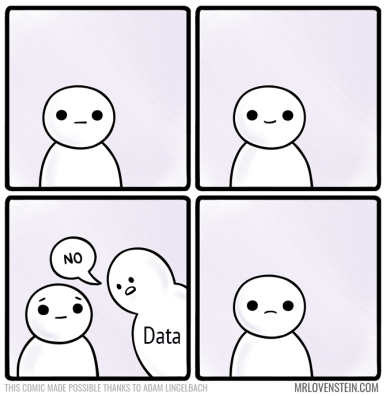


R Workshop: Data Manipulation, Analysis and Graphing

AJ Smit & Robert Schlegel

2017-09-07

Contents

Part I

Day 1

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

1 Preliminaries

1.1 Venue, date and time

This workshop will take place in the week of **4 September – 8 September 2017**, from **9:00–16:00** each day. We will meet in the Zoology building on Rhodes campus.

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 –
- Optional Exercise: A primer on R

Day 2 – Show and tell

- Interactive Session: The basics of `ggplot2`
- – break –
- Interactive Session: The basics of `ggplot2`
- Interactive Session: Brewing colours in `ggplot2`
- – lunch –

- Assignment: DIY report
- – break –
- Student presentations

Day 3 – Going deeper

- Interactive Session: Mapping in `ggplot2`
- – break –
- Interactive Session: Mapping in `ggplot2`
- – lunch –
- Interactive Session: Mapping in Google
- – break –
- Interactive Session: Next level mapping

Day 4 – The Enlightened Researcher

- Presentation: Reproducible research, a.k.a. `Rmarkdown`
- – break –
- Interactive Sessions: The `tidyverse`
- – lunch –
- Interactive Session: "Tidy" data

Day 5 – The world is yours

- Assignment: Final project
- – break –
- Student presentations

1.3 About this Workshop

The aim of this five-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. Day 5 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 no homework but there are 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 learn 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 is becoming the global standard. There are several reasons for this trend:

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 Mac OS X, 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

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, there are 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

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) enable 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!

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[1]. 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] http://cran.r-project.org

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[2] 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 August 2017.

— Cheers, AJ and Robert

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[3] — the R graphics bible
- A Compendium of Clean Graphs in R. Version 2.0[4] — using R's base graphics
- R for Data Science[5] — data analysis using tidy principles
- R Markdown[6] — reproducible reports in R
- bookdown: Authoring Books and Technical Documents with R Markdown[7] — writing books in R
- Shiny[8] — 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 salmon in a bold code (monospaced) font, *e.g.* `tidyr`.

[2] http://www.rstudio.com
[3] https://github.com/hadley/ggplot2-book
[4] http://shinyapps.org/apps/RGraphCompendium/index.php
[5] http://r4ds.had.co.nz/workflow-basics.html
[6] http://rmarkdown.rstudio.com
[7] https://bookdown.org/yihui/bookdown
[8] https://shiny.rstudio.com

- Functions are in salmon, and the code font is followed by parentheses, like `plot()`, or `summary()`.
- Other R objects, such as data, function arguments or variable names are again in salmon font, but without parentheses, such as `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] -26.10  21.41  -1.02   5.78  -8.80  -1.72  13.13  -3.83   2.93
R> [10]  13.71
```

Consult these resources for more about R code style :

- Google's R style guide[9]
- The tidyverse style guide[10]
- Hadley Wickham's advanced R style guide[11]

We can also insert math 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 .pdf document 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[12]. 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! – and you can get 10% extra for this workshop. Note that you will need to complete the exercises in the chapter, An R workflow, before this will be possible. You will then have to demonstrate to Rob or myself how you accomplished this amazing feat (not tricky, really, if you like technical things like we do…).

Figure 1.1: The Rmarkdown workflow.

You will notice that this repository uses GitHub[13], 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.

The R session information when compiling this book is shown below:

[9] https://google.github.io/styleguide/Rguide.xml
[10] http://style.tidyverse.org
[11] http://adv-r.had.co.nz/Style.html
[12] https://github.com/ajsmit/Intro_R_Workshop
[13] https://github.com

```
 sessionInfo()
R> R version 3.4.1 (2017-06-30)
R> Platform: x86_64-apple-darwin15.6.0 (64-bit)
R> Running under: macOS Sierra 10.12.6
R>
R> Matrix products: default
R> BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib
R> LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib
R>
R> locale:
R> [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
R>
R> attached base packages:
R> [1] stats     grDevices utils     datasets  graphics  methods
R> [7] base
R>
R> other attached packages:
R>  [1] maps_3.2.0      zoo_1.8-0       lubridate_1.6.0 knitr_1.17
R>  [5] jpeg_0.1-8      png_0.1-7       gridExtra_2.2.1 dplyr_0.7.2
R>  [9] purrr_0.2.3     readr_1.1.1     tidyr_0.7.1     tibble_1.3.4
R> [13] ggplot2_2.2.1   tidyverse_1.1.1 raster_2.5-8    sp_1.2-5
R>
R> loaded via a namespace (and not attached):
R>  [1] reshape2_1.4.2   haven_1.1.0      lattice_0.20-35  colorspace_1.3-2
R>  [5] htmltools_0.3.6  yaml_2.1.14      rlang_0.1.2      foreign_0.8-69
R>  [9] glue_1.1.1       modelr_0.1.1     readxl_1.0.0     bindrcpp_0.2
R> [13] bindr_0.1        plyr_1.8.4       stringr_1.2.0    munsell_0.4.3
R> [17] gtable_0.2.0     cellranger_1.1.0 rvest_0.3.2      psych_1.7.5
R> [21] evaluate_0.10.1  forcats_0.2.0    parallel_3.4.1   broom_0.4.2
R> [25] Rcpp_0.12.12     backports_1.1.0  scales_0.5.0     jsonlite_1.5
R> [29] mnormt_1.5-5     hms_0.3          digest_0.6.12    stringi_1.1.5
R> [33] bookdown_0.5     grid_3.4.1       rprojroot_1.2    tools_3.4.1
R> [37] magrittr_1.5     lazyeval_0.2.0   pkgconfig_2.0.1  xml2_1.1.1
R> [41] assertthat_0.2.0 rmarkdown_1.6    httr_1.3.1       R6_2.2.2
R> [45] nlme_3.1-131     compiler_3.4.1
```

Strange events permit themselves the luxury of occurring.

Charlie Chan

Without data you're just another person with an opinion.

W. Edwards Deming

2 RStudio

2.1 Setting up the workspace

2.1.1 General settings

Before we start using RStudio (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.

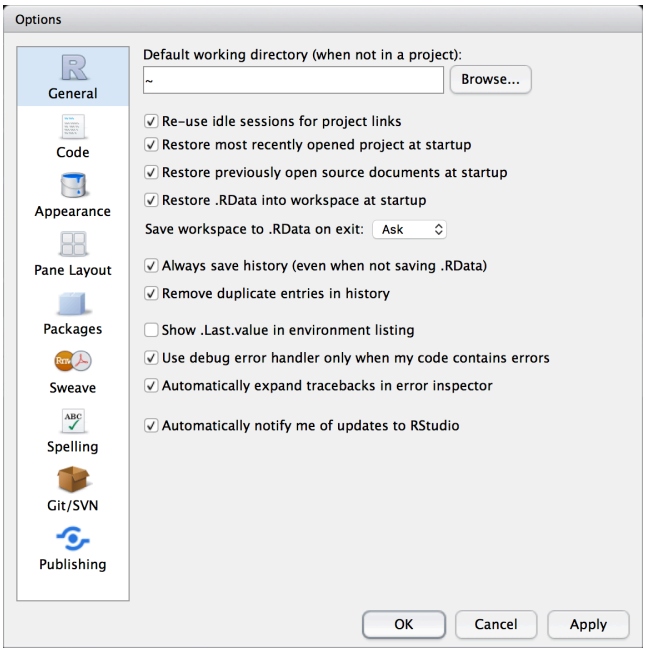


Figure 2.1: The RStudio Preferences menu.

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 RProjects

A very nifty way of managing workflow in RStudio is through the built-in functionality of RProjects. 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 sandard. Better yet, RProjects integrate seemlessly into version control software (e.g. GitHub) and allow 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 RProjects 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 (we'll be writing our own functions soon!) 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[1]. There are currently (9 April 2017) 10,413 packages available for R, with more than 200,000 functions in them!

If the thought of searching for and finding R packages is daunting, a good place to start is the R Task View[2] page. This page curates collections of packages for general tasks you might encounter, such as Experimental Design, 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 of the installation on your hard drive 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)
```

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

[1] https://cran.r-project.org/web/packages/
[2] http://cran.r-project.org/web/views/

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 an/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
R> [1] FALSE
```

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.

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] 4.53
sd(apples)
R> [1] 0.751
```

> ***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: 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)

2.4.3 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"                 "activities.df"      "apples"
R>  [4] "b"                 "begin_figure"       "columns"
R>  [7] "d"                 "dist.points"        "distance.per.month"
R> [10] "draw_legends"      "end_figure"         "include_graphics"
R> [13] "is_latex"          "knitr_first_plot"   "knitr_last_plot"
R> [16] "location.clean"    "plot_hook_bookdown" "shift.vec"
R> [19] "version"
```

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?

2.4.4 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)
```

To see the plots reproduced in Figures 2.2 and 2.3, simply type the following into the **Console**:

```
x <- seq(0, 2, by = 0.01)
y <- 2 * sin(2 * pi * (x - 1/4))
plot(x, y, col = "salmon", cex = 0.4)
```

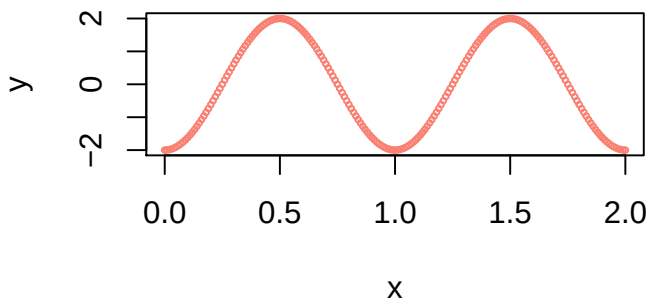


Figure 2.2: An example R plot done in base graphics.

```
# library(ggplot2)
ggplot() +
  geom_point(aes(x = x, y = y), shape = 21, col = "salmon", fill = "white")
```

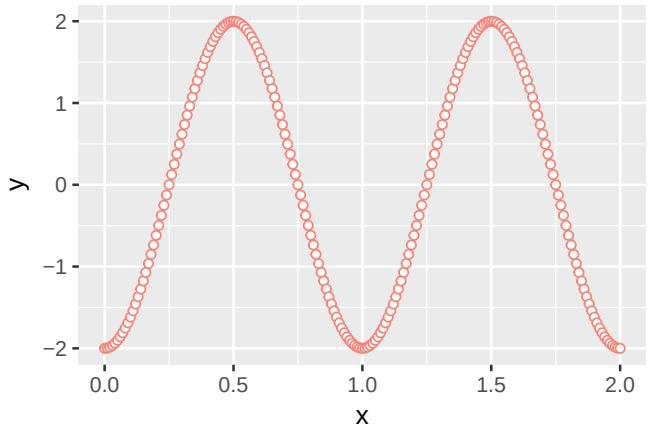


Figure 2.3: The same plot as above, but assembled with `ggplot2`.

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

3

An R workflow

3.1 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 `dat`, 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 mor-

phometric properties of the plants (*e.g.* the mass of the fronds (`blade_weight`), the frond length (`blade_length`), *etc.*).

3.2 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.

3.2.1 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.

3.2.2 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`.

3.2.3 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 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 statistical data. There are several other ways to read in data, but for the purposes of this Course, 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.

There are many arguments that you can use to customise reading of your data, but most important are:

1. `file`: the name of the data file to be read (this needs to include its path if it is not in your specified working directory); note that file names must be placed within quotation marks
2. `header`: is a logical argument (`TRUE`/`FALSE`) that specifies whether R reads the first line of your file as the names of the variables it contains
3. `quote`: By default, character strings can be quoted by either single ('...') or double ("...") quotes and usually do not need to be changed when exporting data as .csv from Excel.

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.table()`, which itself allows you to read in many types of files besides `.csv`. To find out more, type `?read.table` (see the output in Figure 3.1).

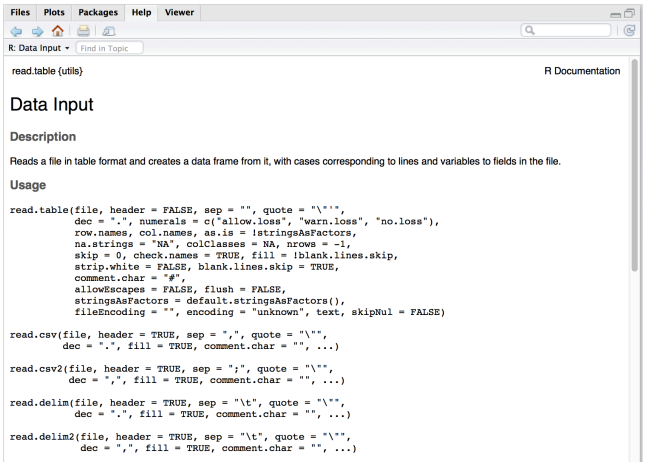


Figure 3.1: A portion of the help page produced by the above command.

3.2.4 Loading a file

To load the `laminaria.csv` file we created, and assign it to an object name in R, enter the following code into the R console:

```
dat <- read.csv("data/laminaria.csv")
```

If one clicks on the newly created `dat` 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(dat)
```

This will of course delete our variable, so we will import it in again:

```
dat <- read.csv("data/laminaria.csv")
```

Managing variables

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

3.3 Working with data

3.3.1 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(dat)
```

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(dat, n = 3)
tail(dat, n = 2)
```

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

```
glimpse(dat)
```

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(dat)
```

3.3.2 Tidyverse sneak peek

Before we begin to manipulate our data further we need to briefly introduce ourselves to the `tidyverse`. In order to do so we must first install and activate this package with the following code if we have not already done so for Chapter 2:

```
# install.packages("tidyverse")
library(tidyverse)
```

We must then introduce 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 functions:

- Pick observations (rows) by their values (`filter()`).
- Pick variables (columns) by their names (`select()`).
- Create new variables (columns) from existing ones (`mutate()`).
- Create groupings within a dataset (`group_by()`).

- Collapse many values in a group down to a single summary (`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.

3.3.3 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:

```
dat %>% # 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.

```
dat %>%
  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:

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

The function `filter()` has two arguments: the first is a dataframe (we specify `dat` 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.

```
dat_kom <- dat %>%
  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?

3.3.4 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:

```
dat %>% # 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:

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

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 April 2017

# 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
```

```
dat <- read.csv("data/laminaria.csv")

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

# Subsetting data
dat %>% # 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?
dat %>%
  filter(site == "Kommetjie") %>%
  nrow()

# The row with the greatest length
dat %>% # 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.

3.3.5 Summary of all variables in a dataframe

Import the data into a dataframe called `dat` 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(dat)
```

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.

3.3.6 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:

```
dat %>% # Chose the dataframe
  summarise(mean = 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:

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

```
dat %>%
  summarise(mean = 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:

```
dat %>%
  summarise(mean = 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.

3.3.7 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.

```
dat %>% # Select 'dat'
  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:

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

then:

```
dat %>%
  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.

3.3.8 Factor summaries

These sorts of summary statistics are fine for continuous variables, but we might want something a little different for factors (*i.e.*, categorical variables with different levels). Try typing:

```
dat %>%
  select(site) %>%
  table()
```

This returns a table with a column corresponding to each unique level of the categorical variable `site`. In the first line is the value of the level (in this case, the names of the places where

the kelps were collected); in the second line is the frequency of occurrence of that level in the variable (the number of kelps measured at each site).

Of course, this is easily extended to two-way tables. Try typing:

```
dat %>%
  select(region, site) %>%
  table()
```

This lists the number of observations at each `site` and at each `region`, with `sites` as columns, and `regions` as rows.

You can also of course assign the results of a call to `table()` to an object. Try:

```
site_by_region <- dat %>%
  select(region, site) %>%
  table()
```

For higher-order data summaries, `ftable()` is useful for creating "flat contingency tables":

```
dat %>%
  select(region, site) %>%
  ftable()
```

3.4 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:

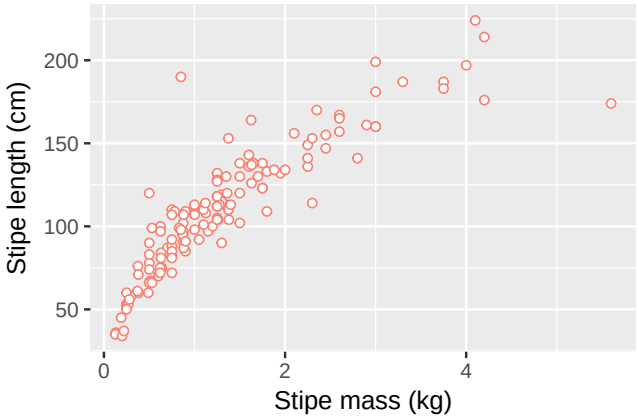
```
write.csv(site_by_region, file = "data/kelp_summary.csv", row.names = TRUE)
```

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 last argument simply tells R to add a column of values specifying row names (in this case, the names of the regions). Of course, if you don't want row names (which might be the case if you were writing the whole dataframe to a file), simply replace the `TRUE` with `FALSE`. The resultant file can sadly be opened in Excel.

3.5 Graphics

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



3.6 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.

We already know how to list and delete objects, but often you will want to dump all objects from memory; this is particularly useful to have at the start and end of a script. This can be done by typing:

```
rm(list = ls())
```

The parameter `list` provides a list of objects to be deleted (you could combine all existing object names together in quotes using `c()`, but more easily the function `ls()` gives all the object names in memory). 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.

3.7 Working directories

At the beginning of this session we glossed over this topic by setting the working directory via RStudios 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 laptop):

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

It will be different for you, but copy it into your script 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.

Absence of evidence is not evidence of absence.

Carl Sagan

Correlation doesn't imply causation, but it does waggle its eyebrows suggestively and gesture furtively while mouthing 'look over there'.

Anomynous

4

A primer on R

Please note that the following chapter departs from the syntax employed by the `tidyverse`, as utilised throughout this workshop, in favour of the base R syntax. This may be changed in the future, but has been left for now in order to better highlight the fundamental machinations of the R language, upon which the `tidyverse` is based.

4.1 Dataframes

The "workhorse" data-containing structures you will use extensively in R are called *dataframes*. In fact, almost all of the work you do in R will be done directly with dataframes or will involve converting data into a dataframe. A dataframe is used for storing data as tables, with a table defined by a collection of vectors of similar or dissimilar data types but all of the same length. Don't worry if any of those terms are unknown or daunting. We will cover them in detail just now. But first we need to see what a dataframe looks like in order to provide context for all of the parts they consist of. After we have covered all of the terms used for data in R we will learn some methods of creating our own dataframes.

To load a dataframe into R is quite simple when the data are already in the ".Rdata" format. Let's load a small dataframe that was prepared for this class and see. The file extension ".Rdata" does not mean necessarily that the data are in a dataframe (table) format. This file extensions is actually a form of data compression unique to R and could hold anything from a single letter to the results of a complex species distribution model. For the following line of code to work we must make sure we are in the "Intro_R_Workshop" project.

```
load("data/intro_data.Rdata")
```

Upon loading the data frame we see in the **Environment** tab that there is a little blue circle next to our object. If we click on that we see a summary of each column. First it says what the data type for that column is and then shows the first several values therein.

If you click on the "intro_data" word in your **Environment** tab it will open it in your **Source Editor** and allow you to click on the columns to organise them by ascending or descending

order. Note that this does not change the dataframe, it is only a visual aid.

4.2 Basic data types

There are several basic R data types that you frequently encounter in daily work. These include but are not limited to *numeric*, *integer*, *logical*, *character*, *factor* and *date* classes. All of these data types are present in our "intro_data" dataframe for us to see practical examples. We will create our own examples as we go along.

4.2.1 Numeric

Numeric data with decimal values are called *numeric* in R. It is the default computational data type. If we look at our data frame we see that the following columns are numeric: lon, lat, NA.perc, mean, min and max. What sort of data are these?

Let's create our own numeric object by assigning a decimal value to a variable `x` as follows, `x` will be of numeric type:

```
x <- 1.2 # assign 1.2 to x
x # print the value of x
R> [1] 1.2
class(x) # what is the class of x?
R> [1] "numeric"
```

Furthermore, even if we assign a number to a variable `k` that doesn't have a decimal place, it is still being saved as a numeric value:

```
k <- 1
k
R> [1] 1
class(k)
R> [1] "numeric"
```

If we want to really be certain that `k` is or is not an integer we use `is.integer()`:

```
is.integer(k) # is k an integer?
R> [1] FALSE
```

4.2.2 Integer

An *integer* in R is a numeric value that does not have a decimal place. It may only be a round whole number. Integers are often used for count data and when converting qualitative data to numbers for data analysis. In our dataframe we may see that we have two integer columns: depth and length. Why are these integers?

In order to create your own integer variable(s) in R, we use the `as.integer()`. We can be assured that `y` is indeed an integer by checking with `is.integer()`:

```
y <- as.integer(13)
y
R> [1] 13
class(y)
R> [1] "integer"
```

```
is.integer(y) # is it an integer?
R> [1] TRUE
```

If we really have to, we can coerce a numeric value into an integer with the same `as.integer()` function:

```
z <- as.integer(pi)
z
R> [1] 3
class(z)
R> [1] "integer"
is.integer(z) # is it an integer?
R> [1] TRUE
```

4.2.3 Logical

There are several *logic* values in R. We are mostly going to be concerned with the two main values we will be encountering: TRUE and FALSE. Note that all letters must be upper case. In our dataframe we see that only the "thermo" column is logical. This column tells us whether or not the data were collected with a thermometer or not.

Logical values (`TRUE` or `FALSE`) are often created via comparison between variables:

```
x <- 1; y <- 2 # sample values
z <- x > y
z
R> [1] FALSE
class(z)
R> [1] "logical"
```

In order to perform logical operations we mostly use `&` (and), `|` (or), and `!` (negation):

```
u <- TRUE; v <- FALSE; w <- TRUE; x <- FALSE
u & v
R> [1] FALSE
u & w
R> [1] TRUE
v & x
R> [1] FALSE
u | v
R> [1] TRUE
!u
R> [1] FALSE
```

Although these logical operators can be immensely useful in more advanced R programming, we will not go into too much detail in this introductory course. For more information on the logical operators, see the R help material:

```
help("&")
```

One final thing to note about logic in R is that it can be useful to perform arithmetic on logical values. `TRUE` has the value 1, while `FALSE` has value 0:

```
as.integer(TRUE) # the numeric value of TRUE
R> [1] 1
```

```
as.integer(FALSE) # the numeric value of FALSE
R> [1] 0
sum(as.integer(intro_data$thermo))
R> [1] 10
```

What is this telling us?

4.2.4 Character

In our dataframe we see that only the "src" column has the *character* values. This column is showing us which government body etc. collected the data in that row. At the use of a very familiar word, character, one may think this data type must be the most straightforward. This is not necessarily so as character values are used to represent *string* values in R. Because computers do not understand text the same way we do, they tend to handle this information differently. This allows us to do some pretty wild stuff with character values, but we won't be getting into that in this course as it quickly becomes very technical and generally speaking isn't very useful in a daily application.

If however we wanted to convert an object to a character value we would do so with `as.character()`:

```
d <- as.character(pi)
class(d)
R> [1] "character"
```

This can be useful if you have data that you want to be characters, but for one reason or another R has decided to make it a different data type.

If you want to join two character objects they can be concatenated with the `paste()` function:

```
a <- "fluffy"; b <- "bunny"
paste(a, b)
R> [1] "fluffy bunny"
paste(a, b, sep = "-")
R> [1] "fluffy-bunny"
```

More functions for string manipulation can be found in the R documentation — type `help("sub")` at the command prompt. You may also wish to install Hadley Wickham's nifty `stringr` package for more cool ways to work with character strings.

4.2.5 Factor

Factor values are somewhat difficult to explain and often even more difficult to understand. Factor values appear the same as character values when we look at them in a spreadsheet. But they are not the same. This will lead to much wailing and gnashing of teeth. So why then do factors exist and why would we use them? Factors allow us to numerically order names non-alphabetically, for example. This then allows one to order a list of research sites in geographical order.

We will see many examples of factors during this course but for now look at the "site" column in our dataframe. If we click on this column a couple of times we see that it reorders all the data based on ascending or descending order of the sites. But that order is not alphabetical, it is based on the *levels* within the factor column. Each factor value in a column is assigned a

level integer value (e.g. 1, 2, 3, 4, etc.). If multiple values in a factor column are the same, they receive the same level value as well.

If we want to see what the levels within a factor column are we use `levels()`:

```
levels(intro_data$site)
R>  [1] "Port Nolloth"  "St Helena Bay" "Saldanha Bay"  "Muizenberg"
R>  [5] "Cape Agulhas"  "Mossel Bay"    "Tsitsikamma"   "Humewood"
R>  [9] "Hamburg"       "Durban"        "Richards Bay"  "Sodwana"
```

We will discuss in the next session what that `$` means. But for now, are you able to see what the pattern is in the levels of the site listing?

If we want to create our own factors we will use `as.factor()`:

```
f <- as.factor(letters[1:5])
levels(f)
R> [1] "a" "b" "c" "d" "e"
```

And if we want to change the order of our factor levels we use `factor()`:

```
f <- factor(f, levels = c("b", "a", "c", "e", "d"))
levels(f)
R> [1] "b" "a" "c" "e" "d"
```

Another reason for using factors to re-order our data, as we shall see tomorrow, is that this allows us to control the order in which values are plotted.

4.2.6 Dates

Figure 4.1: Dates.

4.3 Vectors

A vector, by definition, is a one-dimensional sequence of data elements of the same basic type (class). Members in a vector are officially called components. Basically, a vector is a column. Indeed, a dataframe is nothing more than a collection of vectors stuck together. If we wanted to create a vector from our dataframe we would do this:

```
lonely_vector <- intro_data$NA.perc
```

Notice that we may not click on the object `lonely_vector` in our **Environment** tab. This is because it is no longer two-dimensional. If we want to visualise the data we need to enter it into the console or run it from our script:

```
lonely_vector
R>  [1]  6 41 32  4 28 26  8  3  6 67 38 16
```

Let's create some vectors of our own:

```
primes1 <- c(3, 5, 7)
primes1
R> [1] 3 5 7
class(primes1)
R> [1] "numeric"

p1 <- pi
p2 <- 5
p3 <- 7

primes2 <- c(p1, p2, p3)
primes2
R> [1] 3.14 5.00 7.00
class(primes2)
R> [1] "numeric"
is.numeric(primes2)
R> [1] TRUE
is.integer(primes2) # integers coerced into floating point numbers
R> [1] FALSE
```

We can also have vectors of logical values or character strings, and we can use the function `length()` to see how many components each has:

```
tf <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
tf
R> [1]  TRUE FALSE  TRUE FALSE FALSE
length(tf)
R> [1] 5
cs <- c("Mary", "has", "a", "silly", "lamb")
cs
R> [1] "Mary"  "has"   "a"     "silly" "lamb"
length(cs)
R> [1] 5
```

Of course one would seldom enter data into R using the `c()` (combine) function, but it is useful for short calculations. More often than not one would import data from Excel (urgh!) or

something more reputable. The kinds of data one can read into R are remarkable. We will get to that later on.

We can also combine vectors in many ways, and the simplest way is the append one after the other:

```
primes12 <- c(primes1, primes2)
primes12
R> [1] 3.00 5.00 7.00 3.14 5.00 7.00

nonSense <- c(primes12, cs)
nonSense
R>  [1] "3"                "5"                "7"
R>  [4] "3.14159265358979" "5"                "7"
R>  [7] "Mary"             "has"              "a"
R> [10] "silly"            "lamb"
class(nonSense)
R> [1] "character"
```

In the code fragment above, notice how the numeric values are being coerced into character strings when the two vectors of dissimilar class are combined. This is necessary so as to maintain the same primitive data type for members in the same vector.

4.4 Vector indices

What if we want to extract one or a few components from the vector? Easy… We retrieve values in a vector by declaring an index inside a single square bracket `[]` operator. For example, the following shows how to retrieve a vector component. Since the vector index is 1-based (*i.e.* the first component in a vector is numbered `1`), we use the index position `7` for retrieving the seventh member:

```
nonSense[7] # find the seventh component in the vector
R> [1] "Mary"
# or combine them in interesting ways...
paste(nonSense[7], nonSense[8], nonSense[4], nonSense[10], "bunnies", sep = " ")
R> [1] "Mary has 3.14159265358979 silly bunnies"
```

If the index given is negative, it will remove the value whose position has the same absolute value as the negative index. For example, the following creates a vector slice with the third member removed. However, if an index is out-of-range, a missing value will be reported via the symbol `NA`:

```
a <- c(2, 6, 3, 8, 13)
a
R> [1]  2  6  3  8 13
a[-3]
R> [1]  2  6  8 13
a[10]
R> [1] NA
```

4.5 Vector creation

R has many funky ways of creating vectors. This process is important to understand because we will need to build on it to create our own dataframes. Here are some examples of vector creation:

```
seq(1:10) # assign them to a variable if you want to...
R>  [1]  1  2  3  4  5  6  7  8  9 10
seq(from = 0, to = 100, by = 10)
R>  [1]   0  10  20  30  40  50  60  70  80  90 100
seq(0, 100, len = 10) # one may omit from and to
R>  [1]   0.0  11.1  22.2  33.3  44.4  55.6  66.7  77.8  88.9 100.0
seq(1, 9, by = pi)
R> [1] 1.00 4.14 7.28
rep(13, times = 13)
R>  [1] 13 13 13 13 13 13 13 13 13 13 13 13 13
rep(seq(1:5), times = 6)
R>  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
a <- rnorm(20, mean = 13, sd = 0.13) # random numbers with known mean and sd
rep(a, 5) # one may omit the times argument
R>   [1] 13.1 12.9 13.2 12.9 13.1 12.9 13.1 13.0 13.1 13.1 13.0 13.0 13.3
R>  [14] 12.9 12.9 13.1 12.9 12.6 12.9 12.9 13.1 12.9 13.2 12.9 13.1 12.9
R>  [27] 13.1 13.0 13.1 13.1 13.0 13.0 13.3 12.9 12.9 13.1 12.9 12.6 12.9
R>  [40] 12.9 13.1 12.9 13.2 12.9 13.1 12.9 13.1 13.0 13.1 13.1 13.0 13.0
R>  [53] 13.3 12.9 12.9 13.1 12.9 12.6 12.9 12.9 13.1 12.9 13.2 12.9 13.1
R>  [66] 12.9 13.1 13.0 13.1 13.1 13.0 13.0 13.3 12.9 12.9 13.1 12.9 12.6
R>  [79] 12.9 12.9 13.1 12.9 13.2 12.9 13.1 12.9 13.1 13.0 13.1 13.1 13.0
R>  [92] 13.0 13.3 12.9 12.9 13.1 12.9 12.6 12.9 12.9
rep(c("A", "B", "C"), 3)
R> [1] "A" "B" "C" "A" "B" "C" "A" "B" "C"
rep(c("A", "B", "C"), each = 3)
R> [1] "A" "A" "A" "B" "B" "B" "C" "C" "C"
x <- c("01-31-1960", "02-13-1960", "06-23-1977", "01-01-2013")
class(x)
R> [1] "character"
z <- as.Date(x, "%m-%d-%Y")
class(z) # introducing the date class
R> [1] "Date"
seq(as.Date("2013-12-30"), as.Date("2014-01-04"), by = "days")
R> [1] "2013-12-30" "2013-12-31" "2014-01-01" "2014-01-02" "2014-01-03"
R> [6] "2014-01-04"
seq(as.Date("2013-12-01"), as.Date("2016-01-31"), by = "months")
R>  [1] "2013-12-01" "2014-01-01" "2014-02-01" "2014-03-01" "2014-04-01"
R>  [6] "2014-05-01" "2014-06-01" "2014-07-01" "2014-08-01" "2014-09-01"
R> [11] "2014-10-01" "2014-11-01" "2014-12-01" "2015-01-01" "2015-02-01"
R> [16] "2015-03-01" "2015-04-01" "2015-05-01" "2015-06-01" "2015-07-01"
R> [21] "2015-08-01" "2015-09-01" "2015-10-01" "2015-11-01" "2015-12-01"
R> [26] "2016-01-01"
seq(as.Date("2000/1/1"), by = "month", length.out = 12)
R>  [1] "2000-01-01" "2000-02-01" "2000-03-01" "2000-04-01" "2000-05-01"
R>  [6] "2000-06-01" "2000-07-01" "2000-08-01" "2000-09-01" "2000-10-01"
```

```
R> [11] "2000-11-01" "2000-12-01"
# and many more...
```

4.6 Vector arithmetic

Arithmetic operations of vectors are performed component-by-component, *i.e.*, componentwise. For example, suppose we have vectors `a` and `b`:

```
a <- c(1, 3, 5, 7)
b <- c(1, 2, 4, 8)
```

Then we multiply `a` by `5`…

```
a * 5
R> [1]  5 15 25 35
```

… and see that each component of `a` is multiplied by `5`. In other words, the shorter vector (here `5`) is recycled. Now multiply `a` with `b`…

```
a * b
R> [1]  1  6 20 56
```

…and we see that the components in one vector matches those in the other one-for-one. Similarly for subtraction, addition and division, we get new vectors via componentwise operations. Try this here now a few times with your own vectors.

But what if one vector is somewhat shorter than the other? The *recycling rule* comes into play. If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. For example, the following vectors `u` and `v` have different lengths, and their sum is computed by recycling values of the shorter vector `u`:

```
v <- rep(2, len = 13)
u <- rep(c(1, 20), len = 5)
v + u
R> Warning in v + u: longer object length is not a multiple of shorter
R> object length
R>  [1]  3 22  3 22  3  3 22  3 22  3  3 22  3
```

4.7 Dataframe creation

The most rudimentary way to create a dataframe is to create several vectors and then assemble them into a dataframe using `cbind()` — this is a function that combines by column. For instance:

```
# create three vectors of different types
vec1 <- rep(c("A", "B", "C"), each = 5) # a character vector (a facctor)
vec2 <- seq.Date(from = as.Date("1981-01-01"), by = "day",
                 length.out = length(vec1)) # date vector
vec3 <- rnorm(n = length(vec1), mean = 0, sd = 0.35) # numeric vector
# now assemble dataframe
df1 <- cbind(vec1, vec2, vec3)
head(df1)
R>      vec1 vec2   vec3
R> [1,] "A"  "4018" "0.0752432195138573"
```

```
R> [2,] "A"  "4019" "-0.396306874740361"
R> [3,] "A"  "4020" "0.236314939971823"
R> [4,] "A"  "4021" "-0.24164168866817"
R> [5,] "A"  "4022" "-0.0904063752236777"
R> [6,] "B"  "4023" "0.616698041153829"
```

Another way to achieve the same thing is to use the `data.frame()` function that will allow you to achieve all of the above steps at once. Here is the example:

```
df2 <- data.frame(vec1 = rep(c("A", "B", "C"), each = 5),
                  vec2 = seq.Date(from = as.Date("1981-01-01"), by = "day",
                                  length.out = length(vec1)),
                  vec3 = rnorm(n = length(vec1), mean = 2, sd = 0.75))
head(df2, 2)
R>   vec1       vec2 vec3
R> 1    A 1981-01-01 1.37
R> 2    A 1981-01-02 2.05
```

What about the names of the dataframe that you just created? Are you happy that they are descriptive enough? If you aren't, don't fear. There are several different ways in which we can change it. We can assign the existing separate vectors `vec1`, `vec2` and `vec3` to more user-friendly names using the `data.frame()` function, like this:

```
df1 <- data.frame(level = vec1,
                  sample.date = vec2,
                  measurement = vec3)
head(df1, 2)
R>   level sample.date measurement
R> 1     A  1981-01-01      0.0752
R> 2     A  1981-01-02     -0.3963
```

Another way is to change the name after you have created the dataframe using the `colnames()` assignment function, as in:

```
colnames(df2) <- c("level", "sample.date", "measurement")
head(df2, 2)
R>   level sample.date measurement
R> 1     A  1981-01-01        1.37
R> 2     A  1981-01-02        2.05
names(df2)
R> [1] "level"       "sample.date" "measurement"
```

Dataframes are very versatile and we can do many operations on them. A common requirement is to add a column to a dataframe that contains the outcome of some calculation. We could create a new column in the dataframe "on the fly", as in:

```
df2.1 <- df1 # copy the dataframe
df2.1$meas.anom <- df1$measurement - mean(df1$measurement)
df2.1$meas.diff <- df2.1$measurement - df2.1$meas.anom
head(df2.1, 2)
R>   level sample.date measurement meas.anom meas.diff
R> 1     A  1981-01-01      0.0752    0.0302     0.045
R> 2     A  1981-01-02     -0.3963   -0.4413     0.045
```

We can also combine dataframes in different ways. Perhaps you have two (or more) dataframe

that conform to the same layout, *i.e.* they have the same number of columns (although the length of the dataframes may differ), they have the same data type in those columns and the names of those columns are the same. Also, the order of the columns must be identical in all the dataframes. Two separate dataframe with the same structure may, for example, result from two identical experiments that were repeated at different times. We can then stack one on top (*e.g.* combine our experiments) of the other using the *r*ow bind function `rbind()`, as in:

```
nrow(df1) # check the number of rows first
R> [1] 15
nrow(df2)
R> [1] 15
df3 <- rbind(df1, df2)
nrow(df3) # number of rows in the combined dataframe
R> [1] 30
head(df3, 2)
R>   level sample.date measurement
R> 1     A  1981-01-01      0.0752
R> 2     A  1981-01-02     -0.3963
```

But now how do we know how the portions of the stacked dataframe relate to the experiments that resulted in the data in the first place? There is no label to distinguish one experiment from the other. We can fix this by adding a new column to the stacked dataframe that contains the coding for the two experiments. We can achieve it like this:

```
df3$exp.no <- rep(c("exp1", "exp2"), each = nrow(df1))
head(df3, 2)
R>   level sample.date measurement exp.no
R> 1     A  1981-01-01      0.0752   exp1
R> 2     A  1981-01-02     -0.3963   exp1
tail(df3, 2)
R>    level sample.date measurement exp.no
R> 29     C  1981-01-14        2.48   exp2
R> 30     C  1981-01-15        1.98   exp2
```

DIY: Make your own dataframes

For this task, please create two 'stackable' dataframes, but each with a different number of rows. Then stack them and assign an 'index column' so that the two original dataframes can easily be distinguished from one-another in the combined dataframe.

We can combine dataframes in another way — that is, bind columns side-by-side using the function `cbind()`. We used it before to place vectors of the same length next to each other to create a dataframe. This function is similar to `rbind()`, but where `rbind()` fusses over the names of the columns, `cbind()` does not. What does concern `cbind()`, however, is that the number of rows in the two (or more) dataframes that will be "glued" side-by-side is the same. Try it yourself with your own dataframes.

4.8 Dataframe indices

Remember that weird `$` symbol we saw a little while ago? That symbol tells R that you want to see a column (vector) within a dataframe. For example, if we wanted to perform an operation on only one column in `intro_data` in order to ascertain the mean depth (m) of sampling:

```
round(mean(intro_data$depth),2)
R> [1] 1.33
```

If we want to subset only specific values in a dataframe, as we have seen how to do with vectors, we need to consider that we are now working with two dimensions and not one. We still use `[]` but now we must do a little extra. If we want to see how long the time series for Sodwana is we could do this in several ways, here are the three most common in an improving order:

```
# Subset a dataframe using [,]
intro_data[12,9]
R> [1] 4606

# Subset only one column using []
intro_data$length[12]
R> [1] 4606

# Subset from one column using logic for another column
intro_data$length[intro_data$site == "Sodwana"]
R> [1] 4606
```

The important thing to remember here is that when one needs to use a comma when subsetting, the row number is always on the left, and the column number is always on the right. Rows then columns! Tattoo that onto your brain. Or fore-arm if you are the adventurous type. We will go into the subsetting and analysis of dataframes in much more detail in the following session.

One must keep in mind that data in R can become substantially more complex than what we have covered, and the software also distinguishes several other kinds of data "containers": in addition to vectors and dataframes, we also have *lists*, *matrices*, *time series* and *arrays*. The more complex ones, such as arrays, may have more dimensions than the two (rows along dimension 1, columns along dimension 2) that most people are familiar with. We will not delve into these here as they are bit more advanced than the goals of this course.

4.9 Useful information

4.9.1 Operators

There are several operators you can use to help build expressions as shown in Table 4.1.

Table 4.1: Logical operators for use in R.

Operator	Meaning	Example
`<`	less than	`x < 3`
`>`	greater than	`x > 5`
`>=`	greater than or equal to	`x >= 24`
`<=`	less than or equal to	`x <= 19`
`==`	exactly equal to	`x == 666`
`!=`	not equal to	`x != 777`
`&`	AND; returns TRUE if statements on both sides of the `&` are TRUE	`(x > 3) & (y != 5)`
`|`	OR; the pipe symbol `|`; TRUE if a statement on either side of the `|` is TRUE	`(x < 10) | (x > 20)`

Question

The logical AND: If we had said `subset(dat, site == "Kommetjie" & site == "Baboon")`, what would have happened? Why? Try it and see...

4.9.2 Functions

Some example functions covered so far are presented in Table 4.2.

Table 4.2: Common functions used for daily work in R.

Type	Function	What it does	Syntax
Checking a dataframe	`class()`	Gives the class (type of data structure) of the object	`class(dat)`
Numerical calculations	`mean()`, `sd()`	Calculations on a numerical R object (variables, vectors, arrays)	`mean(dat$var)`
Gets working directory	`getwd()`	Gets current working directory	`getwd()`
Sets working directory	`setwd()`	Sets working directory to specified path	`setwd("yourpath")`
Getting your data into R	`read.table()`	Reads file in table format (with specified separator sep) and creates a dataframe from it	`read.table("filename.csv", header= TRUE, sep = ",")`
Getting .csv file into R	`read.csv()`	Reads in .csv file to a dataframe; as above, but 'sep' predefined	`read.csv("filename.csv", header= TRUE)`
List variables	`ls()`	Lists objects in memory	`ls()`
Variable names in dataframe	`names()`	Gives variable names	`names(dat)`
Subsetting a dataframe	`subset()`	Subset a dataframe	`subset(dat, Var1 == "Low" | Var2 <= 5)`
Cleaning up	`rm()`	Removes object(s) from the Environment	`rm(object_name)`; `rm(list = ls())`

Some summary functions are presented in Table 4.3.

Table 4.3: Commonly used and useful summary functions.

Type	Function	What it does	Syntax
Specifying variable in dataframe	`$`	Gives the class (type of data structure) of the object	`dat$blade_length`
	`str()`	Lists variables in your object by name, their data type (continuous, factor) and an indication of the actual data	`str(dat)`
	`head()`	Gives first 6 rows of a dataframe, or number of rows (`n`) specified	`head(dat, 20)`
	`tail()`	Gives last 6 (or `n`) rows of a dataframe	`tail(dat, 10)`
	`names()`	Gives variable names	`names(dat)`
	`summary()`	Tabulates variables in dataframe and provides summary statistics	`summary(dat)`
<!–	`attach()`	Loads dataframe into memory	`attach(dat)`
–> <!–	`detach()`	Removes dataframe from memory	`detach(dat)`
–>	`with()`	Specify a dataframe	`with(dat, mean(blade_length))`
Summary statistics	`mean()`, `sd()`, `range()`, `var()`	Calculates particular statistic for a variable. Use `na.rm = TRUE` if your data contain `NA`s	*e.g.* `mean(dat$blade_length, na.rm = TRUE)`
	`length()`	Number of elements in a variable. Use `na.omit()` if your data contain `NA`s	`length(dat$blade_length)`; `length(na.omit(dat$blade_length))`
Data manipulation	`table()`	Frequency table by variables specified	`table(dat$site, dat$blade_length)`
	`ftable()`	Flat frequency table useful for displaying multidimensional tables	`ftable(dat$catvar1, dat$catvar2, dat$catvar3)`
Saving an object file	`write.csv()`	Save an object to a .csv file	`write.csv(sp_by_site, file = "Laminaria.csv", row.names = TRUE)`

Part II

Day 2

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

5

Graphics with `ggplot2`

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!

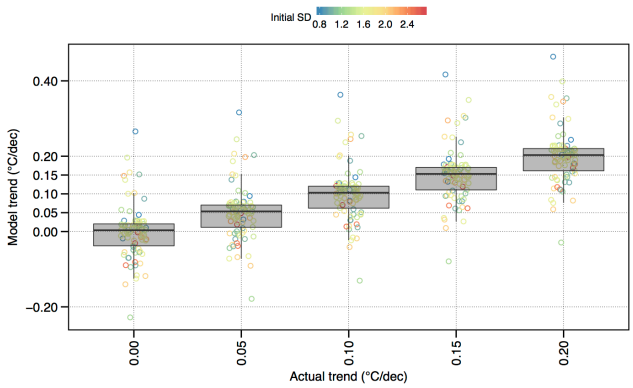


Figure 5.1: The effect of variance (SD) within a temperature time series on the accurate modelling of decadal trends.

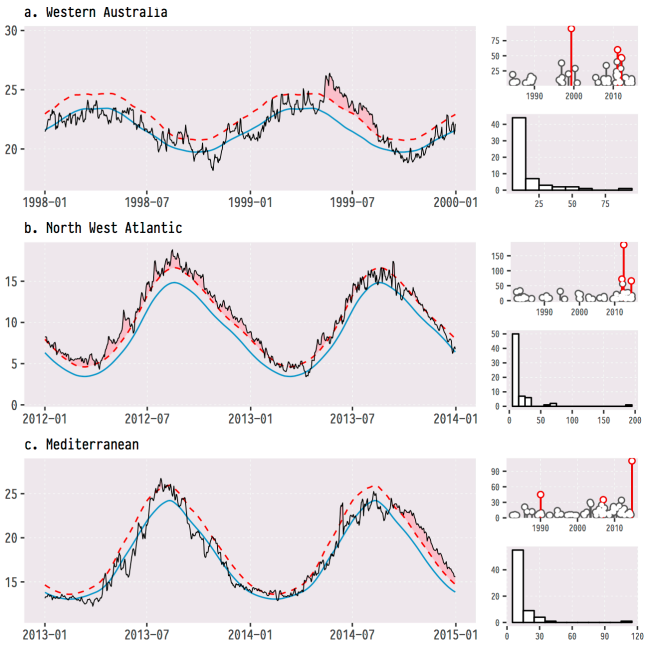


Figure 5.2: The (currently) three most infamous marine heatwaves (MHWs) around the world.

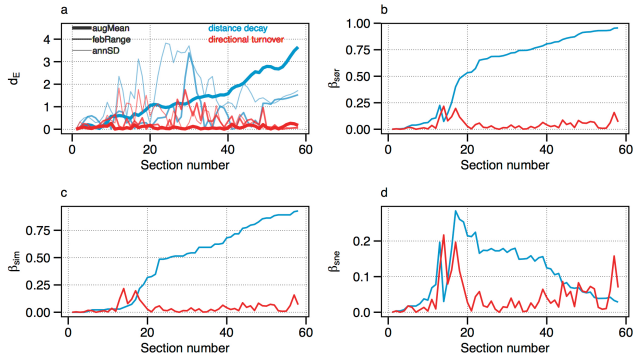


Figure 5.3: Changes in seaweed biodiversity along the South African coastline.

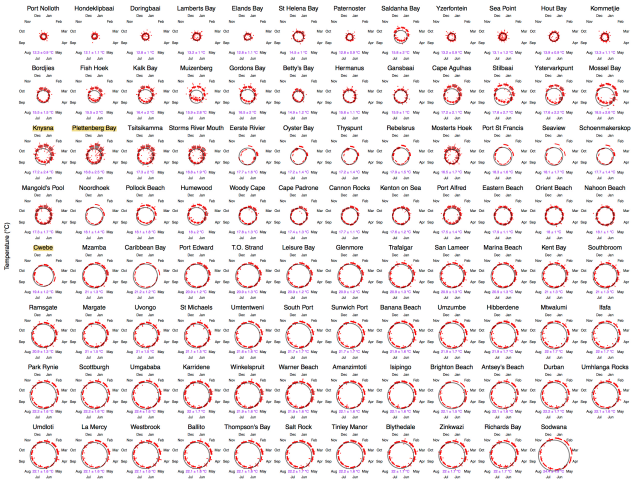


Figure 5.4: Polar plots of monthly temperautres.

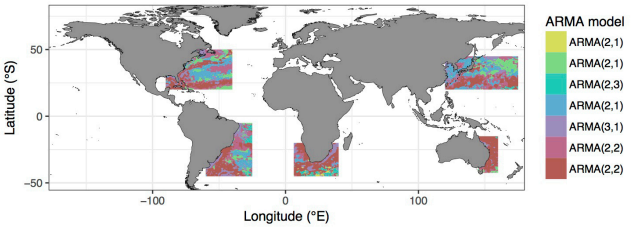


Figure 5.5: Most appropriate autoregressive correlation coefficients for areas around western boundary current.

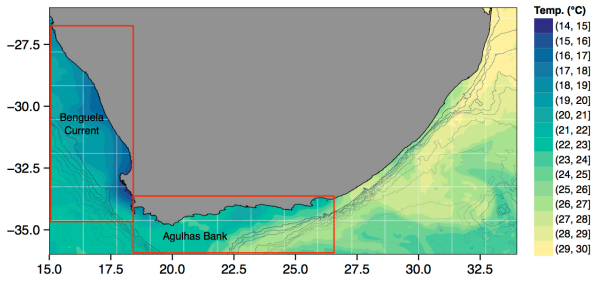


Figure 5.6: The bathymetry of South Africa with SSTs from the MUR product.

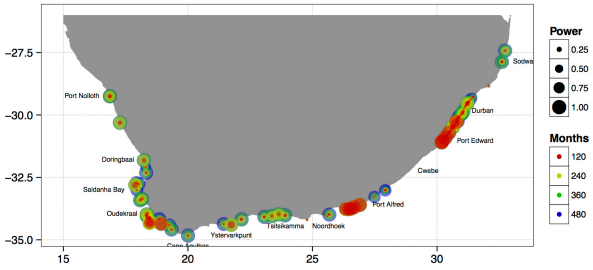


Figure 5.7: The power of the detected decadal trend at each coastal temperature collection site given a hypothetical number of months.

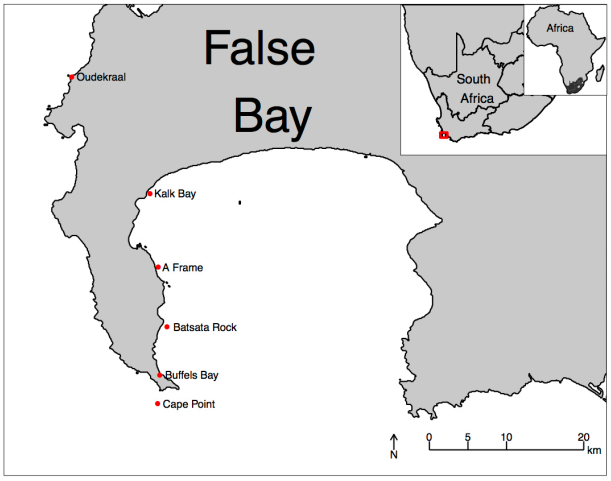


Figure 5.8: An inset map of False Bay.

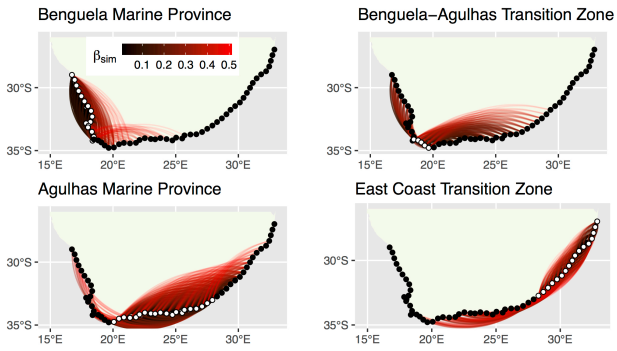


Figure 5.9: The strength of the relationship between each site based on their biodiversity.

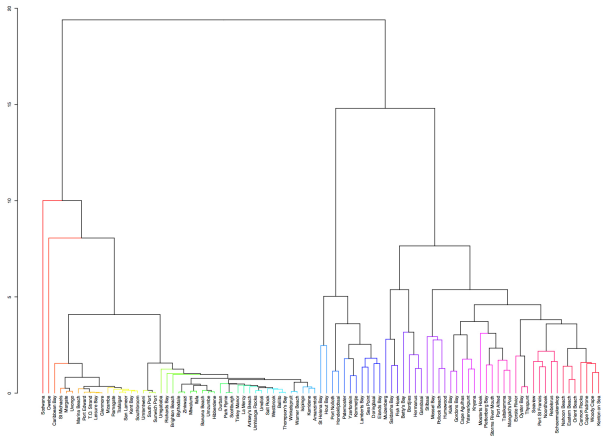


Figure 5.10: A hierarchical cluster analysis.

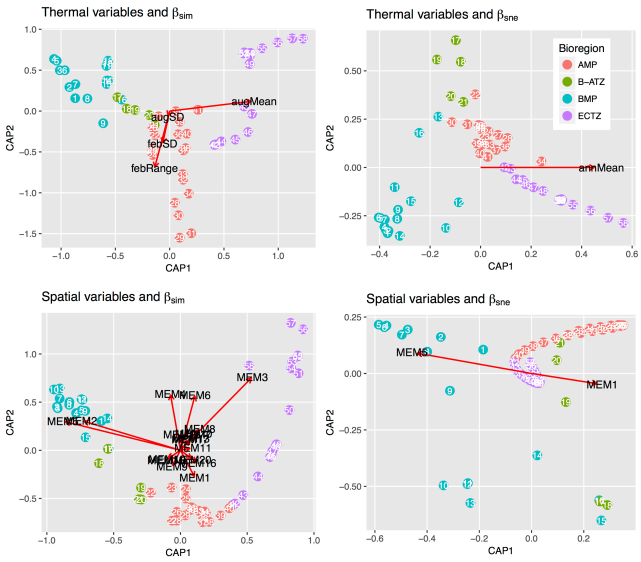


Figure 5.11: CAP analysis showing relationship between abiotic variables and beta diversity with the different sections of South Africa's coastline shown in colour.

5.2 Base R vs. `ggplot2`

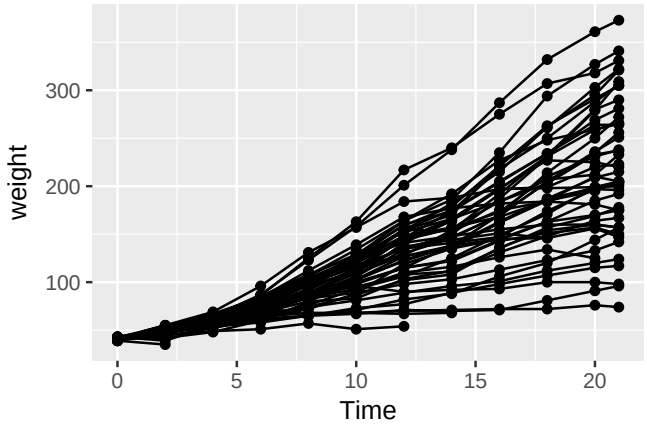
R comes with basic graphing capability, known colloquially (by nerds) as "Base R". The syntax used for this method of creating graphics is often difficult to interpret as there are few human words in the code. In addition to this issue, Base R also does not allow the user enough control over the look 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` (more on this 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)
library(gridExtra) # For creating grids of figures

# Load data
ChickWeight <- datasets::ChickWeight

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

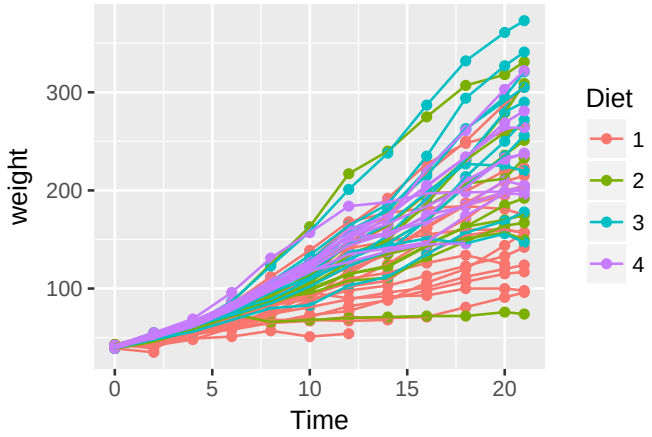


So what is that code doing? We may see from the figure that it is creating a little black dot for every data point, with the `Time` of sampling on the x axis, and the `weight` of the chicken during that time on the y axis. It then connects the dots for each chicken in the dataset. Let's break this code 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 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. The third line takes the same information and creates lines from it. 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.

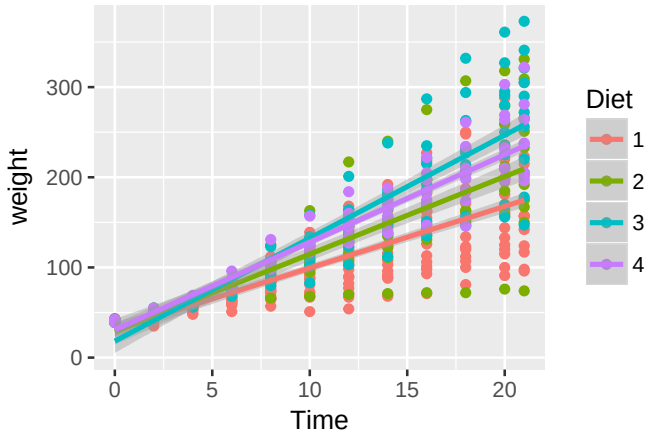
This figure doesn't look like much yet. We saw some examples above that show how sophisticated figures may become. This is a remarkably straight forward task. But don't take my word for it, let's see for ourselves. By adding one more aesthetic to the code above we will now show each `Diet` as a different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

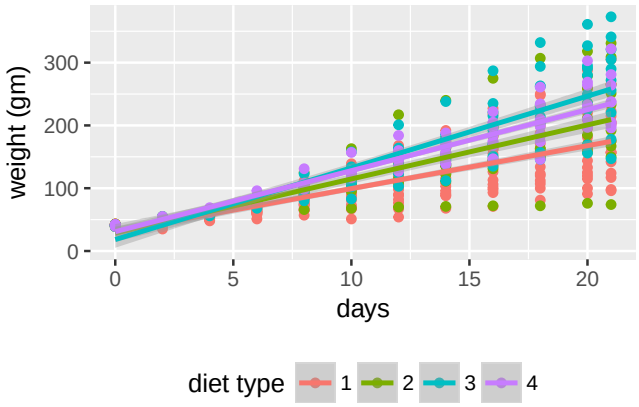


How is a linear model calculated? What patterns do we see in the data now? If you were a chicken, which feed would you want?

5.3 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 `lab()` 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 = "weight (gm)", colour = "diet type") + # Change the labels
  theme(legend.position = "bottom") # Change the legend position
```

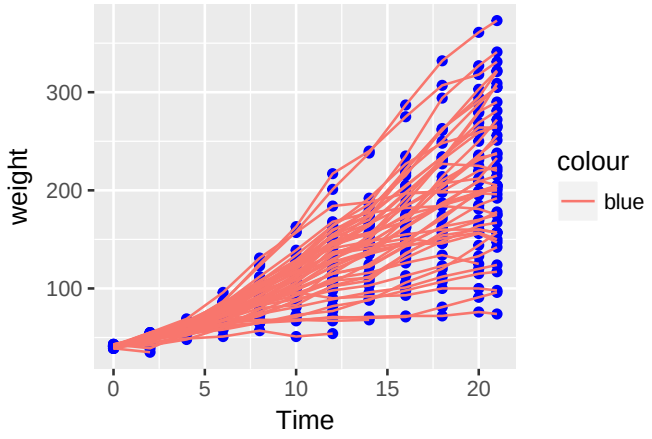


Notice that when we place the legend at the bottom of the figure ggplot automatically makes it horizontal for us. Why do we use "colour" inside of `labs()` to change the legend title?

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

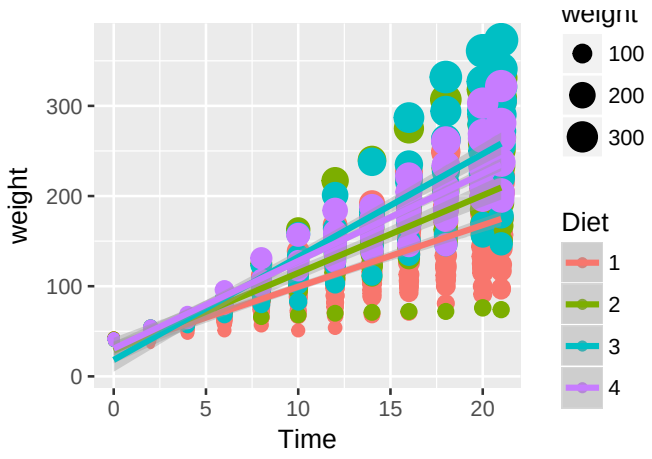
The astute eye will have noticed by now that most arguments we have added to the code have been inside of the `aes()` function. So what exactly is that `aes()` function doing sitting inside of the other functions? The reason for the `aes()` function is that it controls the look of the other functions 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 (jucier) 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.

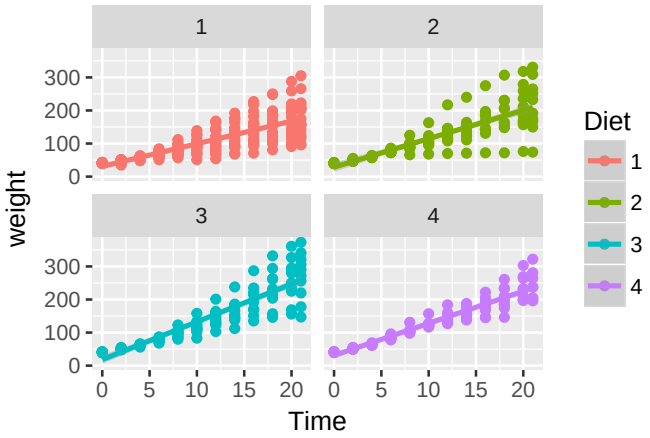
5.5 Faceting

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. We'll see if we can't convince you by the end of this course. For now however, let's just take our understanding of the usability of `ggplot2` one step further by looking at how to create grids of one figure, as well as combine different types of figure into one grid. To do this we are going to need to learn how to create a few new types of figures.

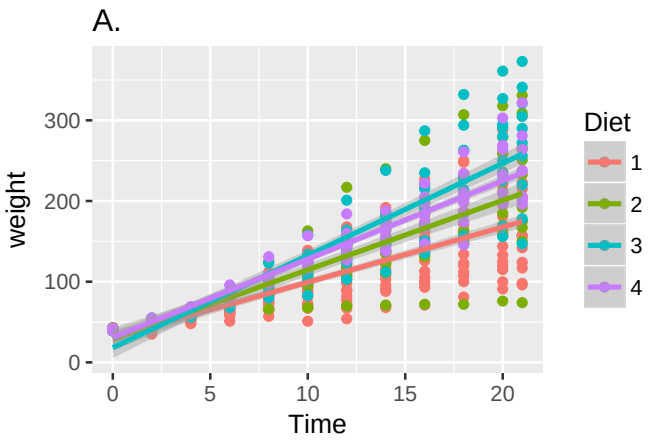
But first, here is how we would facet figure:

```
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
```



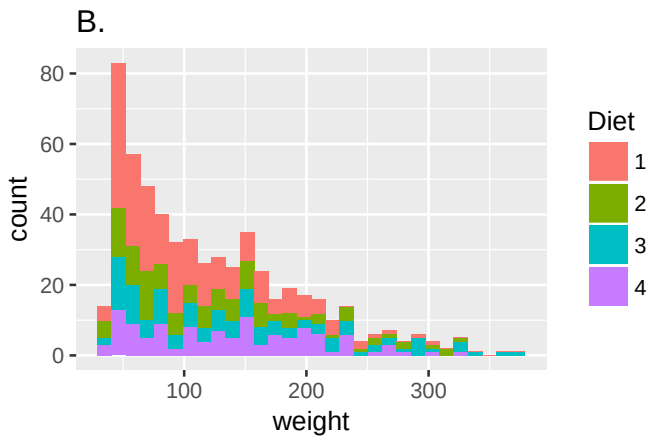
The linear model figure we've already created:

```
lm_1 <- ggplot(data = ChickWeight, aes(x = Time, y = weight, colour = Diet)) +
  geom_point() +
  geom_smooth(method = "lm") +
  ggtitle("A.") # Add a title
lm_1
```



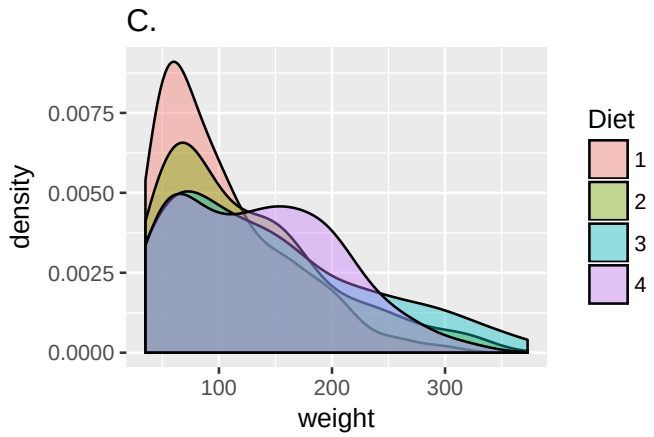
Howto create a histogram:

```
histogram_1 <- ggplot(data = ChickWeight, aes(x = weight)) +
  geom_histogram(aes(fill = Diet)) +
  ggtitle("B.") # Add a title
histogram_1
```



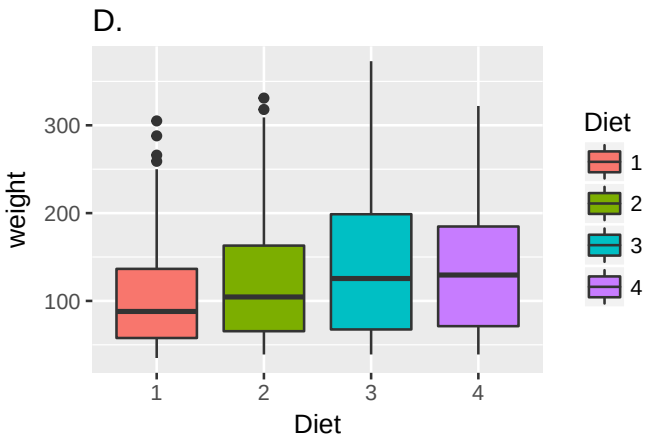
How to create a density plot:

```
density_1 <- ggplot(data = ChickWeight, aes(x = weight)) +
  geom_density(aes(fill = Diet), alpha = 0.4) +
  ggtitle("C.") # Add a title
density_1
```



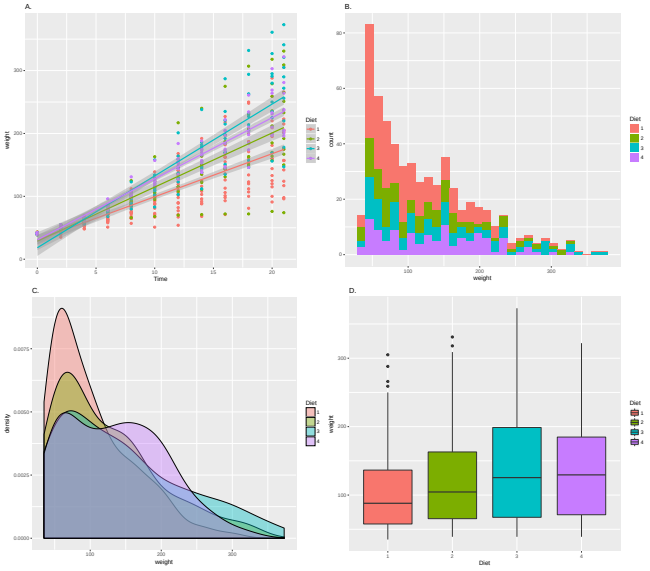
How to create a boxplot:

```
box_1 <- ggplot(data = ChickWeight, aes(x = Diet, y = weight)) +
  geom_boxplot(aes(fill = Diet)) +
  ggtitle("D.") # Add a title
box_1
```



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?

```
grid.arrange(lm_1,histogram_1, density_1, box_1, ncol = 2)
```



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 use the `ggsave()` function.

```
# First we must assign the code to an object name
grid_1 <- grid.arrange(lm_1,histogram_1, density_1, box_1, ncol = 2)

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

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

6

Brewing colours in ggplot2

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.

6.1 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 a new dataframe or twenty. To look at the data that we have available across all of our packages we use the following code:

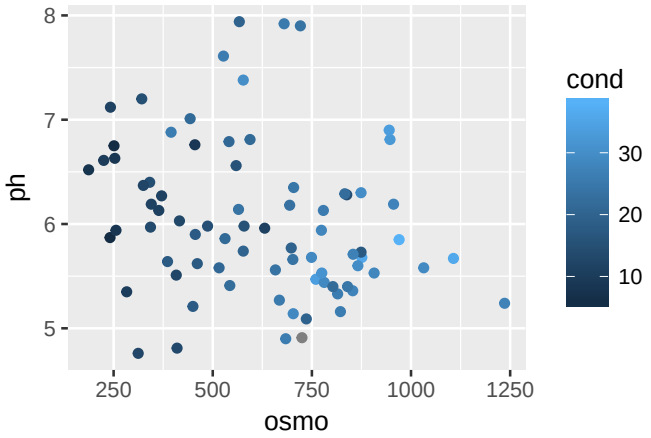
```
data(package = .packages(all.available = TRUE))
```

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. Let's take several minutes to look through all of the data available on our computers and pick out three that we like. One note of caution, in R there are two different forms of data. Wide and long. We will see in depth 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

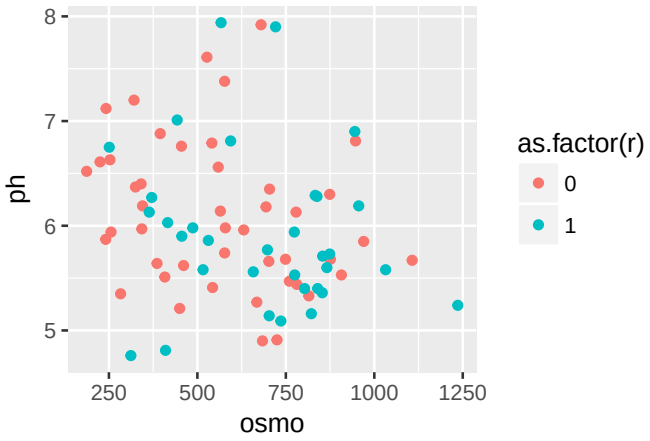
focus. I recommend installing it if you would like access to these data. I have decided to load the `urine` dataframe here. Note that `library(boot)` will not work on your computer if you have not installed the package yet. With these data we will now make a scatterplot with two of the variables, while changing the colour of the dots with a third variable.

```
# Load libraries
library(ggplot2)
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)))
```

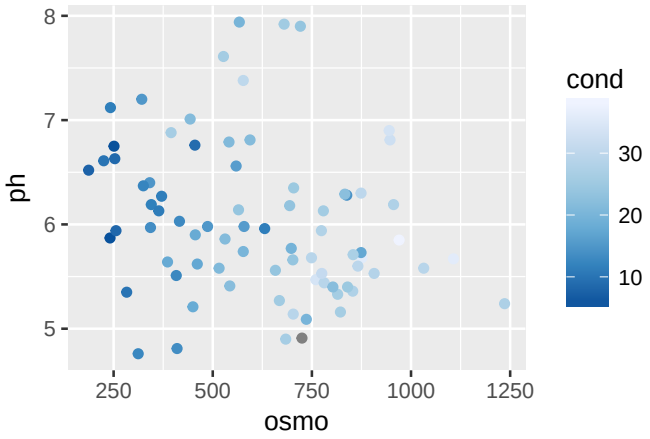


What is the first thing you notice about the difference in the colours? Why did we use `as.factor()` for the colour aesthetic for our points? What happens if we don't use this? Try it now.

6.2 RColorBrewer

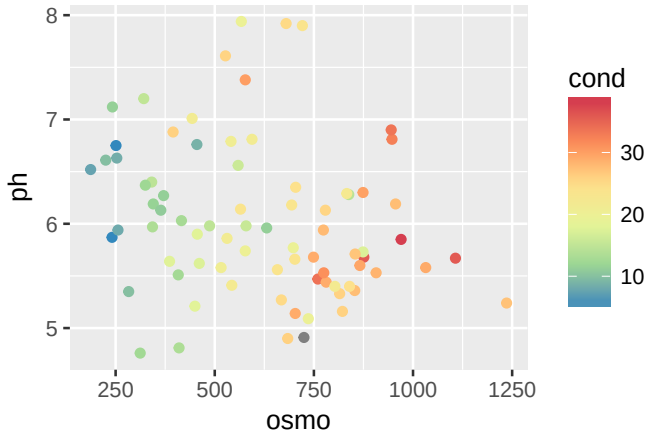
Central to the purpose of `ggplot2` is the creation of beautiful figures. For this reason there are many built in functions that we may use in order to have precise control over the colours we use, as well as additional packages that extend our options even further. The `RColorBrewer` package should 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 cheatsheet we included in the course material shows some different colour brewer palettes. Let's look at how to use those here.

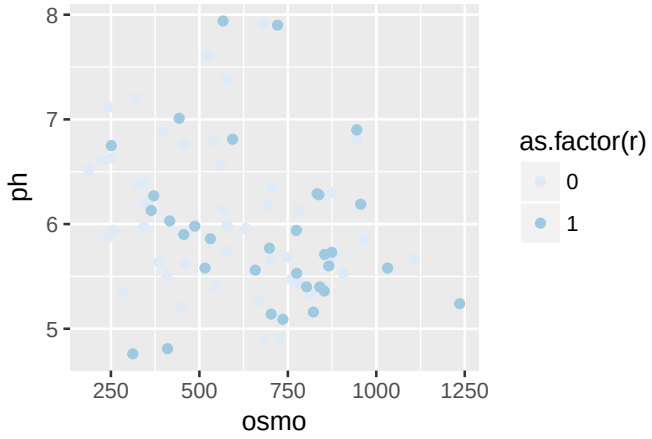
```
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = cond)) +
  scale_colour_distiller(palette = "Spectral")
```



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?

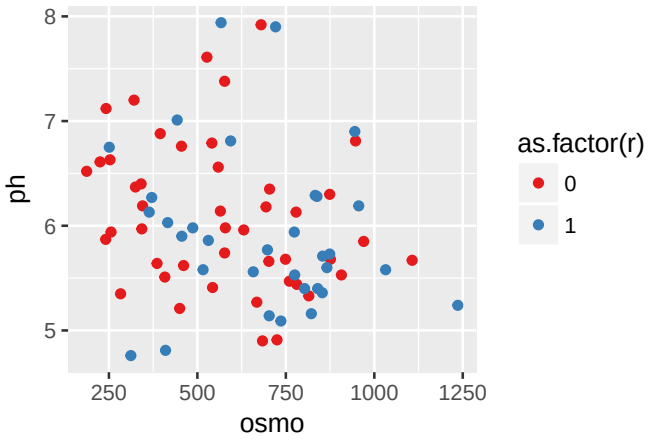
If we want to use colour brewer with a discrete variable we use a slightly different function.

```
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 cheatsheet 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
```



6.3 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!

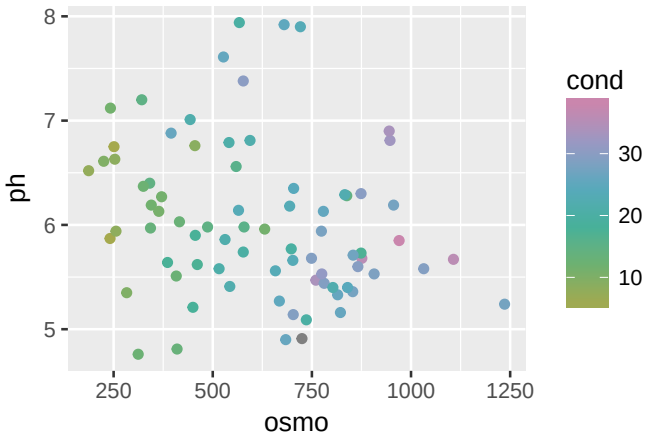
- http://tristen.ca/hcl-picker/#/hlc/6/0.95/48B4B6/345363
- http://tools.medialab.sciences-po.fr/iwanthue/index.php
- http://jsfiddle.net/d6wXV/6/embedded/result/

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.

6.4 Use your own palettes

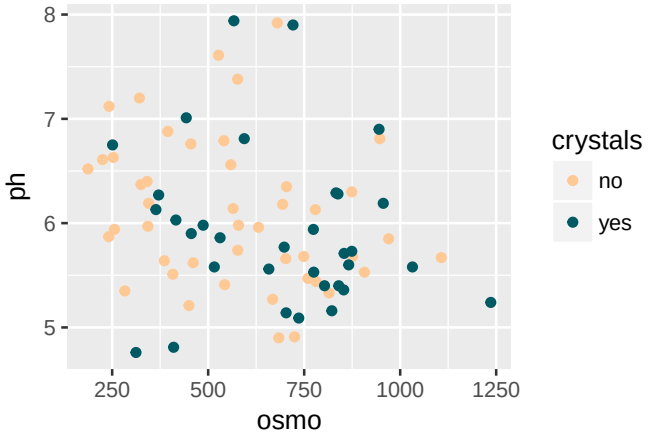
Now that we've had some time to play around with the colour generators let's look at how to use them with our figures. I've used the first web link to create a list of five colours. I then copy and pasted them into the code below, separating them with commas and placing them inside of `c()` and inverted 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.

```
ggplot(data = urine, aes(x = osmo, y = ph)) +
  geom_point(aes(colour = as.factor(r))) +
  scale_colour_manual(values = c("#ffc995", "#005a64"), # 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.

Part III

Day 3

To stay on the map you've got to keep showing up.

Peter Gallagher

There's no map to human behaviour.

Bjork

7

Mapping with ggplot2

Yesterday we learned how to create some basic figures in R and how to change the aesthetics, static values, and colour palettes of those figures. Now we are going to look at how to create maps in R.

Most of the work that we perform as biologists 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 just now, creating maps in R is very straight forward. For that reason we are going to have plenty of time today to learn how to do some more advanced things. Our goal is to produce something similar to the figure below.

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

# Load data
load("data/south_africa_coast.Rdata")
load("data/sa_provinces.Rdata")
load("data/site_list.Rdata")
load("data/rast_annual.Rdata")
load("data/rast_aug.Rdata")
load("data/rast_feb.Rdata")
load("data/MUR.Rdata")

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

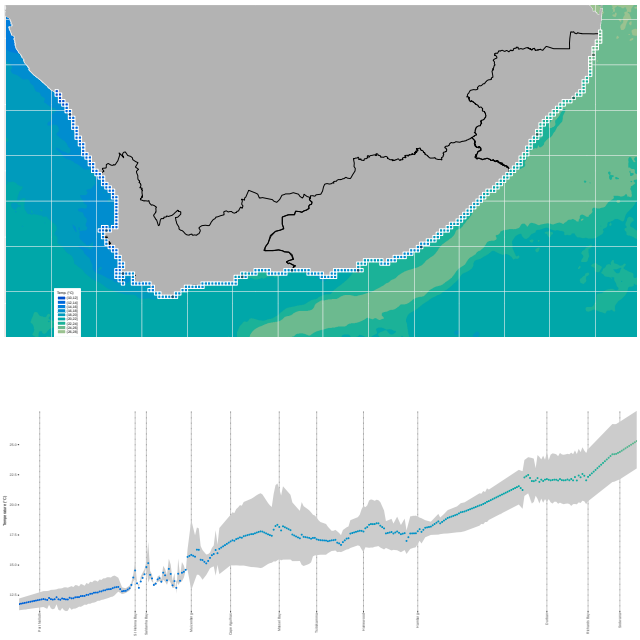


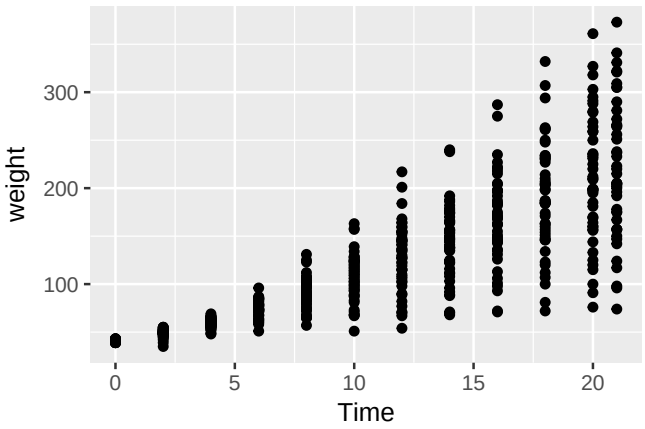
Figure 7.1: The goal for today.

7.1 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 data against a dot plot of the coastline of South Africa.

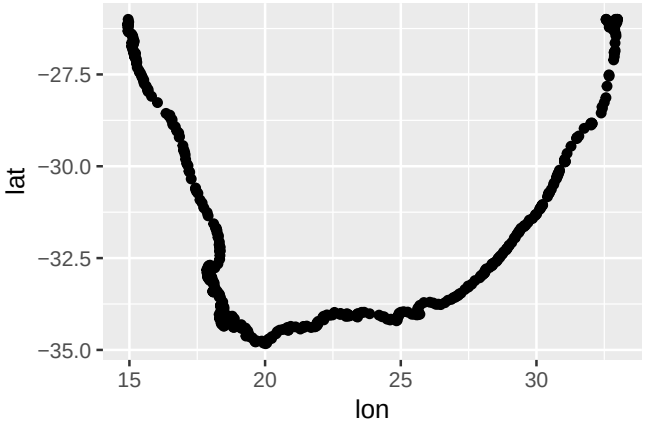
Chicken dots:

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



South Africa 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 the ggplot system this is a simple task.

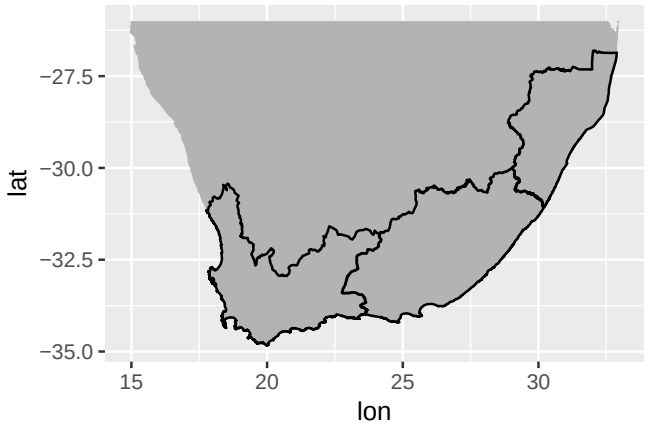
7.2 Creating a map

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. We will start by making a basic landmass background using the `geom_polygon()` function.

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

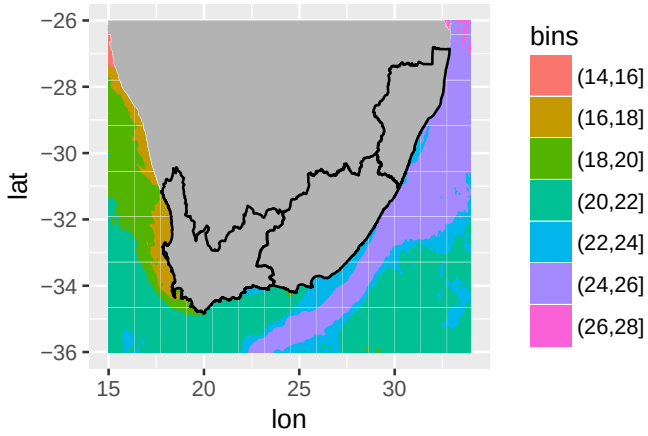
Now our map has a solid picture of South Africa in it. Next up let's add the province borders for the three provinces as seen in 7.1. Notice how we only add one more line of code to do this.

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_polygon(aes(group = group), fill = "grey70") +
  geom_path(data = sa_provinces, aes(group = group)) # The three province borders
```



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?

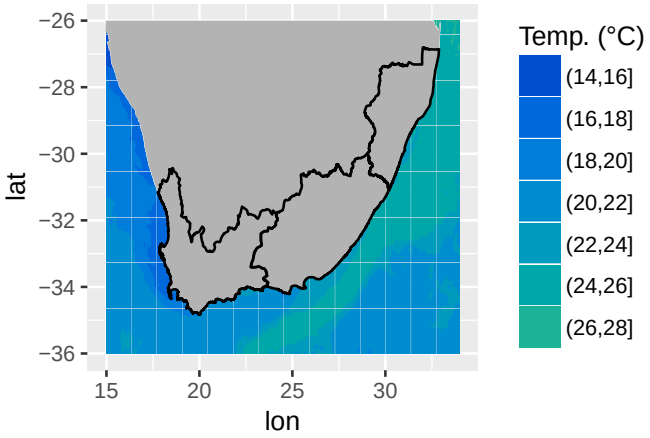
```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_raster(data = MUR, aes(fill = bins)) + # The ocean temperatures
  geom_polygon(aes(group = group), fill = "grey70") +
  geom_path(data = sa_provinces, aes(group = group))
```



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 ggplot. If we want to change them we may do so easily by adding yet one more line of code.

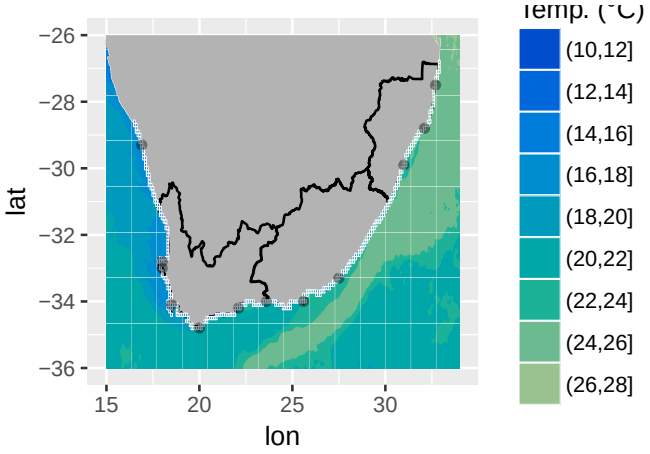
```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_raster(data = MUR, aes(fill = bins)) +
  geom_polygon(aes(group = group), fill = "grey70") +
```

```
  geom_path(data = sa_provinces, aes(group = group)) +
  scale_fill_manual("Temp. (°C)", values = cols11) # Set the colour palette
```



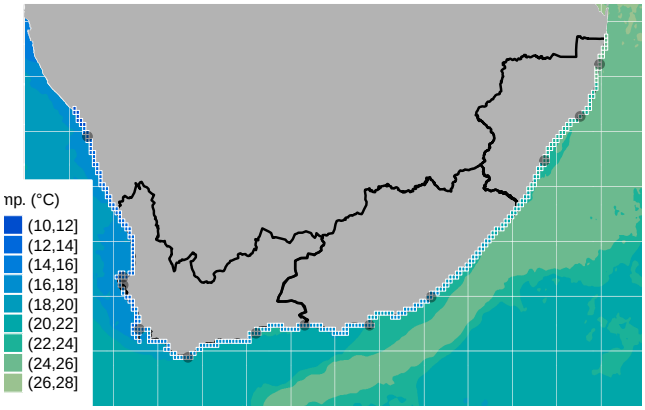
Now there is a colour palette that would make even Jacques Cousteau swoon. When we set the colour palette for a figure in ggplot we must use that colour palette for all other instances of those types of values, too. What this means is that any other discrete values that will be filled in, like the ocean 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 ggplot 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 ggplot changes the colour of the coastal pixels to match the ocean colour automatically. Let's also add somepoints to highlight major cities while we are at it.

```
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_raster(data = MUR, aes(fill = bins)) +
  geom_polygon(aes(group = group), fill = "grey70") +
  geom_path(data = sa_provinces, aes(group = group)) +
  geom_tile(data = rast_annual, aes(fill = bins),
            colour = "white", size = 0.1) + # The coastal temperature values
  geom_point(data = site_list, alpha = 0.4) + # Coastal city locations
  scale_fill_manual("Temp. (°C)", values = cols11)
```



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.

```
fig_top <- ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_raster(data = MUR, aes(fill = bins)) +
  geom_polygon(aes(group = group), fill = "grey70") +
  geom_path(data = sa_provinces, aes(group = group)) +
  geom_tile(data = rast_annual, aes(fill = bins),
            colour = "white", size = 0.1) +
  geom_point(data = site_list, alpha = 0.4) +
  scale_fill_manual("Temp. (°C)", values = cols11) +
  coord_cartesian(expand = 0) + # Remove extra plot area
  theme(axis.title = element_blank(), # Remove the axis labels
        axis.text = element_blank(), # Remove the axis text
        axis.ticks = element_blank(), # Remove the axis ticks
        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(0.11, 0.00) # Fine tune position of legend
        )
fig_top
```

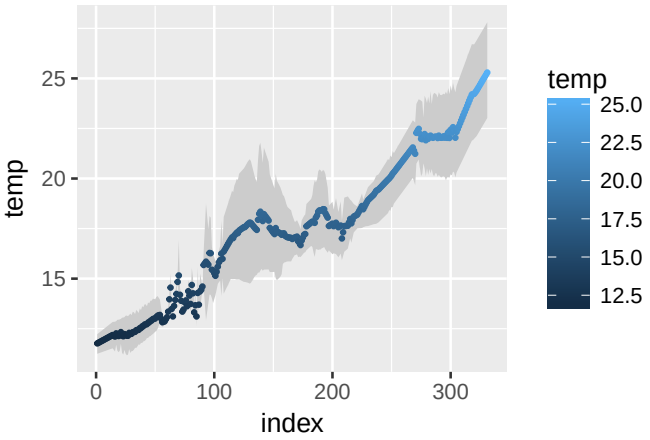


That is a very clean looking map and is going to serve as the top half of the figure we are making together today. For the bottom half we are now going to look at how to create a ribbon in ggplot.

7.3 A ribbon for your troubles

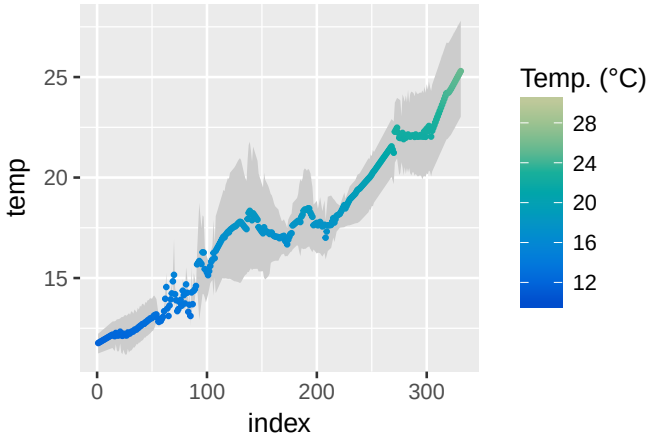
For the bottom portion of our figure we want to add a ribbon that shows the mean temperature along the coast, as well as the coldest (August) and warmest (February) months of the year. This allows us to visualise what the range of temperatures along the coast are.

```
ggplot(data = rast_annual, aes(x = index, y = temp)) +
  geom_ribbon(aes(ymin = rast_aug$temp, ymax = rast_feb$temp), fill = "grey80") +
  geom_point(aes(colour = temp), size = 0.6)
```



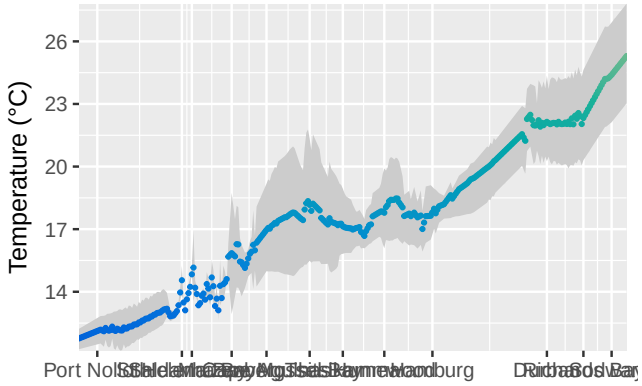
That looks pretty good already. But let's change the colour of the points so that they match the map we are using:

```
ggplot(data = rast_annual, aes(x = index, y = temp)) +
  geom_ribbon(aes(ymin = rast_feb$temp, ymax = rast_aug$temp), fill = "grey80") +
  geom_point(aes(colour = temp), size = 0.6) +
  scale_colour_gradientn("Temp. (°C)", limits = c(10, 30), colours = cols11,
                         breaks = seq(12, 28, 4))
```



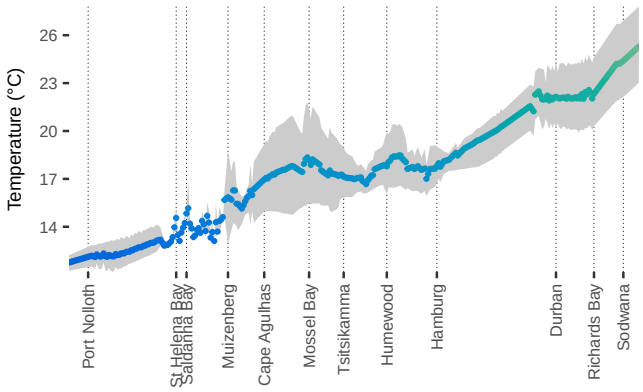
Notice that whereas the colours look the same, for this ribbon we are using temperature as a continuous variable and not a discrete variable. This is why we have to provide the "break" information in the line that changes the colours for our ribbon. The next thing we want to do is change the labels for the x and y axes and remove the legend. Because we are going to have a legend in the top half of the figure, we don't need one for the bottom half.

```
ggplot(data = rast_annual, aes(x = index, y = temp)) +
  geom_ribbon(aes(ymin = rast_feb$temp, ymax = rast_aug$temp), fill = "gray80") +
  geom_point(aes(colour = temp), size = 0.6) +
  scale_colour_gradientn("Temp. (°C)", limits = c(10, 30), colours = cols11,
                         breaks = seq(12, 28, 4)) +
  guides(colour = FALSE) + # Remove the legend
  labs(x = "", y = "Temperature (°C)") + # Change the x and y axis label
  scale_y_continuous(breaks = seq(14, 26, 3),
                     expand = c(0, 0)) + # Change the y axis ticks
  scale_x_continuous(breaks = site_list$index2,
                     labels = site_list$site, expand = c(0, 0)) # Change x axis ticks
```



It may look like we have taken a step backward, but we are about to sort everything out in a very nice way. To do this we will need to manipulate the minutia of the figure using `theme()`. Again, we will label each line to add clarity to the process.

```
fig_bottom <- ggplot(data = rast_annual, aes(x = index, y = temp)) +
  geom_ribbon(aes(ymin = rast_feb$temp, ymax = rast_aug$temp), fill = "gray80") +
  geom_point(aes(colour = temp), size = 0.6) +
  scale_colour_gradientn("Temp. (°C)", limits = c(10, 30), colours = cols11,
                         breaks = seq(12, 28, 4)) +
  guides(colour = FALSE) +
  labs(x = "", y = "Temperature (°C)") +
  scale_y_continuous(breaks = seq(14, 26, 3),
                     expand = c(0, 0)) +
  scale_x_continuous(breaks = site_list$index2,
                     labels = site_list$site, expand = c(0, 0)) +
  theme(panel.background = element_blank(), # Remove the panel background
        panel.border = element_blank(), # Remove the panel border
        plot.background = element_blank(), # Remove the background
        panel.grid.major.x = element_line(colour = "black",
                                          linetype = "dotted",
                                          size = 0.2), # Change the x axis gridlines
        panel.grid.major.y = element_line(colour = NA), # Remove the y axis gridlines
        axis.text.x = element_text(angle = 90,
                                   hjust = 1,
                                   vjust = 0.5,
                                   size = 7), # Change the x axis labels
        axis.text.y = element_text(size = 7), # Change the y axis labels
        axis.title.y = element_text(size = 8) # Change y axis title size
        )
fig_bottom
```



And there we have it. This is a very nice ribbon figure showing the annual range of temperatures along the coast of South Africa. But it relies on the map figure to be complete.

7.4 Combining figures

We learned yesterday how to combine figures in ggplot. With that line of code we saw how to combine all of the figures in equal grids. Now we want the map panel of the figure to be twice as high as the ribbon panel. The following line of code will now round out the whole process for today.

```
# Create an object that contains all of the map data
final <- grid.arrange(fig_top, fig_bottom, layout_matrix = cbind(c(1,1,2), c(1,1,2)))
ggsave(plot = final, "figures/map_complete.pdf")

knitr::include_graphics("figures/map_complete.pdf")
```

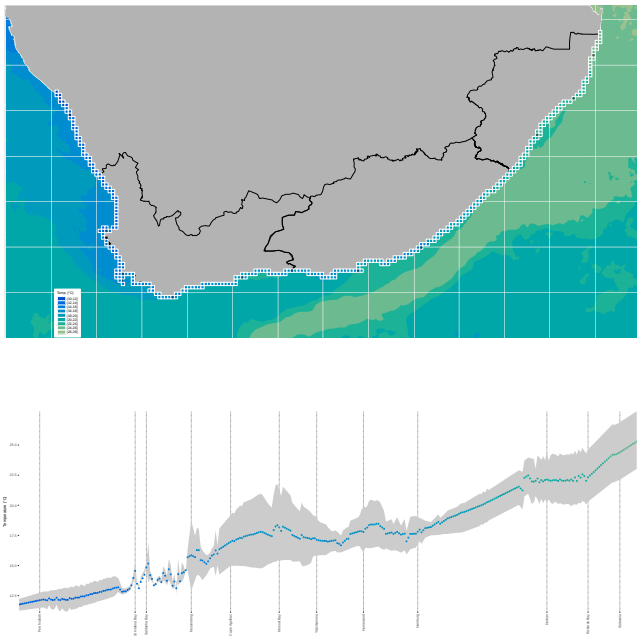


Figure 7.2: The finished product of our efforts.

Why doesn't someone just tell Dora about Google maps so we can all live in a normal world.

Owl City

The only thing Google has failed to do, so far, is fail.

John Battelle

8

Mapping with Google

Now that we've seen how to produce complex maps with ggplot we are going to learn how to create maps using Google maps. 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. So let's get started!

8.1 ggmap

The package we will need for this tut is, as you may have guessed, `ggmap`. It is a bit beefy so please start installing it now.

```
# install.packages("ggmap")
library(ggmap)
library(ggplot2)
```

With our new package installed and activated we now need to load some data that we will use for plotting on top of the Google map.

```
load("data/cape_point_sites.Rdata")
```

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

8.2 Mapping Cape Point

To create our Google map we are going to use 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 map view as it will download faster, and looks tidier. But if we look in the help file we may see a description of all the different map types available. There are a bunch!

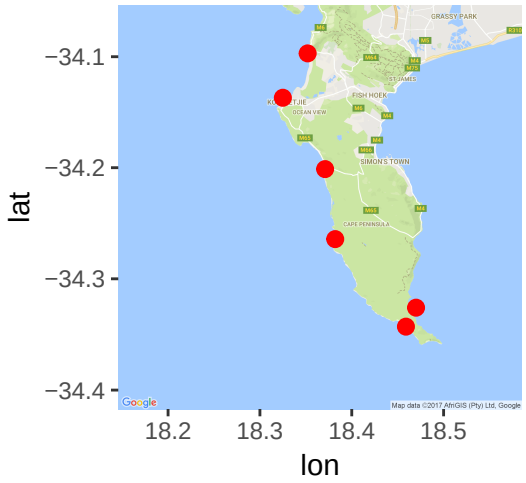
> ***Internet connections***
>
> Much of the code in this document relies on a healthy Internet connection to run. The downloading of Google maps, for example, can be a tenuous process without a stable connection. For that reason (almost) all of the lines of code that require an Internet connection are commented out in this book.

```
# cape_point <- get_map(location = c(lon = 18.36519, lat = -34.2352581),
#                       zoom = 11, maptype = 'roadmap')
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 is to treat the Google data we downloaded as though it is just any ordinary `ggplot2` object. The same as the ones we created in class yesterday and today. For this reason we may use `+` to add new lines of ggplot 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()`.

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

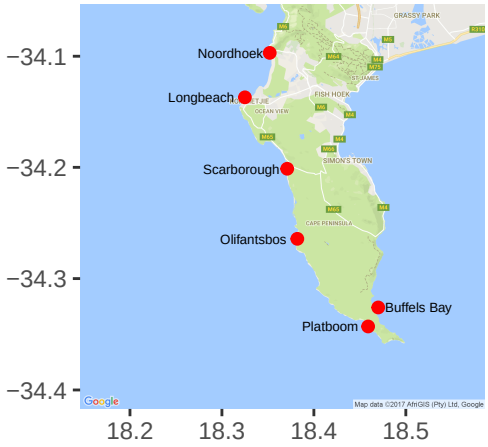


Pretty cool huh?! You may do this for anywhere in the world just as easy as this. The only thing you need to keep in mind is that the `lon`/`lat` coordinates for Google appear to be slightly different than the global standard. It is very curious. This is why the points in the code above have a little bit added or subtracted from their coordinates.

To round out this short session let's clean this figure up a bit and add text labels for the sites.

```
ggmap(cape_point) +
  geom_point(data = cape_point_sites, aes(x = lon+0.002, y = lat-0.007),
             colour = "red", size =  1.75) +
  geom_text(data = cape_point_sites[3,],
            aes(lon+0.002, lat-0.007, label = site),
            hjust = -0.1, vjust = 0.5, size = 1.75) + # Only the third site
```

```
geom_text(data = cape_point_sites[-3,],
          aes(lon+0.002, lat-0.007, label = site),
          hjust = 1.1, vjust = 0.5, size = 1.75) + # All of the other sites
labs(x = "", y = "")
```



And there you have it. A nice quick workflow for adding your data 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.

Knowing where things are, and why, is essential to rational decision making

Jack Dangermond

Here be dragons

Unknown

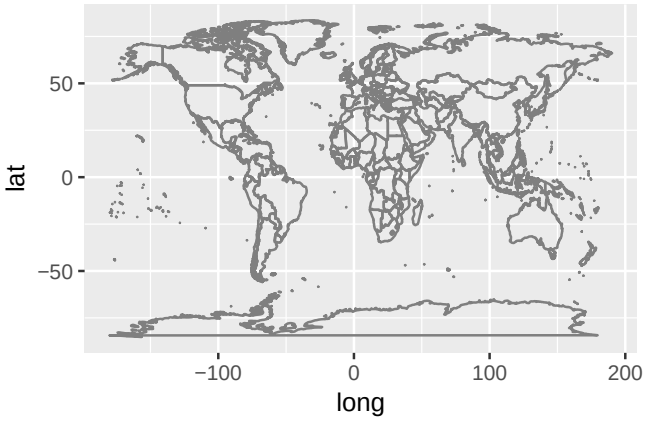
9

Next level mapping

9.1 Built in shape files

In the first module today we learned how to create a map of South Africa using a shape files we had saved on our computer. We are now going to learn how to use the shape files that were downloaded onto our computer with the `tidyverse`.

```
# Load libraries
library(tidyverse)
library(gridExtra)
library(grid)

# The global shape file
ggplot() +
  borders()
```

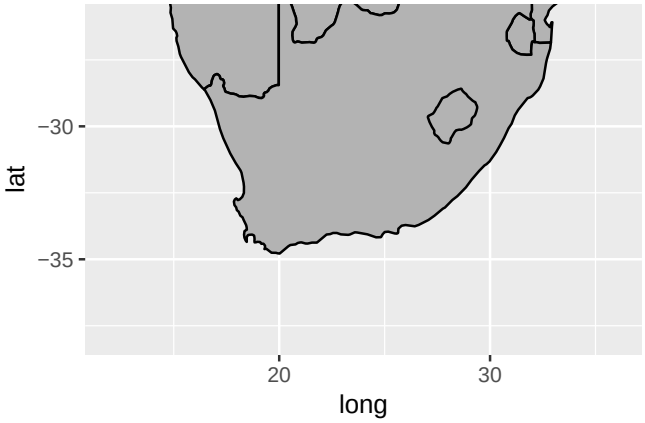


Jikes! It's as simple as that to load a map of the whole planet. Usually we are not going to want to make a map of the entire planet, so let's see how to focus on just South Africa.

```
ggplot() +
  borders(regions = "South Africa",
          colour = "black", fill = "grey70") # Set colour and fill
```

Surprisingly, the international borders for South Africa do not account for Swaziland. That is an error. We also see in the above map that the exclusion of the bottom parts of the countries bordering on South Africa give it the appearance of an island. This isn't great, so let's consider a different option for selecting the area around South Africa.

```
south_africa <- ggplot() +
  borders(fill = "grey70", colour = "black") +
  coord_cartesian(xlim = c(12, 36), ylim = c(-38, -26))
south_africa
```

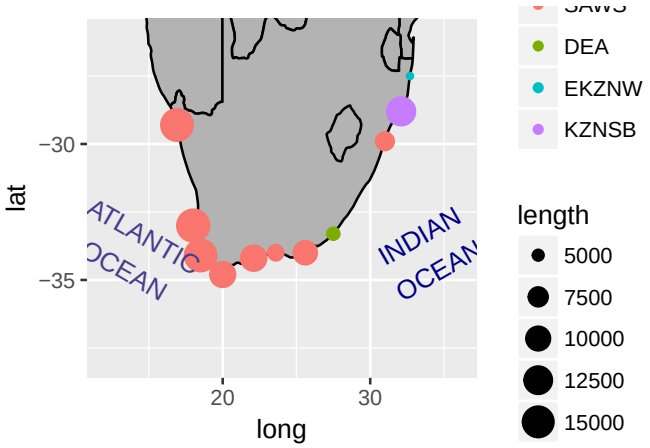


There we have a much nicer looking map of southern Africa. Up next we will remind ourselves how we add points and labels, then we will see how to add scale bars and insets.

9.2 Bells and whistles

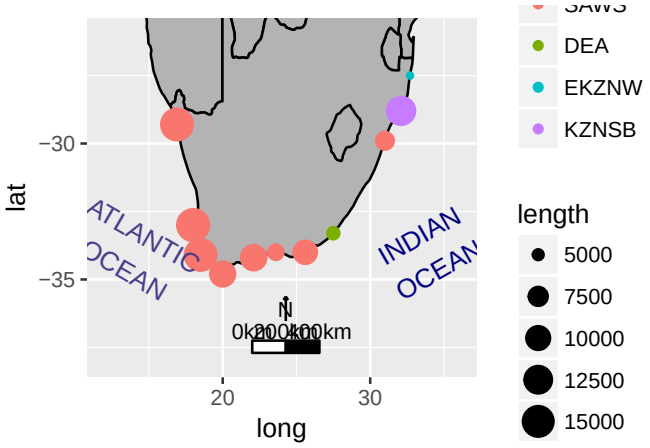
A map is almost always going to need some labels and some other visual cues. We saw in the previous chapter how to add labels, below we will see how this may differ if we want to add just one label at a time. This can be useful if each label needs to be different from all other labels for whatever reason.

```
# Load SACTN site data
load("data/site_list.Rdata")

# Note that because we saved out previous
# map as 'south_africa`, we may now add
# directly to it
south_africa_labels <- south_africa +
  # Add some points
  geom_point(data = site_list, aes(x = lon, y = lat, colour = src, size = length)) +
  # geom_text(data = site_list, aes(x = lon, y = lat, label = site), vjust = 1.5) +
  annotate("text", label = "INDIAN\nOCEAN", x = 34.00, y = -34.0,
           size = 4.0, angle = 30, colour = "navy") +
  annotate("text", label = "ATLANTIC\nOCEAN", x = 14.00, y = -34.0,
           size = 4.0, angle = 330, colour = "darkslateblue")
south_africa_labels
```



With our fancy labels and points added, let's next insert a scale bar. There is no default scale bar function in the `tidyverse`, rather we will need to import one that has been created by a kind Samaritan.

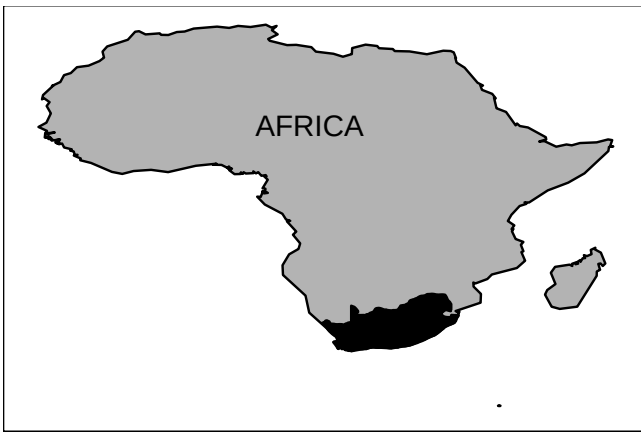
```
# Load the function for creating scale bars
source("functions/scale.bar.func.R")
R> Checking rgeos availability: TRUE

# Add a scale bar to our map
south_africa_scale <- south_africa_labels +
    scaleBar(lon = 22.0, lat = -37.7, distanceLon = 200, distanceLat = 50,
             distanceLegend = 90, dist.unit = "km", arrow.length = 100,
             arrow.distance = 130, arrow.North.size = 3)
south_africa_scale
```



9.3 Insetting

In order to inset a smaller map inside of a bigger map, we first need to create the smaller map. Let's make a map of Africa for this purpose. The built in shape files aren't going to be terribly useful for this, so we will use a shape files of Africa that we already have saved on our computer.

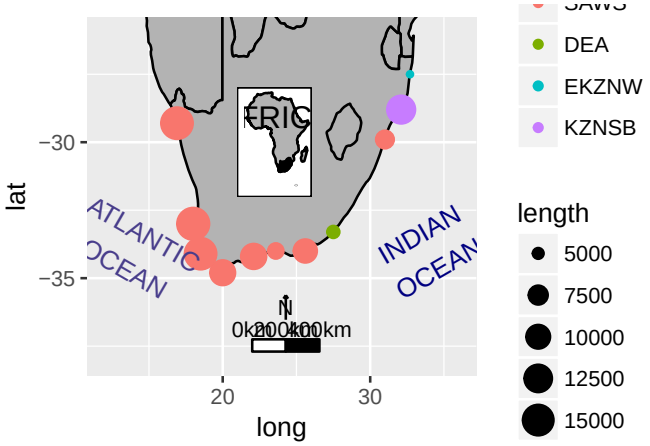
```
# Load Africa shape
load("data/africa_coast.Rdata")

# Create map
africa <- ggplot(data = africa_coast, aes(x = lon, y = lat)) +
  geom_polygon(aes(group = group), colour = "black", fill = "grey70") +
  borders(regions = "South Africa", colour = "black", fill = "black") +
  annotate("text", x = 15, y = 15, label = "AFRICA", size = 4.5) +
  theme_void() +
  theme(plot.background = element_rect(fill = "white", colour = "black"))
africa
```



And now to inset this map of Africa into our map of South 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.

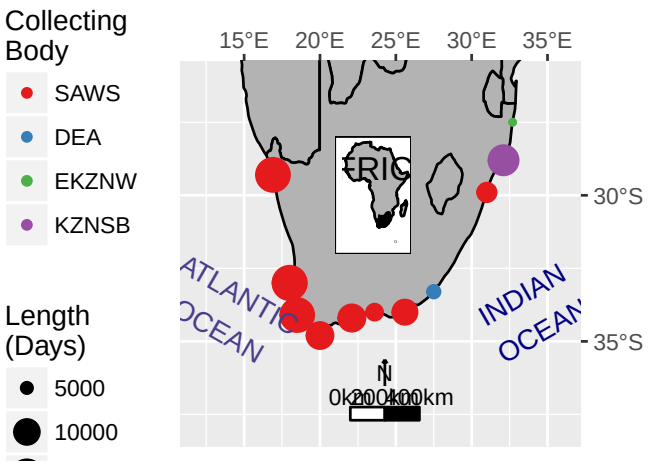
```
south_africa_inset <- south_africa_scale +
  annotation_custom(grob = ggplotGrob(africa),
                    xmin = 21, xmax = 26,
                    ymin = -32, ymax = -28)
south_africa_inset
```



9.4 Rounding it out

There are a lot of exciting things going on in this figure. To round out this chapters let's tweak the label and legend titles to make the figure more presentable.

```
south_africa_final <- south_africa_inset +
  scale_x_continuous(breaks = seq(15, 35, 5),
                     labels = scales::unit_format("°E", sep = ""),
                     position = "top") +
  scale_y_continuous(breaks = seq(-35, -30, 5),
                     labels = c("35°S", "30°S"),
                     position = "right") +
  scale_color_brewer(name = "Collecting\nBody", palette = "Set1") +
  scale_size_continuous(name = "Length\n(Days)", breaks = c(5000, 10000, 15000)) +
  labs(x = "", y = "") +
  theme(legend.position = "left")
south_africa_final
```




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

Part IV

Day 4

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

Dream up a book on Monday, publish it on Friday.

Jill Novak

10 R Markdown

The workshop pdf we have been using for the last three days is actually produced from R Markdown and you may view the R Markdown document (the .Rmd file) to see all of the code that created this pdf in its native state. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. R Markdown is based on the language Markdown, which is another computer language, somewhere in between LaTeX and Microsoft Word. R Markdown differs from Markdown in that it is also able to understand the R code we give it. Furthermore, RStudio has built into it the capabilities necessary to use R Markdown "out of the box". For more details on using R Markdown see http://rmarkdown.rstudio.com. In order to use this tutorial effectively, please examine the R Markdown file ("Intro_R_2017.Rmd") in conjunction with the output document ("Intro_R_2017.pdf"). If you double click on the "Intro_R_2017.Rmd" file it will open in the RStudio editor.

10.1 Quick examples

Below is just a quick overview of the many common things one will need to know to put an RMarkdown document together.

10.1.1 Text

This is text in *italics*.

And this text is in **bold**.

This text is in `code font`.

You can embed an R code chunk like this and show it and the data it produces:

```
library(ggmap)
load("data/cape_point_sites.Rdata")
# cape_point <- get_map(location = c(lon = 18.36519, lat = -34.2352581),
#                       zoom = 11, maptype = 'roadmap')
```

Table 10.1: A table of the first 10 rows of the mtcars data.

	mpg	cyl	disp	hp	drat	wt	qsec	vs
Mazda RX4	21.0	6	160	110	3.90	2.62	16.5	0
Mazda RX4 Wag	21.0	6	160	110	3.90	2.88	17.0	0
Datsun 710	22.8	4	108	93	3.85	2.32	18.6	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.21	19.4	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.0	0
Valiant	18.1	6	225	105	2.76	3.46	20.2	1
Duster 360	14.3	8	360	245	3.21	3.57	15.8	0
Merc 240D	24.4	4	147	62	3.69	3.19	20.0	1
Merc 230	22.8	4	141	95	3.92	3.15	22.9	1
Merc 280	19.2	6	168	123	3.92	3.44	18.3	1

```
load("data/cape_point.Rdata")
str(cape_point)
R>  'ggmap' chr [1:1280, 1:1280] "#A3CCFF" "#A3CCFF" "#A3CCFF" ...
R>  - attr(*, "bb")='data.frame':   1 obs. of  4 variables:
R>   ..$ ll.lat: num -34.4
R>   ..$ ll.lon: num 18.1
R>   ..$ ur.lat: num -34.1
R>   ..$ ur.lon: num 18.6
R>  - attr(*, "source")= chr "google"
R>  - attr(*, "maptype")= chr "roadmap"
R>  - attr(*, "zoom")= num 11
```

You can also embed R output directly into sentences as in this example:

Some site details are in this list:

- site name — Platboom
- longitude — 18.457
- latitude — -34.336

10.1.2 Tables

There are many ways to produce tables in R Markdown. A short search will provide many alternatives. The `xtable` package is another excellent choice as this provides even more options for how your table output will appear. Here we provide one example:

```
knitr::kable(
  head(mtcars[, 1:8], 10), booktabs = TRUE,
  caption = 'A table of the first 10 rows of the mtcars data.'
)
```

Try looking up the help file for `?kable()` to learn more about what may be done with this function.

10.1.3 Images

Images stored on your computer, such as 10.1, can be embedded in your document and even cross referenced.

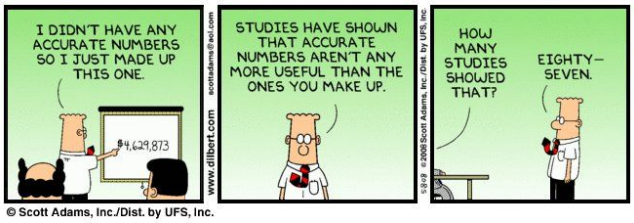


Figure 10.1: Only data will tell.

Notice that in order to display images in this way we need to make sure R knows we are using the `knitr` package function `include_graphics()`, which allow one to display images of all sorts of file types without any fuss.

You can also embed any plots produced by R, for example:

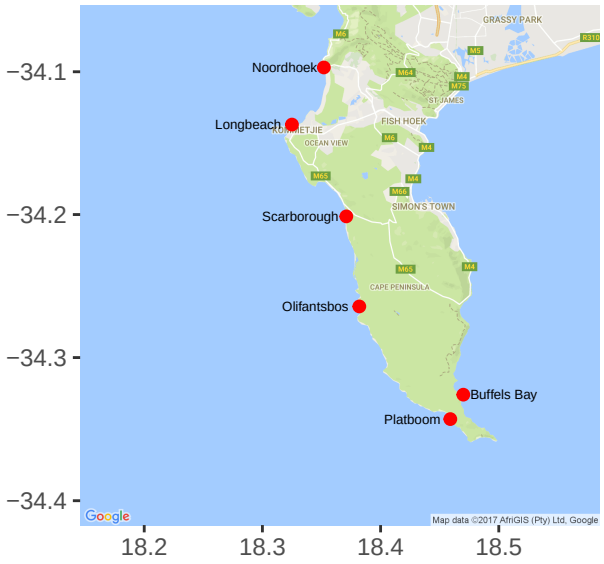


Figure 10.2: My first R plot.

Notice above how the first line specifying the start of the R code includes some specifications with regards to the size of the figure, its caption, etc. Note too that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot. The options `results = TRUE`, `message = TRUE` and `warning = TRUE` have similar functions. Info on the other code chunk options can be found at the R Markdown website[1] or in the Cheatsheets and other documentation accessible via the RStudio "Help" in the menu bar.

HTML hyperlinks

See how the code above also demonstrates how to embed links to external websites.

[1] http://rmarkdown.rstudio.com/authoring_rcodechunks.html

10.1.4 References

We can also have some references… This document was made using the R software (R Core Team 2017) and various add-on packages (Oksanen et al. 2017). The *vegan* package was produced by Oksanen et al. (2017) some time ago.

10.2 Creating a document

Even though we may immediately begin authoring documents with RStudio, we are limited to .html and .doc file types. If we want to author .pdf files, such as the one we are reading now, we must install LaTeX on our computers. This installation process is beyond the scope of this course but there are many resources available online to aid one in the process and the software is, of course, free.

Whether or not you have LaTeX installed, when you click the **Knit** button (with the option to create multiple document kinds) a document will be generated that includes both content as well as the output of any embedded R code chunks (portions of R code surrounded by code that denotes the R commands) within the document. R code chunks can be used to render R output into documents or to simply display code for illustration, as outlined above.

This is a terribly basic demonstration, but since beautiful documentation already exists I suggest you go and find the necessary examples on the R Markdown website indicated above for a more in-depth account of how to use it.

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

11

The Tidyverse

Note: This session was taken *mostly as is* from Hadley Wickham's book, R for Data Science[1].

The Tidyverse[2] 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 two years or so 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 (which, if you have not yet downloaded it, you can do so now). The packages making up the Tidyverse are shown in Figure 11.1.

Figure 11.1: The Tidyverse by Hadley Wickham and co.

[1] http://r4ds.had.co.nz
[2] http://tidyverse.org

```
# install.packages("tidyverse")
library(tidyverse)
```

Take careful note of the conflicts message that's printed when you load the tidyverse. It tells you that `dplyr` overwrites some functions in base R. If you want to use the base version of these functions after loading `tidyverse`, you'll need to use their full names: `stats::filter()` and `stats::lag()`.

You are by now familiar with `ggplot2`, and now we will turn our focus to '**tidyr**`** and more specifically **`**dplyr**' within that.

11.1 Effcient data handling with `dplyr`

We have seen before how to select data in various ways: we can create subsets using the `subset()` command, which allows the selection of different rows of data that fulfil some logical criteria; we can use the square-bracket notation to select only some elements of a dataframe, *e.g.* `dat[1, 2]`, which will select the data element in the first row and in the second column; or we can select entire rows and columns by using `dat[1, ]` or `dat[, 2]`, respectively. These are very basic examples, and there are other ways too. All of these ways utilise functions that come with the standard R installation.

Now we will look at one of the packages in the Tidyverse, *i.e.* `dplyr`. The `dplyr` functions in my own opinion are more powerful and more intuitive than the 'traditional' ways available to us. But to use the functions to their fullest extent we need to learn to think somewhat differently. So, let me introduce the *pipe* command, `%>%`. This command allows us to feed the output from one command directly into another subsequent command, and in doing so it creates a more readable code because the need to nest functions is reduced, and it frees us from having to create multiple intermediate variables that could clog up the memory. Overall our efficiency improves.

I will demonstrate the five main `dplyr` functions that allow you to solve the vast majority of your data manipulation challenges by working through some examples.

- Pick observations by their values (`filter()`).
- Reorder the rows (`arrange()`).
- Pick variables by their names (`select()`).
- Create new variables with functions of existing variables (`mutate()`).
- Collapse many values down to a single summary (`summarise()`).

We will use the South African Coastal Temperature Network data, which we examine here. This is how data analysis normally starts:

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

# copy the data to a dataframe with a shorter name, and delete the original
temp <- SACTNmonthly_v4.0
rm(SACTNmonthly_v4.0)

# see what is in the data
head(temp)
R>           site src       date temp depth type
R> 1 Port Nolloth DEA 1991-02-01 11.5     5  UTR
```

```
R> 2 Port Nolloth DEA 1991-03-01 12.0     5  UTR
R> 3 Port Nolloth DEA 1991-04-01 12.0     5  UTR
R> 4 Port Nolloth DEA 1991-05-01 11.9     5  UTR
R> 5 Port Nolloth DEA 1991-06-01 12.2     5  UTR
R> 6 Port Nolloth DEA 1991-07-01 12.5     5  UTR
levels(temp$site)[1:20]
R>  [1] "Port Nolloth"  "Hondeklipbaai" "Doringbaai"    "Lamberts Bay"
R>  [5] "Elands Bay"    "St Helena Bay" "Paternoster"   "Saldanha Bay"
R>  [9] "Dassen Island" "Yzerfontein"   "Sea Point"     "Oudekraal"
R> [13] "Hout Bay"      "Kommetjie"     "Buffelsbaai"   "Bordjies"
R> [17] "Bordjies Deep" "Fish Hoek"     "Kalk Bay"      "Muizenberg"
unique(temp$src)
R> [1] "DEA"   "SAWS"  "DAFF"  "UWC"   "SAEON" "KZNSB" "EKZNW"
unique(temp$type)
R> [1] "UTR"    "thermo"
```

All of these five functions can be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result. Let's dive in and see how these verbs work.

11.2 Filter rows with `filter()`

`filter()` allows you to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame. For example, we can select all monthly temperatures recorded at Humewood during 1990:

```
# Load this for the year() function
library(lubridate)

# Filter the data
filter(temp, site == "Humewood", year(date) == 1990)
R>        site  src       date temp depth   type
R> 1  Humewood SAWS 1990-01-01 21.9     0 thermo
R> 2  Humewood SAWS 1990-02-01 18.6     0 thermo
R> 3  Humewood SAWS 1990-03-01 18.6     0 thermo
R> 4  Humewood SAWS 1990-04-01 17.3     0 thermo
R> 5  Humewood SAWS 1990-05-01 16.4     0 thermo
R> 6  Humewood SAWS 1990-06-01 15.9     0 thermo
R> 7  Humewood SAWS 1990-07-01 15.7     0 thermo
R> 8  Humewood SAWS 1990-08-01 16.1     0 thermo
R> 9  Humewood SAWS 1990-09-01 16.4     0 thermo
R> 10 Humewood SAWS 1990-10-01 17.1     0 thermo
```

```
R> 11 Humewood SAWS 1990-11-01 18.0     0 thermo
R> 12 Humewood SAWS 1990-12-01 20.1     0 thermo
```

When you run that line of code, `dplyr` executes the filtering operation and returns a new dataframe. `dplyr` functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, `<-`:

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

R either prints out the results, or saves them to a variable. If you want to do both, you can wrap the assignment in parentheses:

```
(humewood_1990 <- filter(temp, site == "Humewood", year(date) == 1990))
R>        site  src       date temp depth   type
R> 1  Humewood SAWS 1990-01-01 21.9     0 thermo
R> 2  Humewood SAWS 1990-02-01 18.6     0 thermo
R> 3  Humewood SAWS 1990-03-01 18.6     0 thermo
R> 4  Humewood SAWS 1990-04-01 17.3     0 thermo
R> 5  Humewood SAWS 1990-05-01 16.4     0 thermo
R> 6  Humewood SAWS 1990-06-01 15.9     0 thermo
R> 7  Humewood SAWS 1990-07-01 15.7     0 thermo
R> 8  Humewood SAWS 1990-08-01 16.1     0 thermo
R> 9  Humewood SAWS 1990-09-01 16.4     0 thermo
R> 10 Humewood SAWS 1990-10-01 17.1     0 thermo
R> 11 Humewood SAWS 1990-11-01 18.0     0 thermo
R> 12 Humewood SAWS 1990-12-01 20.1     0 thermo
```

11.2.1 Comparisons

To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal).

When you're starting out with R, the easiest mistake to make is to use `=` instead of `==` when testing for equality. When this happens you'll get an informative error:

```
filter(temp, year(date) = 1990)
R> Error: <text>:1:25: unexpected '='
R> 1: filter(temp, year(date) =
R>                            ^
```

There's another common problem you might encounter when using `==`: floating point numbers. These results might surprise you!

```
sqrt(2) ^ 2 == 2
R> [1] FALSE
1/49 * 49 == 1
R> [1] FALSE
```

Computers use finite precision arithmetic (they obviously can't store an infinite number of digits!) so remember that every number you see is an approximation. Instead of relying on `==`, use `near()`:

```
near(sqrt(2) ^ 2,  2)
R> [1] TRUE
near(1 / 49 * 49, 1)
R> [1] TRUE
```

11.2.2 Logical operators

Multiple arguments to `filter()` are combined with "and": every expression must be true in order for a row to be included in the output. For other types of combinations, you'll need to use Boolean operators yourself: `&` is "and", `|` is "or", and `!` is "not". Figure 11.2 shows the complete set of Boolean operations.

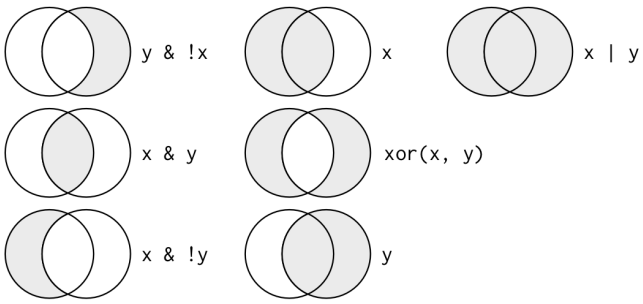


Figure 11.2: Complete set of boolean operations. 'x' is the left-hand circle, 'y' is the right-hand circle, and the shaded region show which parts each operator selects.

The following code finds all temperatures recorded at Pollock Beach during February or March:

```
filter(temp, site == "Pollock Beach", month(date) == 2 | month(date) == 3)
```

The order of operations doesn't work like English. You can't write `filter(temp, site == "Pollock Beach", month(date) == 2 | 3)`, which you might literally translate into "finds all temperatures recorded at Pollock Beach in February or March". Instead it finds all months that equal `2 | 3`, an expression that evaluates to `TRUE`. In a numeric context (like here), `TRUE` becomes one, so this finds all temperatures in January, not February or March. This is quite confusing!

A useful short-hand for this problem is `x %in% y`. This will select every row where `x` is one of the values in `y`. We could use it to rewrite the code above:

```
feb_mar <- filter(temp, site == "Pollock Beach", month(date) %in% c(2, 3))
```

Sometimes you can simplify complicated subsetting by remembering De Morgan's law: `!(x & y)` is the same as `!x | !y`, and `!(x | y)` is the same as `!x & !y`. For example, if you 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:

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

Whenever you start using complicated, multipart expressions in `filter()`, consider making them explicit variables instead. That makes it much easier to check your work.

11.2.3 Missing values

`filter()` only includes rows where the condition is `TRUE`; it excludes both `FALSE` and `NA` values. If you want to preserve missing values, ask for them explicitly:

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)
R> # A tibble: 1 x 1
R>       x
R>   <dbl>
R> 1     3
filter(df, is.na(x) | x > 1)
R> # A tibble: 2 x 1
R>       x
R>   <dbl>
R> 1    NA
R> 2     3
```

11.3 Arrange rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
# Here we ony visualise 10 rows
head(arrange(temp, depth, temp), 10)
R>             site  src       date  temp depth   type
R> 1      Sea Point SAWS 1990-07-01  9.64     0 thermo
R> 2     Muizenberg SAWS 1984-07-01  9.71     0 thermo
R> 3     Doringbaai SAWS 2000-12-01  9.77     0 thermo
R> 4  Hondeklipbaai SAWS 2003-06-01  9.78     0 thermo
R> 5      Sea Point SAWS 1984-06-01 10.00     0 thermo
R> 6     Muizenberg SAWS 1992-07-01 10.19     0 thermo
R> 7  Hondeklipbaai SAWS 2005-07-01 10.33     0 thermo
R> 8  Hondeklipbaai SAWS 2003-07-01 10.34     0 thermo
R> 9      Sea Point SAWS 2000-12-01 10.38     0 thermo
R> 10    Muizenberg SAWS 1984-08-01 10.39     0 thermo
```

Use `desc()` to re-order by a column in descending order:

```
# Here we ony visualise 10 rows, again
head(arrange(temp, desc(temp)), 10)
R>             site   src       date temp depth type
R> 1        Sodwana   DEA 2000-02-01 28.3    18  UTR
R> 2        Sodwana   DEA 1999-03-01 28.0    18  UTR
R> 3        Sodwana   DEA 1998-03-01 27.9    18  UTR
R> 4        Sodwana   DEA 1998-02-01 27.8    18  UTR
R> 5        Sodwana   DEA 1996-02-01 27.7    18  UTR
R> 6        Sodwana   DEA 2000-03-01 27.5    18  UTR
R> 7        Sodwana   DEA 2000-01-01 27.5    18  UTR
R> 8  Leadsmanshoal EKZNW 2007-02-01 27.5    10  UTR
```

```
R> 9         Sodwana EKZNW 2005-01-01 27.5    12  UTR
R> 10        Sodwana EKZNW 2007-02-01 27.4    12  UTR
```

Missing values are always sorted at the end:

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
R> # A tibble: 3 x 1
R>       x
R>   <dbl>
R> 1     2
R> 2     5
R> 3    NA
arrange(df, desc(x))
R> # A tibble: 3 x 1
R>       x
R>   <dbl>
R> 1     5
R> 2     2
R> 3    NA
```

11.4 Select columns with `select()`

It's not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

`select()` is not terribly useful with the `temp` data because we only have 6 variables, but you can still get the general idea:

```
# Select columns by name
head(select(temp, site, date, temp))
R>           site       date temp
R> 1 Port Nolloth 1991-02-01 11.5
R> 2 Port Nolloth 1991-03-01 12.0
R> 3 Port Nolloth 1991-04-01 12.0
R> 4 Port Nolloth 1991-05-01 11.9
R> 5 Port Nolloth 1991-06-01 12.2
R> 6 Port Nolloth 1991-07-01 12.5
# Select all columns between site and temp (inclusive)
head(select(temp, site:temp))
R>           site src       date temp
R> 1 Port Nolloth DEA 1991-02-01 11.5
R> 2 Port Nolloth DEA 1991-03-01 12.0
R> 3 Port Nolloth DEA 1991-04-01 12.0
R> 4 Port Nolloth DEA 1991-05-01 11.9
R> 5 Port Nolloth DEA 1991-06-01 12.2
R> 6 Port Nolloth DEA 1991-07-01 12.5
# Select all columns except those from date to depth (inclusive)
head(select(temp, -(date:depth)))
R>           site src type
```

```
R> 1 Port Nolloth DEA  UTR
R> 2 Port Nolloth DEA  UTR
R> 3 Port Nolloth DEA  UTR
R> 4 Port Nolloth DEA  UTR
R> 5 Port Nolloth DEA  UTR
R> 6 Port Nolloth DEA  UTR
```

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")`: matches names that begin with "abc".
- `ends_with("xyz")`: matches names that end with "xyz".
- `contains("ijk")`: matches names that contain "ijk".
- `matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters. We won't go into this in depth in this workshop.
- `num_range("x", 1:3)` matches `x1`, `x2` and `x3`.

See `?select` for more details.

`select()` can be used to rename variables, but it's rarely useful because it drops all of the variables not explicitly mentioned. Instead, use `rename()`, which is a variant of `select()` that keeps all the variables that aren't explicitly mentioned:

```
head(rename(temp, source = src))
R>           site source       date temp depth type
R> 1 Port Nolloth    DEA 1991-02-01 11.5     5  UTR
R> 2 Port Nolloth    DEA 1991-03-01 12.0     5  UTR
R> 3 Port Nolloth    DEA 1991-04-01 12.0     5  UTR
R> 4 Port Nolloth    DEA 1991-05-01 11.9     5  UTR
R> 5 Port Nolloth    DEA 1991-06-01 12.2     5  UTR
R> 6 Port Nolloth    DEA 1991-07-01 12.5     5  UTR
```

Another option is to use `select()` in conjunction with the `everything()` helper. This is useful if you have a handful of variables you'd like to move to the start of the data frame.

```
head(select(temp, type, src, everything()))
R>   type src         site       date temp depth
R> 1  UTR DEA Port Nolloth 1991-02-01 11.5     5
R> 2  UTR DEA Port Nolloth 1991-03-01 12.0     5
R> 3  UTR DEA Port Nolloth 1991-04-01 12.0     5
R> 4  UTR DEA Port Nolloth 1991-05-01 11.9     5
R> 5  UTR DEA Port Nolloth 1991-06-01 12.2     5
R> 6  UTR DEA Port Nolloth 1991-07-01 12.5     5
```

11.5 Add new variables with `mutate()`

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`.

`mutate()` always adds new columns at the end of your dataset. Remember that when you're in RStudio, the easiest way to see all the columns is `View()` (assuming you don't want to click on the object in your **Environment** tab.

```
head(mutate(temp, kelvin = temp + 273.15))
R>           site src       date temp depth type kelvin
R> 1 Port Nolloth DEA 1991-02-01 11.5     5  UTR    285
R> 2 Port Nolloth DEA 1991-03-01 12.0     5  UTR    285
R> 3 Port Nolloth DEA 1991-04-01 12.0     5  UTR    285
R> 4 Port Nolloth DEA 1991-05-01 11.9     5  UTR    285
R> 5 Port Nolloth DEA 1991-06-01 12.2     5  UTR    285
R> 6 Port Nolloth DEA 1991-07-01 12.5     5  UTR    286
```

Note that you can refer to columns that you've just created:

```
head(mutate(temp,
            kelvin = temp + 273.15,
            kelvin_base = kelvin - temp))
R>           site src       date temp depth type kelvin kelvin_base
R> 1 Port Nolloth DEA 1991-02-01 11.5     5  UTR    285         273
R> 2 Port Nolloth DEA 1991-03-01 12.0     5  UTR    285         273
R> 3 Port Nolloth DEA 1991-04-01 12.0     5  UTR    285         273
R> 4 Port Nolloth DEA 1991-05-01 11.9     5  UTR    285         273
R> 5 Port Nolloth DEA 1991-06-01 12.2     5  UTR    285         273
R> 6 Port Nolloth DEA 1991-07-01 12.5     5  UTR    286         273
```

If you only want to keep the new variables, use `transmute()`:

```
head(transmute(temp, kelvin = temp + 273.15))
R>   kelvin
R> 1    285
R> 2    285
R> 3    285
R> 4    285
R> 5    285
R> 6    286
```

11.5.1 Useful creation functions

There are many functions for creating new variables that you can use with `mutate()`. The key property is that the function must be vectorised: it must take a vector of values as input, return a vector with the same number of values as output. There's no way to list every possible function that you might use, but here's a selection of functions that are frequently useful:

- Arithmetic operators: `+`, `-`, `*`, `/`, `^`. These are all vectorised, using the so called "recycling rules". If one parameter is shorter than the other, it will be automatically extended to be the same length. This is most useful when one of the arguments is a single number: `temp * 10`, `temp * 9 / 5 + 32`, etc.

  Arithmetic operators are also useful in conjunction with the aggregate functions you'll learn about later. For example, `x / sum(x)` calculates the proportion of a total, and `y - mean(y)` computes the difference from the mean.

- Logs: `log()`, `log2()`, `log10()`. Logarithms are an incredibly useful transformation for dealing with data that ranges across multiple orders of magnitude. They also convert multiplicative relationships to additive.

  All else being equal, I recommend using `log2()` because it's easy to interpret: a difference

of 1 on the log scale corresponds to doubling on the original scale and a difference of -1 corresponds to halving.

- Offsets: `lead()` and `lag()` allow you to refer to leading or lagging values. This allows you to compute running differences (e.g. `x - lag(x)`) or find when values change (`x != lag(x)`). They are most useful in conjunction with `group_by()`, which you'll learn about shortly.

```
(x <- 1:10)
R>  [1]  1  2  3  4  5  6  7  8  9 10
lag(x)
R>  [1] NA  1  2  3  4  5  6  7  8  9
lead(x)
R>  [1]  2  3  4  5  6  7  8  9 10 NA
```

- Cumulative and rolling aggregates: R provides functions for running sums, products, mins and maxes: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`; and **`dplyr`** provides `cummean()` for cumulative means. If you need rolling aggregates (i.e. a sum computed over a rolling window), try the **`RcppRoll`** package.

```
x
R>  [1]  1  2  3  4  5  6  7  8  9 10
cumsum(x)
R>  [1]  1  3  6 10 15 21 28 36 45 55
cummean(x)
R>  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

- Logical comparisons, `<`, `<=`, `>`, `>=`, `!=`, which you learned about earlier. If you're doing a complex sequence of logical operations it's often a good idea to store the interim values in new variables so you can check that each step is working as expected.

- Ranking: there are a number of ranking functions, but you should start with `min_rank()`. It does the most usual type of ranking (e.g. 1st, 2nd, 2nd, 4th). The default gives smallest values the small ranks; use `desc(x)` to give the largest values the smallest ranks.

```
y <- c(1, 2, 2, NA, 3, 4)
min_rank(y)
R> [1]  1  2  2 NA  4  5
min_rank(desc(y))
R> [1]  5  3  3 NA  2  1
```

  If `min_rank()` doesn't do what you need, look at the variants `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()`, `ntile()`. See their help pages for more details.

```
row_number(y)
R> [1]  1  2  3 NA  4  5
dense_rank(y)
R> [1]  1  2  2 NA  3  4
percent_rank(y)
R> [1] 0.00 0.25 0.25   NA 0.75 1.00
cume_dist(y)
R> [1] 0.2 0.6 0.6  NA 0.8 1.0
```

11.6 Grouped summaries with `summarise()`

The last key verb is `summarise()`. It collapses a data frame to a single row:

```
summarise(temp, temp = mean(temp, na.rm = TRUE))
R>   temp
R> 1 19.3
```

`summarise()` is not terribly useful unless we pair it with `group_by()`. This changes the unit of analysis from the complete dataset to individual groups. Then, when you use the **dplyr** verbs on a grouped data frame they'll be automatically applied "by group". For example, if we applied exactly the same code to a data frame grouped by site, we get the average temperature per site:

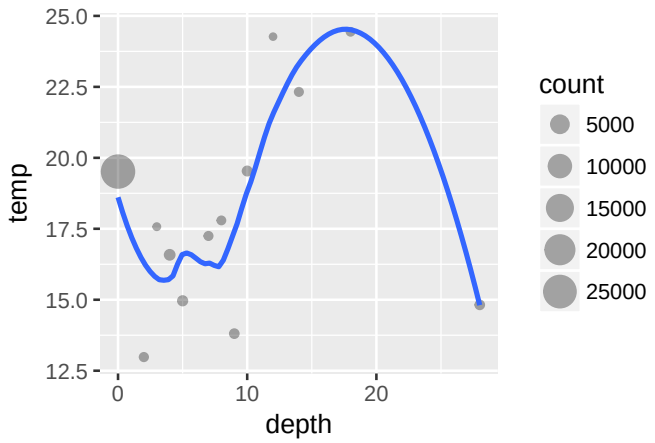
```
by_site <- group_by(temp, site, src)
head(summarise(by_site, temp = mean(temp, na.rm = TRUE)))
R> # A tibble: 6 x 3
R> # Groups:   site [4]
R>            site   src  temp
R>          <fctr> <chr> <dbl>
R> 1  Port Nolloth   DEA  11.8
R> 2  Port Nolloth  SAWS  12.3
R> 3 Hondeklipbaai  SAWS  12.1
R> 4    Doringbaai  DAFF  13.3
R> 5    Doringbaai  SAWS  12.8
R> 6  Lamberts Bay  SAWS  13.2
```

Together `group_by()` and `summarise()` provide one of the tools that you'll use most commonly when working with **dplyr**: grouped summaries. But before we go any further with this, we need to introduce a powerful new idea: the pipe.

11.6.1 Combining multiple operations with the pipe

Imagine that we want to explore the relationship between the average temperate and he depth of collection at each site. Using what you know about **dplyr**, you might write code like this:

```
by_depth <- group_by(temp, depth)
depths <- summarise(by_depth,
  count = n(),
  temp = mean(temp, na.rm = TRUE)
)

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



There are three steps to prepare these data:

1. Group sites by depth.
2. Summarise to compute number of entries per depth as well as the mean temperature.

This code is a little frustrating to write because we have to give each intermediate data frame a name, even though we don't care about it. Naming things requires active effort, so this slows down our analysis.

There's another way to tackle the same problem with the pipe, `%>%`:

```
depths <- temp %>%
  group_by(depth) %>%
  summarise(
    count = n(),
    temp = mean(temp, na.rm = TRUE)
  )
```

This focuses on the transformations, not what's being transformed, which makes the code easier to read. You can read it as a series of imperative statements: group, then summarise. As suggested by this reading, a good way to pronounce `%>%` when reading code is "then".

Behind the scenes, `x %>% f(y)` turns into `f(x, y)`, and `x %>% f(y) %>% g(z)` turns into `g(f(x, y), z)` and so on. You can use the pipe to rewrite multiple operations in a way that you can read left-to-right, top-to-bottom. We'll use piping frequently from now on because it considerably improves the readability of code.

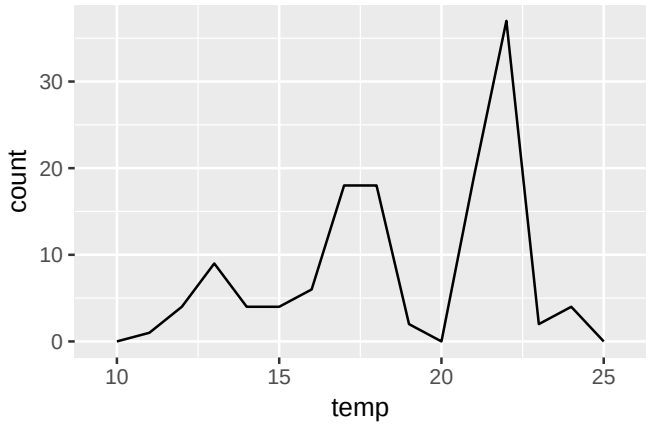
Working with the pipe is one of the key criteria for belonging to the tidyverse. The only exception is ggplot2: it was written before the pipe was discovered. Unfortunately, the next iteration of ggplot2, ggvis, which does use the pipe, isn't quite ready for prime time yet.

11.6.2 Counts

Whenever you do any aggregation, it's always a good idea to include either a count (`n()`), or a count of non-missing values (`sum(!is.na(x))`). That way you can check that you're not drawing conclusions based on very small amounts of data. For example, let's look at the sites based on their mean temperatures:

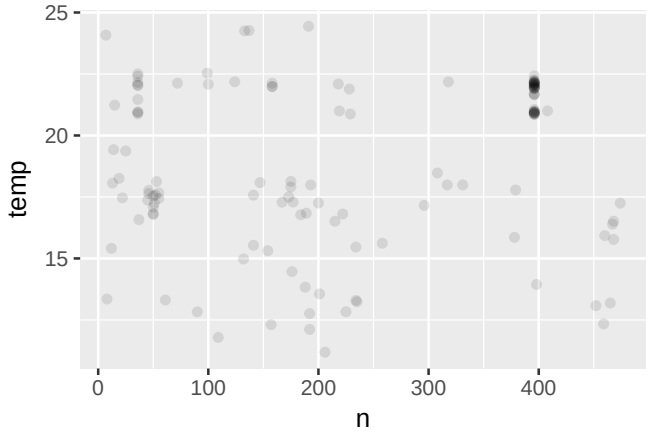
```
temp_2 <- temp %>%
  group_by(site, src) %>%
```

```
  summarise(
    temp = mean(temp, na.rm = T)
  )

ggplot(data = temp_2, mapping = aes(x = temp)) +
  geom_freqpoly(binwidth = 1)
```



This figure would have you believe that the majority of the coastline is around 24°C but we can guess that there is probably more going on than that. We can get more insight if we draw a scatterplot of number of samples per site vs. mean temperatures:

```
temp_3 <- temp %>%
  group_by(site, src) %>%
  summarise(
    temp = mean(temp, na.rm = TRUE),
    n = n()
  )

ggplot(data = temp_3, mapping = aes(x = n, y = temp)) +
  geom_point(alpha = 1/10)
```

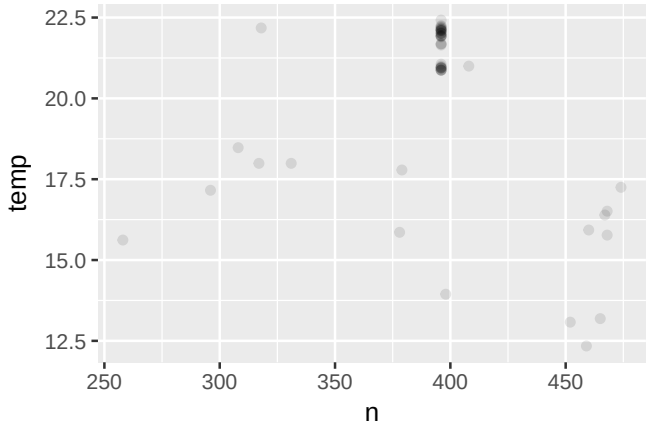


Not surprisingly, we are able to see another facet of the data when we visualise the values

differently.

When looking at this sort of plot, it's often useful to filter out the groups with the smallest numbers of observations, so you can see more of the pattern and less of the extreme variation or undue influence possible from shorter time series. This is what the following code does, as well as showing you a handy pattern for integrating `ggplot2` into `dplyr` flows. It's a bit painful that you have to switch from `%>%` to `+`, but once you get the hang of it, it's quite convenient.

```
temp_3 %>%
  filter(n > 240) %>%
  ggplot(mapping = aes(x = n, y = temp)) +
    geom_point(alpha = 1/10)
```



11.6.3 Useful summary functions

Just using means, counts, and sum can get you a long way, but R provides many other useful summary functions:

- Measures of location: we've used `mean(x)`, but `median(x)` is also useful. The mean is the sum divided by the length; the median is a value where 50% of `x` is above it, and 50% is below it.

```
head(temp %>%
  group_by(site, src) %>%
  summarise(
    mean_temp = mean(temp, na.rm = T),
    median_temp = median(temp, na.rm = T)
  )
)
R> # A tibble: 6 x 4
R> # Groups:   site [4]
R>            site   src mean_temp median_temp
R>          <fctr> <chr>     <dbl>       <dbl>
R> 1  Port Nolloth   DEA      11.8        11.8
R> 2  Port Nolloth  SAWS      12.3        12.3
R> 3 Hondeklipbaai  SAWS      12.1        12.0
R> 4    Doringbaai  DAFF      13.3        13.1
R> 5    Doringbaai  SAWS      12.8        12.7
```

```
R> 6  Lamberts Bay  SAWS      13.2        13.1
```

- Measures of spread: `sd(x)`, `IQR(x)`, `mad(x)`. The mean squared deviation, or standard deviation or sd for short, is the standard measure of spread. The interquartile range `IQR()` and median absolute deviation `mad(x)` are robust equivalents that may be more useful if you have outliers.

```
# Why is the variance in temperature at some sites greater than to others?
head(temp %>%
  group_by(site, src) %>%
  summarise(temp_sd = sd(temp, na.rm = T),
            temp_IQR = IQR(temp, na.rm = T),
            temp_mad = mad(temp, na.rm = T)) %>%
  arrange(desc(temp_sd)),
  6)
R> # A tibble: 6 x 5
R> # Groups:   site [5]
R>              site   src temp_sd temp_IQR temp_mad
R>            <fctr> <chr>   <dbl>    <dbl>    <dbl>
R> 1      Muizenberg  SAWS    2.76     4.77     3.54
R> 2        Stilbaai  SAWS    2.72     4.70     3.50
R> 3      Mossel Bay  SAWS    2.65     4.60     3.17
R> 4         De Hoop  DAFF    2.51     4.34     2.85
R> 5      Mossel Bay   DEA    2.51     4.16     2.78
R> 6 Plettenberg Bay  SAWS    2.50     2.60     1.88
```

- Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`. Quantiles are a generalisation of the median. For example, `quantile(x, 0.25)` will find a value of `x` that is greater than 25% of the values, and less than the remaining 75%.

```
head(temp %>%
  group_by(site, src) %>%
  summarise(
    first = min(date),
    last = max(date)
  ) %>%
  arrange(desc(last)),
  10)
R> # A tibble: 10 x 4
R> # Groups:   site [10]
R>              site   src      first       last
R>            <fctr> <chr>     <dttm>     <dttm>
R> 1     Buffelsbaai   UWC 2015-05-01 2016-04-01
R> 2   Eerste Rivier SAEON 2012-01-01 2016-03-01
R> 3       Oudekraal  DAFF 2003-02-01 2016-02-01
R> 4         De Hoop  DAFF 2014-05-01 2016-02-01
R> 5       Noordhoek SAEON 2011-10-01 2016-02-01
R> 6 Schoenmakerskop SAEON 2011-10-01 2016-01-01
R> # ... with 4 more rows
```

- Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`. These work similarly to `x[1]`, `x[2]`, and `x[length(x)]` but let you set a default value if that position does not exist (i.e. you're trying to get the 3rd element from a group that only has two elements). For example, we can

find the first and last departure for each day:

```
head(temp %>%
  group_by(site, src) %>%
  summarise(
    first_date = first(date),
    last_date = last(date)
  )
)
R> # A tibble: 6 x 4
R> # Groups:   site [4]
R>            site   src first_date  last_date
R>          <fctr> <chr>     <dttm>     <dttm>
R> 1  Port Nolloth   DEA 1991-02-01 2000-02-01
R> 2  Port Nolloth  SAWS 1973-07-01 2011-09-01
R> 3 Hondeklipbaai  SAWS 1990-09-01 2006-08-01
R> 4    Doringbaai  DAFF 2010-11-01 2015-11-01
R> 5    Doringbaai  SAWS 1990-09-01 2006-08-01
R> 6  Lamberts Bay  SAWS 1972-11-01 2011-08-01
```

- Counts: You've seen `n()`, which takes no arguments, and returns the size of the current group. To count the number of non-missing values, use `sum(!is.na(x))`. To count the number of distinct (unique) values, use `n_distinct(x)`.

```
# Which type of instrumentation is used most?
temp %>%
  group_by(type) %>%
  summarise(n_sites = n_distinct(site, src)) %>%
  arrange(desc(n_sites))
R> # A tibble: 2 x 2
R>     type n_sites
R>    <chr>   <int>
R> 1 thermo      86
R> 2    UTR      42
```

Counts are so useful that dplyr provides a simple helper if all you want is a count:

```
head(temp %>%
  count(site, src)
  )
R> # A tibble: 6 x 3
R>            site   src     n
R>          <fctr> <chr> <int>
R> 1  Port Nolloth   DEA   109
R> 2  Port Nolloth  SAWS   459
R> 3 Hondeklipbaai  SAWS   192
R> 4    Doringbaai  DAFF    61
R> 5    Doringbaai  SAWS   192
R> 6  Lamberts Bay  SAWS   465
```

You can optionally provide a weight variable. For example, you could use this to "count" (sum) the total temperature recorded at a site (you would never actually do this):

```
head(temp %>%
  count(site, src, wt = temp)
)
R> # A tibble: 6 x 3
R>            site   src     n
R>           <fctr> <chr> <dbl>
R> 1  Port Nolloth   DEA  1120
R> 2  Port Nolloth  SAWS  5566
R> 3 Hondeklipbaai  SAWS  2241
R> 4    Doringbaai  DAFF   812
R> 5    Doringbaai  SAWS  2335
R> 6  Lamberts Bay  SAWS  6066
```

- Counts and proportions of logical values: `sum(x > 10)`, `mean(y == 0)`. When used with numeric functions, `TRUE` is converted to 1 and `FALSE` to 0. This makes `sum()` and `mean()` very useful: `sum(x)` gives the number of `TRUE`s in `x`, and `mean(x)` gives the proportion.

```
# How many recordings per source are above 15C?
head(temp %>%
  na.omit() %>%
  group_by(src) %>%
  summarise(n_15 = sum(temp > 15))
)
R> # A tibble: 6 x 2
R>     src  n_15
R>   <chr> <int>
R> 1  DAFF   246
R> 2   DEA  1388
R> 3 EKZNW   369
R> 4 KZNSB 15313
R> 5 SAEON   573
R> 6  SAWS  4882
```

11.6.4 Grouping by multiple variables

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll up a dataset:

```
temp_4 <- group_by(temp, site, src, date)
head(per_date   <- summarise(temp_4, recordings = n()))
R> # A tibble: 6 x 4
R> # Groups:   site, src [1]
R>           site   src       date recordings
R>         <fctr> <chr>     <dttm>      <int>
R> 1 Port Nolloth   DEA 1991-02-01          1
R> 2 Port Nolloth   DEA 1991-03-01          1
R> 3 Port Nolloth   DEA 1991-04-01          1
R> 4 Port Nolloth   DEA 1991-05-01          1
R> 5 Port Nolloth   DEA 1991-06-01          1
R> 6 Port Nolloth   DEA 1991-07-01          1
head(per_src <- summarise(per_date, recordings = sum(recordings)))
R> # A tibble: 6 x 3
```

```
R> # Groups:   site [4]
R>            site   src recordings
R>          <fctr> <chr>      <int>
R> 1  Port Nolloth   DEA        109
R> 2  Port Nolloth  SAWS        459
R> 3 Hondeklipbaai  SAWS        192
R> 4    Doringbaai  DAFF         61
R> 5    Doringbaai  SAWS        192
R> 6  Lamberts Bay  SAWS        465
head(per_site  <- summarise(per_src, recordings = sum(recordings)))
R> # A tibble: 6 x 2
R>            site recordings
R>          <fctr>      <int>
R> 1  Port Nolloth        568
R> 2 Hondeklipbaai        192
R> 3    Doringbaai        253
R> 4  Lamberts Bay        465
R> 5    Elands Bay         90
R> 6 St Helena Bay        176
```

Be careful when progressively rolling up summaries: it's OK for sums and counts, but you need to think about weighting means and variances, and it's not possible to do it exactly for rank-based statistics like the median. In other words, the sum of groupwise sums is the overall sum, but the median of groupwise medians is not the overall median.

11.6.5 Ungrouping

If you need to remove grouping, and return to operations on ungrouped data, use `ungroup()`.

```
temp_4 %>%
  ungroup() %>% # no longer grouped by site
  summarise(recordings = n())  # all recordings
R> # A tibble: 1 x 1
R>   recordings
R>        <int>
R> 1      29572
```

11.7 Grouped mutates (and filters)

Grouping is most useful in conjunction with `summarise()`, but you can also do convenient operations with `mutate()` and `filter()`:

- Find values based on rank:

```
temp %>%
  group_by(src) %>%
  filter(rank(desc(temp)) == 3)
R> # A tibble: 7 x 6
R> # Groups:   src [7]
R>           site   src       date  temp depth   type
R>         <fctr> <chr>     <dttm> <dbl> <dbl>  <chr>
R> 1  Buffelsbaai   UWC 2016-01-01  16.8     2    UTR
```

```
R> 2      De Hoop  DAFF 2015-12-01  21.2     0    UTR
R> 3        Dwesa SAEON 2010-04-01  22.0     0    UTR
R> 4 Warner Beach KZNSB 2012-03-01  26.2     0 thermo
R> 5 Richards Bay  SAWS 1991-03-01  25.3     0 thermo
R> 6      Sodwana   DEA 1998-03-01  27.9    18    UTR
R> # ... with 1 more rows
```

- Find all groups bigger than a threshold:

```
temp_5 <- temp %>%
  group_by(site, src) %>%
  filter(n() > 240)
```

- Standardise to compute per group metrics:

```
head(temp %>%
  mutate(anom = temp - mean(temp, na.rm = T)) %>%
  select(site:date, anom, depth, type)
)
R>           site src       date  anom depth type
R> 1 Port Nolloth DEA 1991-02-01 -7.80     5  UTR
R> 2 Port Nolloth DEA 1991-03-01 -7.28     5  UTR
R> 3 Port Nolloth DEA 1991-04-01 -7.31     5  UTR
R> 4 Port Nolloth DEA 1991-05-01 -7.41     5  UTR
R> 5 Port Nolloth DEA 1991-06-01 -7.06     5  UTR
R> 6 Port Nolloth DEA 1991-07-01 -6.73     5  UTR
```

Now, let us select two of the sites only. Let us take all the temperatures at Paternoster and at Oudekraal, and then we calculate a mean and standard deviation for each of these sites.

```
# library(dplyr)
temp %>%
  filter(site == "Paternoster" | site == "Oudekraal") %>%
  group_by(site) %>%
  summarise(mean = mean(temp, na.rm = TRUE),
            sd = sd(temp, na.rm = TRUE))
R> # A tibble: 2 x 3
R>          site  mean    sd
R>        <fctr> <dbl> <dbl>
R> 1 Paternoster  13.1  1.14
R> 2   Oudekraal  12.3  1.36
```

What if we had more sites than only two as in the above example? This is a better way than using the | (or) operator:

```
selected_sites <- c("Paternoster", "Oudekraal", "Muizenberg", "Humewood")
temp %>%
  filter(site %in% selected_sites) %>%
  group_by(site) %>%
  summarise(temp_mean = mean(temp, na.rm = TRUE),
            temp_sd = sd(temp, na.rm = TRUE))
R> # A tibble: 4 x 3
R>          site temp_mean temp_sd
R>        <fctr>     <dbl>   <dbl>
R> 1 Paternoster      13.1    1.14
```

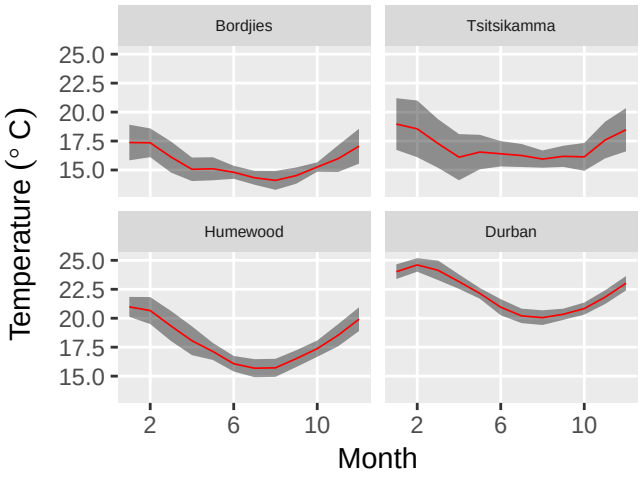
```
R> 2   Oudekraal      12.3    1.36
R> 3  Muizenberg      15.9    2.76
R> 4    Humewood      18.0    2.03
```

Okay, thinking back to the spreadsheet example we worked on the first day of the R workshop… here it is repeated in a more efficient way. We need to calculate an overall mean for each month at each site over all of the years. Here is how:

```
clim <- temp %>%
  select(-depth, -type) %>%
  mutate(month = month(date)) %>%
  group_by(site, month) %>%
  summarise(temp_mean = mean(temp, na.rm = TRUE),
            temp_sd = sd(temp, na.rm = TRUE))
# use various means to your disposal to see what's inside the new dataframe...
```

We can also bring `ggplot2` into this pipeline:

```
temp %>%
  filter(site %in% c("Bordjies", "Tsitsikamma", "Humewood", "Durban")) %>%
  select(-depth, -type) %>%
  mutate(month = month(date)) %>%
  group_by(site, month) %>%
  summarise(temp_mean = mean(temp, na.rm = TRUE),
            temp_sd = sd(temp, na.rm = TRUE)) %>%
  ggplot(aes(month, temp_mean, site)) +
  geom_ribbon(aes(ymin = temp_mean - temp_sd, ymax = temp_mean + temp_sd),
              fill = "black", alpha = 0.4) +
  geom_line(col = "red", size = 0.3) +
  facet_wrap(~site) +
  scale_x_continuous(breaks = seq(2, 12, 4)) +
  labs(x = "Month", y = expression(paste("Temperature ", (degree~C)))) +
  theme(aspect.ratio = 0.6,
        strip.text = element_text(size = 6),
        panel.grid.minor = element_blank())
```



I'm a tidy, neat person. But I'm not a maniac.

Jamie Lee Curtis

There is nothing quite so good as burial at sea. It is simple, tidy, and not very incriminating.

Alfred Hitchcock

12

Tidy data with `tidyr`

Tidying data forms part of the series of steps one does immediately after importing data. It precedes data transformation and visualisation (Figure 12.1).

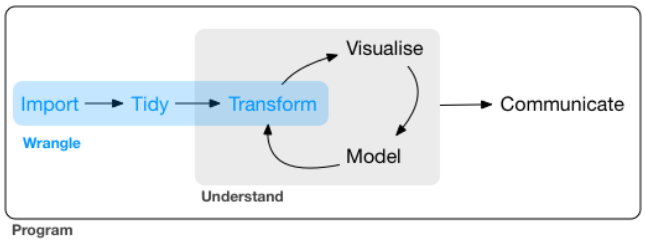


Figure 12.1: Data tidying in the data processing pipeline.

In order to follow along with these examples, you need to load the `tidyverse` packages if you have not already. The example dataset that we will be using here was produced by the World Health Organization, and it reports the new cases of tuberculosis infections for the world's countries:

```
library(tidyverse)
?who # a dataset in the 'tidyr' package; command displays the help file
```

The help file displays this at the bottom:

> "The data uses the original codes given by the World Health Organization. The column names for columns five through 60 are made by combining new_ to a code for method of diagnosis (`rel` = relapse, `sn` = negative pulmonary smear, `sp` = positive pulmonary smear, `ep` = extrapulmonary) to a code for gender (`f` = female, `m` = male) to a code for age group (`014` = 0-14 yrs of age, `1524` = 15-24 years of age, `2534` = 25 to 34 years of age, `3544` = 35 to 44 years of age, `4554` = 45 to 54 years of age, `5564` = 55 to 64 years of age, `65` = 65 years of age or older)."

Here we will focus on a portion of the data. You can represent the same underlying data in multiple ways. The example below shows the same data organised in four different ways. Each dataset shows the same values of four variables `country`, `year`, `population` and `cases`, but each dataset organises the values in a different way. Also worth noticing is that the dataframe is now presented as a *tibble*, which is automagically created using the `as_tibble()` function that lives within the Tidyverse package called **`tibble`**. For all (most?) practical purposes it is the same as a normal dataframe, but with some convenience properties that make printing it to screen a bit less annoying in some ways.

```
table1
R> # A tibble: 6 x 4
R>       country  year  cases population
R>         <chr> <int>  <int>      <int>
R> 1 Afghanistan  1999    745   19987071
R> 2 Afghanistan  2000   2666   20595360
R> 3      Brazil  1999  37737  172006362
R> 4      Brazil  2000  80488  174504898
R> 5       China  1999 212258 1272915272
R> 6       China  2000 213766 1280428583
table2
R> # A tibble: 12 x 4
R>       country  year       type     count
R>         <chr> <int>      <chr>     <int>
R> 1 Afghanistan  1999      cases       745
R> 2 Afghanistan  1999 population  19987071
R> 3 Afghanistan  2000      cases      2666
R> 4 Afghanistan  2000 population  20595360
R> 5      Brazil  1999      cases     37737
R> 6      Brazil  1999 population 172006362
R> # ... with 6 more rows
table3
R> # A tibble: 6 x 3
R>       country  year              rate
R> *       <chr> <int>             <chr>
R> 1 Afghanistan  1999      745/19987071
R> 2 Afghanistan  2000     2666/20595360
R> 3      Brazil  1999   37737/172006362
R> 4      Brazil  2000   80488/174504898
R> 5       China  1999 212258/1272915272
R> 6       China  2000 213766/1280428583

# Spread across two tibbles
table4a  # cases
R> # A tibble: 3 x 3
R>       country `1999` `2000`
R> *       <chr>  <int>  <int>
R> 1 Afghanistan    745   2666
R> 2      Brazil  37737  80488
R> 3       China 212258 213766
table4b  # population
R> # A tibble: 3 x 3
R>       country     `1999`     `2000`
```

```
R> *       <chr>      <int>      <int>
R> 1 Afghanistan   19987071   20595360
R> 2      Brazil  172006362  174504898
R> 3       China 1272915272 1280428583
```

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset, will be much easier to work with inside the tidyverse.

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

Figure 12.2 shows the rules visually.

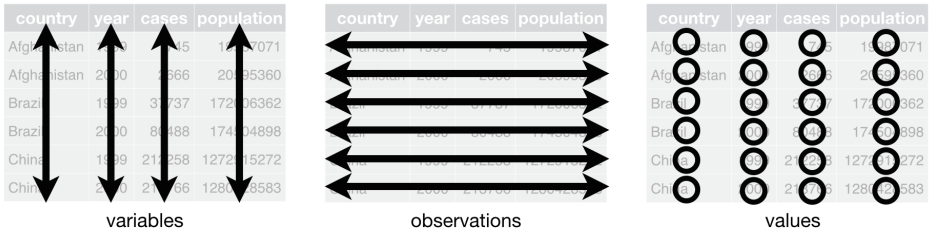


Figure 12.2: Following three rules make a dataset tidy — variables are in columns, observations are in rows, and values are in cells.

These three rules are interrelated because it's impossible to only satisfy two of the three. That interrelationship leads to an even simpler set of practical instructions:

1. Put each dataset in a tibble.
2. Put each variable in a column.

In this example, only `table1` is tidy. It's the only representation where each column is a variable. This representation is also sometimes called *long* data in contrast to *wide* data, where some variables are sometimes spread across several columns, making it wider in shape than the corresponding long data.

Why ensure that your data are tidy? There are two main advantages:

1. There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.
2. There's a specific advantage to placing variables in columns because it allows R's vectorised nature to shine. As you learned in mutate and summary functions, most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

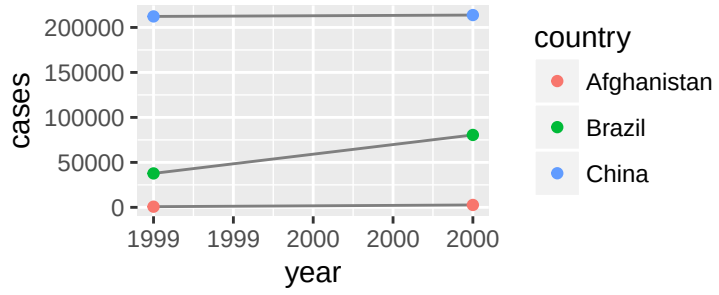
`dplyr`, `ggplot2`, and all the other packages in the tidyverse are designed to work with tidy data. Here are a couple of small examples showing how you might work with `table1`.

```
# compute rate per 10,000
mutate(table1, rate = cases / population * 10000)
```

```
R> # A tibble: 6 x 5
R>       country  year  cases population  rate
R>         <chr> <int>  <int>      <int> <dbl>
R> 1 Afghanistan  1999    745   19987071 0.373
R> 2 Afghanistan  2000   2666   20595360 1.294
R> 3      Brazil  1999  37737  172006362 2.194
R> 4      Brazil  2000  80488  174504898 4.612
R> 5       China  1999 212258 1272915272 1.667
R> 6       China  2000 213766 1280428583 1.669
```

```
# compute cases per year
count(table1, year, wt = cases)
R> # A tibble: 2 x 2
R>    year      n
R>   <int>  <int>
R> 1  1999 250740
R> 2  2000 296920
```

```
# visualise changes over time
ggplot(table1, aes(year, cases)) +
  geom_line(aes(group = country), colour = "grey50") +
  geom_point(aes(colour = country)) + theme(aspect.ratio = 0.6)
```



12.1 Spreading and gathering

The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons:

1. Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.
2. Data are often organised to facilitate some use other than analysis. For example, data are often organised to make entry as easy as possible.

This means for most real life analyses you'll need to do some tidying. The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. The second step is to resolve one of two common problems:

1. One variable might be spread across multiple columns.
2. One observation might be scattered across multiple rows.

Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in tidyr: `gather()` and `spread()`.

12.1.1 Gathering

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take `table4a`: the column names `1999` and `2000` represent values of the `year` variable, and each row represents two observations, not one.

```
table4a
R> # A tibble: 3 x 3
R>       country `1999` `2000`
R> *       <chr>  <int>  <int>
R> 1 Afghanistan    745   2666
R> 2      Brazil  37737  80488
R> 3       China 212258 213766
```

To tidy a dataset like this, we need to *gather* those columns into a new pair of variables. To describe that operation we need three parameters:

1. The set of columns that represent values, not variables. In this example, those are the columns `1999` and `2000`.
2. The name of the variable whose values form the column names. I call that the `key`, and here it is `year`.
3. The name of the variable whose values are spread over the cells. I call that `value`, and here it's the number of `cases`.

Together those parameters generate the call to `gather()`:

```
 gather(table4a, `1999`, `2000`, key = "year", value = "cases")
R> # A tibble: 6 x 3
R>       country  year  cases
R>         <chr> <chr>  <int>
R> 1 Afghanistan  1999    745
R> 2      Brazil  1999  37737
R> 3       China  1999 212258
R> 4 Afghanistan  2000   2666
R> 5      Brazil  2000  80488
R> 6       China  2000 213766
```

The columns to gather are specified with `dplyr::select()` style notation. Here there are only two columns, so we list them individually. Note that "1999" and "2000" are non-syntactic names so we have to surround them in backticks. To refresh your memory of the other ways to select columns, see select.

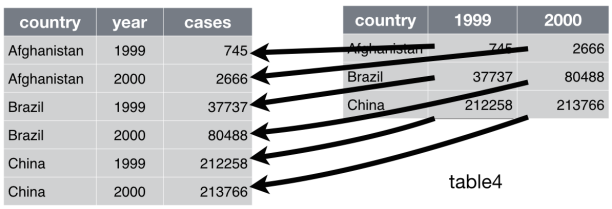


Figure 12.3: Gathering `table4` into a tidy form.

In the final result, the gathered columns are dropped, and we get new `key` and `value` columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in Figure 12.3. We can use `gather()` to tidy `table4b` in a similar fashion. The only difference is the variable stored in the cell values:

```
gather(table4b, `1999`, `2000`, key = "year", value = "population")
R> # A tibble: 6 x 3
R>       country  year population
R>         <chr> <chr>      <int>
R> 1 Afghanistan  1999   19987071
R> 2      Brazil  1999  172006362
R> 3       China  1999 1272915272
R> 4 Afghanistan  2000   20595360
R> 5      Brazil  2000  174504898
R> 6       China  2000 1280428583
```

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we need to use `dplyr::left_join()`.

```
tidy4a <- gather(table4a, `1999`, `2000`, key = "year", value = "cases")
tidy4b <- gather(table4b, `1999`, `2000`, key = "year", value = "population")
left_join(tidy4a, tidy4b)
R> Joining, by = c("country", "year")
R> # A tibble: 6 x 4
R>       country  year  cases population
R>         <chr> <chr>  <int>      <int>
R> 1 Afghanistan  1999    745   19987071
R> 2      Brazil  1999  37737  172006362
R> 3       China  1999 212258 1272915272
R> 4 Afghanistan  2000   2666   20595360
R> 5      Brazil  2000  80488  174504898
R> 6       China  2000 213766 1280428583
```

12.1.2 Spreading

Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows. For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

```
table2
R> # A tibble: 12 x 4
R>       country  year       type     count
R>         <chr> <int>      <chr>     <int>
R> 1 Afghanistan  1999      cases       745
R> 2 Afghanistan  1999 population  19987071
R> 3 Afghanistan  2000      cases      2666
R> 4 Afghanistan  2000 population  20595360
R> 5      Brazil  1999      cases     37737
R> 6      Brazil  1999 population 172006362
R> # ... with 6 more rows
```

To tidy this up, we first analyse the representation in similar way to `gather()`. This time, however, we only need two parameters:

1. The column that contains variable names, the `key` column. Here, it's `type`.
2. The column that contains values forms multiple variables, the `value` column. Here it's `count`.

Once we've figured that out, we can use `spread()`, as shown programmatically below, and visually in Figure 12.4.

```
spread(table2, key = type, value = count)
R> # A tibble: 6 x 4
R>       country  year  cases population
R> *       <chr> <int>  <int>      <int>
R> 1 Afghanistan  1999    745   19987071
R> 2 Afghanistan  2000   2666   20595360
R> 3      Brazil  1999  37737  172006362
R> 4      Brazil  2000  80488  174504898
R> 5       China  1999 212258 1272915272
R> 6       China  2000 213766 1280428583
```

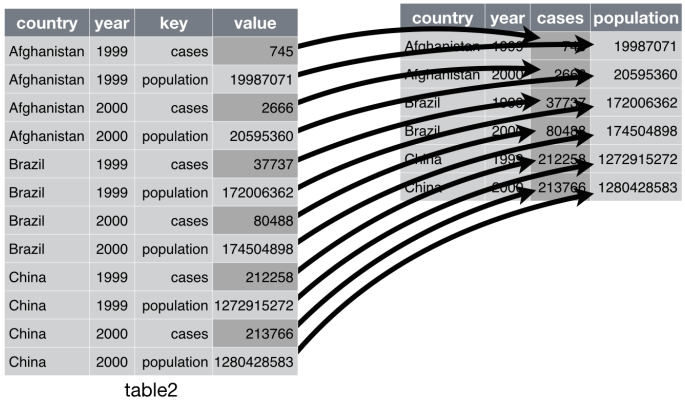


Figure 12.4: Spreading 'table2' makes it tidy

As you might have guessed from the common `key` and `value` arguments, `spread()` and `gather()` are complements. `gather()` makes wide tables narrower and longer; `spread()` makes long tables shorter and wider.

12.2 Separating and uniting

So far you've learned how to tidy `table2` and `table4`, but not `table3`. `table3` has a different problem: we have one column (`rate`) that contains two variables (`cases` and `population`). To fix this problem, we'll need the `separate()` function. You'll also learn about the complement of `separate()`: `unite()`, which you use if a single variable is spread across multiple columns.

12.2.1 Separate

`separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take `table3`:

```
table3
R> # A tibble: 6 x 3
R>       country  year              rate
R> *       <chr> <int>             <chr>
R> 1 Afghanistan  1999      745/19987071
R> 2 Afghanistan  2000     2666/20595360
R> 3      Brazil  1999   37737/172006362
R> 4      Brazil  2000   80488/174504898
R> 5       China  1999 212258/1272915272
R> 6       China  2000 213766/1280428583
```

The `rate` column contains both `cases` and `population` variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into, as shown in Figure 12.5 and the code below.

```
separate(table3,  rate, into = c("cases", "population"))
R> # A tibble: 6 x 4
R>       country  year  cases population
R> *       <chr> <int>  <chr>      <chr>
R> 1 Afghanistan  1999    745   19987071
R> 2 Afghanistan  2000   2666   20595360
R> 3      Brazil  1999  37737  172006362
R> 4      Brazil  2000  80488  174504898
R> 5       China  1999 212258 1272915272
R> 6       China  2000 213766 1280428583
```

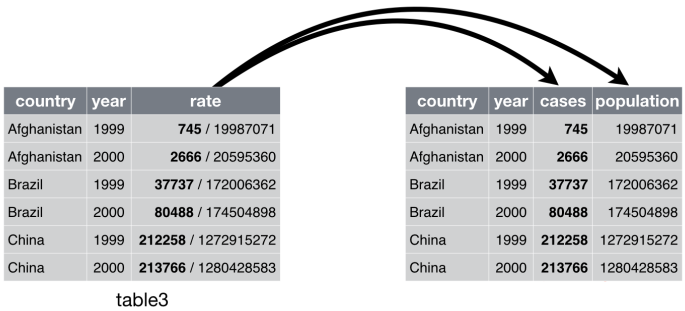


Figure 12.5: Separating 'table3' makes it tidy

By default, `separate()` will split values wherever it sees a non-alphanumeric character (i.e. a

character that isn't a number or letter). For example, in the code above, `separate()` split the values of `rate` at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the `sep` argument of `separate()`. For example, we could rewrite the code above as:

```
separate(table3, rate, into = c("cases", "population"), sep = "/")
```

Look carefully at the column types: you'll notice that `case` and `population` are character columns. This is the default behaviour in `separate()`: it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using `convert = TRUE`:

```
separate(table3, rate, into = c("cases", "population"), convert = TRUE)
R> # A tibble: 6 x 4
R>       country  year  cases population
R> *       <chr> <int>  <int>      <int>
R> 1 Afghanistan  1999    745   19987071
R> 2 Afghanistan  2000   2666   20595360
R> 3      Brazil  1999  37737  172006362
R> 4      Brazil  2000  80488  174504898
R> 5       China  1999 212258 1272915272
R> 6       China  2000 213766 1280428583
```

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far-left of the strings; negative value start at -1 on the far-right of the strings. When using integers to separate strings, the length of `sep` should be one less than the number of names in `into`.

You can use this arrangement to separate the last two digits of each year. This make this data less tidy, but is useful in other cases, as you'll see in a little bit.

```
separate(table3, year, into = c("century", "year"), sep = 2)
R> # A tibble: 6 x 4
R>       country century  year              rate
R> *       <chr>   <chr> <chr>             <chr>
R> 1 Afghanistan      19    99       745/19987071
R> 2 Afghanistan      20    00      2666/20595360
R> 3      Brazil      19    99    37737/172006362
R> 4      Brazil      20    00    80488/174504898
R> 5       China      19    99 212258/1272915272
R> 6       China      20    00 213766/1280428583
```

12.2.2 Unite

`unite()` is the inverse of `separate()`: it combines multiple columns into a single column. You'll need it much less frequently than `separate()`, but it's still a useful tool to have in your back pocket.

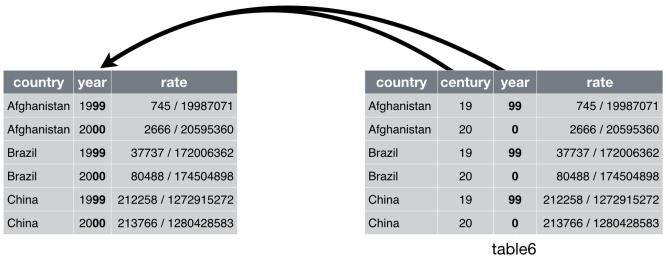


Figure 12.6: Uniting ‘table5‘ makes it tidy

We can use `unite()` to rejoin the *century* and *year* columns that we created in the last example. That data is saved as `tidyr::table5`. `unite()` takes a data frame, the name of the new variable to create, and a set of columns to combine, again specified in `dplyr::select()` style:

```
 unite(table5, new, century, year)
R> # A tibble: 6 x 3
R>       country   new              rate
R> *       <chr> <chr>             <chr>
R> 1 Afghanistan 19_99       745/19987071
R> 2 Afghanistan 20_00      2666/20595360
R> 3      Brazil 19_99    37737/172006362
R> 4      Brazil 20_00    80488/174504898
R> 5       China 19_99 212258/1272915272
R> 6       China 20_00 213766/1280428583
```

In this case we also need to use the `sep` argument. The default will place an underscore (_) between the values from different columns. Here we don't want any separator so we use `""`:

```
unite(table5, new, century, year, sep = "")
R> # A tibble: 6 x 3
R>       country   new              rate
R> *       <chr> <chr>             <chr>
R> 1 Afghanistan  1999       745/19987071
R> 2 Afghanistan  2000      2666/20595360
R> 3      Brazil  1999    37737/172006362
R> 4      Brazil  2000    80488/174504898
R> 5       China  1999 212258/1272915272
R> 6       China  2000 213766/1280428583
```

Part V

Day 5

Everyone should have their mind blown once a day.

Neil deGrasse Tyson

Somewhere, something incredible is waiting to be known.

Carl Sagan

13 Final project

You now have all of the skills required to perform a full data analysis in R yourself. The aim then of this final project is to spend two hours interrogating your data, performing analyses, creating figures, and authoring & publishing all of it via R Markdown.

13.1 Instructions

In groups no larger than 5, and using `data/ecklonia.csv`, think of an idea that interests you that you can test. The possibilities are very broad. Avoid asking the same question as any other group. At the end of the following ~2 hours you (or your group) should have an R Markdown file (.Rmd) that may be successfully compiled to a .html or .doc format.

13.2 Requirements

There is no specific page limit/ requirement but with figures, the length of this document should not exceed 3-5 pages. Below is a list of the requirements to be achieved. Remember, all of your work (writing and code) must be entered into a .Rmd file that you then knit into a document that shows your text, code and figures seamlessly together.

- Content:
  - Aims and objectives
  - State a hypotheses
  - Provide a summary table of the relevant data
  - Methodology
    - Perform a statistical test
  - Results (What do the results show?)
  - Create 2 figures:
    - Avoid default looking figures
    - Your figures should be publication quality
  - Conclusion (What does it all mean?)

    - References:
        * End with a reference section
        * Strive to have at least 2 references
- Final product:
    - The .Rmd file
    - The knited .HTML and .doc files
- Bonus:
    - Demonstrate a data transformation not shown in class (+5%)
    - Create a graph type we have not shown in class (+5%)
        * Or create a custom theme for your figures

13.3 Example

Here we show a brief example based on work with the SACTN data. Note that this is not a template to follow, just some ideas to help you along with your work. You'll want your final report to look more professional.

```
# Load packages
library(ggplot2)

# Load temperature data
SACTN_data <- read.csv("data/SACTN_data.csv")
SACTN_data <- subset(SACTN_data, site != "Knysna")
SACTN_data$date <- as.Date(SACTN_data$date)

# Load site information
load("data/site_list.Rdata")
site_list <- subset(site_list, site == "Port Nolloth" | site == "Muizenberg")

# Load SA map
load("data/south_africa_coast.Rdata")
```

13.3.1 Aims and objectives

The aim of this research is to show if there are statistical differences between the seawater temperature at two different sites. The objective is to produce statistical and graphical results that answer this question.

13.3.2 Hypothesis

> H0: There is no difference between the seawater temperature at Port Nolloth and Muizenberg H1: There is a difference between the seawater temperature at Port Nolloth and Muizenberg

13.3.3 Methodology

SAWS (South African Weather Service) and their citizen scientists collected seawater temperature at the three sites shown in Figure 1. The difference between the data at the two sites was calculated with a t-test (remember to also check for similarity of variance and homoscedasticity).

```
# Map showing sample sites
ggplot(data = south_africa_coast, aes(x = lon, y = lat)) +
  geom_polygon(aes(group = group), fill = "grey70", colour = "black") +
  geom_point(data = site_list, aes(fill = site), size = 5, shape = 25, alpha = 0.6) +
  scale_fill_manual(values = c("deeppink", "chocolate")) +
  labs(x = "", y = "") +
  coord_equal() +
  theme(legend.background = element_rect(colour = "white"),
        legend.justification = c(0, 0),
        legend.position = c(0.4, 0.5),
        panel.background = element_rect(fill = "powderblue")
        )
```

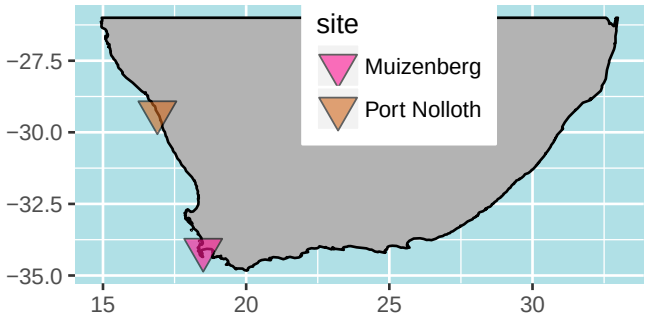


Figure 13.1: Map of South Africa showing the two sites in this study.

```
# Create two  different dataframes for ease of use
PN <- subset(SACTN_data, site == "Port Nolloth")
MZ <- subset(SACTN_data, site == "Muizenberg")

# Perform t-test
results <- t.test(PN$temp, MZ$temp)
```

13.3.4 Results

With a t value of -26.51 and a p-value of 0.00 we see that there is a significant difference between the seawater temperatures at Muizenberg and Port Nolloth. This difference in temperature is evident when we visualise the data. Below in Figure 2 we see the data displayed with box and whisker plots to sow the difference in the spread of the data. Also shown in Figure 3 is a line graph of these two different time series.

```
ggplot(data = SACTN_data, aes(y = temp, x = site)) +
  geom_boxplot(aes(fill = site)) +
  geom_point(position = "jitter", alpha = 0.2) +
  scale_fill_manual(values = c("deeppink", "chocolate")) +
  guides(fill = FALSE) +
  labs(x = "", y = "Temperature (°C)")
```

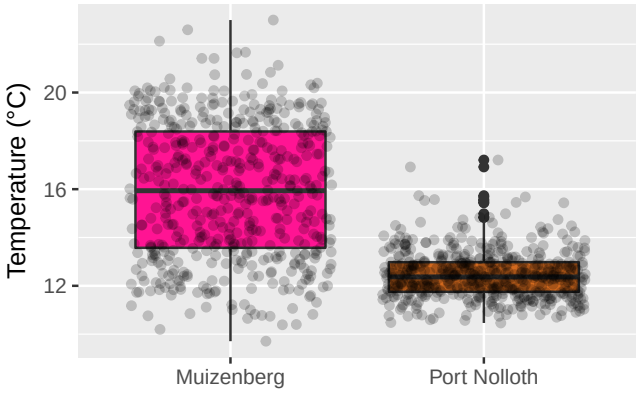


Figure 13.2: Boxplot showing range of temepratures at two sites.

```
ggplot(data = SACTN_data, aes(x = date, y = temp, colour = site)) +
  geom_line(aes(group = site)) +
  labs(x = "", y = "Temperature (°C)") +
  theme(axis.text.x = element_text(angle = 45)) +
  theme_linedraw() +
  theme(legend.position = "top")
```

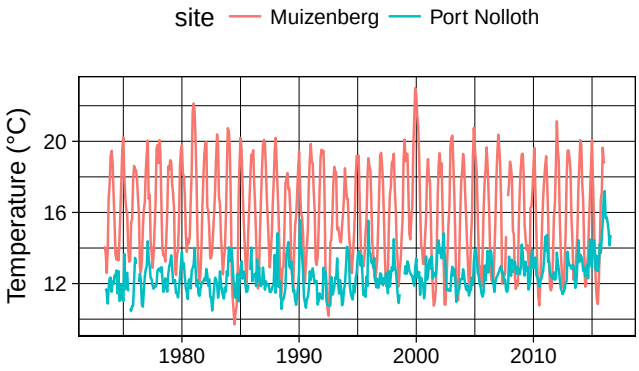


Figure 13.3: The two time series used in this study.

13.3.5 Conclusion

The significant difference in seawater temperature between Port Nolloth and Muizenberg implies that we may see different species assemblages at the two different sites. Because the mean temperature at Muizenberg (15.91°C) is greater than Port Nolloth (12.46°C) we may infer that the ecosystem at Muizenberg will have more species that may occur in temperate transition zones than Port Nolloth.

The analyses performed and the figures included in this document were done using the R software (R Core Team 2017).

References

Oksanen, Jari, F. Guillaume Blanchet, Michael Friendly, Roeland Kindt, Pierre Legendre, Dan McGlinn, Peter R. Minchin, et al. 2017. *Vegan: Community Ecology Package*. https://CRAN.R-project.org/package=vegan.

R Core Team. 2017. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.