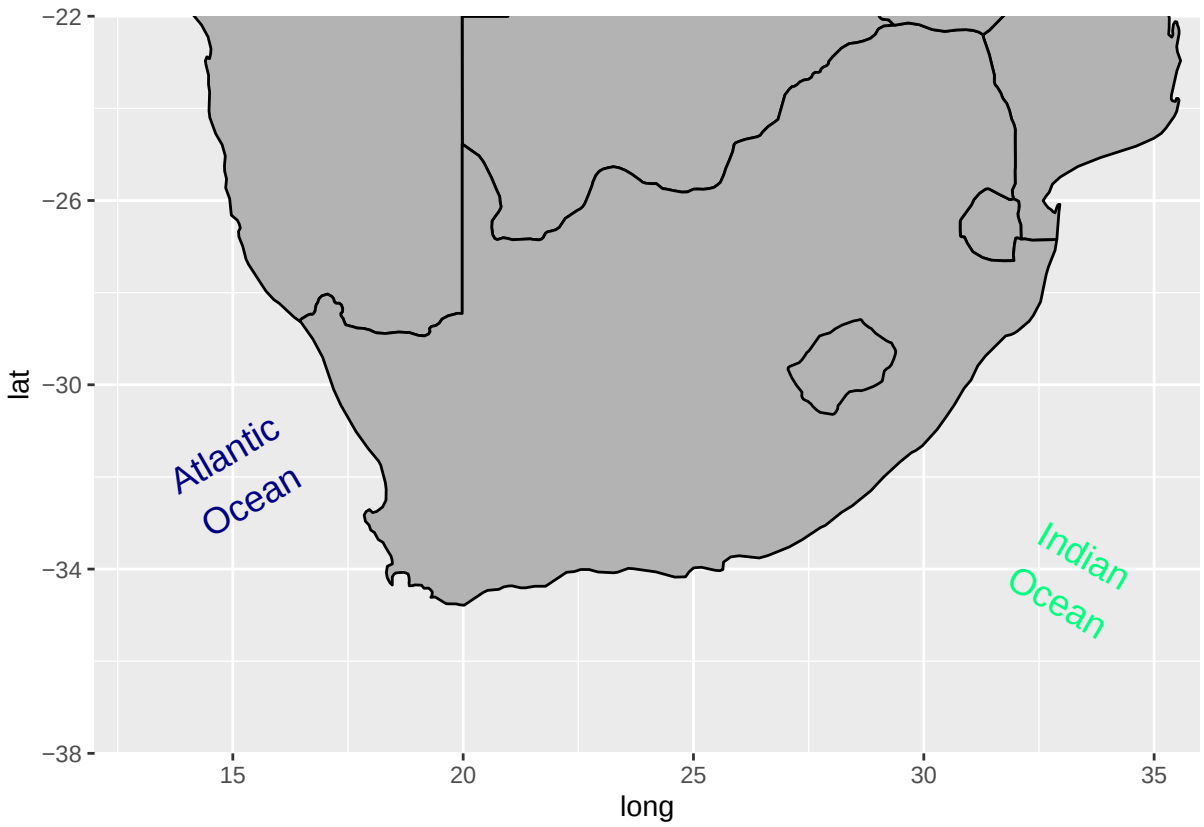


Figure 9.3: Map of southern Africa with specific labels.

9.3 Scale bars

With our fancy labels added, let's insert a scale bar next. There is no default scale bar function in the **tidyverse**, which is why we have loaded the **ggsn** package. This package is devoted to adding scale bars and North arrows to **ggplot2** figures. There are heaps of options so we'll just focus on one of them for now. It is a bit finicky so to get it looking exactly how we want it requires some guessing and checking. Please feel free to play around with the coordinates below. We may see the list of available North arrow shapes by running `northSymbols()`.

```
sa_3 <- sa_2 +
  # scalebar(x.min = 22, x.max = 26, y.min = -36, y.max = -35, # Set location of bar
  #          dist = 200, height = 1, st.dist = 0.8, st.size = 4, # Set particulars
  #          transform = TRUE, model = "WGS84") + # Set appearance
  north(x.min = 22.5, x.max = 25.5, y.min = -33, y.max = -31, # Set location of symbol
        scale = 1.2, symbol = 16)
sa_3
```

9.4 Insetting

In order to inset a smaller map inside of a larger map we must first create the smaller map. We have already loaded just such a map of Africa so we will use that for this example.

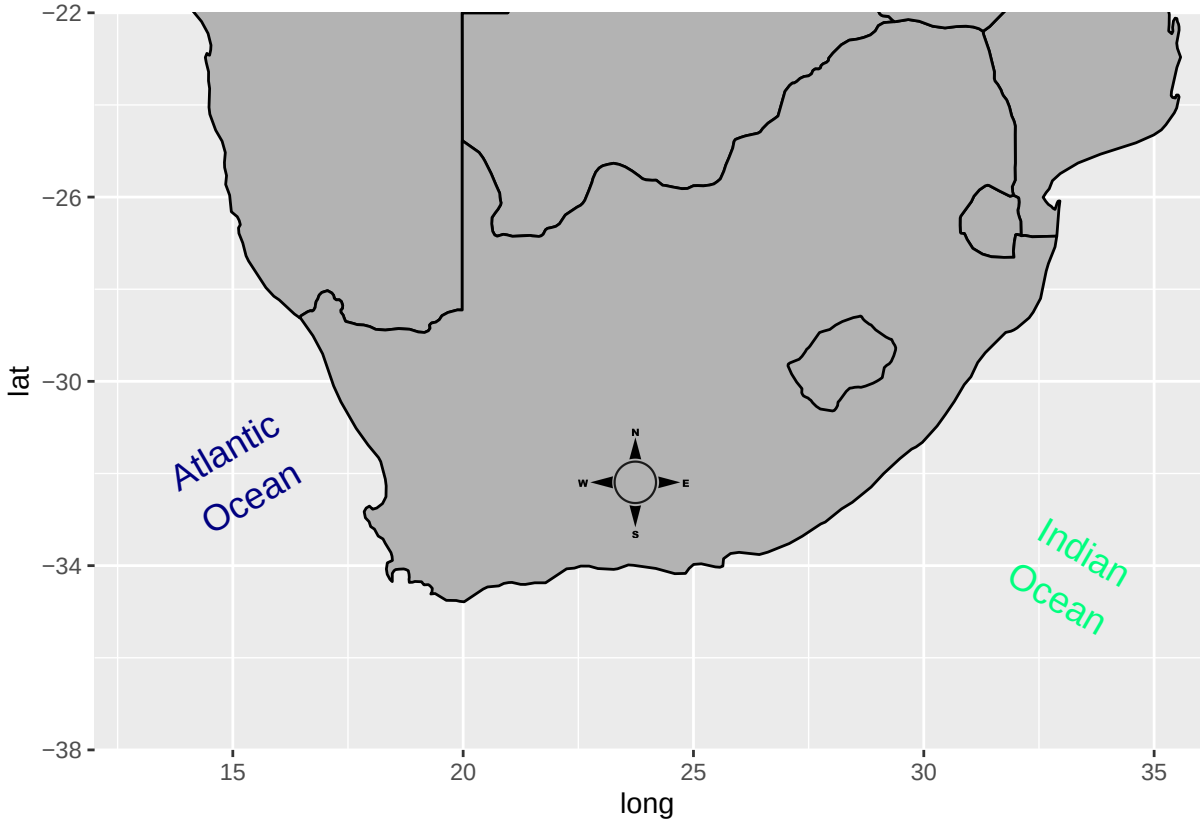
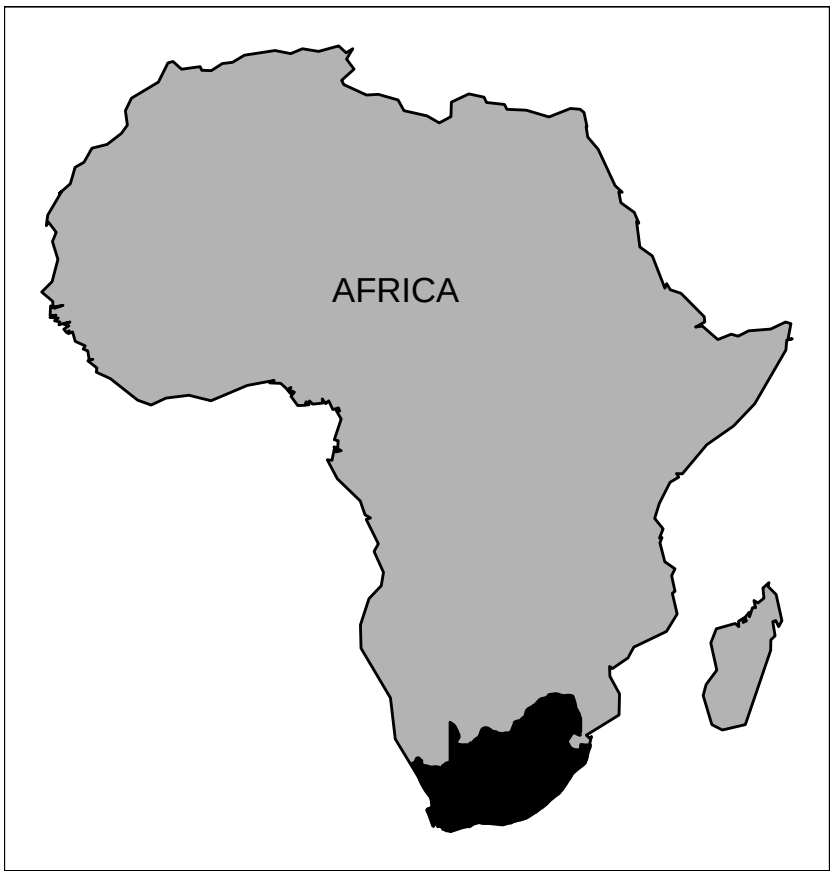


Figure 9.4: Map of southern Africa with labels and a scale bar.

```
africa_map
```



And now to inset this map of Africa into our map of southern Africa we will need to learn how to create a 'grob'. This is very simple and does not require any extra work on our part. Remember that **ggplot2** objects are different from normal objects (i.e. dataframes), and that they have their own way of storing and accessing data. In order to convert any sort of thing into a format that ggplot understands we convert it into a grob, as shown below. Once converted, we may then plop it onto our figure/map wherever we please. Both of these steps are accomplished with the single function `annotation_custom()`. This is also a good way to add logos or any other sort of image to a map/figure. You can really go completely bananas. It's even possible to add GIFs. Such happy. Much excite. Very wonderment.

```
sa_4 <- sa_3 +
  annotation_custom(grob = ggplotGrob(africa_map),
                    xmin = 20.9, xmax = 26.9,
                    ymin = -30, ymax = -24)
sa_4
```

9.5 Rounding it out

There are a lot of exciting things going on in our figure now. To round out our adventures in mapping let's tweak the lon/lat labels to a more prestigious convention. There are two ways to do this. One of which requires us to install the **scales** package. Don't worry, it's a small one!

```
sa_final <- sa_4 +
  scale_x_continuous(breaks = seq(16, 32, 4),
                     labels = c("16°E", "20°E", "24°E", "28°E", "32°E"),
                     position = "bottom") +
  scale_y_continuous(breaks = seq(-36, -24, 4),
                     labels = c("36.0°S", "32.0°S", "28.0°S", "24.0°S"),
                     position = "right") +
  labs(x = "", y = "")
sa_final
```

And lastly we save the fruits of our labours.

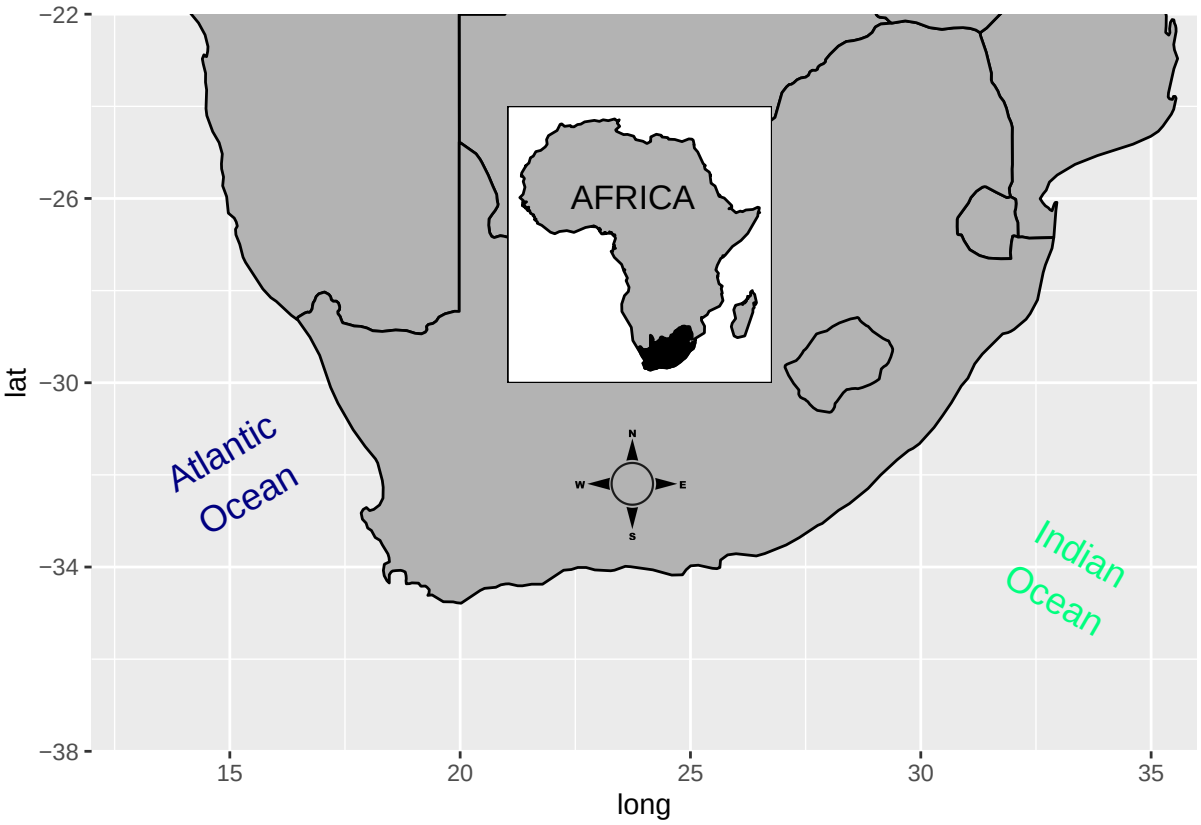


Figure 9.5: Map of southern Africa, with labels, scale bar, and an inset map of Africa.

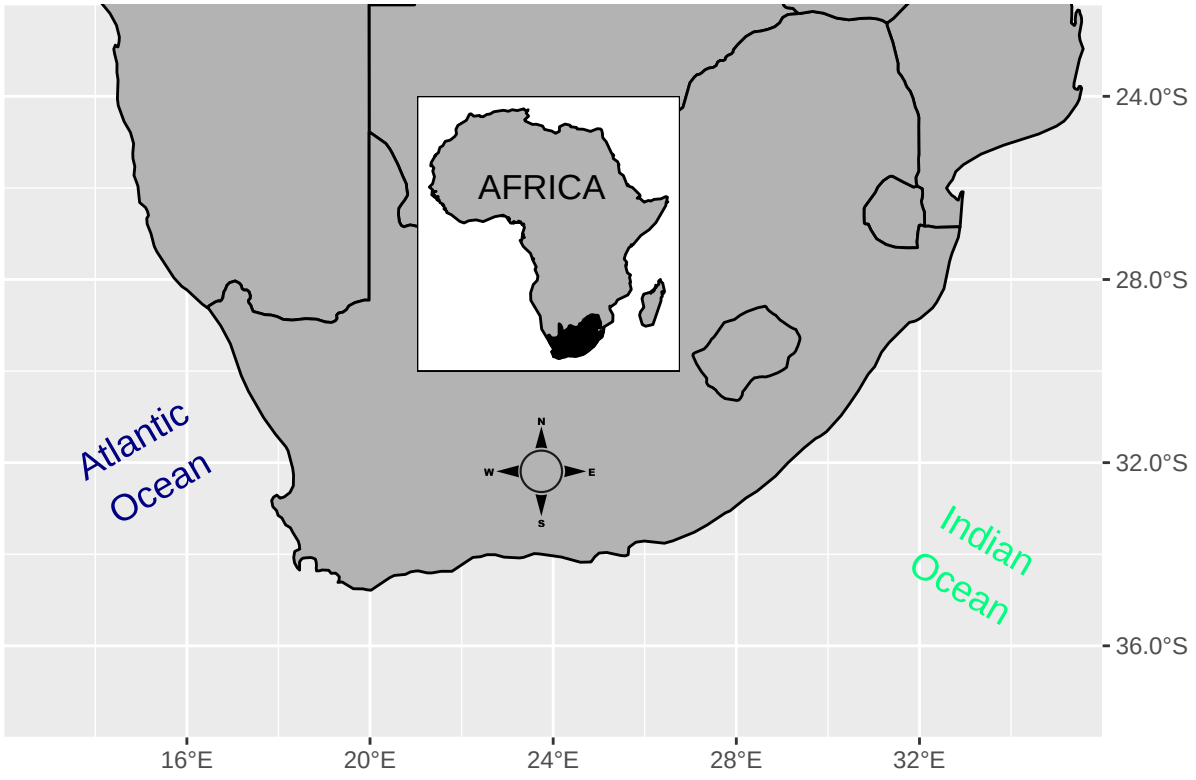


Figure 9.6: The final map with all of the bells and whistles.

```
ggsave(plot = sa_final, filename = "figures/southern_africa_final.pdf",
       height = 6, width = 8)
```

9.6 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>      ggsn    scales   forcats   stringr     dplyr     purrr     readr     tidyr
R>   "0.5.0"   "1.1.0"   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"
R>    tibble   ggplot2 tidyverse
R>   "2.1.3"   "3.2.1"   "1.3.0"
```

Chapter 10

Mapping with Google

> "The only thing Google has failed to do, so far, is fail."
>
> — John Battelle
>
> "I'm afraid that if you look at a thing long enough, it loses all of its meaning."
>
> — Andy Warhol

Now that we've seen how to produce complex maps with **ggplot2** we are going to learn how to bring Google maps into the mix. Some kind hearted person has made a package for R that allows us to do this relatively easily. But that means we will need to install another new package.

After we learn how to use Google maps we will take some time to learn additional mapping techniques that we may want for creating publication quality figures.

10.1 ggmap

The package we will need for this tut is **ggmap**. It is a bit beefy so let's get started on installing it now. Once it's done we will activated the packages we need and load some site points for mapping.

```
# Load libraries
library(tidyverse)
library(ggmap)

# Load data
load("data/cape_point_sites.RData")
```

Take a moment to look at the data we loaded. What does it show?

10.2 Mapping Cape Point

We will be creating our Google map in two steps. The first step is to use the `get_map()` function to tell Google what area of the world we want a map of, as well as what sort of map we want. Remember how Google maps can be switched from satellite view to map view? We may do that in R as well. For now we are only going to use the satellite view so that we may insert out own labels. But if we look in the help file we may see a description of all the different map types available. There are a bunch!

The `get_map()` function relies on a healthy Internet connection to run. The downloading of Google maps can be a tenuous process without a stable connection. For that reason, if your Internet connection is not great you may run the commented out line of code in favour of loading the Google map. This is a pre-saved file of the output of the code chunk below. Please rather run the `get_map()` function first, and if it won't connect only then load the saved file as shown below.

```
cape_point <- get_map(location = c(lon = 18.36519, lat = -34.2352581),
                      zoom = 10, maptype = 'satellite')
# load("data/cape_point.RData")
```

If we look in the environment panel in the top right corner, what do we see? What do we think the code above is doing?

The second step in the process is to treat the Google data we downloaded as though it is just any ordinary **ggplot2** object. The same as the ones we created yesterday and today. For this reason we may use `+` to add new lines of **ggplot2** code to the Google object we downloaded in order to show site locations etc. Let's first just see how the map looks when we add some points. Note that we do not use the function `ggplot()` at the beginning of our code, but rather `ggmap()`.

```
cp_1 <- ggmap(cape_point) +
  geom_point(data = cape_point_sites, aes(x = lon+0.002, y = lat-0.007),
             colour = "red", size =  2.5) +
  labs(x = "", y = "")
cp_1
```

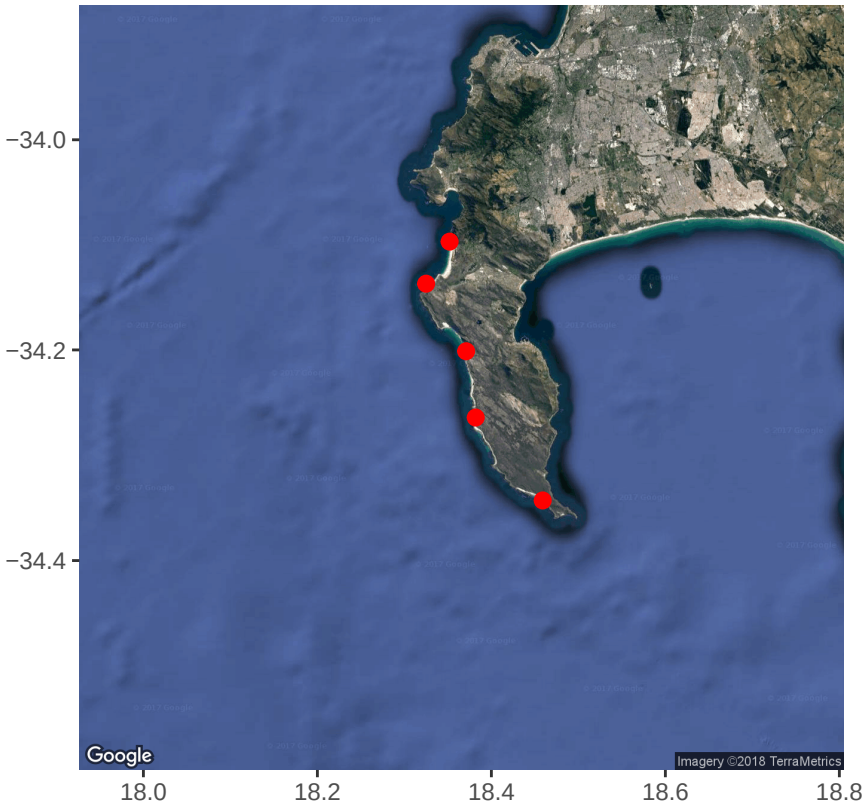


Figure 10.1: Google map of Cape Point with some site locations highlighted with red points.

Pretty cool huh?! You may do this for anywhere in the world just as easy as this. The only thing you need to keep in mind is that the lon/lat coordinates for Google appear to be slightly different than the global standard. This is why the x and y `aes()` values in the code above have a little bit added or subtracted from them. That one small issue aside, this is a nice quick workflow for adding your site locations to a Google map background. Play around with the different map types you can download and try it for any place you can think of.

10.3 Site labels

The previous figure shows our sites along Cape Point, which is great, but which site is which?! We need site labels. This is a relatively straightforward process, so we've given a complicated example of how to do so here.

```
cp_2 <- cp_1 +
  geom_text(data = cape_point_sites, # Choose dataframe
            aes(lon+0.002, lat-0.007, label = site), # Set aesthetics
            hjust = 1.15, vjust = 0.5, # Adjust vertical and horizontal
            size = 3, colour = "white") # Adjust appearance
cp_2
```

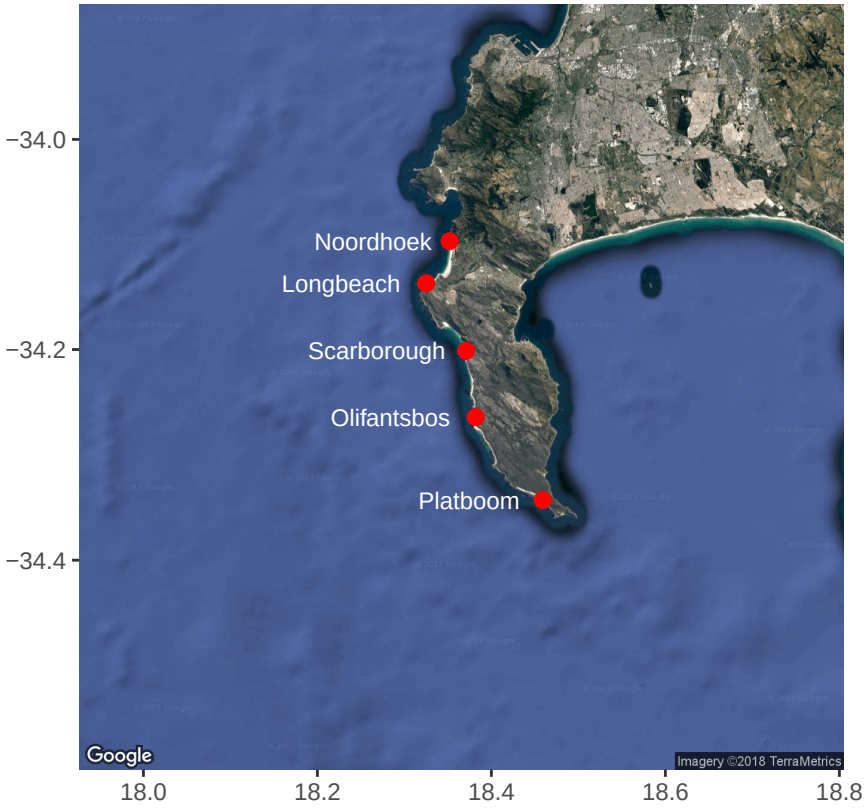


Figure 10.2: Google map of Cape Point with site point labeled

10.4 DIY maps

Now that we have learned how to make conventional maps, as well as fancy Google maps, it is time for us to branch out once again at let our creative juices flow. Please group up as you see fit and create your own beautiful map of wherever you like. Bonus points for faceting in additional figures showing supplementary information. Feel free to use either conventional maps or the Google alternative. Same as yesterday, we will be walking the room to help with any issues that may arise.

10.5 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>     ggmap   forcats   stringr     dplyr     purrr     readr     tidyr    tibble
R>   "3.0.0"   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"   "2.1.3"
R>   ggplot2 tidyverse
R>   "3.2.1"   "1.3.0"
```

Chapter 11

Tidy data

> "Order and simplification are the first steps toward the mastery of a subject."
>
> — Thomas Mann
>
> "Get your facts first, and then you can distort them as much as you please."
>
> — Mark Twain

The Tidyverse is a collection of R packages that adhere to the *tidy data* principles of data analysis and graphing. The purpose of these packages is to make working with data more efficient. The core Tidyverse packages were created by Hadley Wickham, but over the last few years other individuals have added some packages to the collective, which has significantly expanded our data analytical capabilities through improved ease of use and efficiency. The Tidyverse packages can be loaded collectively by calling the `tidyverse` package, as we have seen throughout this workshop. The packages making up the Tidyverse are shown in Figure 11.1.

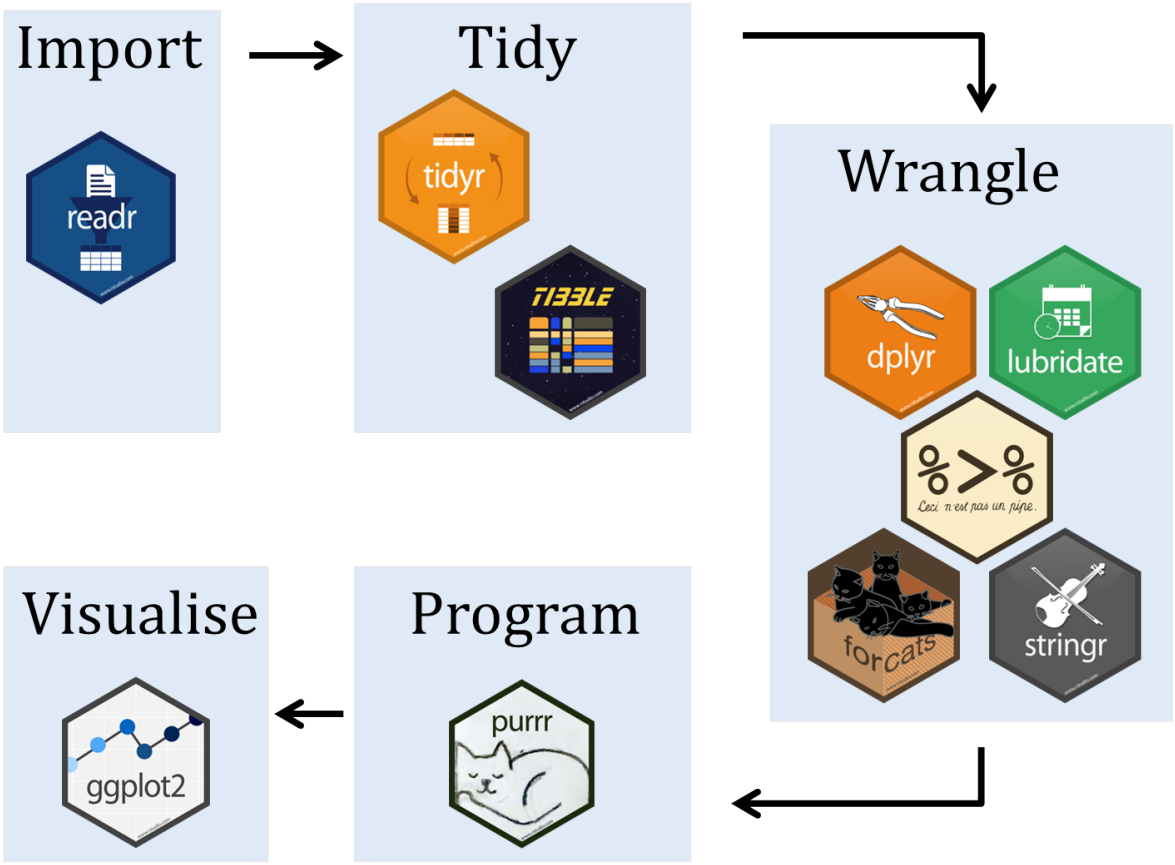


Figure 11.1: The Tidyverse by Hadley Wickham and co.

```
library(tidyverse)
```

As we may see in the following figure (Figure 11.2), the tidying of ones data should be the second step in any workflow, after the loading of the data.

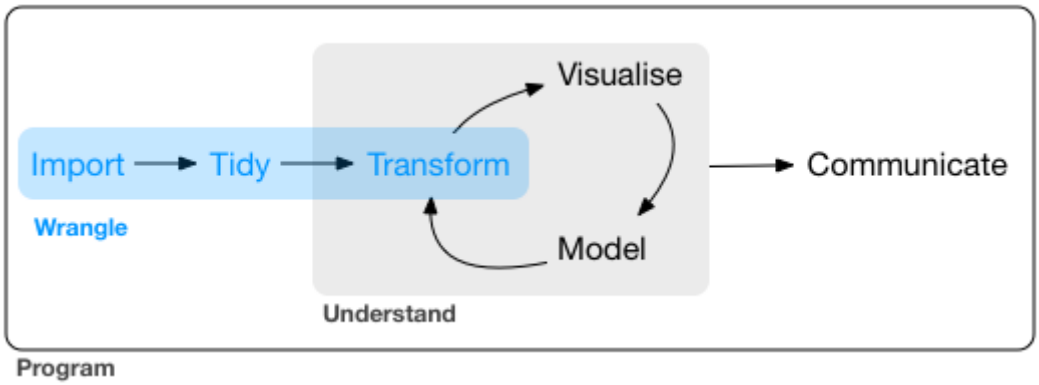


Figure 11.2: Data tidying in the data processing pipeline. Reproduced from [R for Data Science](http://r4ds.had.co.nz/workflow-basics.html)

But what exactly are **tidy data**? It is not just a a buzz word, there is a real definition. In three parts, to be exact. Taken from Hadley Wickham's R for Data Science:

1. Each variable must have its own column—*i.e.* each of the things **known about** the data in its own column.
2. Each observation must have its own row—each of the things **measured** in its own row; for example, in the *Laminaria* data the measured things include `blade_length`, `blade_weight`, *etc.*, and they can all be captured in one column named, for example, `measurement`.

3. Each value must have its own cell—each of the things that is known and measured must be in its own cell.

This is represented graphically in F 11.3. One will generally satisfy these three rules effortlessly simply by never putting more than one dataset in a file, and never putting more (or less) than one variable in the same column. We will go over this several more times today so do not fret if those guidelines are not immediately clear.

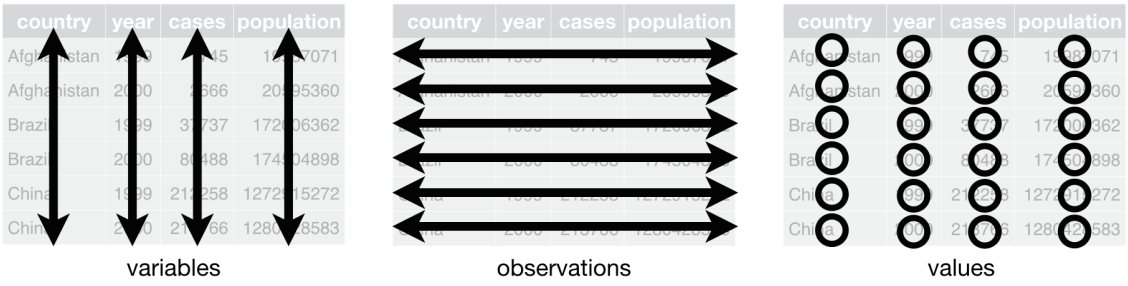


Figure 11.3: Following three rules make a dataset tidy — variables are in columns, observations are in rows, and values are in cells. Reproduced from [R for Data Science](http://r4ds.had.co.nz/workflow-basics.html)

In order to illustrate the meaning of this three part definition, we are going to learn how to manipulate a non-tidy dataset into a tidy one. To do so we will need to learn a few new, very useful functions. Let's load our demo dataset to get started. This snippet from the SACTN dataset contains only data for 2008-2009 for three time series, with some notable (untidy) changes. The purpose of the following exercises is not only to show how to tidy data, but to also illustrate that these steps may be done more quickly in R than MS Excel, allowing for ones raw data to remain exactly how they were collected, with all of the manipulations performed on them documented in an R script. This is a centrally important part of reproducible research.

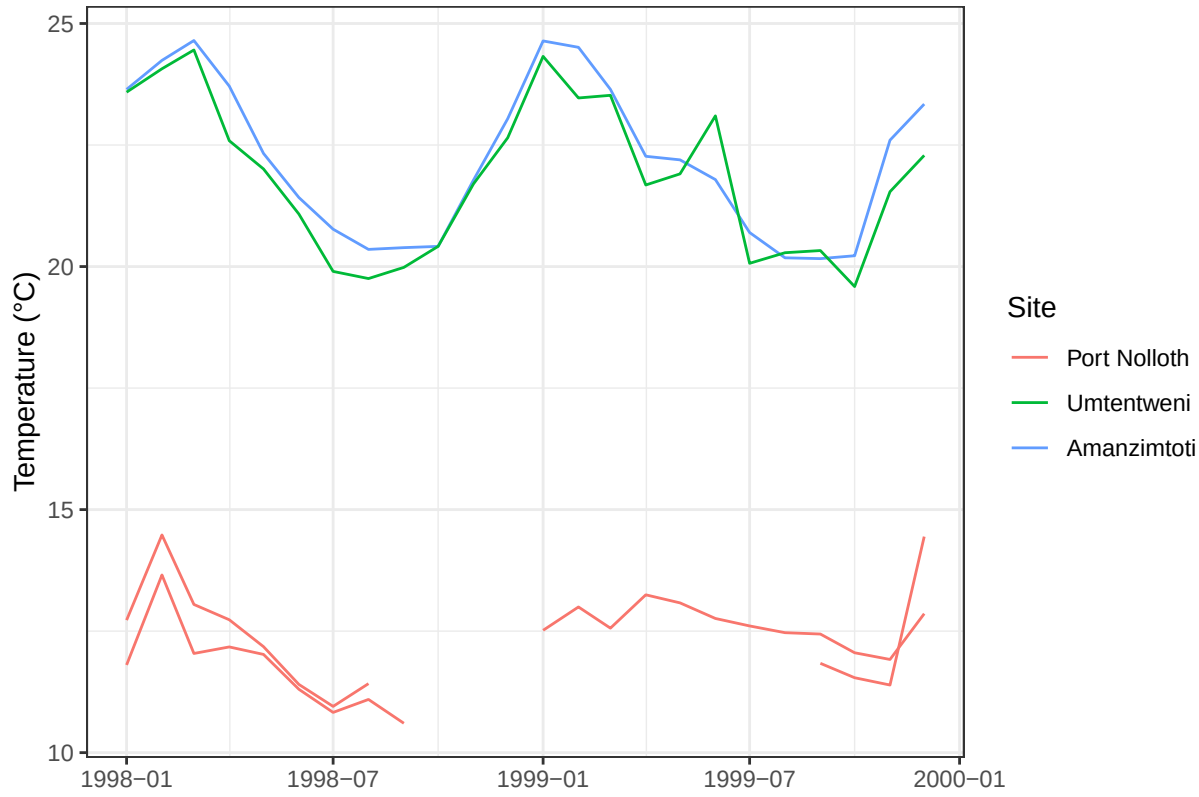
```
load("data/SACTN_mangled.RData")
```

With our data loaded let's now have a peek at them. We will first see that we have loaded not one, but five different objects into our environment pane in the top right of our RStudio window. These all contain the exact same data in different states of disrepair. As one may guess, some of these datasets will be easier to use than others.

```
SACTN1
SACTN2
SACTN3

# Spread across two dataframes
SACTN4a
SACTN4b
```

We start off by looking at `SACTN1`. If these data look just like all of the other SACTN data we've used thus far that's because they are. These are how tidy data should look. No surprises. In fact, because these data are already tidy it is very straightforward to use them for whatever purposes we may want. Making a time series plot, for example.

```
ggplot(data = SACTN1, aes(x = date, y = temp)) +
  geom_line(aes(colour = site, group = paste0(site, src))) +
  labs(x = "", y = "Temperature (°C)", colour = "Site") +
  theme_bw()
```



`%>%`
Remember that this funny series of symbols is the pipe operator. It combines consequetive rows of code together so that they run as though they were one 'chunk'. We will be seeing this symbol a lot today. The keyboard shortcut for `%>%` is **ctrl-shift-m**.

11.1 Gathering and spreading

Before *tidy* became the adjective used to describe neatly formatted data, people used to say *long*. This is because well organised dataframes tend to always be longer than they are wide (with the exception of species assemblage data). The opposite of *long* data are *wide* data. If one ever finds a dataset that is wider than it is long then this is

probably because the person that created them saved one variable across many columns.

As we sit here and read through these examples it may seem odd that so much effort is being spent on something so straightforward as tidy data. Surely this is too obvious to devote an entire day of work to it? Unfortunately not. As we go out into the wild world of 'real life data', we tend to find that very few datasets (especially those collected by hand) are tidy. Rather they are plagued by any number of issues. The first step then for tidying up the data are to have a look at them and discern what are the observations that were made/recorded (the 'measurements'), and what are the variables within those observations. We also need to know something about *how* or *where* the data were collected, or what threy represent — this is information *about* the data, *i.e.* some meta-data. Let's have a look now at `SACTN2` for an example of what *wide* data look like, and how to fix it.

11.1.1 Gathering

In `SACTN2` we can see that the `src` column has been removed and that the temperatures are placed in columns that denote the collecting source. This may at first seem like a reasonable way to organise these data, but it is not tidy because the collecting source is one variable, and so should not take up more than one column. We need to `gather()` these source columns back together. We do this by telling `gather()` what the names of the columns are we want to squish together. We then tell it the name of the **key** column. This is the column that will contain all of the old column names we are gathering. In this case we will call it `src`. The last piece of this puzzle is the **value** column. This is where we decide what the name of the column will be for values we are gathering up. In this case we will name it `temp`, because we are gathering up the temperature values that were incorrectly spread out by month.

```
SACTN2_tidy <- SACTN2 %>%
  gather(DEA, KZNSB, SAWS, key = "src", value = "temp")
```

11.1.2 Spreading

Should ones data be too long, meaning when individual observations are spread across multiple rows, we will need to use `spread()` to rectify the situation. This is generally the case when we have two or more variables stored within the same column, as we may see in `SACTN3`. This is not terribly common as it would require someone to put quite a bit of time into making a dataframe this way. But never say never. To spread data we first tell R what the name of the column is that contains more than one variable, in this case the 'var' column. We then tell R what the name of the column is that contains the values that need to be spread, in this case the 'val' column. Notice here that these two column names are not given in inverted commas. This is because with gather, we were creating two new columns, and so we fed them to R as character vectors (*i.e.* inside of inverted commas). With the spread function we are naming existing columns and so we give them to R without inverted commas.

```
SACTN3_tidy1 <- SACTN3 %>%
  spread(key = var, value = val)
```

11.2 Separating and uniting

We've now covered how to make our dataframes longer or wider depending on their tidiness. Now we will look at how to manage our columns when they contain more (or less) than one variable, but the overall dataframe does not need to be made wider or longer. This is generally the case when one has a column with two variables, or two or more variables are spread out across multiple columns, but there is still only one observation per row. Let's see some examples to make this more clear.

11.2.1 Separate

If we look at `SACTN4a` we see that we no longer have a `site` and `src` column. Rather these have been replaced by an `index` column. This is an efficient way to store these data, but it is not tidy because the site and source of each observation have now been combined into one column (variable). Remember, tidy data calls for each of the things known about the data to be its own variable. To re-create our `site` and `src` columns we must `separate()` the `index` column. First we give R the name of the column we want to separate, in this case `index`. Next we must say what the names of the new columns will be. Remember that because we are creating new column names we feed these into R within inverted commas. Lastly we should tell R how to separate the `index` column. If we look at the data we may see that the values we want to split up are separated with '/', so that is what we give to R. Often times the `separate()` function is able to guess correctly, but it is better to be explicit.

```
SACTN4a_tidy <- SACTN4a %>%
  separate(col = index, into = c("site", "src"), sep = "/ ")
```

11.2.2 Separating dates using `mutate()`

Although the `date` column represents an example of a date type (a kind of data in its own right), one might also want to split this column into its constituent parts, *i.e.* create separate columns for day, month, and year. In this case we can spread these components of the date vector into three columns using the `mutate()` function and some functions in the **lubridate** package (contained within **tidyverse**).

```
SACTN_tidy2 <- SACTN4a %>%
  separate(col = index, into = c("site", "src"), sep = "/ ") %>%
  mutate(day = lubridate::day(date),
         month = lubridate::month(date),
         year = lubridate::year(date))
```

Note that when the date is split into component parts the data are no longer tidy (see below).

11.2.3 Unite

It is not uncommon that field/lab instruments split values across multiple columns while they are making recordings. I see this most often with date values. Often the year, month, and day values are given in different columns. There are uses for the data in this way, though it is not terribly tidy. We usually want the date of any observation to be shown in just one column. If we look at `SACTN4b` we will see that there is a `year`, `month`, and `day` column. To `unite()` them we must first tell R what we want the united column to be labelled, in this case we will use `date`. We then list the columns to be united, her this is `year`, `month`, and `day`. Lastly we must specify if we want the united values to have a separator between them. The standard separator for date values is '-'.

```
SACTN4b_tidy <- SACTN4b %>%
  unite(year, month, day, col = "date", sep = "-")
```

11.3 Joining

We will end this session with the concept of joining two different dataframes. Remember that one of the rules of tidy data is that only one complete dataset is saved per dataframe. This rule then is violated not only when additional data are stored where they don't belong, but also when necessary data are saved elsewhere. If we look back at `SACTN4a` and `SACTN4b` we will see that they are each missing different columns. Were we to **join** these dataframes together they would complete each other. The `tidyverse` provides us with several methods of doing this, but we will demonstrate here only the most common technique. The function `left_join()` is so named because it joins two or more dataframes together based on the matching of columns from the left to the right. It combines values together where it sees that they match up, and adds new rows and columns where they do not.

```
SACTN4_tidy <- left_join(SACTN4a_tidy, SACTN4b_tidy)
```

```
R> Joining, by = c("site", "src", "date")
```

As we see above, if we let `left_join()` do it's thing it will make a plan for us and find the common columns and match up the values and observations for us as best it can. It then returns a message letting us know what it's done. That is a pleasant convenience, but we most likely want to exert more control over this process than that. In order to specify the columns to be used for joining we must add one more argument to `left_join()`. The `by` argument must be fed a list of column names in inverted commas if we want to specify how to join our dataframes. Not that when we run this it does not produce a message as we have provided enough explicit information that the machine is no longer needing to think for itself.

```
SACTN4_tidy <- left_join(SACTN4a_tidy, SACTN4b_tidy, by = c("site", "src", "date"))
```

There are also other kinds of joins: see for example also `inner_join`, `right_join`, `full_join`, `semi-join`, `nest_join`, and `anti_join` in the **dplyr** package contained within **tidyverse**.

11.4 But why though?

At this point one may be wondering what the point of all of this is. Sure it's all well and good to see how to tidy one's data in R, but couldn't this be done more quickly and easily in MS Excel? Perhaps, yes, with a small dataset. But remember, (for many) the main reason we are learning R is to ensure that we are performing reproducible research. This means that every step in our workflow must be documented. And we accomplish this by writing R scripts, and part of the purpose of these scripts is to ensure that we capture all the steps taken to get the raw data into a neat and tidy format that can unambiguously be read into R in the format that will quicky get us up

and running with our analyses and visualisations.

11.5 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R>   forcats   stringr     dplyr     purrr     readr     tidyr    tibble   ggplot2
R>   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"   "2.1.3"   "3.2.1"
R> tidyverse
R>   "1.3.0"
```

Chapter 12

Tidier data

> "Knowing where things are, and why, is essential to rational decision making."
>
> — Jack Dangermond
>
> "The mind commands the body and it obeys. The mind orders itself and meets resistance."
>
> — Frank Herbert, Dune

On Day 1 already we walked ourselves through a tidy workflow. We saw how to import data, how to manipulate it, run a quick analysis or two, and create figures. In the previous session we filled in the missing piece of the workflow by also learning how to tidy up our data within R. For the remainder of today we will be revisiting the 'transform' portion of the tidy workflow. In this session we are going to go into more depth on what we learned in Day 1, and in the last session we will learn some new tricks. Over these two sessions we will also become more comfortable with the *pipe* command `%>%`, while practising writing tidy code.

There are five primary data transformation functions that we will focus on here:

- Arrange observations (rows) with `arrange()`
- Filter observations (rows) with `filter()`
- Select variables (columns) with`select()`
- Create new variables (columns) with `mutate()`
- Summarise variables (columns) with `summarise()`

We will use the full South African Coastal Temperature Network dataset for these exercises. Before we begin however we will need to cover two new concepts.

```
# Load libraries
library(tidyverse)
library(lubridate)

# Load the data from a .RData file
load("data/SACTNmonthly_v4.0.RData")

# Copy the data as a dataframe with a shorter name
SACTN <- SACTNmonthly_v4.0

# Remove the original
rm(SACTNmonthly_v4.0)
```

12.1 Comparison operators

The assignment operator (←) is a symbol that we use to assign some bit of code to an object in our environment. Likewise, comparison operators are symbols we use to compare different objects. This is how we tell R how to decide to do many different things. We will see these symbols often out in the 'real world' so let's spend a moment now getting to know them better. Most of these should be very familiar to us:

- Greater than: `>`
- Greater than or equal to: `⩾`
- Less than: `<`
- Less than or equal to: `⩽`
- Equal to: `==`
- Not equal to: `≠`

It is important here to note that `==` is for comparisons and `=` is for maths. They are **not** interchangeable, as we may see in the following code chunk. This is one of the more common mistakes one makes when writing code. Luckily the error message this creates should provide us with the clues we need to figure out that we have made this specific mistake.

```
SACTN %>%
  filter(site = "Amanzimtoti")
```

```
R> Error: `site` (`site = "Amanzimtoti"`) must not be named, do you need `==`?
```

12.2 Logical operators

Comparison operators are used to make direct comparisons between specific things, but logical operators are used more broadly when making logical arguments. Logic is central to most computing so it is worth taking the time to cover these symbols explicitly here. R makes use of the same *Boolean logic* symbols as many other platforms, including Google, so some (or all) of these will likely be familiar. We will generally only use three:

- and: `&`
- or: `|`
- not: `!`

When writing a line of tidy code we tend to use these logical operator to combine two or more arguments that use comparison operators. For example, the following code chunk uses the `filter()` function to find all temperatures recorded at Pollock Beach during December **OR** January. Don't worry if the following line of code is difficult to piece out, but make sure you can locate which symbols are comparison operators and which are logical operators. Please note that for purposes of brevity all of the outputs in this section are limited to ten lines, but when one runs these code chunks on ones own computer they will be much longer.

```
SACTN %>%
  filter(site == "Pollock Beach", month(date) == 12 | month(date) == 1)
```

```
R>             site  src       date     temp depth   type
R> 1  Pollock Beach SAWS 1999-12-01 19.95000     0 thermo
R> 2  Pollock Beach SAWS 2000-01-01 19.03333     0 thermo
R> 3  Pollock Beach SAWS 2000-12-01 19.20000     0 thermo
R> 4  Pollock Beach SAWS 2001-01-01 18.32667     0 thermo
R> 5  Pollock Beach SAWS 2001-12-01 20.59032     0 thermo
R> 6  Pollock Beach SAWS 2002-01-01 21.47097     0 thermo
R> 7  Pollock Beach SAWS 2002-12-01 19.78065     0 thermo
R> 8  Pollock Beach SAWS 2003-01-01 20.64516     0 thermo
R> 9  Pollock Beach SAWS 2003-12-01 20.48710     0 thermo
R> 10 Pollock Beach SAWS 2004-01-01 21.34839     0 thermo
```

We will look at the interplay between comparison and logical operators in more depth in the following session after we have reacquainted ourselves with the main transformation functions we need to know.

12.3 Arrange observations (rows) with `arrange()`

First up in our greatest hits reunion tour is the function `arrange()`. This very simply arranges the observations (rows) in a dataframe based on the variables (columns) it is given. If we are concerned with ties in the ordering

of our data we provide additional columns to `arrange()`. The importance of the columns for arranging the rows is given in order from left to right.

```
SACTN %>%
  arrange(depth, temp)
```

```
R>             site  src       date      temp depth   type
R> 1      Sea Point SAWS 1990-07-01  9.635484     0 thermo
R> 2     Muizenberg SAWS 1984-07-01  9.708333     0 thermo
R> 3     Doringbaai SAWS 2000-12-01  9.772727     0 thermo
R> 4  Hondeklipbaai SAWS 2003-06-01  9.775000     0 thermo
R> 5      Sea Point SAWS 1984-06-01 10.000000     0 thermo
R> 6     Muizenberg SAWS 1992-07-01 10.193548     0 thermo
R> 7  Hondeklipbaai SAWS 2005-07-01 10.333333     0 thermo
R> 8  Hondeklipbaai SAWS 2003-07-01 10.340909     0 thermo
R> 9      Sea Point SAWS 2000-12-01 10.380645     0 thermo
R> 10    Muizenberg SAWS 1984-08-01 10.387097     0 thermo
```

If we would rather arrange our data in descending order, as is perhaps more often the case, we simply wrap the column name we are arranging by with the `desc()` function as shown below.

```
SACTN %>%
  arrange(desc(temp))
```

```
R>             site   src       date     temp depth type
R> 1        Sodwana   DEA 2000-02-01 28.34648    18  UTR
R> 2        Sodwana   DEA 1999-03-01 28.04890    18  UTR
R> 3        Sodwana   DEA 1998-03-01 27.87781    18  UTR
R> 4        Sodwana   DEA 1998-02-01 27.76452    18  UTR
R> 5        Sodwana   DEA 1996-02-01 27.73637    18  UTR
R> 6        Sodwana   DEA 2000-03-01 27.52637    18  UTR
R> 7        Sodwana   DEA 2000-01-01 27.52291    18  UTR
R> 8  Leadsmanshoal EKZNW 2007-02-01 27.48132    10  UTR
R> 9        Sodwana EKZNW 2005-01-01 27.45619    12  UTR
R> 10       Sodwana EKZNW 2007-02-01 27.44054    12  UTR
```

It must also be noted that when arranging data in this way, any rows with `NA` values will be sent to the bottom of the dataframe. This is not always ideal and so must be kept in mind.

12.4 Filter observations (rows) with `filter()`

When simply arranging data is not enough, and we need to remove rows of data we do not want, `filter()` is the tool to use. For example, we can select all monthly temperatures recorded at the `site` Humewood during the `year` 1990 with the following code chunk:

```
SACTN %>%
  filter(site == "Humewood", year(date) == 1990)
```

```
R>        site  src       date     temp depth   type
R> 1  Humewood SAWS 1990-01-01 21.87097     0 thermo
R> 2  Humewood SAWS 1990-02-01 18.64286     0 thermo
R> 3  Humewood SAWS 1990-03-01 18.61290     0 thermo
R> 4  Humewood SAWS 1990-04-01 17.30000     0 thermo
R> 5  Humewood SAWS 1990-05-01 16.35484     0 thermo
R> 6  Humewood SAWS 1990-06-01 15.93333     0 thermo
R> 7  Humewood SAWS 1990-07-01 15.70968     0 thermo
R> 8  Humewood SAWS 1990-08-01 16.09677     0 thermo
R> 9  Humewood SAWS 1990-09-01 16.41667     0 thermo
R> 10 Humewood SAWS 1990-10-01 17.14194     0 thermo
```

Remember to use the assignment operator (←, keyboard shortcut **alt -**) if one wants to create an object in the environment with the new results.

```
humewood_90s <- SACTN %>%
  filter(site == "Humewood", year(date) %in% seq(1990, 1999, 1))
```

It must be mentioned that `filter()` also automatically removes any rows in the filtering column that contain `NA` values. Should one want to keep rows that contain missing values, insert the `is.na()` function into the line of code in question. To illustrate this let's filter the temperatures for the Port Nolloth data collected by the DEA that were at or below 11°C OR were missing values. We'll put each argument on a separate line to help keep things clear. Note how R automatically indents the last line in this chunk to help remind us that they are in fact part of the same argument. Also note how I have put the last bracket at the end of this argument on it's own line. This is not required, but I like to do so as it is a very common mistake to forget the last bracket.

```
SACTN %>%
  filter(site == "Port Nolloth", # First give the site to filter
         src == "DEA", # Then specify the source
         temp <= 11 | # Temperatures at or below 11°C OR
           is.na(temp) # Include missing values
         )
```

12.5 Select variables (columns) with `select()`

When one loads a dataset that contains more columns than will be useful or required it is preferable to shave off the excess. We do this with the `select()` function. In the following four examples we are going to remove the `depth` and `type` columns. There are many ways to do this and none are technically better or faster. So it is up to the user to find a favourite technique.

```
# Select columns individually by name
SACTN %>%
  select(site, src, date, temp)

# Select all columns between site and temp like a sequence
SACTN %>%
  select(site:temp)

# Select all columns except those stated individually
SACTN %>%
  select(-date, -depth)

# Select all columns except those within a given sequence
  # Note that the '-' goes outside of a new set of brackets
  # that are wrapped around the sequence of columns to remove
SACTN %>%
  select(-(date:depth))
```

We may also use `select()` to reorder the columns in a dataframe. In this case the inclusion of the `everything()` function may be a useful shortcut as illustrated below.

```
# Change up order by specifying individual columns
SACTN %>%
  select(temp, src, date, site)

# Use the everything function to grab all columns
# not already specified
SACTN %>%
  select(type, src, everything())

# Or go bananas and use all of the rules at once
  # Remember, when dealing with tidy data,
  # everything may be interchanged
SACTN %>%
  select(temp:type, everything(), -src)
```

12.6 Create new variables (columns) with `mutate()`

When one is performing data analysis/statistics in R this is likely because it is necessary to create some new values that did not exist in the raw data. The previous three functions we looked at (`arrange()`, `filter()`, `select()`) will prepare us to create new data, but do not do so themselves. This is when we need to use `mutate()`. We must however be very mindful that `mutate()` is only useful if we want to create new variables (columns) that are a function of one or more *existing* columns. This means that any column we create with `mutate()` will always have the same number of rows as the dataframe we are working with. In order to create a new column we must first tell R what the name of the column will be, in this case let's create a column named `kelvin`. The second step is to then tell R what to put in the new column. AS you may have guessed, we are going to convert the `temp` column into Kelvin (°K) by adding 273.15 to every row.

```
SACTN %>%
  mutate(kelvin = temp + 273.15))
```

```
R>            site src       date     temp depth type   kelvin
R> 1   Port Nolloth DEA 1991-02-01 11.47029     5  UTR 284.6203
R> 2   Port Nolloth DEA 1991-03-01 11.99409     5  UTR 285.1441
R> 3   Port Nolloth DEA 1991-04-01 11.95556     5  UTR 285.1056
R> 4   Port Nolloth DEA 1991-05-01 11.86183     5  UTR 285.0118
R> 5   Port Nolloth DEA 1991-06-01 12.20722     5  UTR 285.3572
R> 6   Port Nolloth DEA 1991-07-01 12.53810     5  UTR 285.6881
R> 7   Port Nolloth DEA 1991-08-01 11.25202     5  UTR 284.4020
R> 8   Port Nolloth DEA 1991-09-01 11.29208     5  UTR 284.4421
R> 9   Port Nolloth DEA 1991-10-01 11.37661     5  UTR 284.5266
R> 10 Port Nolloth DEA 1991-11-01 10.98208     5  UTR 284.1321
```

This is a very basic example and `mutate()` is capable of much more than simple addition. We will get into some more exciting examples during the next session.

12.7 Summarise variables (columns) with `summarise()`

Finally this brings us to the last tool for this section. To create new columns we use `mutate()`, but to calculate any sort of summary/statistic from a column that will return fewer rows than the dataframe has we will use `summarise()`. This makes `summarise()` much more powerful than the other functions in this section, but because it is able to do more, it can also be more unpredictable, making it's use potentially more challenging. We will almost always end op using this function in our work flows however so it behoves us to become well acquainted with it. The following chunk very simply calculates the overall mean temperature for the entire SACTN.

```
SACTN %>%
  summarise(mean_temp = mean(temp, na.rm = TRUE))
```

```
R>   mean_temp
R> 1  19.26955
```

Note how the above chunk created a new dataframe. This is done because it cannot add this one result to the previous dataframe due to the mismatch in the number of rows. Were we to want to create additional columns with other summaries we may do so within the same `summarise()` function. These multiple summaries are displayed on individual lines in the following chunk to help keep things clear.

```
SACTN %>%
  summarise(mean_temp = mean(temp, na.rm = TRUE),
            sd_temp = sd(temp, na.rm = TRUE),
            min_temp = min(temp, na.rm = TRUE),
            max_temp = max(temp, na.rm = TRUE)
            )
```

```
R>   mean_temp  sd_temp min_temp max_temp
R> 1  19.26955 3.682122 9.136322 28.34648
```

Creating summaries of the *entire* SACTN dataset in this way is not appropriate as we should not be combining time series from such different parts of the coast. In order to calculate summaries within variables we will need to learn how to use `group_by()`, which in turn will first require us to learn how to chain multiple functions together

within a pipe (%>%). That is how we will begin the next session for today. Finishing with several tips on how to make our data the tidiest that it may be.

12.8 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R> lubridate   forcats   stringr     dplyr     purrr     readr     tidyr    tibble
R>   "1.7.4"   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"   "2.1.3"
R>   ggplot2 tidyverse
R>   "3.2.1"   "1.3.0"
```

Chapter 13

Tidiest data

> "Conducting data analysis is like drinking a fine wine. It is important to swirl and sniff the wine, to unpack the complex bouquet and to appreciate the experience. Gulping the wine doesn't work."
>
> — Daniel B. Wright
>
> "If you torture the data long enough, it will confess to anything."
>
> — Ronald Coase

In the previous session we covered the five main transformation functions one would use in a typical tidy workflow. But to really unlock their power we need to learn how to use them with `group_by()`. This is how we may calculate statistics based on the different grouping variables within our data, such as sites or species or soil types, for example. Let's begin by loading the **`tidyverse`** package and the SACTN data if we haven't already.

```
# Load libraries
library(tidyverse)
library(lubridate)

# load the data from a .RData file
load("data/SACTNmonthly_v4.0.RData")

# Copy the data as a dataframe with a shorter name
SACTN <- SACTNmonthly_v4.0

# Remove the original
rm(SACTNmonthly_v4.0)
```

13.1 Group observations (rows) by variables (columns) with `group_by()`

With the SACTN dataset loaded we will now look at the effect that `group_by` has on other `tidyverse` functions. First we shall create a new object called `SACTN_depth`. If we look at this object in the RStudio GUI it will not appear to be any different from `SACTN`. We may think of `group_by()` as working behind the scenes in order to better support the five main functions. We see that `group_by()` is working as intended when we create summaries from the `SACTN_depth` dataframe. Remember that one does not need to put the last bracket own it's own line. I like to do so in order to reduce the chances that I will forget to type it after the final argument within the last function in the code chunk.

```
# Group by depth
SACTN_depth <- SACTN %>%
  group_by(depth)

# Calculate mean temperature by depth
SACTN_depth_mean <- SACTN_depth %>%
  summarise(mean_temp = mean(temp, na.rm = TRUE),
  count = n()
)
```

```
# Visualise the results
SACTN_depth_mean
```

```
R> # A tibble: 13 x 3
R>    depth mean_temp count
R>    <dbl>     <dbl> <int>
R>  1     0      19.5 26299
R>  2     2      13.0   237
R>  3     3      17.6   141
R>  4     4      16.6   529
R>  5     5      15.0   408
R>  6     7      17.2   227
R>  7     8      17.8   201
R>  8     9      13.8   311
R>  9    10      19.5   362
R> 10    12      24.3   137
R> 11    14      22.3   223
R> 12    18      24.4   191
R> 13    28      14.8   306
```

Let's visualise our newly created summary dataframe and see what we get.

```
# Why does the relationship between depth and temperature look so odd?
ggplot(data = SACTN_depth_mean, mapping = aes(x = depth, y = mean_temp)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
```

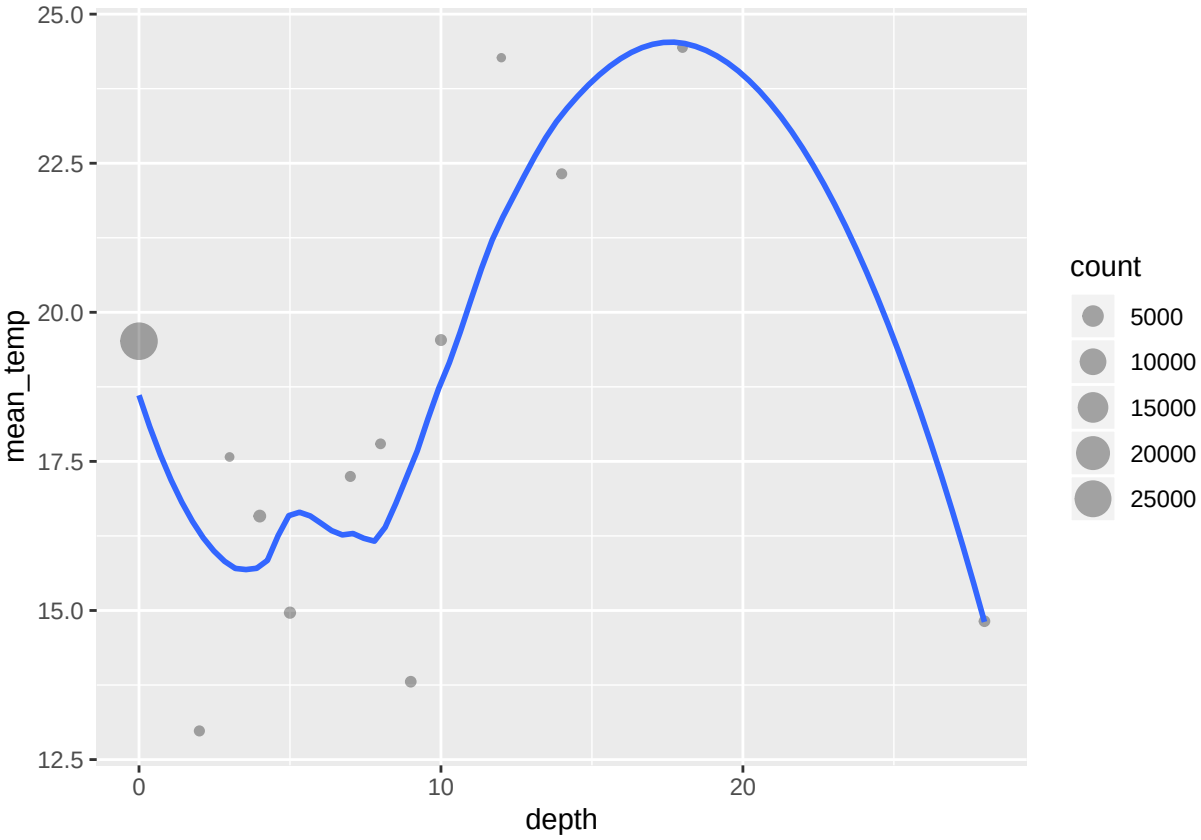


Figure 13.1: Relationship between depth and mean temperature.

13.1.1 Grouping by multiple variables

As one may have guessed by now, grouping is not confined to a single column. One may use any number of columns to perform elaborate grouping measures. Let's look at some ways of doing this with the SACTN data.

```
# Create groupings based on temperatures and depth
SACTN_temp_group <- SACTN %>%
  group_by(round(temp), depth)

# Create groupings based on source and date
SACTN_src_group <- SACTN %>%
  group_by(src, date)

# Create groupings based on date and depth
SACTN_date_group <- SACTN %>%
  group_by(date, depth)
```

Now that we've created some grouped dataframes, let's think of some ways to summarise these data.

13.1.2 Ungrouping

Once we level up our **tidyverse** skills we will routinely be grouping variables while calculating statistics. This then poses the problem of losing track of which dataframes are grouped and which aren't. Happily, to remove any grouping we just use `ungroup()`. No arguments required, just the empty function by itself. Too easy.

```
SACTN_ungroup <- SACTN_date_group %>%
  ungroup()
```

13.2 Chain functions with the pipe (%>%)

This now brings us to the last major concept we need to become confident with for this workshop. Everything we have learned thus far builds on everything else in a modular way, with most concepts and steps therein being interchangeable. The pipe takes all of the things we have learned and takes them to the next level. And the best part about it is that it requires us to learn nothing new. We've been doing it all along, perhaps without even realising it. Let's see what I mean. The following chunk does in one command what the first chunk in the previous section took two steps. This is not only faster, but saves us from having to create intermediate dataframes that only slow down our computer and clog up our environment.

```
SACTN_depth_mean_2 <- SACTN %>% # Choose a base dataframe
  group_by(depth) %>% # Group by thedepth column
  summarise(mean_temp = mean(temp, na.rm = TRUE), # Calculate means
  count = n() # Count observations
) # Safety measure
```

Take a moment and compare the `SACTN_depth_mean_2` object that we've just created against the `SACTN_depth_mean` object we created at the beginning of this session. Same same.

Not only does this keep our workflow tidier, it also makes it easier to read for ourselves, our colleagues, and most importantly, our future selves. When we look at the previous code chunk we can think of it as a paragraph in a research report, with each line a sentence. If I were to interpret this chunk of code in plain English I think it would sound something like this:

> In order to create the `SACTN_depth_mean_2` dataframe I first started by taking the original SACTN data.
> I then grouped the data into different depth categories.
> After this I calculated the mean temperature for each depth category, as well as counting the number of observations within each depth group.

Just like paragraphs in a human language may vary in length, so too may code chunks. There really is no limit. This is not to say that it is encouraged to attempt to reproduce a code chunk of comparable length to anything Marcel Proust would have written. It is helpful to break things up into pieces of a certain size. What that size is though is open to the discretion of the person writing the code. It is up to you to find out for yourself what works best for you.

13.3 Group all the functions!

We've played around quite a bit with grouping and summarising, but that's not all we can do. We can use `group_by()` very nicely with `filter()` and `mutate()` as well. Not so much with `arrange()` and `select()` as these are designed to work on the entire dataframe at once, without any subsetting.

We can do some rather imaginative things when we combine all of these tools together. In fact, we should be able to accomplish almost any task we can think of. For example, what if we wanted to create a new object that was a subset of only the sites in the `SACTN` that had at least 30 years (360 months) of data?

```
SACTN_30_years <- SACTN %>%
  group_by(site, src) %>%
  filter(n() > 360)
```

Or what if we wanted to calculate anomaly data for each site?

```
SACTN_anom <- SACTN %>%
  group_by(site, src) %>%
  mutate(anom = temp - mean(temp, na.rm = T)) %>%
  select(site:date, anom, depth, type) %>%
  ungroup()
```

Now, let's select only two sites and calculate their mean and standard deviations. Note how whichever columns we give to `group_by()` will be carried over into the new dataframe created by `summarise()`.

```
SACTN %>%
  filter(site == "Paternoster" | site == "Oudekraal") %>%
  group_by(site, src) %>%
  summarise(mean_temp = mean(temp, na.rm = TRUE),
            sd_temp = sd(temp, na.rm = TRUE))
```

```
R> # A tibble: 3 x 4
R> # Groups:   site [2]
R>   site        src   mean_temp sd_temp
R>   <fct>       <chr>     <dbl>   <dbl>
R> 1 Paternoster DEA        12.8   0.879
R> 2 Paternoster SAWS       13.6   1.40
R> 3 Oudekraal   DAFF       12.3   1.36
```

13.4 Going deeper

We learned in the previous session that one should avoid using comparison operators to compare logical arguments as this tends to not produce the results one would expect. Below we see what happens when we try to repeat the code chunk above, but using a logical operator within a comparison operator.

```
SACTN %>%
  filter(site == "Paternoster" | "Oudekraal") %>% # This line has been changed/shortened
  group_by(site, src) %>%
  summarise(mean_temp = mean(temp, na.rm = TRUE),
            sd_temp = sd(temp, na.rm = TRUE))
```

```
R> Error in site == "Paternoster" | "Oudekraal": operations are possible only for numeric, logical or complex types
```

Oh no, we broke it! This is a common error while learning to write code so do try to keep this rule in mind as it can cause a lot of headaches. An easy way to spot this problem is if ones line of code has more logical operators than comparison operators you're probably going to have a bad time. This is doubly unfortunate as we would need to write less code if this were not so. Happily, there is a shortcut for just this problem, `%in%`. Whenever we want to use operators to filter by more than two things, it is most convenient to create an object that contains the names or numbers that we want to filter by. We then replace our comparison and logical operators with that one simple symbol (`%in%`).

```
# First create a character vector containing the desired sites
selected_sites <- c("Paternoster", "Oudekraal", "Muizenberg", "Humewood")

# Then calculate the statistics
SACTN %>%
  filter(site %in% selected_sites) %>%
  group_by(site, src) %>%
  summarise(mean_temp = mean(temp, na.rm = TRUE),
            sd_temp = sd(temp, na.rm = TRUE))
```

```
R> # A tibble: 5 x 4
R> # Groups:   site [4]
R>   site        src   mean_temp sd_temp
R>   <fct>       <chr>     <dbl>   <dbl>
R> 1 Paternoster DEA        12.8   0.879
R> 2 Paternoster SAWS       13.6   1.40
R> 3 Oudekraal   DAFF       12.3   1.36
R> 4 Muizenberg  SAWS       15.9   2.76
R> 5 Humewood    SAWS       18.0   2.03
```

The `%in%` operator can be a very useful shortcut, but sometime we cannot avoid the comparison and logical operator dance. For example, if one wanted to find temperatures at Port Nolloth that were over 10°C but under 15°C you could use either of the following two filters. Remember that whenever we see a `,` in the filter function it is the same as the `&` logical operator. Of the two different techniques shown below, I would be more inclined to use the first one. The fewer symbols we use to write our code the better. Both for readability and error reduction.

```
SACTN %>%
  filter(site == "Port Nolloth", temp > 10, temp < 15)

SACTN %>%
  filter(site == "Port Nolloth", !(temp <= 10 | temp >= 15))
```

As one may imagine, performing intricate logical arguments like this may get out of hand rather quickly. It is advisable to save intermediate steps in a complex workflow to avoid too much heartache. Where exactly these 'fire breaks' should be made is up to the person writing the code.

13.5 Pipe into ggplot2

It is also possible to combine piped code chunks and **ggplot2** chunks into one 'combi-chunk'. I prefer not to do this as I like saving the new dataframe I have created as an object in my environment before visualising it so that if anything has gone wrong (as things tend to do) I can more easily find the problem.

Regardless of what one may or may not prefer to do, the one thing that must be mentioned about piping into **ggplot2** is that when we start with the `ggplot()` function we switch over from the pipe (`%>%`) to the plus sign (`+`). There are currently efforts to address this inconvenience, but are not yet ready for public consumption.

```
SACTN %>% # Choose starting dataframe
  filter(site %in% c("Bordjies", "Tsitsikamma", "Humewood", "Durban")) %>% # Select sites
  select(-depth, -type) %>% # Remove depth and type columns
  mutate(month = month(date), # Create month column
         index = paste(site, src, sep = "/ ")) %>% # Create individual site column
  group_by(index, month) %>% # Group by individual sites and months
  summarise(mean_temp = mean(temp, na.rm = TRUE), # Calculate mean temperature
            sd_temp = sd(temp, na.rm = TRUE)) %>% # Calculate standard deviation
  ggplot(aes(x = month, y = mean_temp)) + # Begin with ggplot, switch from '%>%' to '+'
  geom_ribbon(aes(ymin = mean_temp - sd_temp, ymax = mean_temp + sd_temp),
              fill = "black", alpha = 0.4) + # Create a ribbon
  geom_line(col = "red", size = 0.3) + # Create lines within ribbon
  facet_wrap(~index) + # Facet by individual sites
  scale_x_continuous(breaks = seq(2, 12, 4)) + # Control x axis ticks
  labs(x = "Month", y = "Temperature (°C)") + # Change labels
  theme_dark() # Set theme
```

13.6 Additional useful functions

There is an avalanche of useful functions to be found within the **tidyverse**. In truth, we have only looked at functions from three packages: **ggplot2**, **dplyr**, and **tidyr**. There are far, far too many functions even within these three packages to cover within a week. But that does not mean that the functions in other packages, such as **purrr** are not also massively useful for our work. More on that tomorrow. For now we will see how the inclusion of a handful of choice extra functions may help to make our workflow even tidier.

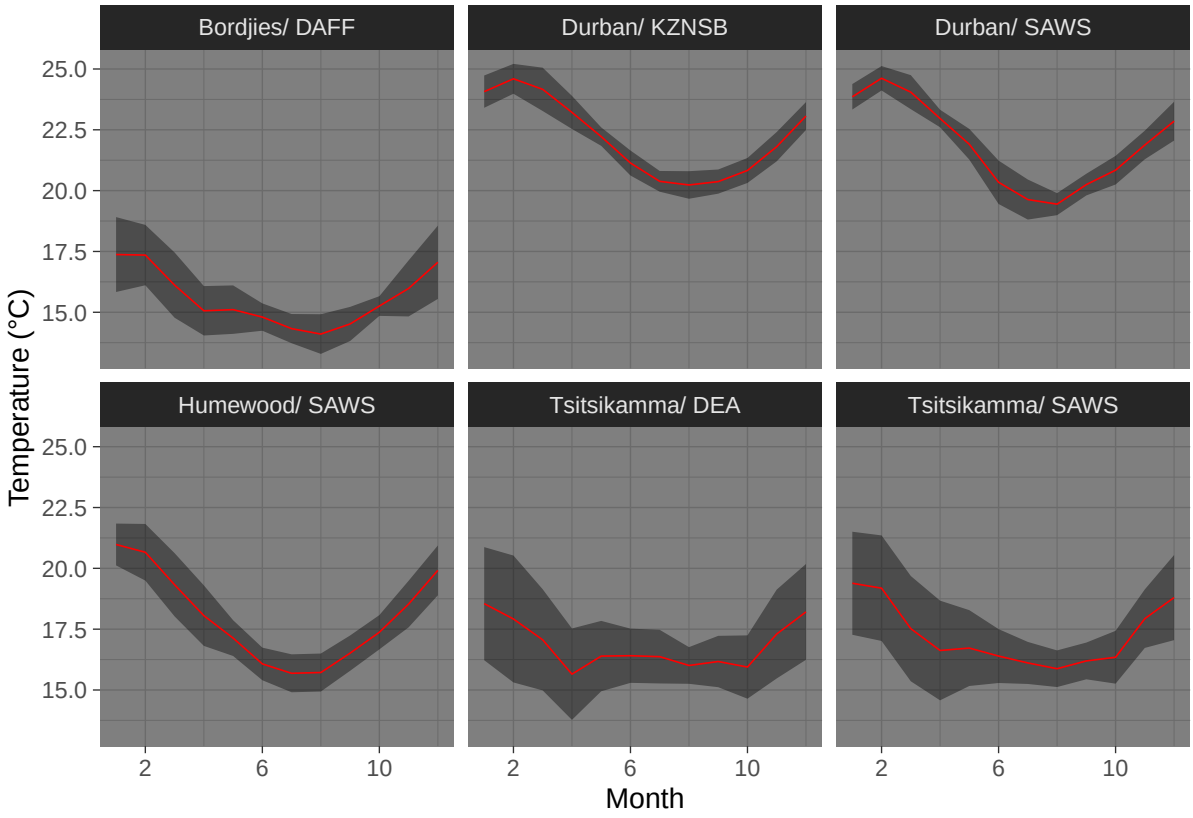


Figure 13.2: Line and ribbon plots for the climatologies of several sites.

13.6.1 Rename variables (columns) with `rename()`

We have seen that we select columns in a dataframe with `select()`, but if we want to rename columns we have to use, you guessed it, `rename()`. This functions works by first telling R the new name you would like, and then the existing name of the column to be changed. This is perhaps a bit back to front, but such is life on occasion.

```
SACTN %>%
  rename(source = src)
```

```
R>             site source       date     temp depth type
R> 1  Port Nolloth    DEA 1991-02-01 11.47029     5  UTR
R> 2  Port Nolloth    DEA 1991-03-01 11.99409     5  UTR
R> 3  Port Nolloth    DEA 1991-04-01 11.95556     5  UTR
R> 4  Port Nolloth    DEA 1991-05-01 11.86183     5  UTR
R> 5  Port Nolloth    DEA 1991-06-01 12.20722     5  UTR
R> 6  Port Nolloth    DEA 1991-07-01 12.53810     5  UTR
R> 7  Port Nolloth    DEA 1991-08-01 11.25202     5  UTR
R> 8  Port Nolloth    DEA 1991-09-01 11.29208     5  UTR
R> 9  Port Nolloth    DEA 1991-10-01 11.37661     5  UTR
R> 10 Port Nolloth    DEA 1991-11-01 10.98208     5  UTR
```

13.6.2 Create a new dataframe for a newly created variable (column) with `transmute()`

If for whatever reason one wanted to create a new variable (column), as one would do with `mutate()`, but one does not want to keep the dataframe from which the new column was created, the function to use is `transmute()`.

```
SACTN %>%
  transmute(kelvin = temp + 273.15)
```

```
R>   [1] 284.6203 285.1441 285.1056 285.0118 285.3572 285.6881 284.4020 284.4421
R>   [9] 284.5266 284.1321
```

This makes a bit more sense when paired with `group_by()` as it will pull over the grouping variables into the new

dataframe. Note that when it does this for us automatically it will provide a message in the console.

```
SACTN %>%
  group_by(site, src) %>%
  transmute(kelvin = temp + 273.15)
```

```
R> # A tibble: 10 x 3
R> # Groups:   site, src [1]
R>    site         src   kelvin
R>    <fct>        <chr>  <dbl>
R>  1 Port Nolloth DEA     285.
R>  2 Port Nolloth DEA     285.
R>  3 Port Nolloth DEA     285.
R>  4 Port Nolloth DEA     285.
R>  5 Port Nolloth DEA     285.
R>  6 Port Nolloth DEA     286.
R>  7 Port Nolloth DEA     284.
R>  8 Port Nolloth DEA     284.
R>  9 Port Nolloth DEA     285.
R> 10 Port Nolloth DEA     284.
```

13.6.3 Count observations (rows) with `n()`

We have already seen this function sneak it's way into a few of the code chunks in the previous session. We use `n()` to count any grouped variable automatically. It is not able to be given any arguments, so we must organise our dataframe in order to satisfy it's needs. It is the diva function of the **`tidyverse`**; however, it is terribly useful as we usually want to know how many observations our summary stats are based. First we will run some stats and create a figure without documenting `n`. Then we will include `n` and see how that changes our conclusions.

```
SACTN_n <- SACTN %>%
  group_by(site, src) %>%
  summarise(mean_temp = round(mean(temp, na.rm = T))) %>%
  arrange(mean_temp) %>%
  ungroup() %>%
  select(mean_temp) %>%
  unique()

ggplot(data = SACTN_n, aes(x = 1:nrow(SACTN_n), y = mean_temp)) +
  geom_point() +
  labs(x = "", y = "Temperature (°C)") +
  theme(axis.text.x = element_blank(),
        axis.ticks.x = element_blank())
```

This looks like a pretty linear distribution of temperatures within the SACTN dataset. But now let's change the size of the dots to show how frequently each of these mean temperatures is occurring.

```
SACTN_n <- SACTN %>%
  group_by(site, src) %>%
  summarise(mean_temp = round(mean(temp, na.rm = T))) %>%
  ungroup() %>%
  select(mean_temp) %>%
  group_by(mean_temp) %>%
  summarise(count = n())

ggplot(data = SACTN_n, aes(x = 1:nrow(SACTN_n), y = mean_temp)) +
  geom_point(aes(size = count)) +
  labs(x = "", y = "Temperature (°C)") +
  theme(axis.text.x = element_blank(),
        axis.ticks.x = element_blank())
```

We see now when we include the count (`n`) of the different mean temperatures that this distribution is not so even. There appear to be humps around 17°C and 22°C. Of course, we've created dot plots here just to illustrate

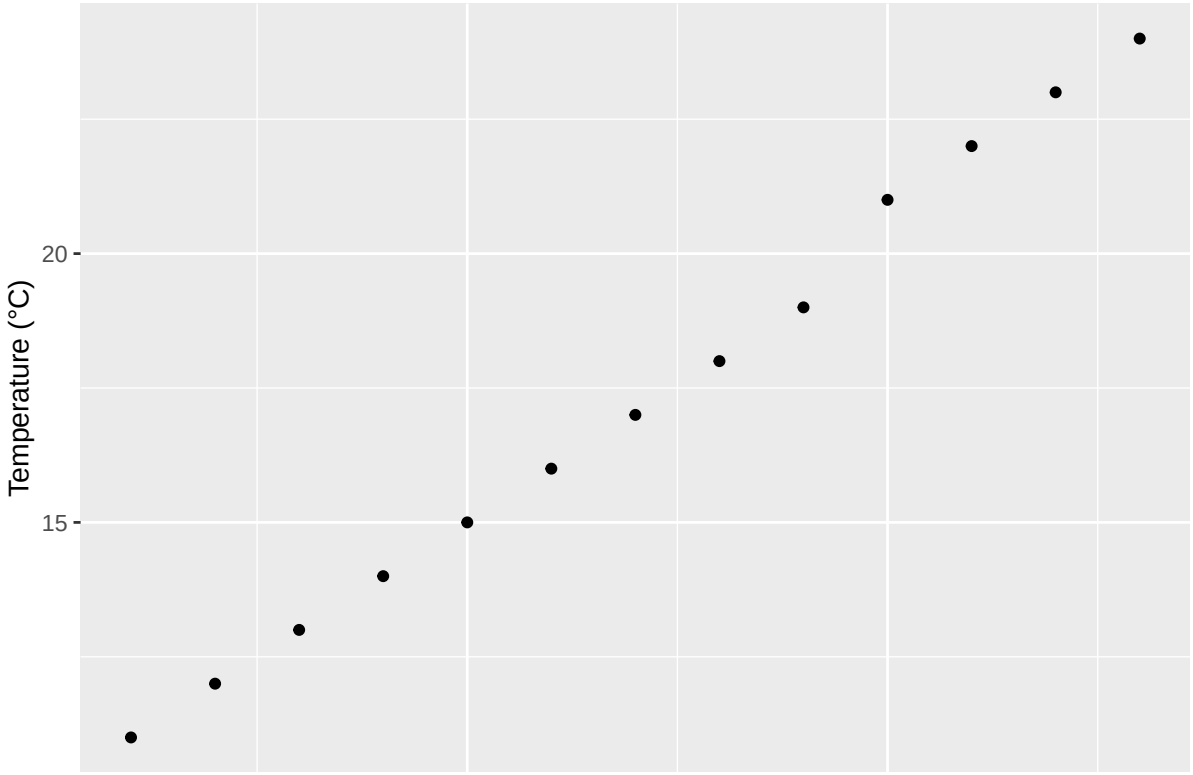


Figure 13.3: Dot plot showing range of mean temperatures for the time series in the SACTN dataset.

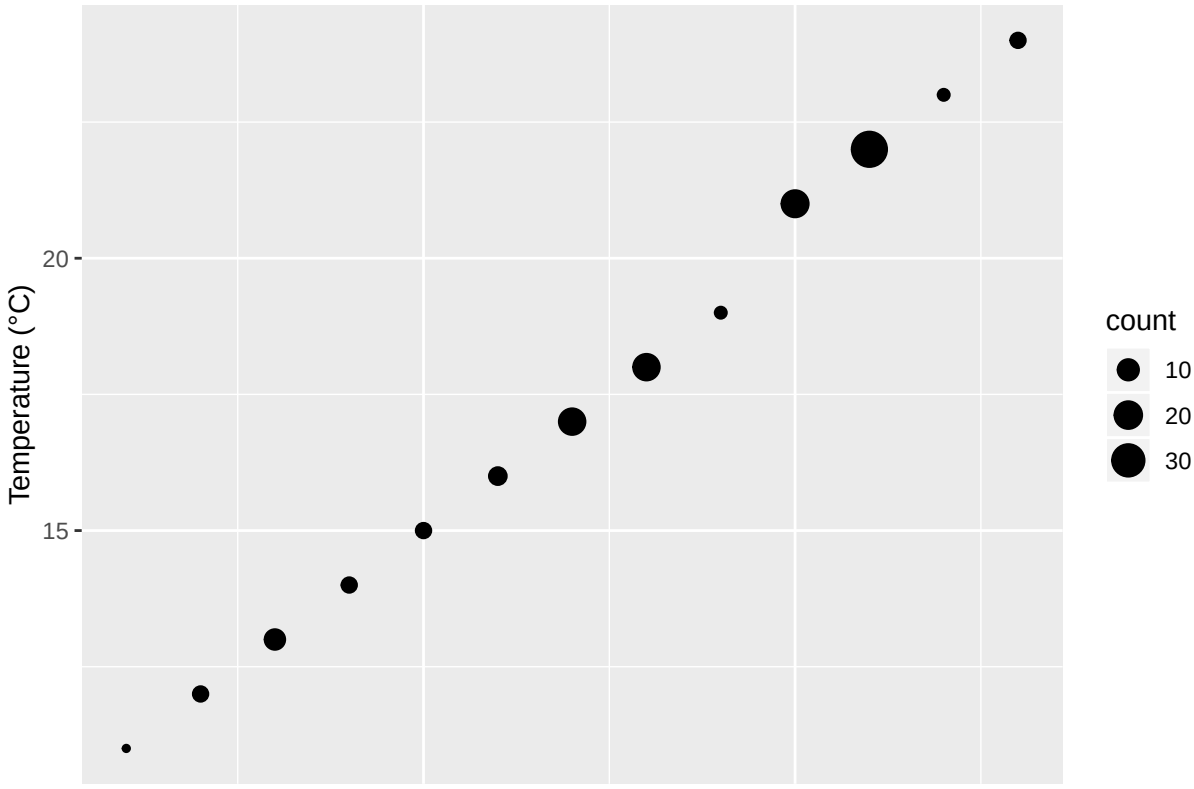


Figure 13.4: Dot plot showing range of mean temperatures for the time series in the SACTN dataset with the size of each dote showing the number of occurences of each mean.

this point. In reality if one were interested in a distribution like this one would use a histogram, or better yet, a density polygon.

```
SACTN %>%
  group_by(site, src) %>%
  summarise(mean_temp = round(mean(temp, na.rm = T))
            ) %>%
  ungroup() %>%
  ggplot(aes(x = mean_temp)) +
  geom_density(fill = "seagreen", alpha = 0.6) +
  labs(x = "Temperature (°C)")
```

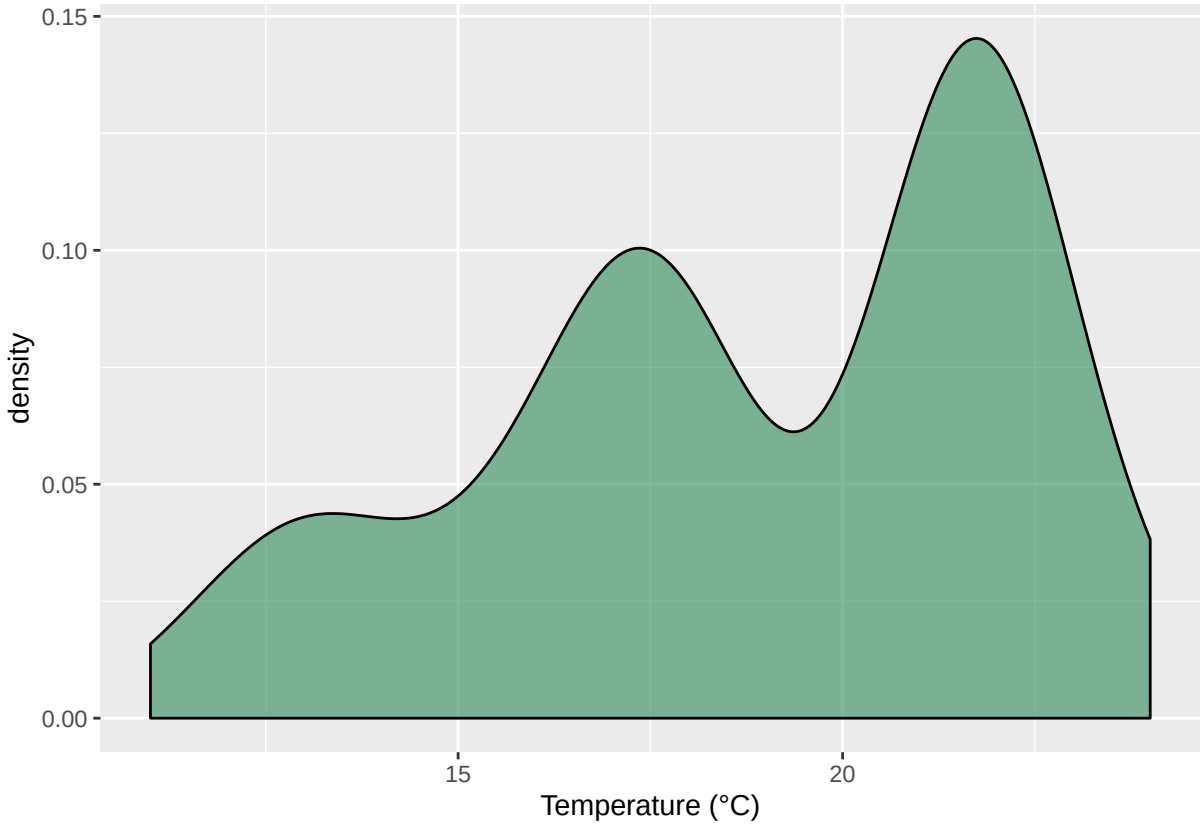


Figure 13.5: Frequency distribution of mean temperature for each time series in the SACTN dataset.

13.6.4 Select observations (rows) by number with `slice()`

If one wants to select only specific rows of a dataframe, rather than using some variable like we do for `filter()`, we use `slice()`. The function expects us to provide it with a series of integers as seen in the following code chunk. Try playing around with these values and see what happens

```
# Slice a seqeunce of rows
SACTN %>%
  slice(10010:10020)

# Slice specific rows
SACTN %>%
  slice(c(1,8,19,24,3,400))

# Slice all rows except these
SACTN %>%
  slice(-(c(1,8,4)))

# Slice all rows except a sequence
```

```
SACTN %>%
  slice(-(1:1000))
```

It is discouraged to use slice to remove or select specific rows of data as this does not discriminate against any possible future changes in ones data. Meaning that if at some point in the future new data are added to a dataset, re-running this code will likely no longer be selecting the correct rows. This is why `filter()` is a main function, and `slice()` is not. This auxiliary function can however still be quite useful when combined with arrange.

```
# The top 5 variable sites as measured by SD
SACTN %>%
  group_by(site, src) %>%
  summarise(sd_temp = sd(temp, na.rm = T)) %>%
  ungroup() %>%
  arrange(desc(sd_temp)) %>%
  slice(1:5)
```

```
R> # A tibble: 5 x 3
R>   site       src   sd_temp
R>   <fct>      <chr>   <dbl>
R> 1 Muizenberg SAWS     2.76
R> 2 Stilbaai   SAWS     2.72
R> 3 Mossel Bay SAWS     2.65
R> 4 De Hoop    DAFF     2.51
R> 5 Mossel Bay DEA      2.51
```

13.6.5 Summary functions

There is a near endless sea of possibilities when one starts to become comfortable with writing R code. We have seen several summary functions used thus far. Mostly in straightforward ways. But that is one of the fun things about R, the only limits to what we may create are within our mind, not the program. Here is just one example of a creative way to answer a straightforward question: 'What is the proportion of recordings above 15°C per source?'. Note how we may refer to columns we have created within the same chunk. There is no need to save the intermediate dataframes if we choose not to.

```
SACTN %>%
  na.omit() %>%
  group_by(src) %>%
  summarise(count = n(),
            count_15 = sum(temp > 15)) %>%
  mutate(prop_15 = count_15/count) %>%
  arrange(prop_15)
```

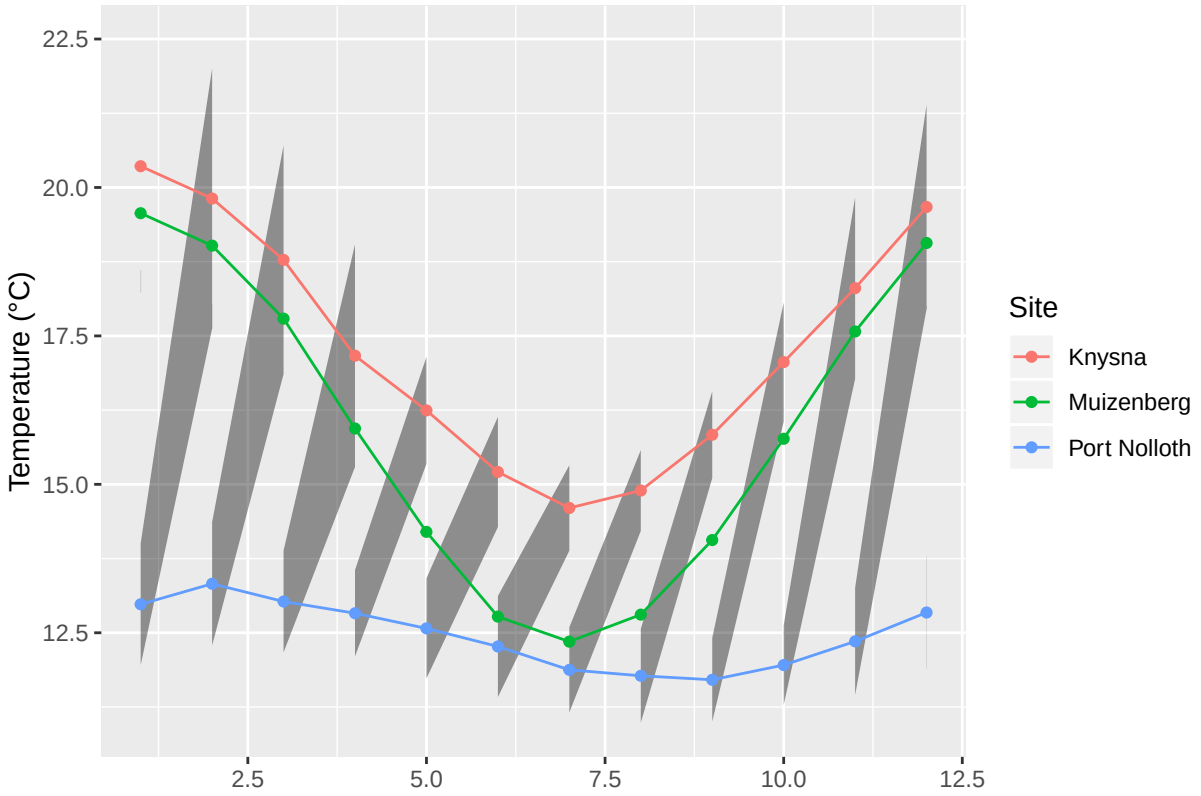
```
R> # A tibble: 7 x 4
R>   src   count count_15 prop_15
R>   <chr> <int>    <int>   <dbl>
R> 1 DAFF    641      246   0.384
R> 2 SAWS   8636     4882   0.565
R> 3 UWC      12        7   0.583
R> 4 DEA    2087     1388   0.665
R> 5 SAEON   596      573   0.961
R> 6 EKZNW   369      369   1
R> 7 KZNSB 15313    15313   1
```

13.7 The new age *redux*

Remember the spreadsheet example from the first day of the R workshop? Here it is repeated in a more efficient way. Now with bonus ribbons! In this chunk we see how to load, transform, and visualise the data all in one go. One would not normally do this, but it sure is snappy!

```
read_csv("data/SACTN_data.csv") %>% # Load the SACTN Day 1 data
  mutate(month = month(date)) %>% # Then create a month abbreviation column
  group_by(site, month) %>% # Then group by sites and months
```

```
  summarise(mean_temp = mean(temp, na.rm = TRUE), # Lastly calculate the mean
            sd_temp = sd(temp, na.rm = TRUE)) %>% # and the SD
  ggplot(aes(x = month, y = mean_temp)) + # Begin ggplot
  geom_ribbon(aes(ymin = mean_temp - sd_temp, ymax = mean_temp + sd_temp),
              fill = "black", alpha = 0.4) + # Create a ribbon
  geom_point(aes(colour = site)) + # Create dots
  geom_line(aes(colour = site, group = site)) + # Create lines
  labs(x = "", y = "Temperature (°C)", colour = "Site") # Change labels
```



13.8 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R> lubridate   forcats   stringr     dplyr     purrr     readr     tidyr    tibble
R>   "1.7.4"   "0.4.0"   "1.4.0"   "0.8.3"   "0.3.3"   "1.3.1"   "1.0.0"   "2.1.3"
R>   ggplot2 tidyverse
R>   "3.2.1"   "1.3.0"
```

Chapter 14

Recap

> "Everyone should have their mind blown once a day."
>
> — Neil deGrasse Tyson
>
> "Somewhere, something incredible is waiting to be known."
>
> — Carl Sagan

Over the past four days we have covered quite a bit of ground. By now it is our hope that after having participated in this workshop you will feel confident enough using R to branch out on your own and begin applying what you have learned to your own research.

Above all, remember the tidy principles you have leaned here and endeavour to apply them to all facets of your work. The more uniformly tidy your work becomes, the more compounding benefits you will begin to notice.

14.1 The future

The content we have covered in this workshop is only the beginning. We have looked down upon the tidyverse, it's multitudinous spiralling arms stretching out away from us in all directions. The next step is to begin to investigate the specific branches of the R tree of knowledge that interest us most. Or are most relevant to our work. The following list contains some further suggestions for workshops that are available:

- R for biologists
- R for environmental science
- R for oceanographers
- Advanced visualisations
- Multivariate analysis
- Species distribution modelling
- Reproducible research
- Basic stats

For further information or inquiries about additional training please contact Robert Schlegel: robwschlegel@gmail.com .

14.2 Today

For the rest of today we will now open the floor to questions and suggestions that we may work through as a group.

14.3 Session info

```
installed.packages()[names(sessionInfo()$otherPkgs), "Version"]
```

```
R> character(0)
```