Intro to R for Limnology

Cory Sauve

10/01/2020

Contents

1 Welcome!

Welcome to the written part of visualizing lake data in R! Everything covered in the lecture videos is covered here in much more detail. You'll also find information of how to install and set up R and RStudio, getting help, and additional R resources.

1.1 Workshop Contents

- **Getting Started**
  - Covers how to install the required software and how to organize everything you'll need
- **Data Analysis Crash Course**
  - Reviews some R basics and most of the functions you'll be using to create your figures
- **On to University Lake**
  - Covers how to import, manipulate, and visualize your University Lake Data
- **Troubleshooting**
  - Provides some basic troubleshooting tips if you run into problems
- **Ok, what's next?**
  - List of resources to look at if you want to learn more about R

1.2 A Suggested Workflow

There are multiple ways to successfully complete your figures. Here are a few strategies based on your potential situation:

1. Completely New to R

- Read completely through the *Getting Started* section and download all the materials you'll need
- Work through the *Data Analysis Crash Course*
- Watch the videos and follow along with the code outline
- Refer back to the **On to University Lake** and **Troubleshooting** sections if you get stuck.

2. Have some R Experience

- Read completely through the *Getting Started* section and download all the materials you'll need
- Skim the *Data Analysis Crash Course* section
- Watch the videos and follow along with the code outline
- Refer back to the **On to University Lake** and **Troubleshooting** sections if you get stuck.

3. Decided to start this the night before

- Make some coffee or drink a Red Bull
- Read completely through the *Getting Started* section and download all the materials you'll need
- Watch the videos (maybe at 1.5x speed) and follow along with the code outline
- If you get stuck, refer back to this guide (see **Troubleshooting**, **On to University Lake**)

2 Getting Started

2.1 Materials You'll Need

- **Computer**
  - Ideally one that runs Windows, macOS, or Linux. You can make a Chromebook work for what we're doing but will take a little more effort.
  - Ideally *your* computer. It's helpful to know that your files will be in the same place you left them (and to know R and R package versions will be the same). This isn't 100% necessary - and you will be able to finish everything regardless - but working on your own computer is definitely a proactive approach to avoid issues down the road.
- **Code Outline & Example Data**
  - Found on **Canvas**
    - Two example datasets: `water-chemisty.csv` and `plankton.csv`. Download and save in your project folder.
    - Code outline: `limno-workshop-student.Rmd`. Outlines the code used to make your figures (follows the video lectures). Download and save in your project folder.
- **Video lectures**
  - All lecture videos are hosted on YouTube **here**
    - **Importing and Manipulating the Data**
    - **Making the Water Chemistry Figures**
    - **Making the Plankton Figures**
    - **Making the Light Figure**
- **Written Materials**
  - Found on **Canvas**
    - `limno-workshop.pdf` or `limno-workshop.html`: contains everything covered in the video lectures, plus more.
    - `Required Figures for Lab Reports`: examples of all the figures you need to make

2.2 Install R and RStudio

We will use the open-source programming language **R** for this workshop. It's free (Yay!) and relatively easy to install on your own computer. We'll use **RStudio** to access R. You have several options to set up R and RStudio on your computer:

- **R & R Studio on your computer (recommended)**
  - R can be installed by:
    1. Go to **CRAN**, the Comprehensive R Archive Network
    2. Select the `Download R for` link that is appropriate for your computer
    3. Download the latest release by clicking the corresponding link
    4. Double click on the downloaded file (check `Downloads`). Follow the prompts to install.
    5. If you are using macOS, you'll also need to install XQuartz **here**.
  - RStudio can be installed by:
    1. Go to **RStudio**
    2. Click the `Download` button under the free, open-source license of **RStudio Desktop**
    3. Download the file that is appropriate for your computer, open the downloaded file, and follow the prompts.

  - If you already have R and/or R Studio installed, I **highly recommend** you re-install the most recent version of both. If you don't want to do that, make sure you update all CRAN packages with the following command:

```
update.packages(ask = FALSE, checkBuilt = TRUE)
```

- **RStudio Cloud**
  - While R is free and widely supported, sometimes it can be a headache to install and configure. If you would like to avoid these potential headaches (or have a Chromebook and/or not using you own computer), **RStudio Cloud** allows you to run a full instance of RStudio in your browser. There's a generous free tier that allows you to do everything without installing anything! All you have to do is set up a free account **here**.

2.3 Packages

R has thousands of packages that enhance the capabilities of R. You'll need to install several:

- **tidyverse**: A collection of R packages for data science
- **here**: To help with file paths
- **rmarkdown**: To create reproducible analyses
- **palmerpenguins**: An example dataset
- **patchwork**

How to install:

1. **Open RStudio**.
2. **Install packages**. On the command line (>) on the left of the screen, type the following commands:

```
install.packages("rmarkdown", dependenices = TRUE)
install.packages("tidyverse", dependencies = TRUE)
install.packages("here", dependencies = TRUE)
install.packages("palmerpenguins")
```

- **Optional packages**: You may want to eventually knit R Markdown documents to PDF. To do so install the `tinytex` by:

```
install.packages("tinytex")

# Once installed, run:
tinytex::install_tinytex()
```

2.4 Creating a Working Directory and R Project

A working directory is simply the folder on your computer where all your coding project files live. It is also one of the most important things to set up and keep safe. Create a folder on your computer where you normally keep your files. You're not going to want to move this folder once you make it so chose wisely. As for the name, make it short, all lowercase, and do not have spaces. Names like *code*, *r-work*, *code-work* would all be good choices.

Once you have a working directory set up it's time to create a project. In RStudio:

1. Go to *File > New Project*
2. Select *New Directory > New Project*
3. Name the folder (remember all lowercase, no spaces. limnology would probably be a good idea)
4. Select the file path that goes to your working directory folder
5. Click *Create Project*

There should now be a new folder in your working directory with whatever you named your project. In that folder, there will be a file called *your_project_name.Rproj*. If you open that file, R will open a new RStudio session that starts at your project root.

I also recommend creating a folder in your project folder called *data*. This is where all your raw data files should live.

---

3 Data Analysis Crash Course

3.1 First, some basics

3.1.1 R as a calculator

At its most basic, R is a calculator:

```
1 + 9
```

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

```
1 / 200 * 30
```

```
## [1] 0.15
```

```
(59 + 73 + 2) / 3
```

```
## [1] 44.66667
```

```
3 ^ 4
```

```
## [1] 81
```

R also supports all the basic log and trig functions you can find on a scientific calculator...

```
sqrt(2)
```

```
## [1] 1.414214
```

```
sin(pi / 2)
```

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

```
log(4) # natural log
```

```
## [1] 1.386294
```

```
log10(4) # common log
```

```
## [1] 0.60206
```

Note that we used functions here. R has a bunch of built-in functions for you to use. We can also define our own. If you want to see what a function does, simply put a question mark in front of it. Any information about the function will show up in the bottom right under the **Help** tab.

```
?log()
```

3.1.2 Objects in R

Essentially everything we create in R is an object. Objects can be anything from numbers to figures to dataframes. To create an object, use the `<-` operator:

```
x <- 3 * 4
```

Note that if you run this code, nothing shows up. However, if you look under the **Environment** tab (top right) there should be a value for the object `x`. To view the object, we can simply call the object name directly:

```
x
```

```
## [1] 12
```

We can assign more than numbers to objects. In this case, we can make a character string:

```
my_string <- "R is kinda cool"
```

We can do math with objects:

```
x * 2
```

```
## [1] 24
```

Or we can do math with multiple objects

```
one_fish <- 1
two_fish <- 2
one_fish + two_fish
```

```
## [1] 3
```

And finally, we can make new objects from other objects

```
three_fish <- one_fish + two_fish
three_fish
```

```
## [1] 3
```

3.1.3 Combining values with `c()`

Combining values into a vector is an essential function in R. Vectors are the building block of dataframes and R is very good at doing vector math. As you get more advanced in R, vectors will constitute the majority of you analysis.

To create a vector, we can use the `c()` function. We won't go over every data type that a vector can support, but you'll mostly come across vectors with numeric or character data. Let's make a vector of numeric values between 1-10, called `a_numeric_vector`:

```
a_numeric_vector <- c(1:10)
a_numeric_vector
```

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

We can also easily create a character vector using `c()`:

```
a_char_vector <- c("this", "is", "a", "character", "vector")
a_char_vector
```

```
## [1] "this"      "is"        "a"         "character" "vector"
```

3.1.4 Creating and Importing dataframes

Most of the data you will come across data stored in things like Excel spreadsheets and comma-separated values files (ie. csv). R allows you to import a variety of file types with ease. You may also run into times where it's just easier to create a dataframe directly in R. We'll cover how to do both.

We'll go over how to import data in two forms: Excel (.xlsx) and comma-separate values file (.csv). To import a .csv file, we can use the `read_csv()` function in the `readr` package. `readr` already comes with the `tidyverse` library so no need to install anything else! Remember to use the `here` package to help with file paths:

```
library(tidyverse)
library(here)

my_dataframe <- read_csv(here("folder/with/the/data", "my_awesome_data.csv"))
```

If you want to import Excel files, you'll need to install the package `readxl`. Remember that packages are installed with `install.packages()` and to only install a package once (hint: do it directly on the command line)

```
install.packages("readxl", dependencies = TRUE)
```

After `readxll` is installed, a similar strategy to `read_csv` can be employed with `read_excel()`:

```
library(readxl)

my_dataframe <- read_excel(here("folder/with/the/data", "my_awesome_excel_data.xlsx"))
```

Let's move on to creating your own dataframes in R. Say you have some catch data from a pond you surveyed with the species names, how many you collected, and how long your surveyed for. You have two options to make a dataframe directly in R. We'll be using the `tibble` package in the `tidyverse`.

- **Create vectors for each variable and then combine with `tibble()`:**

```
species <- c("Largemouth bass", "Bluegill", "Central stoneroller")
catch_n <- c(50, 15, 60)
effort_min <- c(10, 10, 10)

my_fish_dataframe <- tibble(species, catch_n, effort_min)

my_fish_dataframe
```

```
## # A tibble: 3 x 3
##   species             catch_n effort_min
##   <chr>                 <dbl>      <dbl>
## 1 Largemouth bass          50         10
## 2 Bluegill                 15         10
## 3 Central stoneroller      60         10
```

- **Create a dataframe all at once with `tribble()`**

```
my_fish_dataframe <-
  tribble(
    ~species, ~catch_n, ~effort_min,
    "Largemouth bass", 50, 10,
    "Bluegill", 15, 10,
    "Central stoneroller", 60, 10
  )

my_fish_dataframe
```

```
## # A tibble: 3 x 3
##   species             catch_n effort_min
##   <chr>                 <dbl>      <dbl>
## 1 Largemouth bass          50         10
## 2 Bluegill                 15         10
## 3 Central stoneroller      60         10
```

3.1.5 The Mighty %>%

Using a pipe (`%>%`) in R is an incredibly powerful tool. The `%>%` allows you to write code that reads "left to right" instead of from "inside out". Technically, the `%>%` is found in the `magrittr` package but is ready for use from the `tidyverse`.

To show how pipes work, let's look at a comparison on how code looks without and with a pipe. Let's first create a vector of random values and call it x. Note the use of the `c()` (combine) function here:

```
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)
```

Now let's say that we want to exponentiate each value in `x`, sum those values, and then round them to a whole number. The pipe-less way of doing so would look like this:

```
round(sum(exp(x)))
```

```
## [1] 16
```

Using a pipe allows us to take these functions and start a logical progression from left (start) to right (finish)

```
x %>%
  exp() %>%
  sum() %>%
  round()
```

```
## [1] 16
```

3.2 Penguins!

Before we take a look at the University Lake data, let's get some practice working with actual data in R. We're going to use the `palmerpenguins` package we installed earlier. This package contains two datasets with size measurements for three penguin species in the Palmer Archipelago, Antarctica.

3.2.1 Required packages

You'll need to load several package before we move on to analyzing the penguin data. Remember the `library()` function loads packages into your current session. So let's load the packages we'll need:

```
library(tidyverse)
library(palmerpenguins)
library(patchwork)
```

3.2.2 Taking a look at the data

Normally we would have to load the data separately. However, the penguins data we are after is actually automatically loaded into our environment when we loaded the package `palmerpenguins`.

The next step in a typical data analysis workflow is to take a look at the data you are working with. It's important to remember that data can be large. The term "big data" gets thrown around a lot in the data science world. A simple definition that I like is that if the computer you are using crashes when you try to work with your dataset, then your data are indeed large. Obviously this means that "big data" is a highly subjective term.

Because of this, it's helpful to inspect your data without committing all of it to your computers memory. R gives you a few options to do that. Let's first look at the structure that the penguin data are in. We can do so by calling the `glimpse()` function from the `dplyr` package that came installed with the `tidyverse`:

```
glimpse(penguins)
```

```
## Rows: 344
## Columns: 8
## $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, A...
## $ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torge...
## $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34....
## $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18....
## $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, ...
## $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 347...
## $ sex               <fct> male, female, female, NA, female, male, female, m...
## $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2...
```

We can see that our dataset is organized in row and columns containing size measurements for several penguin species. The handy thing about `glimpse()` is that it also tells you the dimensions of the dataset (344 rows X 8 columns) and lists all of the data types for each column.

We can view *all* of the data using the `View()` function. When you call `View()`, it will open a new tab in your window where you can view the entire dataset.

```
View(penguins)
```

Another useful set of functions to get a quick look at your data are the `head()` and `tail()` functions. These allow you to look at the first and last rows that are in your dataset. Let's take a look at the first rows in the *penguins* dataset:

```
head(penguins)
```

```
## # A tibble: 6 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex
##   <fct>   <fct>           <dbl>         <dbl>            <int>       <int> <fct>
## 1 Adelie  Torge~           39.1          18.7              181        3750 male
## 2 Adelie  Torge~           39.5          17.4              186        3800 fema~
## 3 Adelie  Torge~           40.3          18                195        3250 fema~
## 4 Adelie  Torge~           NA            NA                 NA          NA <NA>
## 5 Adelie  Torge~           36.7          19.3              193        3450 fema~
## 6 Adelie  Torge~           39.3          20.6              190        3650 male
## # ... with 1 more variable: year <int>
```

You can customize the number of rows that are returned using the `n =` argument within `head()` or `tail()`. For example, let's say we wanted to see the last 10 rows:

```
tail(penguins, n = 10)
```

```
## # A tibble: 10 x 8
##    species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##    <fct>   <fct>           <dbl>         <dbl>            <int>       <int>
##  1 Chinst~ Dream            50.2          18.8              202        3800
##  2 Chinst~ Dream            45.6          19.4              194        3525
##  3 Chinst~ Dream            51.9          19.5              206        3950
##  4 Chinst~ Dream            46.8          16.5              189        3650
##  5 Chinst~ Dream            45.7          17                195        3650
##  6 Chinst~ Dream            55.8          19.8              207        4000
##  7 Chinst~ Dream            43.5          18.1              202        3400
##  8 Chinst~ Dream            49.6          18.2              193        3775
##  9 Chinst~ Dream            50.8          19                210        4100
## 10 Chinst~ Dream            50.2          18.7              198        3775
## # ... with 2 more variables: sex <fct>, year <int>
```

3.2.3 Counting some penguins

We saw from both `head()` and `tail()` that we have penguin measurements for different species and islands. It would be helpful for our analysis to get an idea of how individuals are distributed across species and islands. We can quickly do this using the `count()` function to see how may individuals there are per species:

```
penguins %>%
  count(species)
```

```
## # A tibble: 3 x 2
##   species       n
##   <fct>     <int>
## 1 Adelie      152
## 2 Chinstrap    68
## 3 Gentoo      124
```

We can go one step further to see how many individuals there are per species per island:

```
penguins %>%
  count(species, island, .drop = FALSE)
```

```
## # A tibble: 9 x 3
##   species   island        n
##   <fct>     <fct>     <int>
## 1 Adelie    Biscoe       44
## 2 Adelie    Dream        56
## 3 Adelie    Torgersen    52
## 4 Chinstrap Biscoe        0
## 5 Chinstrap Dream        68
## 6 Chinstrap Torgersen     0
## 7 Gentoo    Biscoe      124
## 8 Gentoo    Dream         0
## 9 Gentoo    Torgersen     0
```

You may want to arrange the number of each in either ascending or descending order. We can do this with `arrange()` and `desc()`

```
# Ascending order
penguins %>%
  count(species, island, .drop = FALSE) %>%
  arrange(n)
```

```
## # A tibble: 9 x 3
##   species   island        n
##   <fct>     <fct>     <int>
## 1 Chinstrap Biscoe        0
## 2 Chinstrap Torgersen     0
## 3 Gentoo    Dream         0
## 4 Gentoo    Torgersen     0
## 5 Adelie    Biscoe       44
## 6 Adelie    Torgersen    52
## 7 Adelie    Dream        56
## 8 Chinstrap Dream        68
## 9 Gentoo    Biscoe      124
```

```
# Descending order
penguins %>%
  count(species, island, .drop = FALSE) %>%
  arrange(desc(n))
```

```
## # A tibble: 9 x 3
##   species   island        n
##   <fct>     <fct>     <int>
## 1 Gentoo    Biscoe      124
## 2 Chinstrap Dream        68
## 3 Adelie    Dream        56
## 4 Adelie    Torgersen    52
## 5 Adelie    Biscoe       44
## 6 Chinstrap Biscoe        0
## 7 Chinstrap Torgersen     0
## 8 Gentoo    Dream         0
## 9 Gentoo    Torgersen     0
```

3.2.4 Group and summarize the species

It's common to want to calculate some descriptive statistics from our data. Many times we want to do so by some form of a categorical variable. For the penguins dataset, it would make sense to look at our data by species.

The combination of `group_by()` and `sumamrize()` in the `dplyr` package allows for us to do this with ease. Let's first determine the mean values for *bill_length_mm* and *bill_depth_mm*:

```
penguins %>%
  group_by(species) %>%
  summarize(across(bill_length_mm:bill_depth_mm, mean, na.rm = TRUE))
```

```
## # A tibble: 3 x 3
##   species   bill_length_mm bill_depth_mm
##   <fct>              <dbl>         <dbl>
## 1 Adelie              38.8          18.3
## 2 Chinstrap           48.8          18.4
## 3 Gentoo              47.5          15.0
```

Notice the use of `across()`. This function allows us to apply functions across a set of columns with one function rather than having to apply one-by-one. It saves us from having to type a lot more for the same thing:

```
penguins %>%
  group_by(species) %>%
  summarize(
    bill_length_mm = mean(bill_length_mm, na.rm = TRUE),
    bill_depth_mm = mean(bill_depth_mm, na.rm = TRUE)
  )
```

Another quick note. The `na.rm = TRUE` argument is critical here. You may have saw a few *NA* values in the data. This means that there are missing values where the *NA's* are. R does not know how to do math with missing values (makes sense) and many functions will throw an error if you don't tell them to ignore them. `na.rm = TRUE` tells functions to ignore missing values when performing some sort of calculation.

We can also summarize columns based on their data types. Let's calculate the mean of all numeric values in the penguins data:

```
penguins %>%
  group_by(species) %>%
  summarize(across(where(is.numeric), mean, na.rm = TRUE))
```

```
## # A tibble: 3 x 6
##   species   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g  year
##   <fct>              <dbl>         <dbl>             <dbl>       <dbl> <dbl>
## 1 Adelie              38.8          18.3              190.       3701. 2008.
## 2 Chinstrap           48.8          18.4              196.       3733. 2008.
## 3 Gentoo              47.5          15.0              217.       5076. 2008.
```

The *year* variable happens to be numeric. It doesn't really make sense to take the mean of a series of years. We can use `select()` to drop this column:

```
penguins %>%
  group_by(species) %>%
  summarize(across(where(is.numeric), mean, na.rm = TRUE)) %>%
  select(-year)
```

```
## # A tibble: 3 x 5
##   species   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>              <dbl>         <dbl>             <dbl>       <dbl>
## 1 Adelie              38.8          18.3              190.       3701.
## 2 Chinstrap           48.8          18.4              196.       3733.
## 3 Gentoo              47.5          15.0              217.       5076.
```

Up until now, we've only applied summarize by a single function. Most of the time we want to calculate multiple descriptive statistics. Luckily `across()` accomplishes this with ease.

```
penguins %>%
  group_by(species) %>%
  summarize(across(bill_length_mm, list(mean, median, min, max, sd), na.rm = TRUE))
```

```
## # A tibble: 3 x 6
##   species bill_length_mm_1 bill_length_mm_2 bill_length_mm_3 bill_length_mm_4
##   <fct>              <dbl>            <dbl>            <dbl>            <dbl>
## 1 Adelie              38.8             38.8             32.1             46
## 2 Chinst~             48.8             49.6             40.9             58
## 3 Gentoo              47.5             47.3             40.9             59.6
## # ... with 1 more variable: bill_length_mm_5 <dbl>
```

Well that worked but do you see the problem here? What the heck does *bill_length_mm_1* mean? We can infer that it corresponds to the first function we included in `list()`, mean. However, we aren't in the business of creating vague column variables. We can use some R magic to rename the columns based off of the function we're using:

```
penguins %>%
  group_by(species) %>%
  summarize(across(bill_length_mm,
                   list(mean = mean, median = median, min = min, max = max, sd = sd),
                   .names = "{col}_{fn}", na.rm = TRUE))
```

```
## # A tibble: 3 x 6
##   species bill_length_mm_~ bill_length_mm_~ bill_length_mm_~ bill_length_mm_~
##   <fct>              <dbl>            <dbl>            <dbl>            <dbl>
## 1 Adelie              38.8             38.8             32.1             46
## 2 Chinst~             48.8             49.6             40.9             58
## 3 Gentoo              47.5             47.3             40.9             59.6
## # ... with 1 more variable: bill_length_mm_sd <dbl>
```

3.2.5 Transforming columns

Another typical task when working with data is to manipulate column variables. The `mutate()` function allows us to either manipulate existing variables or create new ones.

Let's say we need to convert the body mass measurements we took from grams to kilograms, but keeping both variables:

```
penguins %>%
  mutate(
    body_mass_kg = body_mass_g / 1000
  )
```

```
## # A tibble: 344 x 9
##    species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##    <fct>   <fct>           <dbl>         <dbl>            <int>       <int>
##  1 Adelie  Torge~           39.1          18.7              181        3750
##  2 Adelie  Torge~           39.5          17.4              186        3800
##  3 Adelie  Torge~           40.3          18                195        3250
##  4 Adelie  Torge~           NA            NA                 NA          NA
##  5 Adelie  Torge~           36.7          19.3              193        3450
##  6 Adelie  Torge~           39.3          20.6              190        3650
##  7 Adelie  Torge~           38.9          17.8              181        3625
##  8 Adelie  Torge~           39.2          19.6              195        4675
##  9 Adelie  Torge~           34.1          18.1              193        3475
## 10 Adelie  Torge~           42            20.2              190        4250
## # ... with 334 more rows, and 3 more variables: sex <fct>, year <int>,
## #   body_mass_kg <dbl>
```

That was easy! We can see that a new column, `body_mass_kg`, was created based on the existing `body_mass_g` column. Now R isn't great about significant figures and you'll need to be very aware of them when you're calculating additional variables. We can use the `round()` function to easily correct for this. Let's say we only want one decimal place in `body_mass_kg`. We can round one of two ways. We can either round when we calculate the variable or after it is calculated:

```
# During
penguins %>%
  mutate(
    body_mass_kg = round(body_mass_g / 1000, 1)
  )

# After
penguins %>%
mutate(
  body_mass_kg = body_mass_g / 1000,
  body_mass_kg = round(body_mass_kg, 1)
)
```

3.2.6 Transforming everything

Sometimes we need to reshape our data depending on the format the dataframe is in. Both `pivot_longer()` and `pivot_wider()` allow for this. Our current penguin dataframe is in a *wide* format. Say that we want to create a new variable *measurement* that includes the measurement variables and then a `value` variable to contain the value:

```
penguins %>%
  pivot_longer(cols = bill_length_mm:body_mass_g,
               names_to = "measurement",
               values_to = "value")
```

```
## # A tibble: 1,376 x 6
##    species island    sex     year measurement        value
##    <fct>   <fct>     <fct>  <int> <chr>              <dbl>
##  1 Adelie  Torgersen male    2007 bill_length_mm      39.1
##  2 Adelie  Torgersen male    2007 bill_depth_mm       18.7
##  3 Adelie  Torgersen male    2007 flipper_length_mm  181
##  4 Adelie  Torgersen male    2007 body_mass_g       3750
##  5 Adelie  Torgersen female  2007 bill_length_mm      39.5
##  6 Adelie  Torgersen female  2007 bill_depth_mm       17.4
##  7 Adelie  Torgersen female  2007 flipper_length_mm  186
##  8 Adelie  Torgersen female  2007 body_mass_g       3800
##  9 Adelie  Torgersen female  2007 bill_length_mm      40.3
## 10 Adelie  Torgersen female  2007 bill_depth_mm       18
## # ... with 1,366 more rows
```

We can see that `pivot_longer()` gathered all of the measurement variables into `measurement` and the values into `value`. This type of transformation wouldn't be a good idea with the penguin data but `pivot_longer()` can be useful in other cases. We'll see how it's handy when working with our profile data.

3.2.7 Selecting the penguins you want

One final thing before we move on to visualizing the penguin data. It is common to want to filter specific values in a dataframe. The `filter()` function in `dplyr` allows us to quickly do this. Let's say we only want penguins that were measured on Dream Island. We can filter the penguins dataframe with one line of code:

```
penguins %>%
  filter(island == "Dream")
```

```
## # A tibble: 124 x 8
##    species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##    <fct>   <fct>           <dbl>         <dbl>            <int>       <int>
##  1 Adelie  Dream            39.5          16.7              178        3250
##  2 Adelie  Dream            37.2          18.1              178        3900
##  3 Adelie  Dream            39.5          17.8              188        3300
##  4 Adelie  Dream            40.9          18.9              184        3900
##  5 Adelie  Dream            36.4          17                195        3325
##  6 Adelie  Dream            39.2          21.1              196        4150
##  7 Adelie  Dream            38.8          20                190        3950
##  8 Adelie  Dream            42.2          18.5              180        3550
##  9 Adelie  Dream            37.6          19.3              181        3300
## 10 Adelie  Dream            39.8          19.1              184        4650
## # ... with 114 more rows, and 2 more variables: sex <fct>, year <int>
```

We can also filter by multiple matching objects. If we wanted to only include specific penguin species we could do so by using the `%in%` operator:

```
penguins %>%
  filter(species %in% c("Chinstrap", "Gentoo"))
```

Sometimes it makes sense to filter *out* something. We know that we only have three species so instead of filtering *for* two species, we can simply filter *out* the one we don't want by adding `!` in front of the column we're filtering:

```
penguins %>%
  filter(!species == "Adelie")
```

3.2.8 Making a basic figure with `ggplot2`

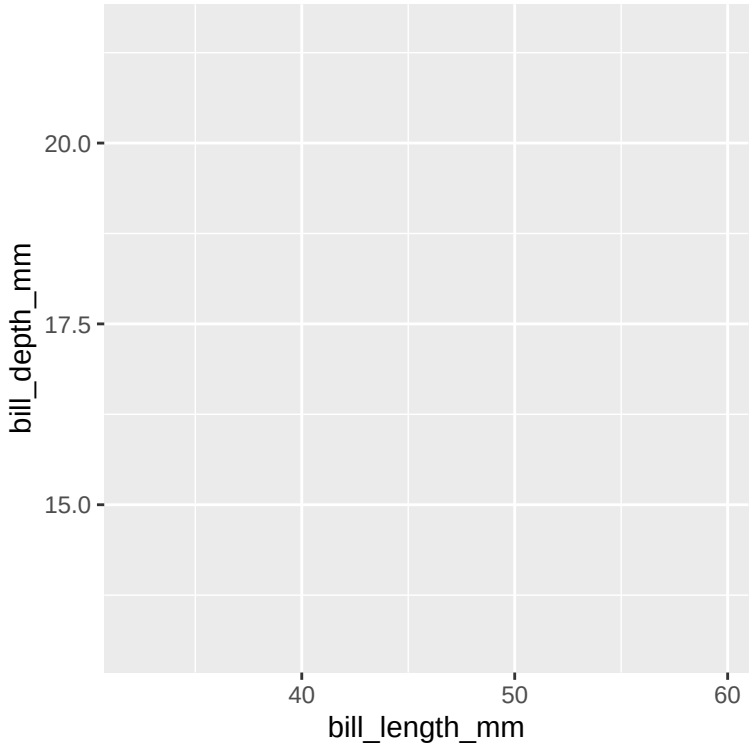
Now that we have a pretty good idea of what's in the penguins data, let's visualize some of it. The `ggplot2` package is by far the most popular plotting package in R. `ggplot2` uses the *Grammar of Graphics* as the underlying theory to make figures. We're not going to get into the theory but know that making figures in `ggplot2` consists of adding *layers* to a plot until you have the final product.

The first step to make any figure is to tell `ggplot2` what data you intend to use. There are multiple ways to do this but I prefer to use the pipe operator and pass the data to the `ggplot()` function like so:

```
penguins %>%
  ggplot()
```

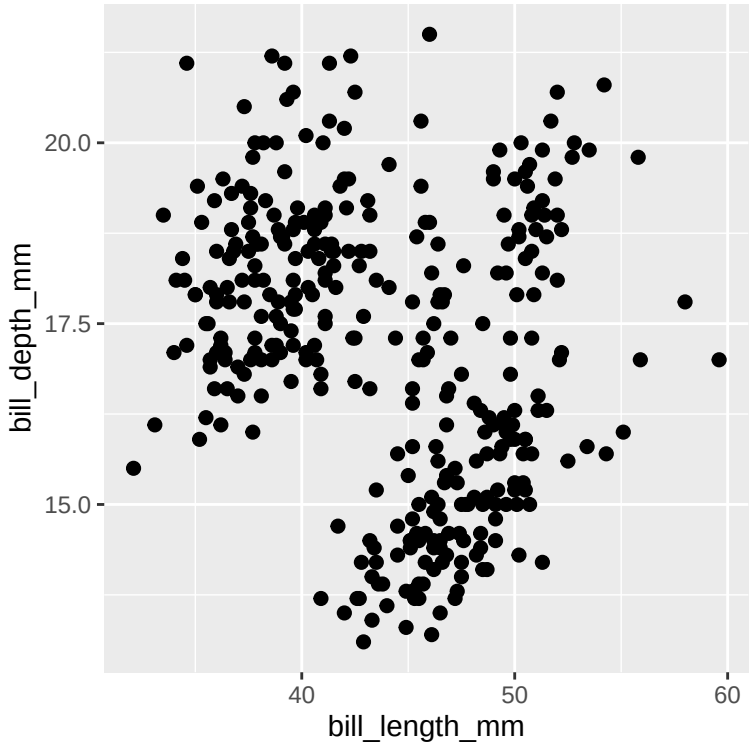
You'll notice that all we see is a blank rectangle. This is because we have not mapped the specific variables we intend to use in the `penguins` dataset. We will do this by mapping the x and y-variables with `aes()`. Let's say we're interested in the relationship between *bill_length_mm* and *bill_depth_mm*:

```
penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm))
```



Ok, so this is still not what we're after. We are seeing the variables mapped to the x and y-axis but we're not seeing any points. Remember that `ggplot2` uses layers to build a figure. We need to add points using `geom_point()`:

```
penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(size = 2)
```

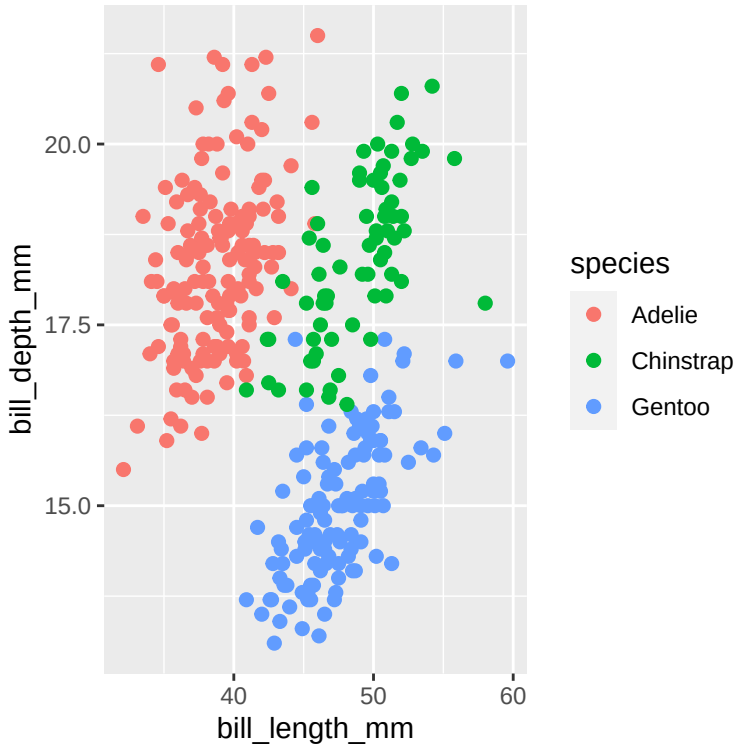


Now we're seeing some data. Congrats, you made your first figure with ggplot! Now let's try to find something meaningful to plot!

3.2.9 Map penguins differently

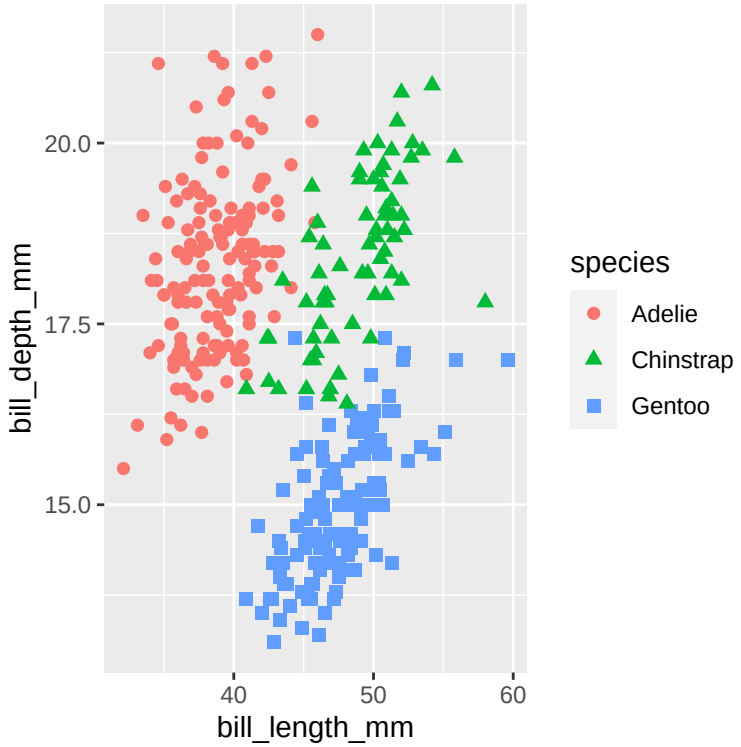
With ggplot, we can map additional variables to aesthetics like shape and color. Let's try to take our last figure and map each species of penguin to a different color:

```
penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(aes(color = species), size = 2)
```



Ok, now we're seeing some clustering between species just by adding some color to our figure! Let's go one step further and also change the shape by species:

```
penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(aes(color = species, shape = species), size = 2)
```

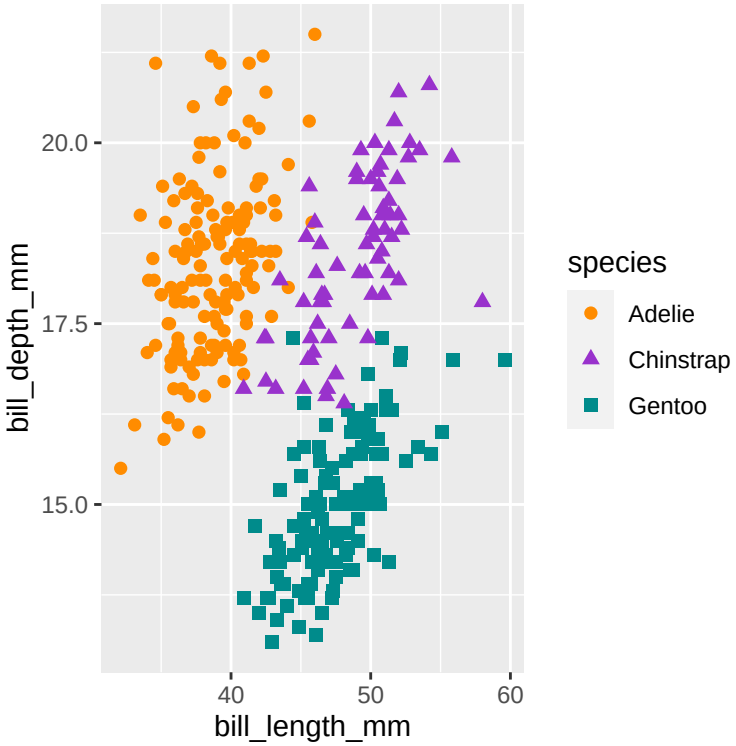


One really important thing to note. See how we mapped the species of penguin to color and shape in `geom_point()`, but not with size? This is because we put the color and shape arguments in the `aes()` function and did not do that for size. Instead, we included size outside of the `aes()` to apply a style to all of the points in `geom_point()`, not to a specific variable.

3.2.10 Add some style to your plot

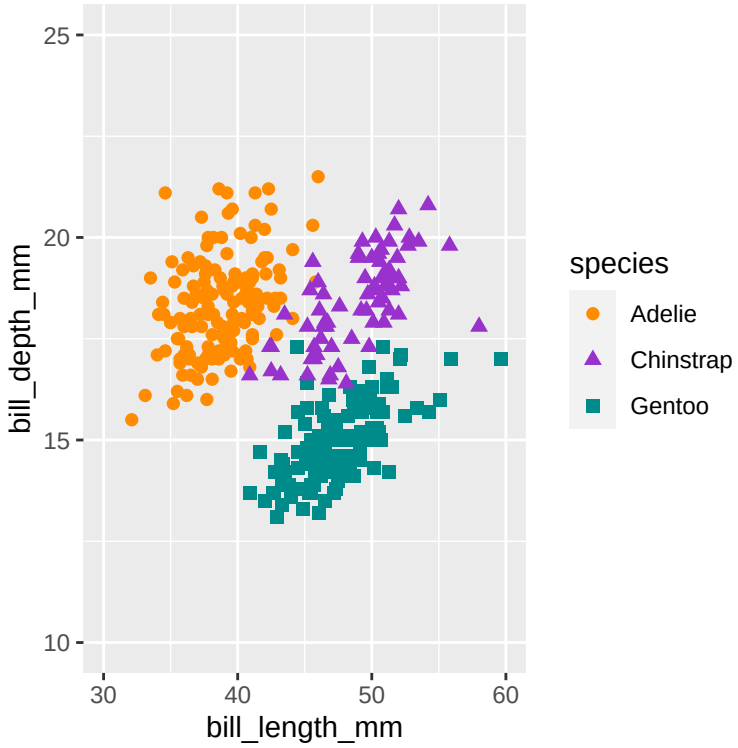
- **Color**
  - We can modify colors of the points in the last plot by manually defining the colors with `scale_color_manual`:

```
penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(aes(color = species, shape = species), size = 2) +
  scale_color_manual(values = c("darkorange","darkorchid","cyan4"))
```



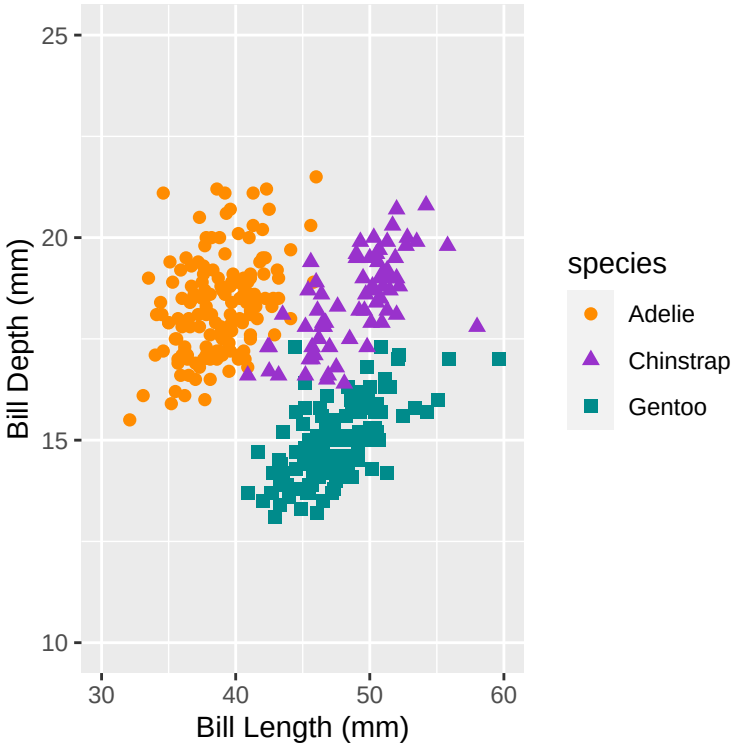
- **Scales**
  - We can also modify the scale of the x and y-axis by adding the following:

```
penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(aes(color = species, shape = species), size = 2) +
  scale_color_manual(values = c("darkorange","darkorchid","cyan4")) +
  coord_cartesian(xlim = c(30, 60), ylim = c(10, 25)) +
  scale_x_continuous(breaks = seq(30, 60, 10)) +
  scale_y_continuous(breaks = seq(10, 25, 5))
```

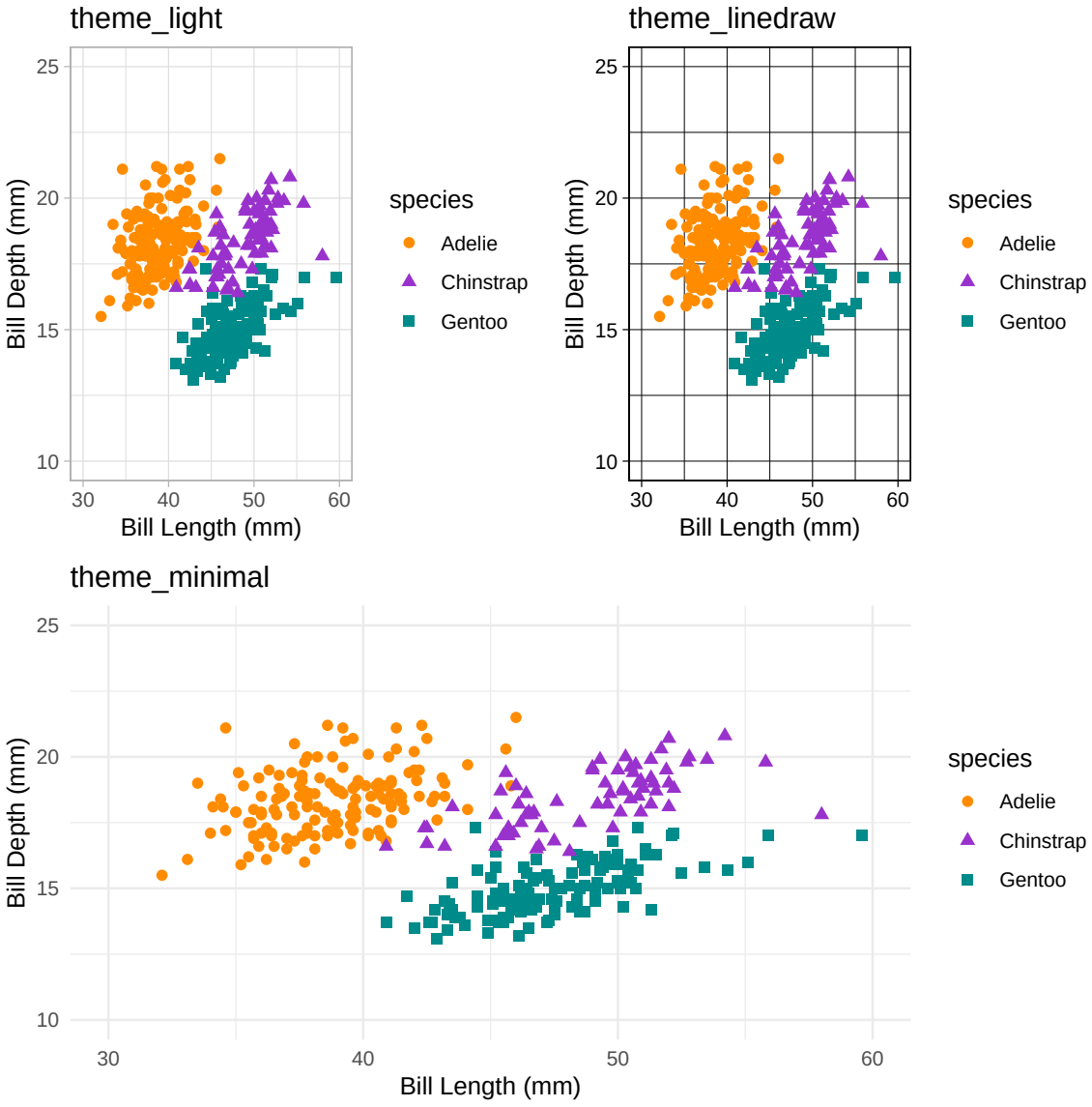


- **Labels**
  - We can modify the the axis labels by:

```
penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(aes(color = species, shape = species), size = 2) +
  scale_color_manual(values = c("darkorange","darkorchid","cyan4")) +
  coord_cartesian(xlim = c(30, 60), ylim = c(10, 25)) +
  scale_x_continuous(name = "Bill Length (mm)", breaks = seq(30, 60, 10)) +
  scale_y_continuous(name = "Bill Depth (mm)", breaks = seq(10, 25, 5))
```



- **Themes**
  - `ggplot2` provides full customization of styling the plot. We'll go into more detail about themes when we work with the University Lake data. Here are a few built-in themes available:

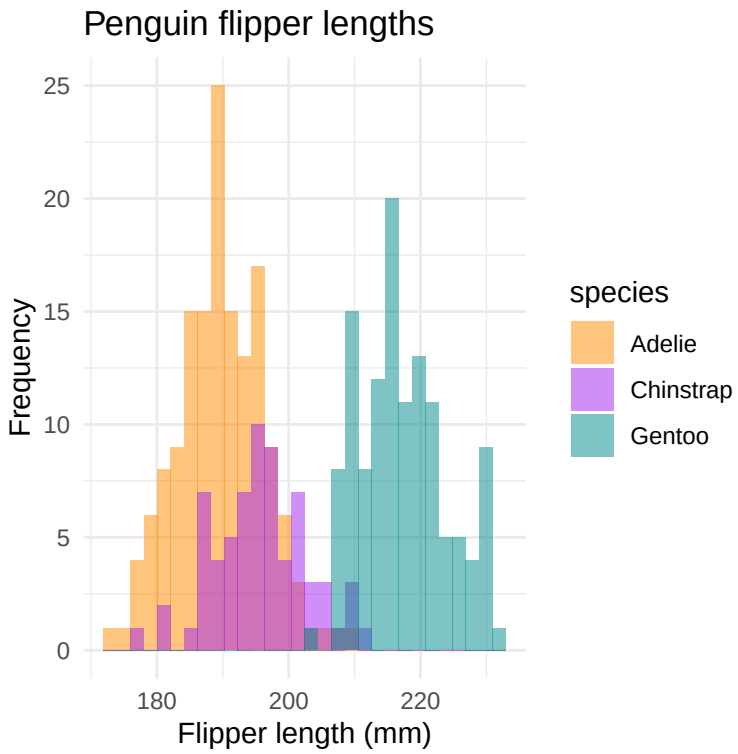


3.2.11 Some additional plots

ggplot2 supports more geoms than we can cover in this guide. But here are a few that you may frequently come across in the future:

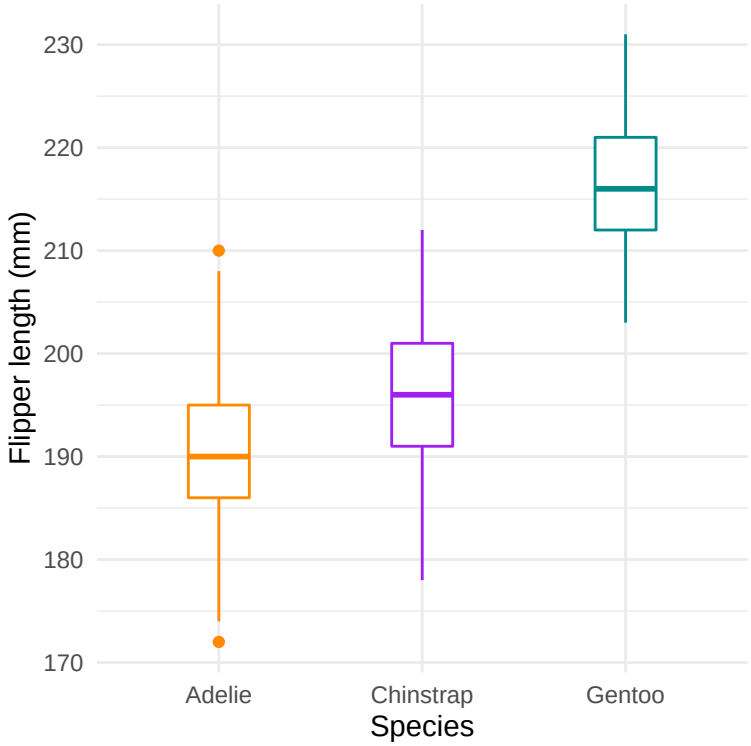
- **Histogram**

```
penguins %>%
  ggplot( aes(x = flipper_length_mm)) +
  geom_histogram(aes(fill = species), alpha = 0.5, position = "identity") +
  scale_fill_manual(values = c("darkorange","purple","cyan4")) +
  labs(x = "Flipper length (mm)", y = "Frequency", title = "Penguin flipper lengths") +
  theme_minimal()
```



- **Boxplot**

```
penguins %>%
  ggplot(aes(x = species, y = flipper_length_mm)) +
  geom_boxplot(aes(color = species), width = 0.3, show.legend = FALSE) +
  scale_color_manual(values = c("darkorange","purple","cyan4")) +
  labs(x = "Species", y = "Flipper length (mm)") +
  theme_minimal()
```



We'll cover everything else you need to know about `ggplot2` when we work with the University Lake data!

---

4 On to University Lake

4.1 Required Libraries

We'll first need to load the packages we installed earlier using `library()`:

```
library(tidyverse)
library(here)
```

```
## here() starts at C:/Users/csauve/Desktop/code/limnology
```

```
library(patchwork)
```

4.2 Importing the data

Our first task is to load the University Lake data into our environment. Since our data are stored as comma-separated values (ie. .csv file), we'll use `here()` to tell R where to find the files and `read_csv()` to import them.

The basic form of what this will look like is:

```
object_name_for_r <- read_csv(here("file/path/to/folder/with/data", "name_of_data_file.csv"))
```

In this case, both data files are stored in the folder **data**. So loading those data into the current environment is done with:

If everything worked, two dataframes will appear in the **Environment** tab in the top right named `water_chem_raw` and `plankton_raw`.

4.3 Working with the water chemistry data

4.3.1 Taking a look

It's a good idea to first get an idea of what format the data are in and what's there. Let's first look at the overall structure with `glimpse()`:

```
glimpse(water_chem_raw)
```

```
## Rows: 12
## Columns: 18
## $ lake_name      <chr> "University", "University", "University", "Universit...
## $ depth          <dbl> 0, 1, 2, 2, 3, 3, 4, 5, 6, 7, 7, 8
## $ sample_type    <chr> NA, NA, NA, "rep", NA, "dup", NA, NA, NA, NA, "rep", NA
## $ temp_c         <dbl> 25.96, 25.32, 24.60, NA, 23.33, 21.48, 16.82, 14.35,...
## $ do_mgl         <dbl> 11.89, 14.78, 11.75, NA, 2.53, 2.02, 0.40, 0.00, 0.0...
## $ do_sat_per     <dbl> 162.2, 183.7, 143.2, NA, 30.0, 23.3, 0.4, 0.0, 0.0, ...
## $ cond_umhos     <dbl> 310.2, 298.6, 340.4, NA, 386.0, 387.7, 390.8, 375.8,...
## $ light_sur_mmol <dbl> 1817, 3609, 1900, NA, 2528, 3671, 3719, 1754, 1519, ...
## $ light_dep_mmol <dbl> 1817.00, 259.10, 23.21, NA, 4.38, 0.00, 0.00, 0.00, ...
## $ ph             <dbl> 8.9, 8.9, 7.4, NA, 8.0, 6.6, 7.0, 7.0, 6.5, 6.8, NA,...
```

```
## $ alk_mgl        <dbl> 95, 91, 110, NA, 102, 132, 164, 152, 171, 170, NA, 206
## $ turb_ntu       <dbl> 11.5, 12.0, 10.2, 11.5, 11.6, 12.5, 31.4, 23.4, 29.6...
## $ srp_mgl        <dbl> 0.004, 0.009, 0.005, 0.003, 0.021, 0.005, 0.008, 0.0...
## $ tp_mgl         <dbl> 0.049, 0.048, 0.050, 0.049, 0.052, 0.054, 0.037, 0.0...
## $ nh3_mgl        <dbl> -0.013, -0.017, 0.011, 0.007, 0.013, 0.017, 0.129, 0...
## $ no3_mgl        <dbl> 0.015, 0.010, 0.011, 0.015, 0.010, 0.011, 0.014, 0.0...
## $ tn_mgl         <dbl> 1.153, 1.132, 0.933, 1.023, 0.955, 1.051, 0.940, 0.9...
## $ chla_ugl       <dbl> 49.837, 61.845, 69.054, NA, 72.105, 75.775, 24.923, ...
```

Remember you can view the *entire* dataset with `View(water_chem_raw)`, the first 6 rows with `head(water_chem_raw)`, or the last 6 rows with `tail(water_chem_raw)`

We can see that in `water_chem_raw`, each row is a unique depth measurement and each column is either a explanatory variable (e.g. lake_name, sample_type) or a parameter we intend to plot.

4.3.2 Control for MDL's

The next step in preparing the water chemistry data for plotting is to control for MDLs, or method detection limits. We'll use the `mutate()` function to override the existing variables to control for the MDLs (Note that you should verify what the current MDLs are and then update the following code accordingly):

```
water_chem_clean <- water_chem_raw %>%
  mutate(
    srp_mgl = ifelse(srp_mgl <= 0.002, 0.002, srp_mgl),
    tp_mgl = ifelse(tp_mgl <= 0.002, 0.002, tp_mgl),
    no3_mgl = ifelse(no3_mgl <= 0.009, 0.009, no3_mgl),
    nh3_mgl = ifelse(nh3_mgl <= 0.015, 0.015, nh3_mgl),
    tn_mgl = ifelse(tn_mgl <= 0.104, 0.104, tn_mgl)
  )
```

4.3.3 Calculating Error

Another task we need to complete is to determine the error indicated by either the duplicate and replicate measurements. We'll first create a custom function to find the error and then apply this function to `water_chemistry_clean`:

```
# Create custom function for error
error_calc <- function(x){
  abs(x - lag(x))
}

# Apply function to data
water_chem_error <- water_chem_clean %>%
  group_by(depth) %>%
  mutate(across(temp_c:chla_ugl, error_calc)) %>%
  ungroup(depth) %>%
  filter(sample_type %in% c("rep", "dup")) %>%
  rename_at(vars(-lake_name, -depth, -sample_type),
            funs(paste0(., sep = "_", "error"))) %>%
  select(-lake_name, -sample_type)
```

4.3.4 Average values

Now that the error between measurements is calculated, we can average the measurements by depth to get the final points for our figures:

```
water_chem_clean <- water_chem_clean %>%
  group_by(depth) %>%
  summarize(across(temp_c:chla_ugl, mean, na.rm = TRUE))
```

4.3.5 Organic Nitrogen

Another variable we need to calculate for our figures is organic nitrogen. We will first define the function `get_orgn()`based off of TN, NO3, and NH3. After that, we can apply to the dataframe to create a new column, `orgn_mgl`

```
get_orgn <- function(tn_mgL, no3_mgL, nh3_mgL){
    org_n <- tn_mgL - (no3_mgL + nh3_mgL)
    return(org_n)
}

# Apply function to df
water_chem_clean <- water_chem_clean %>%
  mutate(orgn_mgl = round(get_orgn(tn_mgl, no3_mgl, nh3_mgl), 3))
```

4.3.6 Percent Light Level

Percent light level is another variable we'll need to put our figures together. Again, we'll define a function and then apply it to the existing dataframe with `mutate()`:

```
# Create function
get_percent_light <- function(light_at_depth, light_at_surface){

  percent_light <- round((light_at_depth / light_at_surface) * 100, 1)
  return(percent_light)
}

# Apply to data
water_chem_clean <- water_chem_clean %>%
  mutate(
    light_level_per = get_percent_light(light_dep_mmol, light_sur_mmol)
  )
```

4.3.7 One Percent Light

The last variable we need to calculate is the one percent light level. However, we'll create a separate object rather than adding to the dataframe since the value applies to the entire profile:

```
# Create function
get_one_percent <- function(depths, light){

  # Determine surface and one percent light
  surface_light <- light[[1]]
  one_percent <- surface_light * 0.01

  # Remove zeros and determine length
  light1 <- light[!light %in% 0]
  len <- length(light1)

  # Make depth vector sample length
  depths1 <- depths[1:len]

  # Calculate one percent light level
  mod <- lm(depths1 ~ log(light1))
  coef <- coef(mod)
  int <- coef[1]
  slope <- coef[2]
  one_percent_light_level <- slope * log(one_percent) + int

  return(one_percent_light_level)
}

# Apply to data
one_percent_light <- round(
                      get_one_percent(
                        water_chem_clean$depth,
                        water_chem_clean$light_dep_mmol
                      ), 1)
```

4.3.8 Sig Figs

One thing we need to control for are significant figures. This is easy to do with `mutate()` and `round()`:

```
water_chem_clean <- water_chem_clean %>%
  mutate(
    temp_c = round(temp_c, 2),
    do_mgl = round(do_mgl, 2),
    do_sat_per = round(do_sat_per, 1),
    cond_umhos = round(cond_umhos, 1),
    light_sur_mmol = round(light_sur_mmol, 0),
    light_dep_mmol = round(light_dep_mmol, 0),
    ph = round(ph, 1),
    alk_mgl = round(alk_mgl, 0),
    turb_ntu = round(turb_ntu, 1),
    srp_mgl = round(srp_mgl, 3),
    tp_mgl = round(tp_mgl, 3),
    nh3_mgl = round(nh3_mgl, 3),
    no3_mgl = round(no3_mgl, 3),
    tn_mgl = round(tn_mgl, 3),
    chla_ugl = round(chla_ugl, 2),
    orgn_mgl = round(orgn_mgl, 3),
    light_level_per = round(light_level_per, 1)
  )
```

4.3.9 Combining with error

Since we saved the error values as a separate dataframe, we want to join them back with the cleaned water chemistry data. We'll use `left_join` to do this:

```
water_chem_clean <- water_chem_clean %>% left_join(water_chem_error, by = c("depth"))
```

4.3.10 Secchi

Similar to the one percent level, Secchi depth is separate from our water chemistry data and needs to be defined:

```
secchi_m <- 0.75
```

4.3.11 Bottom of Epilimnion

Like Secchi and one percent light, the bottom of the epilimnion needs to be defined separately. We don't have a function for this one. Use Figure 6-3 in Wetzel (pg. 76) to estimate.

```
bottom_of_epi <- 1.5
```

4.4 Working with the plankton data

We've already imported the plankton data as `plankton_raw`. Let's get an idea of what those data look like:

```
head(plankton_raw)
```

```
## # A tibble: 6 x 12
##   lake  depth sample_type dolichospermum_~ aphanizomenon_n~ microcystis_nul
##   <chr> <dbl> <chr>                 <dbl>            <dbl>           <dbl>
## 1 Univ~     0 <NA>                  56560            13440            1120
## 2 Univ~     1 <NA>                 147280            19040            1120
## 3 Univ~     2 <NA>                  45749.            7780.            778.
## 4 Univ~     3 <NA>                  68381               0                0
## 5 Univ~     3 dup                  121735            10942            2736
## 6 Univ~     4 <NA>                  20821             5923            2513
## # ... with 6 more variables: ceratium_nul <dbl>, nauplii_nul <dbl>,
## #   bosmina_nul <dbl>, calanoid_nul <dbl>, cyclopoid_nul <dbl>,
## #   chaoborus_nul <dbl>
```

We can see that the data are organized similar to the water chemistry data with the plankton taxa organized by column.

4.4.1 Defining the cube root function

To make the phytoplankton figure more readable, we want to transform the NU/L values using cube-root. R does not include a cube-root function out of the box, but we can easily define one:

```
cube_rt <- function(x){
  x ^ (1/3)
}
```

4.4.2 Tranforming the plankton data

The first step to transform the plankton data is to calculate the minimum, maximum, and mean values for each taxa per depth. Now, we could filter out each taxa and apply these functions to each dataframe. However, that would require a lot of copy-and-paste. Instead, we can apply those functions to all columns with `summarize()` and `across()` and then collect each taxa into the same column:

```
plankton_summary <- plankton_raw %>%
  group_by(depth) %>%
  summarize(across(
    ends_with("nul"),
    list(mean = mean, min = min, max = max),
    .names = "{col}_{fn}")) %>%
  pivot_longer(
    cols = dolichospermum_nul_mean:chaoborus_nul_max,
    names_to = c("taxa", "unit", "stat"),
    names_sep = "_"
  ) %>%
  mutate(taxa = str_to_title(taxa)) %>%
  select(-unit) %>%
  pivot_wider(
    names_from = stat,
    values_from = value
  )
```

4.4.3 Phytoplankton data

To prep the phytoplankton data, we need to apply the cube-root function and then determine the upper and lower bounds for the error bars:

```
phyts <- plankton_summary %>%
  filter(taxa %in% c("Dolichospermum", "Aphanizomenon", "Microcystis", "Ceratium")) %>%
  mutate(
    mean_rt = cube_rt(mean),
    upper_rt = cube_rt(max),
    lower_rt = cube_rt(min),
    upper_bound = abs(mean_rt - upper_rt),
    lower_bound = abs(mean_rt - lower_rt)
  ) %>%
  select(depth, taxa, mean_rt, upper_bound, lower_bound)
```

4.4.4 Zooplankton data

We don't have to apply the cube-root function to the zooplankton taxa. However, we need to determine the upper and lower bounds:

```
zoops <- plankton_summary %>%
  filter(!taxa %in% c("Dolichospermum", "Aphanizomenon", "Microcystis", "Ceratium")) %>%
  rename(lower_bound = min, upper_bound = max)
```

4.5 Making the Water Chemistry Figures

All of the water chemistry figures follow the same format. Once you get the first figure completed, you can easily use it as a template for the others. There are ways to avoid having to copy and paste so much (ie. make a function) but that's a little beyond the scope of this workshop. Feel free to experiment with creating custom plotting functions if you're feeling adventurous.

One quick note about how we are formatting the figures. You may notice in the limnology literature that many profile figures include multiple parameters with multiple scales on the same figure. While this is common, there are a few reasons why we are not doing this:

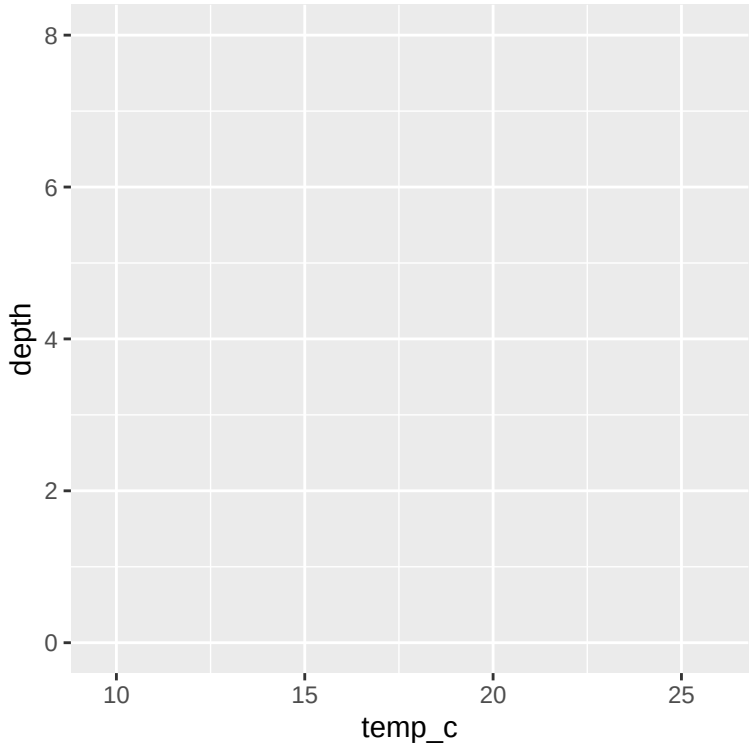
- Combining multiple parameters with *different* scales makes interpretation difficult. This increase in complexity makes you increasingly reliant on different line/point patterns, legends, and colors to explain your figure. Subplots allow you to avoid all of this and make the interpretation clear for the reader.
- ggplot2 does not natively support multiple scales on the same plot for the reason I listed below. It can be done with base graphics if you want to do this in the future
- ggplot2 is considerably easier to use than base graphics. ggplot2 is also widely used in the scientific community and a highly desirable skill to have.

4.5.1 Figure 1 - Mapping the data

Check out the *Required Figures for Lab Reports* document for what Figure 1 is supposed to look like. We essentially need to make a separate figure for each parameter and then combine them together for the final product. We're going to inspect each element of the first subplot separately just to get an idea of how things work. After that, we can skip a lot of steps for the remaining subplots.

We're going to focus on the temperature subplot and the first step is to map the data. Specifically, we're going to define the x and y-variables with `aes()`:

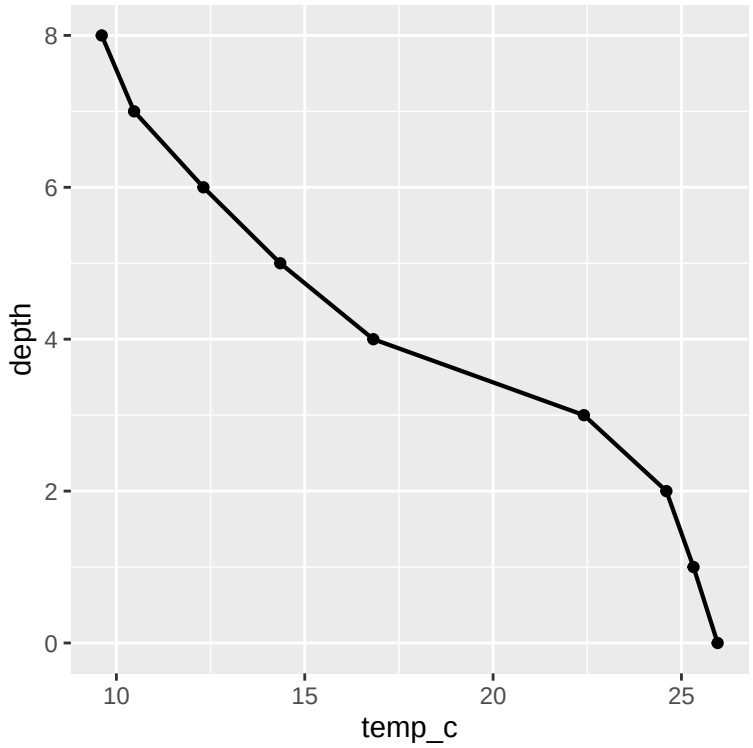
```
water_chem_clean %>%
  ggplot(aes(x = temp_c, y = depth))
```



4.5.2 Figure 1 - Points and Lines

Now that we have the data we want mapped, we can start adding the points and lines with `geom_`:

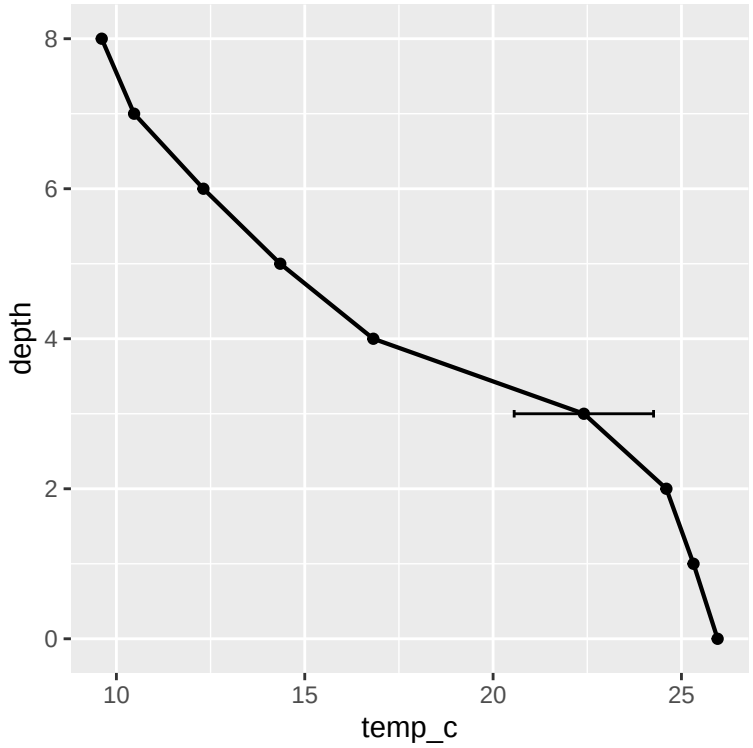
```
water_chem_clean %>%
  ggplot(aes(x = temp_c, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75)
```



4.5.3 Figure 1 - Adding error bars

Our next step is to add the error bars. Remember, that we've already defined the error values so now we just need to call them:

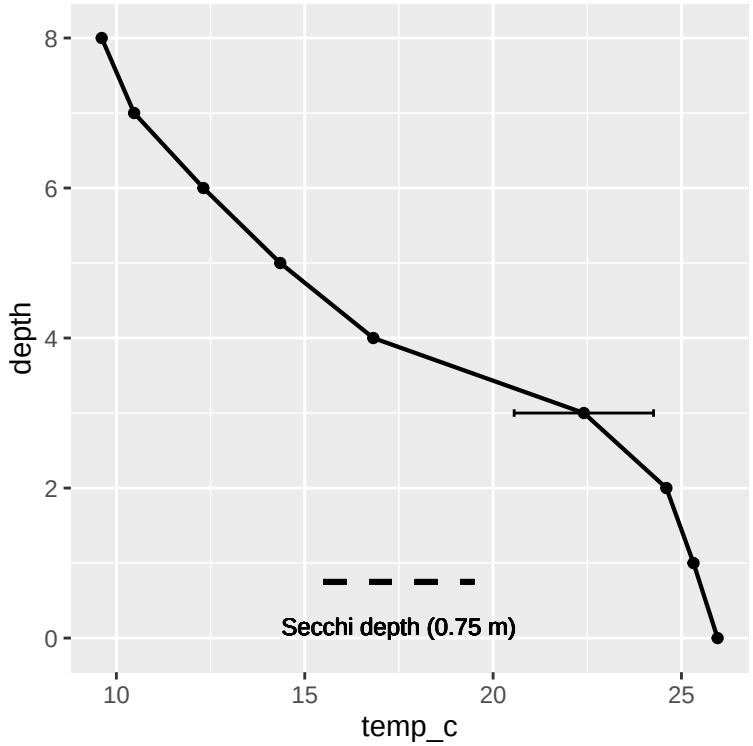
```
water_chem_clean %>%
  ggplot(aes(x = temp_c, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  geom_errorbarh(aes(y = depth,
                     xmin = temp_c - temp_c_error,
                     xmax = temp_c + temp_c_error),
                 height = 0.1)
```



4.5.4 Figure 1 - Adding annotations

We want to add the Secchi depth measurement to our figure. This is a two step process with `geom_segment()` for the line and `geom_text()` to label the line:

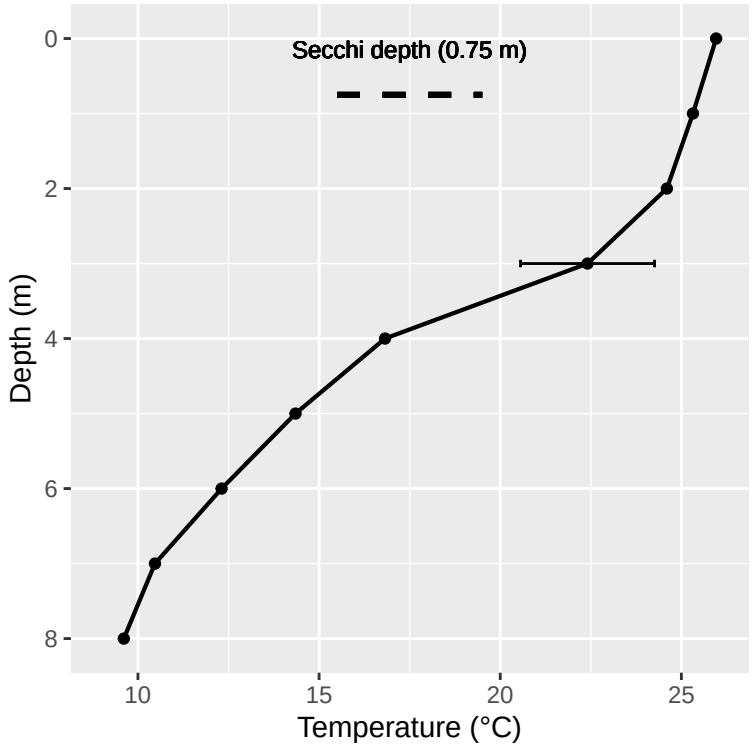
```
water_chem_clean %>%
  ggplot(aes(x = temp_c, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  geom_errorbarh(aes(y = depth,
                     xmin = temp_c - temp_c_error,
                     xmax = temp_c + temp_c_error),
                 height = 0.1) +
  geom_segment(aes(x = 15.5,
                   y = secchi_m,
                   xend = 19.5,
                   yend = secchi_m),
               size = 1, linetype = "dashed") +
  geom_text(aes(x = 17.5, y = 0.15), label = "Secchi depth (0.75 m)", size = 3)
```



4.5.5 Figure 1 - Scale and labels

You've probably noticed that the current plot looks weird. It's because we haven't flipped the y-axis so the surface measurement is at the top of the plot. We also want to control the scale of the x-axis for the labels and then change the axis labels. We can do all of that with only three lines of code:

```
water_chem_clean %>%
  ggplot(aes(x = temp_c, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  geom_errorbarh(aes(y = depth,
                     xmin = temp_c - temp_c_error,
                     xmax = temp_c + temp_c_error),
                 height = 0.1) +
  geom_segment(aes(x = 15.5,
                   y = secchi_m,
                   xend = 19.5,
                   yend = secchi_m),
               size = 1, linetype = "dashed") +
  geom_text(aes(x = 17.5, y = 0.15), label = "Secchi depth (0.75 m)", size = 3) +
  coord_cartesian(xlim = c(9, 26)) +
  scale_y_reverse(name = "Depth (m)") +
  scale_x_continuous(name = "Temperature (°C)")
```



Now that looks better! Last step is to save the plot as an object:

```
p_temp <- water_chem_clean %>%
  ggplot(aes(x = temp_c, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  geom_errorbarh(aes(y = depth,
                     xmin = temp_c - temp_c_error,
                     xmax = temp_c + temp_c_error),
                 height = 0.1) +
  geom_segment(aes(x = 15.5,
                   y = secchi_m,
                   xend = 19.5,
                   yend = secchi_m),
               size = 1, linetype = "dashed") +
  geom_text(aes(x = 17.5, y = 0.15), label = "Secchi depth (0.75 m)", size = 3) +
  coord_cartesian(xlim = c(9, 26)) +
  scale_y_reverse(name = "Depth (m)") +
  scale_x_continuous(name = "Temperature (°C)")
```

4.5.6 Figure 1 - Remaining subplots

So we're done with one out of the four subplots we need. The remaining subplots follow the same pattern as the temperature plot and are completed for you in the `limno-workshop-student` file.

4.5.7 Figure 1 - Combining and adjusting

Now that we have all of the subplots completed all we have to do is combine all the plots and apply a theme to the plot. Be careful to make sure you swap out the `+` and `&` sign when you're putting the figure together:

```
plot1 <- (p_temp | p_do) / (p_turb | p_dosat) +
  plot_annotation(tag_levels = "A", tag_suffix = ".") &
  theme_bw() &
  theme(
    panel.grid = element_blank(),
    panel.border = element_rect(color = "black"),
    axis.text = element_text(color = "black"),
    axis.ticks = element_line(color = "black")
  )
```

4.5.8 Figure 1 - Saving

Finally, we can save our figure with `ggsave()`. For Windows users, make sure you add the `type = "cairo` to adjust for some weird rendering. Also watch the `width =` and `height =` arguments if the plots come out with funky dimensions:

```
ggsave(plot1, file = "figure1.png", device = "png", type = "cairo", width = 7, height = 7)
```

4.5.9 Making Figures 2-5

Figures 2 through 5 are really similar to Figure 1 in terms of code. Make sure you check out the `limno-workshop-student` file. Every plot has a scaffold that outlines every function you will need to create the remaining plots. Refer to the **Required Figures for Lab Reports** on what the layout and labels should look like.

4.6 Making the Plankton Figures

The code required to do the plankton figures is very similar to the water chemistry figures we've already made. We're again going to do through the first subplot in detail and then apply those concepts to the remaining subplots.

While we only went over the first water chemistry plot (Figure 1), we're going to go over how to do both the phytoplankton (Figure 6) and zooplankton (Figure 7) figures.

4.6.1 Figure 6 - Filtering

Remember the `filter()` verb? We can use this to filter out the taxa we want:

```
phyts %>%
  filter(taxa == "Aphanizomenon")
```

4.6.2 Figure 6 - Mapping the data

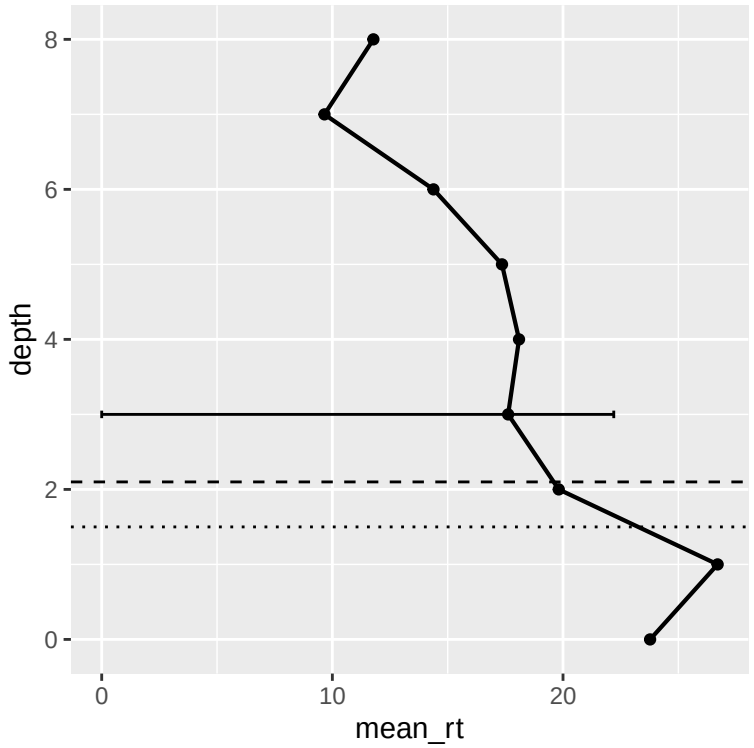
Now we can pass the filtered data to `ggplot()`, map the variables, and then add the appropriate geoms:

```
phyts %>%
  filter(taxa == "Aphanizomenon") %>%
  ggplot(aes(x = mean_rt, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75)
```

4.6.3 Figure 6 - Error bars & Lines

The next step is to add the error bars and draw lines to represent the one percent light level and the bottom of the epilimnion:

```
phyts %>%
  filter(taxa == "Aphanizomenon") %>%
  ggplot(aes(x = mean_rt, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  geom_errorbarh(aes(y = depth,
                     xmin = mean_rt - lower_bound,
                     xmax = mean_rt + upper_bound),
                 height = 0.1) +
  geom_hline(aes(yintercept = one_percent_light), linetype = "dashed") +
  geom_hline(aes(yintercept = bottom_of_epi), linetype = "dotted")
```



4.6.4 Figure 6 - Scale and labels

One thing that is unique to the phytoplankton figure is that we need to directly modify the axis labels. We can do that within `scale_x_continuous()`:

```
p_aphani <- phyts %>%
  filter(taxa == "Aphanizomenon") %>%
  ggplot(aes(x = mean_rt, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  geom_errorbarh(aes(y = depth,
                     xmin = mean_rt - lower_bound,
                     xmax = mean_rt + upper_bound),
                 height = 0.1) +
  geom_hline(aes(yintercept = one_percent_light), linetype = "dashed") +
  geom_hline(aes(yintercept = bottom_of_epi), linetype = "dotted") +
  coord_cartesian(xlim = c(0, 60)) +
  scale_y_reverse(name = "Depth (m)") +
  scale_x_continuous(
    name = "",
    breaks = seq(0, 60, 20),
    labels = c(0, expression(20^3), expression(40^3), expression(60^3))) +
  ggtitle("Aphanizomenon")
```

4.6.5 Figure 6 - Remaining subplots

The remaining subplots look a lot like the last figure we made. The only wrinkle is that on the far right subplot you need to add the labels for the one percent light and bottom of epilimnion lines. The code for the remaining subplots is completed for you in `limno-workshop-student`

4.6.6 Figure 6 - Combining and adjusting

Now that we have the four subplots completed, we can combine the subplots, apply the theme, and save:

```
p_phyt <- (p_aphani| p_ceratium | p_dolicho | p_microcystis) +
  plot_annotation(caption = "Density (#/L)") &
  theme_bw() &
  theme(plot.caption = element_text(hjust = 0.5, size = 12, vjust = 8),
        plot.title = element_text(hjust = 0.5, size = 10, face = "italic"),
        axis.title.y = element_text(size = 12),
        panel.grid = element_blank(),
        panel.border = element_rect(color = "black"),
        axis.text = element_text(color = "black"),
        axis.ticks = element_line(color = "black"))

ggsave(p_phyt, file = "figure6.png", device = "png", type = "cairo", height = 6, width = 10)
```

4.6.7 Figure 7 - Zooplankton

The zooplankton figure is very similar to the phytoplankton figure. Check out the `limno-workshop-student` and the corresponding video to complete the figure.

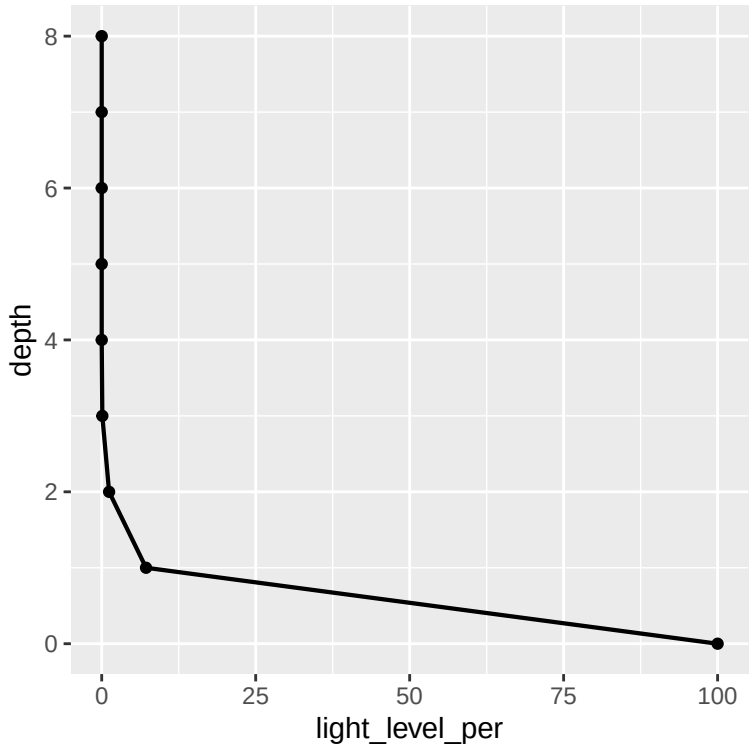
4.7 Making the Light Figure

The light figure is a bit different than what we've done with the previous figures. However, it's going to take considerably less steps to complete (Yay!). No need for subplots here, but there are some wrinkles that you'll need to look out for as we put the light figure together.

4.7.1 Figure 8 - Data, Points, and Lines

Just like the other figures, we'll map the data and add the geoms:

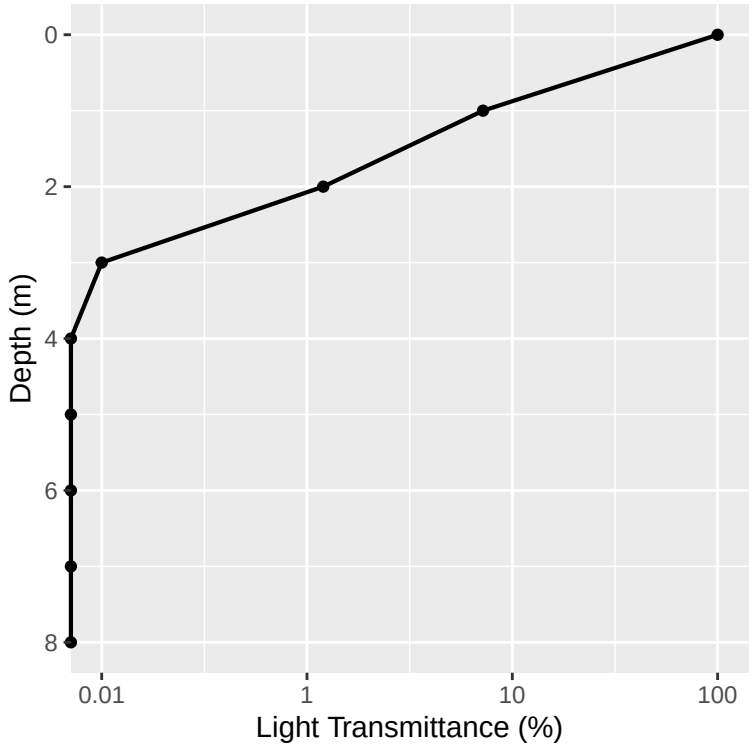
```
water_chem_clean %>%
  ggplot(aes(x = light_level_per, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75)
```



4.7.2 Figure 8 - Scales

To make the figure easier to interpret, we'll log transform the x-axis with `scale_x_log10`:

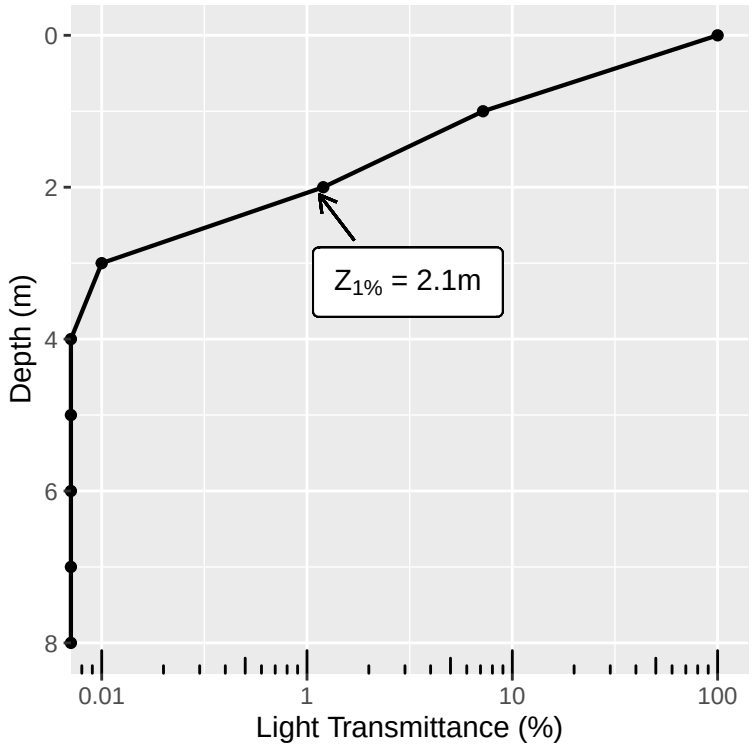
```
water_chem_clean %>%
  ggplot(aes(x = light_level_per, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  coord_cartesian(xlim = c(0.1, 100), clip = 'off') +
  scale_y_reverse(name = "Depth (m)") +
  scale_x_log10(name = "Light Transmittance (%)",
                breaks = c(0.1, 1, 10, 100),
                labels = c("0.01", "1", "10", "100"))
```



4.7.3 Figure 8 - Annotatations and arrows

The next step is to add the annotation and arrow. This takes a little guess-and-check to get the positioning right:

```
water_chem_clean %>%
  ggplot(aes(x = light_level_per, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  coord_cartesian(xlim = c(0.1, 100), clip = 'off') +
  scale_y_reverse(name = "Depth (m)") +
  scale_x_log10(name = "Light Transmittance (%)",
                breaks = c(0.1, 1, 10, 100),
                labels = c("0.01", "1", "10", "100")) +
  annotation_logticks(sides = "b") +
  geom_label(label= expression('Z'['1%']*' = 2.1m'), aes(x = 3.1, y = 3.25),
  label.padding = unit(0.55, "lines"), label.size = 0.35, color = "black", fill="white") +
  geom_segment(
    aes(x = 1.15, xend = 1.7, y = 2.1, yend = 2.7),
    arrow = arrow(ends = "first", type = "open", length = unit(0.25, "cm"))
  )
```



4.7.4 Figure 8 - Themes

Finally, we'll apply the theme and save the final figure

```
p_light <- water_chem_clean %>%
  ggplot(aes(x = light_level_per, y = depth)) +
  geom_point(size = 1.5) +
  geom_path(size = 0.75) +
  coord_cartesian(xlim = c(0.1, 100), clip = 'off') +
  scale_y_reverse(name = "Depth (m)") +
  scale_x_log10(name = "Light Transmittance (%)",
                breaks = c(0.1, 1, 10, 100),
                labels = c("0.01", "1", "10", "100")) +
  annotation_logticks(sides = "b") +
  geom_label(label= expression('Z'['1%']*' = 2.1m'), aes(x = 3.1, y = 2.75),
  label.padding = unit(0.55, "lines"), label.size = 0.35, color = "black", fill="white") +
  geom_segment(
    aes(x = 1.15, xend = 1.7, y = 2.1, yend = 2.7),
    arrow = arrow(ends = "first", type = "open", length = unit(0.25, "cm"))
  ) +
  theme_bw() +
  theme(
    panel.grid = element_blank(),
    panel.border = element_rect(color = "black"),
    axis.text = element_text(color = "black"),
    axis.ticks = element_line(color = "black")
  )

ggsave(p_light, file = "figure8.png", device = "png", type = "cairo")
```

---

5 Troubleshooting Tips

- **Read!**
  - You'd be surprised how much an error message tells you. Things like forgetting to load functions and missing parentheses are easy to determine from the error message.
- **Refresh a lot**
  - The old adage of "Did you turn it off and turn it back on again?" applies to R as well. Sometimes restarting the session will do you wonders. Simply save your work and go to *Session > Restart R*. Another option is to close RStudio entirely, reopen the project, and start fresh with a new environment.
- **When in doubt, Google it!**
  - If you are stuck on a error, the next logical step is to copy the error and paste in your web browser. Most likely someone else has experienced that error and has a solution. The most popular source for these solutions is by far stackoverflow
- **Getting help from RStudio**
  - RStudio does a lot to try to help you with your R problems. There are numerous cheatsheets you can download by going to *Help > Cheatsheets*. RStudio also maintains a super helpful support site here
- **Send me an email**
  - If you have tried the above tips and are still having issues, feel free to email me with your questions. Just make sure you include the error and the code that is giving you issues. My email is <**coryjsauve@gmail.com**>.

---

6 Ok, what's next?

6.1 Good things to read

If you feel motivated to continue to learn R here are some (mostly) free online resources I highly recommend checking out:

- *R for Data Science*
- *Advanced R*
- *R Packages*
- *Introdory Fisheries Analyses with R*
- *Fundamentals of Data Visualization*
- *Data Visualization: A Practical Introduction*
- *Text Mining with R*
- RStudio Education Blog
- R Studio Blog

6.2 Good people to follow

One of the best things about R is that there is an amazing community behind the language. Here are some really great Twitter follows/blogs to check out:

- Hadley Whickham: @hadleywickham (https://hadley.nz)
- Jenny Bryan: @JennyBryan; (https://jennybryan.org/)
- Andrew Heiss: @andrewheiss; (https://www.andrewheiss.com/)
- David Robinson: @drob; (http://varianceexplained.org/)
- Emily Robinson: @robinson_es; (https://hookedondata.org/)
- will Chase @Will_R_Chase; (https://www.williamrchase.com/)
- Julia Silge: @juliasilge; (https://juliasilge.com/)
- Danielle Navarro: @djnavarro; (https://djnavarro.net/)
- Thomas Lin Pedersen: @thomasp85; (https://www.data-imaginist.com/)
- Mine Cetinkaya-Rundel: @minebocek; (https://mine-cr.com)
- Jacqueline Nolis: @skyetetra; (https://jnolis.com)
- Allison Horst: @allison_horst; (https://www.allisonhorst.com/)
- Kieran Healy: @kjhealy; (https://kieranhealy.org/)
- Max Kuhn: @topepos; (http://appliedpredictivemodeling.com/)
- Jordan S Read: @jordansread
- Claus wilke: @ClausWilkel (https://clauswilke.com/)
- Mara Averick: @dataandme
- Winston Change: @winston_chang
- Zev Ross: @zevross
- Studio @rstudio
- R-bloggers: @Rbloggers
- Tom Mock: @thomas_mock
