Data Analysis and Visualization in R for Hakai Ecologists

Data Carpentry (with modifcations)

2018-12-04

Contents

Figure 1:

Chapter 1

Introduction

Data Carpentry's aim is to teach researchers basic concepts, skills, and tools for working with data so that they can get more done in less time, and with less pain. The lessons below were designed for those interested in working with ecology data in R.

This is an introduction to R designed for participants with no programming experience. These lessons can be taught in a day (~ 6 hours). They start with some basic information about R syntax, the RStudio interface, and move through how to import CSV files, the structure of data frames, how to deal with factors, how to add/remove rows and columns, how to calculate summary statistics from a data frame, and a brief introduction to plotting. The last lesson demonstrates how to work with the Hakai database directly from R.

1.1 Chapters

1. Before we start
2. Introduction to R
3. Starting with data
4. Aggregating and analyzing data with dplyr
5. Data visualization with `ggplot2`
6. R and Databases

1.2 Requirements

Data Carpentry's teaching is hands-on, so participants are encouraged to use their own computers to ensure the proper setup of tools for an efficient workflow. *These lessons assume no prior knowledge of the skills or tools*, but working through this lesson requires working copies of the software described below. To most effectively use these materials, please make sure to download the data and install everything *before* working through this lesson.

1.2.1 Setup instructions

R and **RStudio** are separate downloads and installations. R is the underlying statistical computing environment, but using R alone is no fun. RStudio is a graphical integrated development environment (IDE) that makes using R much easier and more interactive. You need to install R before you install RStudio. After

installing both programs, you will need to install the `tidyverse` package from within RStudio. Follow the instructions below for your operating system, and then follow the instructions to install `tidyverse`.

1.2.1.1 Windows

1.2.1.1.1 If you already have R and RStudio installed

- Open RStudio, and click on "Help" > "Check for updates". If a new version is available, quit RStudio, and download the latest version for RStudio.
- To check which version of R you are using, start RStudio and the first thing that appears in the console indicates the version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display which version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it. You can check here for more information on how to remove old versions from your system if you wish to do so.

1.2.1.1.2 If you don't have R and RStudio installed

- Download R from the CRAN website.
- Run the `.exe` file that was just downloaded
- Go to the RStudio download page
- Under *Installers* select **RStudio x.yy.zzz - Windows XP/Vista/7/8** (where x, y, and z represent version numbers)
- Double click the file to install it
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

1.2.1.2 macOS

1.2.1.2.1 If you already have R and RStudio installed

- Open RStudio, and click on "Help" > "Check for updates". If a new version is available, quit RStudio, and download the latest version for RStudio.
- To check the version of R you are using, start RStudio and the first thing that appears on the terminal indicates the version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display which version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it.

1.2.1.2.2 If you don't have R and RStudio installed

- Download R from the CRAN website.
- Select the `.pkg` file for the latest R version
- Double click on the downloaded file to install R
- It is also a good idea to install XQuartz (needed by some packages)
- Go to the RStudio download page
- Under *Installers* select **RStudio x.yy.zzz - Mac OS X 10.6+ (64-bit)** (where x, y, and z represent version numbers)
- Double click the file to install RStudio
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

1.2.1.3 Linux

- Follow the instructions for your distribution from CRAN, they provide information to get the most recent version of R for common distributions. For most distributions, you could use your package manager (e.g., for Debian/Ubuntu run `sudo apt-get install r-base`, and for Fedora `sudo yum`

`install R`), but we don't recommend this approach as the versions provided by this are usually out of date. In any case, make sure you have at least R 3.3.1.
- Go to the RStudio download page
- Under *Installers* select the version that matches your distribution, and install it with your preferred method (e.g., with Debian/Ubuntu `sudo dpkg -i rstudio-x.yy.zzz-amd64.deb` at the terminal).
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

1.2.1.4 For everyone

**After installing R and RStudio, you need to install the `tidyverse` package.

1.3 Contributors

The list of contributors to this lesson is available here,

Page built on: 2018-12-04 11:00:34

Chapter 2

Before We Start

2.0.1 Learning Objectives

- Describe the purpose of the RStudio Script, Console, Environment, and Plots panes.
- Organize files and directories for a set of analyses as an R Project, and understand the purpose of the working directory.
- Use the built-in RStudio help interface to search for more information on R functions.
- Demonstrate how to provide sufficient information for troubleshooting with the R user community.

2.1 What is R? What is RStudio?

The term "`R`" is used to refer to both the programming language and the software that interprets the scripts written using it.

RStudio is currently a very popular way to not only write your R scripts but also to interact with the R software. To function correctly, RStudio needs R and therefore both need to be installed on your computer.

2.2 Why learn R?

2.2.1 R does not involve lots of pointing and clicking, and that's a good thing

The learning curve might be steeper than with other software, but with R, the results of your analysis do not rely on remembering a succession of pointing and clicking, but instead on a series of written commands, and that's a good thing! So, if you want to redo your analysis because you collected more data, you don't have to remember which button you clicked in which order to obtain your results; you just have to run your script again.

Working with scripts makes the steps you used in your analysis clear, and the code you write can be inspected by someone else who can give you feedback and spot mistakes.

Working with scripts forces you to have a deeper understanding of what you are doing, and facilitates your learning and comprehension of the methods you use.

2.2.2 R code is great for reproducibility

Reproducibility is when someone else (including your future self) can obtain the same results from the same dataset when using the same analysis.

R integrates with other tools to generate manuscripts from your code. If you collect more data, or fix a mistake in your dataset, the figures and the statistical tests in your manuscript are updated automatically.

An increasing number of journals and funding agencies expect analyses to be reproducible, so knowing R will give you an edge with these requirements.

2.2.3 R is interdisciplinary and extensible

With 10,000+ packages that can be installed to extend its capabilities, R provides a framework that allows you to combine statistical approaches from many scientific disciplines to best suit the analytical framework you need to analyze your data. For instance, R has packages for image analysis, GIS, time series, population genetics, and a lot more.

2.2.4 R works on data of all shapes and sizes

The skills you learn with R scale easily with the size of your dataset. Whether your dataset has hundreds or millions of lines, it won't make much difference to you.

R is designed for data analysis. It comes with special data structures and data types that make handling of missing data and statistical factors convenient.

R can connect to spreadsheets, databases, and many other data formats, on your computer or on the web.

2.2.5 R produces high-quality graphics

The plotting functionalities in R are endless, and allow you to adjust any aspect of your graph to convey most effectively the message from your data.

2.2.6 R has a large and welcoming community

Thousands of people use R daily. Many of them are willing to help you through mailing lists and websites such as Stack Overflow, or on the RStudio community.

2.2.7 Not only is R free, but it is also open-source and cross-platform

Anyone can inspect the source code to see how R works. Because of this transparency, there is less chance for mistakes, and if you (or someone else) find some, you can report and fix bugs.

2.3 Knowing your way around RStudio

Let's start by learning about RStudio, which is an Integrated Development Environment (IDE) for working with R.

The RStudio IDE open-source product is free under the Affero General Public License (AGPL) v3. The RStudio IDE is also available with a commercial license and priority email support from RStudio, Inc.

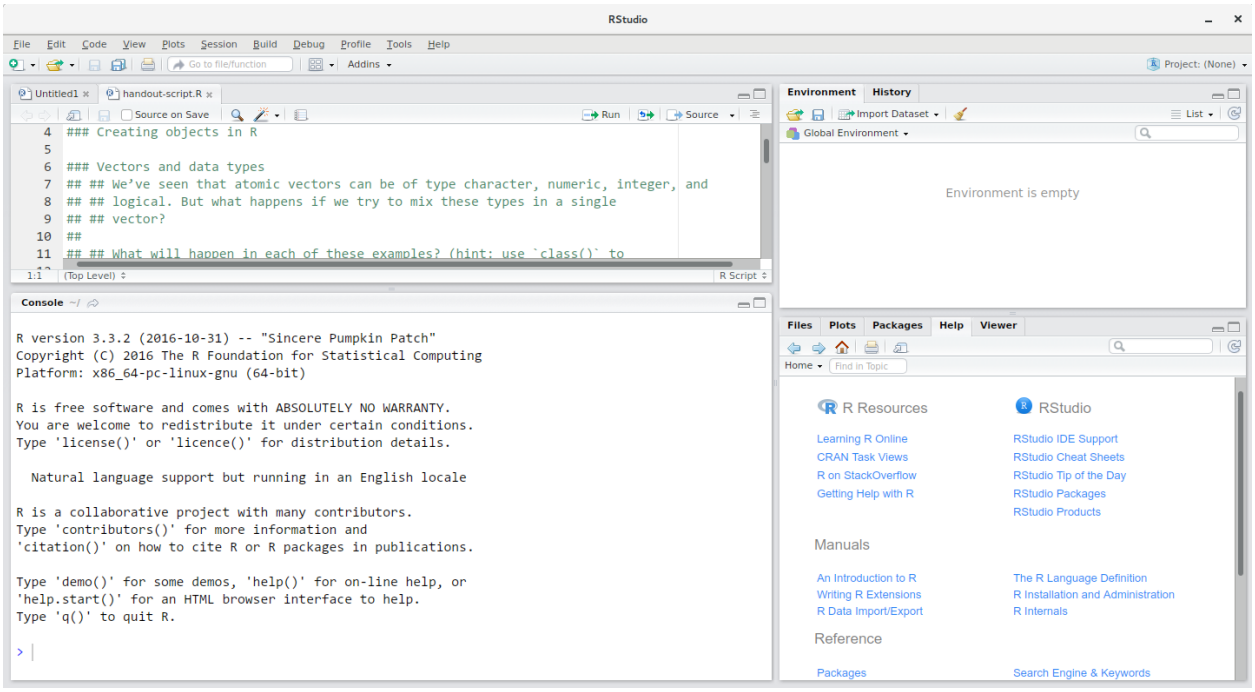


Figure 2.1: RStudio interface screenshot. Clockwise from top left: Source, Environment/History, Files/Plots/Packages/Help/Viewer, Console.

We will use RStudio IDE to write code, navigate the files on our computer, inspect the variables we are going to create, and visualize the plots we will generate. RStudio can also be used for other things (e.g., version control, developing packages, writing Shiny apps) that we will not cover during the workshop.

RStudio is divided into 4 "Panes": the **Source** for your scripts and documents (top-left, in the default layout), your **Environment/History** (top-right), your **Files/Plots/Packages/Help/Viewer** (bottom-right), and the R **Console** (bottom-left). The placement of these panes and their content can be customized (see menu, Tools -> Global Options -> Pane Layout).

One of the advantages of using RStudio is that all the information you need to write code is available in a single window. Additionally, with many shortcuts, autocompletion, and highlighting for the major file types you use while developing in R, RStudio will make typing easier and less error-prone.

2.4 Getting set up

It is good practice to keep a set of related data, analyses, and text self-contained in a single folder, called the **working directory**. All of the scripts within this folder can then use *relative paths* to files that indicate where inside the project a file is located (as opposed to absolute paths, which point to where a file is on a specific computer). Working this way makes it a lot easier to move your project around on your computer and share it with others without worrying about whether or not the underlying scripts will still work.

RStudio provides a helpful set of tools to do this through its "Projects" interface, which not only creates a working directory for you, but also remembers its location (allowing you to quickly navigate to it) and optionally preserves custom settings and open files to make it easier to resume work after a break. Go through the steps for creating an "R Project" for this tutorial below.

1. Start RStudio.
2. Under the `File` menu, click on `New project`. Choose `New directory`, then `New project`.

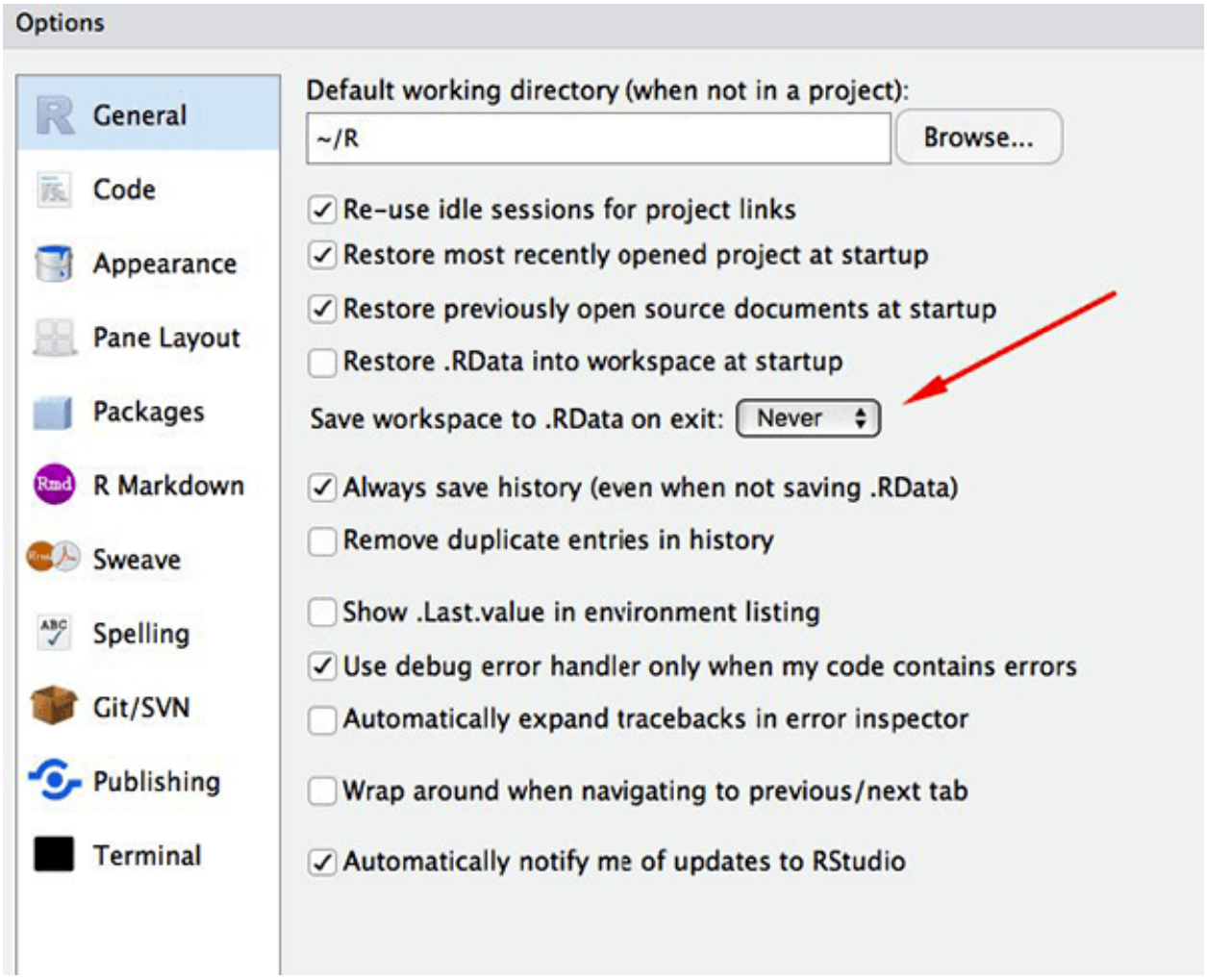


Figure 2.2: Set 'Save workspace to .RData on exit' to 'Never'

3. Enter a name for this new folder (or "directory"), and choose a convenient location for it. This will be your **working directory** for the rest of the day (e.g., `~/intro_to_r`).
4. Click on `Create project`.
5. Download this file and put it in a new folder in your `intro_to_r` folder and call the new folder `data_output`
6. Set Preferences to 'Never' save workspace in RStudio.

RStudio's default preferences generally work well, but saving a workspace to .RData can be cumbersome, especially if you are working with larger datasets. To turn that off, go to Tools –> 'Global Options' and select the 'Never' option for 'Save workspace to .RData' on exit.'

2.4.1 Organizing your working directory

Using a consistent folder structure across your projects will help keep things organized, and will also make it easy to find/file things in the future. This can be especially helpful when you have multiple projects. In general, you may create directories (folders) for **scripts**, **data**, and **documents**.

- `data/` Use this folder to store your raw data and intermediate datasets you may create for the need of a particular analysis. For the sake of transparency and provenance, you should *always* keep a

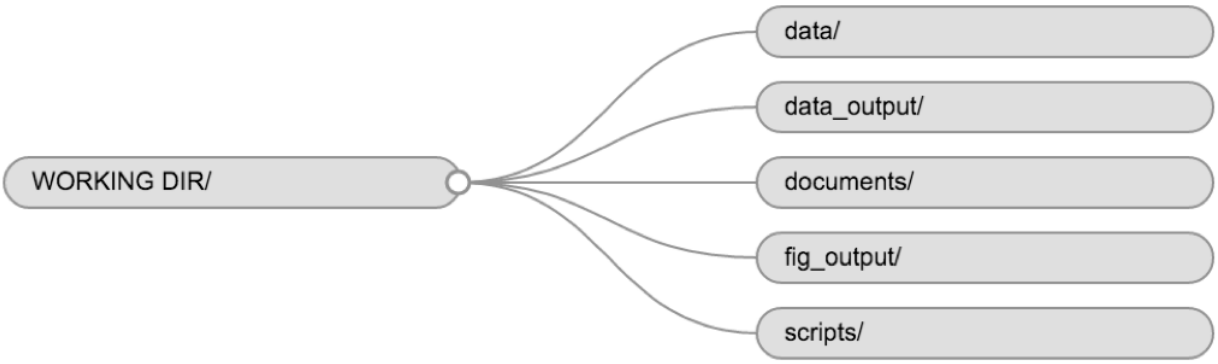


Figure 2.3: Example of a working directory structure.

copy of your raw data accessible and do as much of your data cleanup and preprocessing programmatically (i.e., with scripts, rather than manually) as possible. Separating raw data from processed data is also a good idea. For example, you could have files `data/raw/tree_survey.plot1.txt` and `...plot2.txt` kept separate from a `data/processed/tree.survey.csv` file generated by the `scripts/01.preprocess.tree_survey.R` script.

- **`documents/`** This would be a place to keep outlines, drafts, and other text.
- **`scripts/`** This would be the location to keep your R scripts for different analyses or plotting, and potentially a separate folder for your functions (more on that later).

You may want additional directories or subdirectories depending on your project needs, but these should form the backbone of your working directory.

For this workshop, we will need a `data/` folder to store our raw data, and we will use `data_output/` for when we learn how to export data as CSV files, and `fig_output/` folder for the figures that we will save.

- Under the `Files` tab on the right of the screen, click on `New Folder` and create a folder named `data` within your newly created working directory (e.g., `~/data-carpentry/data`). (Alternatively, type `dir.create("data")` at your R console.) Repeat these operations to create a `data_output/` and a `fig_output` folders.

We are going to keep the script in the root of our working directory because we are only going to use one file and it will make things easier.

Your working directory should now look like this:

2.4.2 The working directory

The working directory is an important concept to understand. It is the place from where R will be looking for and saving the files. When you write code for your project, it should refer to files in relation to the root of your working directory and only need files within this structure.

Using RStudio projects makes this easy and ensures that your working directory is set properly. If you need to check it, you can use `getwd()`. If for some reason your working directory is not what it should be, you can change it in the RStudio interface by navigating in the file browser where your working directory should be, and clicking on the blue gear icon "More", and select "Set As Working Directory". Alternatively you can use `setwd("/path/to/working/directory")` to reset your working directory. However, your scripts should not include this line because it will fail on someone else's computer.

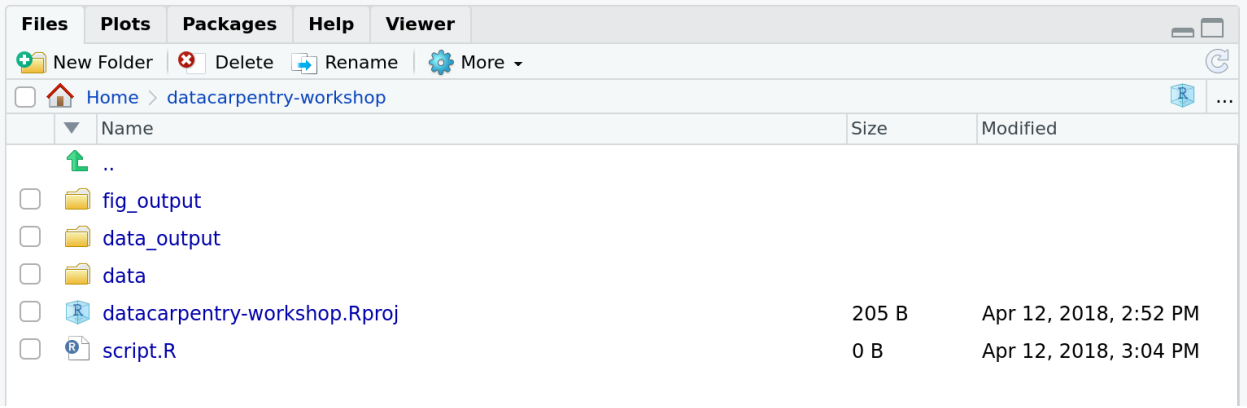


Figure 2.4: How it should look like at the beginning of this lesson

2.5 Interacting with R

The basis of programming is that we write down instructions for the computer to follow, and then we tell the computer to follow those instructions. We write, or *code*, instructions in R because it is a common language that both the computer and we can understand. We call the instructions *commands* and we tell the computer to follow the instructions by *executing* (also called *running*) those commands.

There are two main ways of interacting with R: by using the console or by using script files (plain text files that contain your code). The console pane (in RStudio, the bottom left panel) is the place where commands written in the R language can be typed and executed immediately by the computer. It is also where the results will be shown for commands that have been executed. You can type commands directly into the console and press `Enter` to execute those commands, but they will be forgotten when you close the session.

Because we want our code and workflow to be reproducible, it is better to type the commands we want in the script editor, and save the script. This way, there is a complete record of what we did, and anyone (including our future selves!) can easily replicate the results on their computer.

RStudio allows you to execute commands directly from the script editor by using the `Ctrl` + `Enter` shortcut (on Macs, `Cmd` + `Return` will work, too). The command on the current line in the script (indicated by the cursor) or all of the commands in the currently selected text will be sent to the console and executed when you press `Ctrl` + `Enter`. You can find other keyboard shortcuts in this RStudio cheatsheet about the RStudio IDE.

At some point in your analysis you may want to check the content of a variable or the structure of an object, without necessarily keeping a record of it in your script. You can type these commands and execute them directly in the console. RStudio provides the `Ctrl` + `1` and `Ctrl` + `2` shortcuts allow you to jump between the script and the console panes.

If R is ready to accept commands, the R console shows a `>` prompt. If it receives a command (by typing, copy-pasting or sent from the script editor using `Ctrl` + `Enter`), R will try to execute it, and when ready, will show the results and come back with a new `>` prompt to wait for new commands.

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a `+` prompt. It means that you haven't finished entering a complete command. This is because you have not 'closed' a parenthesis or quotation, i.e. you don't have the same number of left-parentheses as right-parentheses, or the same number of opening and closing quotation marks. When this happens, and you thought you finished typing your command, click inside the console window and press `Esc`; this will cancel the incomplete command and return you to the `>` prompt.

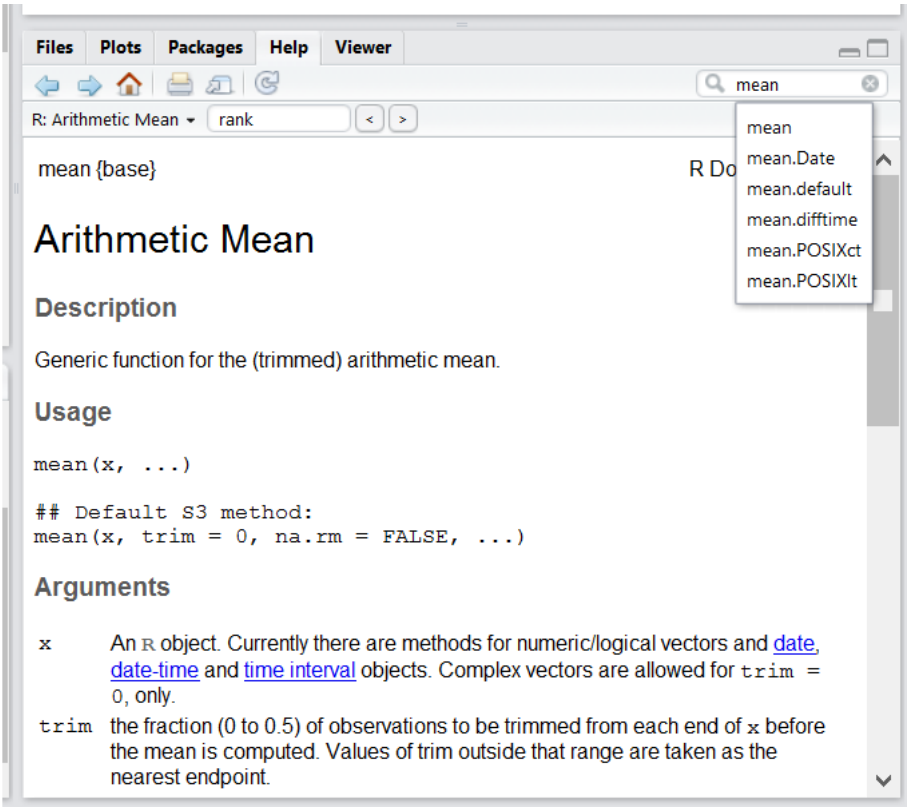


Figure 2.5: RStudio help interface.

2.6 How to learn more after the workshop?

The material we cover during this workshop will give you an initial taste of how you can use R to analyze data for your own research. However, you will need to learn more to do advanced operations such as cleaning your dataset, using statistical methods, or creating beautiful graphics. The best way to become proficient and efficient at R, as with any other tool, is to use it to address your actual research questions. As a beginner, it can feel daunting to have to write a script from scratch, and given that many people make their code available online, modifying existing code to suit your purpose might make it easier for you to get started.

2.7 Seeking help

2.7.1 Use the built-in RStudio help interface to search for more information on R functions

One of the fastest ways to get help, is to use the RStudio help interface. This panel by default can be found at the lower right hand panel of RStudio. As seen in the screenshot, by typing the word “Mean”, RStudio tries to also give a number of suggestions that you might be interested in. The description is then shown in the display window.

2.7.2 I know the name of the function I want to use, but I'm not sure how to use it

If you need help with a specific function, let's say `barplot()`, you can type:

```
?barplot
```

If you just need to remind yourself of the names of the arguments, you can use:

```
args(lm)
```

2.7.3 I want to use a function that does X, there must be a function for it but I don't know which one...

If you are looking for a function to do a particular task, you can use the `help.search()` function, which is called by the double question mark `??`. However, this only looks through the installed packages for help pages with a match to your search request

```
??kruskal
```

If you can't find what you are looking for, you can use the rdocumentation.org website that searches through the help files across all packages available.

Finally, a generic Google or internet search "R <task>" will often either send you to the appropriate package documentation or a helpful forum where someone else has already asked your question.

2.7.4 I am stuck... I get an error message that I don't understand

Start by googling the error message. However, this doesn't always work very well because often, package developers rely on the error catching provided by R. You end up with general error messages that might not be very helpful to diagnose a problem (e.g. "subscript out of bounds"). If the message is very generic, you might also include the name of the function or package you're using in your query.

However, you should check Stack Overflow. Search using the `[r]` tag. Most questions have already been answered, but the challenge is to use the right words in the search to find the answers: http://stackoverflow.com/questions/tagged/r

2.7.5 More resources

- The R-Studio Community Forums
- R for Data Science
- Stack Overflow
- Hakai R Guide for Reproducible Analyses

Chapter 3

Intro to R

3.0.1 Learning Objectives

- Define the following terms as they relate to R: object, assign, call, function, arguments, options.
- Assign values to objects in R.
- Learn how to *name* objects
- Use comments to inform script.
- Solve simple arithmetic operations in R.
- Call functions and use arguments to change their default options.
- Inspect the content of vectors and manipulate their content.
- Subset and extract values from vectors.
- Analyze vectors with missing data.

3.1 Creating objects in R

You can get output from R simply by typing math in the console:

```
3 + 5
12 / 7
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create an object, we need to give it a name followed by the assignment operator `<-`, and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 **goes into** `x`. For historical reasons, you can also use `=` for assignments, but not in every context. Because of the slight differences in syntax, it is good practice to always use `<-` for assignments.

In RStudio, typing Alt + - (push Alt at the same time as the - key) will write `<-` in a single keystroke in a PC, while typing Option + - (push Option at the same time as the - key) does the same in a Mac.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid, but `x2` is). R is case

sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they are the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). If in doubt, check the help to see if the name is already in use. It's also best to avoid dots (`.`) within an object name as in `my.dataset`. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them. It is also recommended to use nouns for object names, and verbs for function names. It's important to be consistent in the styling of your code (where you put spaces, how you name objects, etc.). Using a consistent coding style makes your code clearer to read for your future self and your collaborators. In R, the best style guide is the tidyverse's. The tidyverse's is very comprehensive and may seem overwhelming at first. You can install the **`lintr`** package to automatically check for issues in the styling of your code.

3.1.1 Objects vs. variables

> What are known as `objects` in `R` are known as `variables` in many other programming languages. Depending on the context, `object` and `variable` can have drastically different meanings. However, in this lesson, the two words are used synonymously. For more information see: https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name:

```
weight_kg <- 55    # doesn't print anything
(weight_kg <- 55)  # but putting parenthesis around the call prints the value of `weight_kg`
weight_kg          # and so does typing the name of the object
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

We can also change an object's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

This means that assigning a value to one object does not change the values of other objects For example, let's store the animal's weight in pounds in a new object, `weight_lb`:

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

3.1.2 Comments

The comment character in R is `#`, anything to the right of a `#` in a script will be ignored by R. It is useful to leave notes, and explanations in your scripts. RStudio makes it easy to comment or uncomment a paragraph: after selecting the lines you want to comment, press at the same time on your keyboard Ctrl + Shift + C. If you only want to comment out one line, you can put the cursor at any location of that line (i.e. no need to select the whole line), then press Ctrl + Shift + C.

3.1.3 Challenge

What are the values after each statement in the following?

```
mass <- 47.5            # mass?
age  <- 122             # age?
mass <- mass * 2.0      # mass?
age  <- age - 20        # age?
mass_index <- mass/age  # mass_index?
```

3.1.4 Functions and their arguments

Functions are "canned scripts" that automate more complicated sets of commands including operations assignments, etc. Many functions are predefined, or can be made available by importing R *packages* (more on that later). A function usually gets one or more inputs called *arguments*. Functions often (but not always) return a *value*. A typical example would be the function `sqrt()`. The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called *calling* the function. An example of a function call is:

```
b <- sqrt(a)
```

Here, the value of `a` is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to the object `b`. This function is very simple, because it takes just one argument.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a dataset. We'll see that when we read data files into R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below). Some functions take arguments which may either be specified by the user, or, if left out, take on a *default* value: these are called *options*. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let's try a function that can take multiple arguments: `round()`.

```
round(3.14159)
```

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

Here, we've called `round()` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the `round` function. We can use `args(round)` or look at the help for this function using `?round`.

```
args(round)
```

```
#> function (x, digits = 0)
#> NULL
```

```
?round
```

We see that if we want a different number of digits, we can type `digits=2` or however many we want.

```
round(3.14159, digits = 2)
```

```
#> [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
#> [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits = 2, x = 3.14159)
```

```
#> [1] 3.14
```

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to specify the names of all optional arguments. If you don't, someone reading your code might have to look up the definition of a function with unfamiliar arguments to understand what you're doing.

3.2 Vectors and data types

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of animal weights and assign it to a new object `weight_g`:

```
weight_g <- c(50, 60, 65, 82)
weight_g
```

A vector can also contain characters:

```
animals <- c("mouse", "rat", "dog")
animals
```

Mouse, rat and dog are called elements of the animals vector. The quotes around "mouse", "rat", etc. are essential here. Without the quotes R will assume there are objects called `mouse`, `rat` and `dog`. As these objects don't exist in R's memory, there will be an error message.

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(weight_g)
length(animals)
```

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```
class(weight_g)
class(animals)
```

An **atomic vector** is the simplest R **data type** and is a linear vector of a single type. Above, we saw 2 of the 6 main **atomic vector** types that R uses: `"character"` and `"numeric"` (or `"double"`). These are the basic building blocks that all R objects are built from. The other 4 **atomic vector** types are:

- `"logical"` for `TRUE` and `FALSE` (the boolean data type)
- `"integer"` for integer numbers (e.g., `2L`, the `L` indicates to R that it's an integer)
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1 + 4i`) and that's all we're going to say about them
- `"raw"` for bitstreams that we won't discuss further

You can check the type of your vector using the `typeof()` function and inputting your vector as the argument.

Vectors are one of the many **data structures** that R uses. Other important ones are lists (`list`), matrices (`matrix`), data frames (`data.frame`), factors (`factor`) and arrays (`array`).

3.2.1 Challenge

- We've seen that atomic vectors can be of type character, numeric (or double), integer, and logical. But what happens if we try to mix these types in a single vector?
- What will happen in each of these examples? (hint: use `class()` to check the data type of your objects):

```
num_char <- c(1, 2, 3, "a")
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
tricky <- c(1, 2, 3, "4")
```

- Why do you think it happens?
- How many values in `combined_logical` are `"TRUE"` (as a character) in the following example:

```
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
combined_logical <- c(num_logical, char_logical)
```

- You've probably noticed that objects of different types get converted into a single, shared type within a vector. In R, we call converting objects from one class into another class *coercion*. These conversions happen according to a hierarchy, whereby some types get preferentially coerced into other types. Can you draw a diagram that represents the hierarchy of how these data types are coerced?

3.3 Subsetting vectors

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[2]
```

```
#> [1] "rat"
```

```
animals[c(3, 2)]
```

```
#> [1] "dog" "rat"
```

We can also repeat the indices to create an object with more elements than the original one:

```
more_animals <- animals[c(1, 2, 3, 2, 1, 4)]
more_animals
```

```
#> [1] "mouse" "rat"   "dog"   "rat"   "mouse" "cat"
```

R indices start at 1. Programming languages like Fortran, MATLAB, Julia, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

3.4 Missing data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages). Missing data are represented in vectors as `NA`.

When doing operations on numbers, most functions will return `NA` if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data. You can add the argument `na.rm=TRUE` to calculate the result while ignoring the missing values.

```
heights <- c(2, 4, 4, NA, 6)
mean(heights)
max(heights)
mean(heights, na.rm = TRUE)
max(heights, na.rm = TRUE)
```

If your data include missing values, you may want to become familiar with the functions `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
## Extract those elements which are not missing values.
heights[!is.na(heights)]

## Returns the object with incomplete cases removed. The returned object is an atomic vector of type `"
na.omit(heights)

## Extract those elements which are complete cases. The returned object is an atomic vector of type `"nu
heights[complete.cases(heights)]
```

Recall that you can use the `typeof()` function to find the type of your atomic vector.

3.4.1 Challenge

1. Using this vector of heights in inches, create a new vector with the NAs removed.

   ```
   heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72, 65, 64, 70, 63, 65
   ```

2. Use the function `median()` to calculate the median of the `heights` vector.

Answer

```
heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72, 65, 64, 70, 63, 65)

# 1.
heights_no_na <- heights[!is.na(heights)]
# or
heights_no_na <- na.omit(heights)

# 2.
median(heights, na.rm = TRUE)
```

Now that we have learned how to write scripts, and the basics of R's data structures, we are ready to start working with the Portal dataset we have been using in the other lessons, and learn about data frames.

Chapter 4

Starting with Data

4.0.1 Learning Objectives

- Describe what a data frame is.
- Load external data from a .csv file into a data frame.
- Summarize the contents of a data frame.
- Format dates.

4.1 Presentation of the Survey Data

We are studying the species repartition and weight of animals caught in plots in our study area. The dataset is stored as a comma separated value (CSV) file. Each row holds information for a single animal, and the columns represent:

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

We are going to use the R function `download.file()` to download the CSV file that contains the survey data from figshare, and we will use `read_csv()` to load into memory the content of the CSV file as an

object of class `data.frame`. Inside the download.file command, the first entry is a character string with the source URL ("https://ndownloader.figshare.com/files/2292169"). This source URL downloads a CSV file from figshare. The text after the comma ("data/portal_data_joined.csv") is the destination of the file on your local machine. You'll need to have a folder on your machine called "data" where you'll download the file. So this command downloads a file from figshare, names it "portal_data_joined.csv," and adds it to a preexisting folder named "data."

```
library(tidyverse)

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

You are now ready to load the data:

```
surveys <- read_csv("data/portal_data_joined.csv")
```

```
#> Parsed with column specification:
#> cols(
#>   record_id = col_integer(),
#>   month = col_integer(),
#>   day = col_integer(),
#>   year = col_integer(),
#>   plot_id = col_integer(),
#>   species_id = col_character(),
#>   sex = col_character(),
#>   hindfoot_length = col_integer(),
#>   weight = col_integer(),
#>   genus = col_character(),
#>   species = col_character(),
#>   taxa = col_character(),
#>   plot_type = col_character()
#> )
```

This statement doesn't produce any output because, as you might recall, assignments don't display anything. If we want to check that our data has been loaded, we can see the contents of the data frame by typing its name: `surveys`.

Wow… that was a lot of output. At least it means the data loaded properly. Let's check the top (the first 6 lines) of this data frame using the function `head()`:

```
head(surveys)
```

```
#> # A tibble: 6 x 13
#>   record_id month   day  year plot_id species_id sex   hindfoot_length
#>       <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
#> 1         1     7    16  1977       2 NL         M                  32
#> 2        72     8    19  1977       2 NL         M                  31
#> 3       224     9    13  1977       2 NL         <NA>               NA
#> 4       266    10    16  1977       2 NL         <NA>               NA
#> 5       349    11    12  1977       2 NL         <NA>               NA
#> 6       363    11    12  1977       2 NL         <NA>               NA
#> # ... with 5 more variables: weight <int>, genus <chr>, species <chr>,
#> #   taxa <chr>, plot_type <chr>
```

```
## Try also
## View(surveys)
```

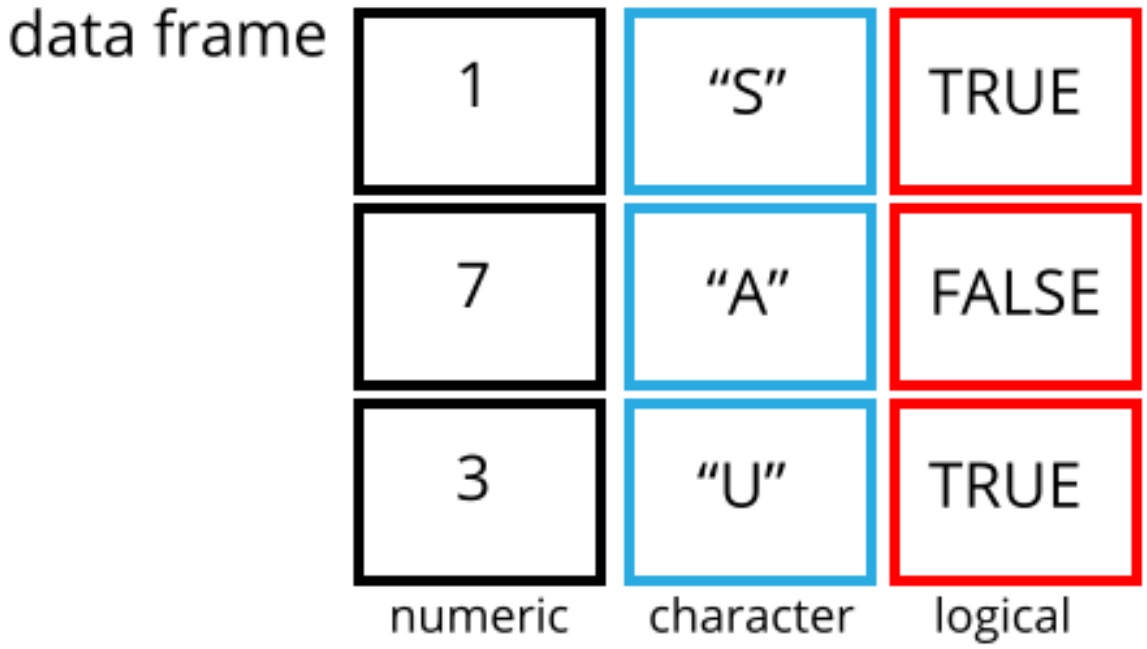


Figure 4.1:

4.2 What are data frames?

Data frames are the *de facto* data structure for most tabular data, and what we use for statistics and plotting.

A data frame can be created by hand, but most commonly they are generated by the functions `read_csv()` or `read.table()`; in other words, when importing spreadsheets from your hard drive (or the web).

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors). For example, here is a figure depicting a data frame comprising a numeric, a character, and a logical vector.

We can see this when inspecting the structure of a data frame with the function `str()`:

```
str(surveys)
```

4.3 Inspecting `data.frame` Objects

We already saw how the functions `head()` and `str()` can be useful to check the content and the structure of a data frame. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data. Let's try them out!

- Size:
  - `dim(surveys)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the **dim**ensions of the object)
  - `nrow(surveys)` - returns the number of rows
  - `ncol(surveys)` - returns the number of columns
- Content:

  - `head(surveys)` - shows the first 6 rows
  - `tail(surveys)` - shows the last 6 rows
- Names:
  - `names(surveys)` - returns the column names (synonym of `colnames()` for `data.frame` objects)
  - `rownames(surveys)` - returns the row names
- Summary:
  - `str(surveys)` - structure of the object and information about the class, length and content of each column
  - `summary(surveys)` - summary statistics for each column

Note: most of these functions are "generic", they can be used on other types of objects besides `data.frame`.

4.3.1 Challenge

Based on the output of `str(surveys)`, can you answer the following questions?

- What is the class of the object `surveys`?
- How many rows and how many columns are in this object?
- How many species have been recorded during these surveys?

Answer

```
str(surveys)
```

```
#> Classes 'tbl_df', 'tbl' and 'data.frame':  34786 obs. of  13 variables:
#>  $ record_id      : int  1 72 224 266 349 363 435 506 588 661 ...
#>  $ month          : int  7 8 9 10 11 11 12 1 2 3 ...
#>  $ day            : int  16 19 13 16 12 12 10 8 18 11 ...
#>  $ year           : int  1977 1977 1977 1977 1977 1977 1977 1978 1978 1978 ...
#>  $ plot_id        : int  2 2 2 2 2 2 2 2 2 2 ...
#>  $ species_id     : chr  "NL" "NL" "NL" "NL" ...
#>  $ sex            : chr  "M" "M" NA NA ...
#>  $ hindfoot_length: int  32 31 NA NA NA NA NA NA NA NA ...
#>  $ weight         : int  NA NA NA NA NA NA NA NA 218 NA ...
#>  $ genus          : chr  "Neotoma" "Neotoma" "Neotoma" "Neotoma" ...
#>  $ species        : chr  "albigula" "albigula" "albigula" "albigula" ...
#>  $ taxa           : chr  "Rodent" "Rodent" "Rodent" "Rodent" ...
#>  $ plot_type      : chr  "Control" "Control" "Control" "Control" ...
#>  - attr(*, "spec")=List of 2
#>   ..$ cols   :List of 13
#>   .. ..$ record_id      : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
#>   .. ..$ month          : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
#>   .. ..$ day            : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
#>   .. ..$ year           : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
#>   .. ..$ plot_id        : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
#>   .. ..$ species_id     : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
#>   .. ..$ sex            : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
#>   .. ..$ hindfoot_length: list()
```

```
#>   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
#>   .. ..$ weight         : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
#>   .. ..$ genus          : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
#>   .. ..$ species        : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
#>   .. ..$ taxa           : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
#>   .. ..$ plot_type      : list()
#>   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
#>   ..$ default: list()
#>   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
#>   ..- attr(*, "class")= chr "col_spec"
```

```
## * class: data frame
## * how many rows: 34786,  how many columns: 13
## * how many species: 48
```

4.4 Indexing and subsetting data frames

Our survey data frame has rows and columns (it has 2 dimensions), if we want to extract some specific data from it, we need to specify the "coordinates" we want from it. Row numbers come first, followed by column numbers. However, note that different ways of specifying these coordinates lead to results with different classes.

```
# first element in the first column of the data frame (as a vector)
surveys[1, 1]
# first element in the 6th column (as a vector)
surveys[1, 6]
# first column of the data frame (as a vector)
surveys[, 1]
# first column of the data frame (as a data.frame)
surveys[1]
# first three elements in the 7th column (as a vector)
surveys[1:3, 7]
# the 3rd row of the data frame (as a data.frame)
surveys[3, ]
# equivalent to head_surveys <- head(surveys)
head_surveys <- surveys[1:6, ]
```

`:` is a special function that creates numeric vectors of integers in increasing or decreasing order, test `1:10` and `10:1` for instance.

You can also exclude certain indices of a data frame using the "`-`" sign:

```
surveys[, -1]          # The whole data frame, except the first column
surveys[-c(7:34786), ] # Equivalent to head(surveys)
```

Data frames can be subset by calling indices (as shown previously), but also by calling their column names directly:

```
surveys["species_id"]       # Result is a data.frame
surveys[, "species_id"]     # Result is a vector
surveys[["species_id"]]     # Result is a vector
```

```
surveys$species_id          # Result is a vector
```

In RStudio, you can use the autocompletion feature to get the full and correct names of the columns.

4.5 Formatting Dates

This next section is taken from Chapter 16 of Hadley Wickham and Garrett Grolemund's book R for Data Science which is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 license.

4.5.1 Introduction

This chapter will show you how to work with dates and times in R. At first glance, dates and times seem simple. You use them all the time in your regular life, and they don't seem to cause much confusion. However, the more you learn about dates and times, the more complicated they seem to get. To warm up, try these three seemingly simple questions:

Does every year have 365 days? Does every day have 24 hours? Does every minute have 60 seconds? I'm sure you know that not every year has 365 days, but do you know the full rule for determining if a year is a leap year? (It has three parts.) You might have remembered that many parts of the world use daylight savings time (DST), so that some days have 23 hours, and others have 25. You might not have known that some minutes have 61 seconds because every now and then leap seconds are added because the Earth's rotation is gradually slowing down.

Dates and times are hard because they have to reconcile two physical phenomena (the rotation of the Earth and its orbit around the sun) with a whole raft of geopolitical phenomena including months, time zones, and DST. This chapter won't teach you every last detail about dates and times, but it will give you a solid grounding of practical skills that will help you with common data analysis challenges.

4.5.2 Prerequisites

This chapter will focus on the **lubridate** package, which makes it easier to work with dates and times in R. lubridate is not part of core tidyverse because you only need it when you're working with dates/times. We will also need nycflights13 for practice data.

```
library(tidyverse)
library(lubridate)
#install.packages('nycflights13')
library(nycflights13)
```

4.6 Creating date/times

There are three types of date/time data that refer to an instant in time:

- A **date**. Tibbles print this as `<date>`.
- A **time** within a day. Tibbles print this as `<time>`.
- A **date-time** is a date plus a time: it uniquely identifies an instant in time (typically to the nearest second). Tibbles print this as `<dttm>`. Elsewhere in R these are called POSIXct, but I don't think that's a very useful name.

In this chapter we are only going to focus on dates and date-times as R doesn't have a native class for storing times. If you need one, you can use the **hms** package.

You should always use the simplest possible data type that works for your needs. That means if you can use a date instead of a date-time, you should. Date-times are substantially more complicated because of the need to handle time zones, which we'll come back to at the end of the chapter.

To get the current date or date-time you can use `today()` or `now()`:

```
today()
now()
```

Otherwise, there are three ways you're likely to create a date/time:

- From a string.
- From individual date-time components.
- From an existing date/time object.

They work as follows.

4.6.1 From strings

Date/time data often comes as strings. You've seen one approach to parsing strings into date-times in date-times. Another approach is to use the helpers provided by lubridate. They automatically work out the format once you specify the order of the component. To use them, identify the order in which year, month, and day appear in your dates, then arrange "y", "m", and "d" in the same order. That gives you the name of the lubridate function that will parse your date. For example:

```
ymd("2017-01-31")
mdy("January 31st, 2017")
dmy("31-Jan-2017")
```

These functions also take unquoted numbers. This is the most concise way to create a single date/time object, as you might need when filtering date/time data. `ymd()` is short and unambiguous:

```
ymd(20170131)
```

`ymd()` and friends create dates. To create a date-time, add an underscore and one or more of "h", "m", and "s" to the name of the parsing function:

```
ymd_hms("2017-01-31 20:11:59")
mdy_hm("01/31/2017 08:01")
```

You can also force the creation of a date-time from a date by supplying a timezone:

```
ymd(20170131, tz = "UTC")
```

4.6.2 From other types

You may want to switch between a date-time and a date. That's the job of `as_datetime()` and `as_date()`:

```
as_datetime(today())
as_date(now())
```

Sometimes you'll get date/times as numeric offsets from the "Unix Epoch", 1970-01-01. If the offset is in seconds, use `as_datetime()`; if it's in days, use `as_date()`.

```
as_datetime(60 * 60 * 10)
as_date(365 * 10 + 2)
```

4.6.3 Exercises

1. What happens if you parse a string that contains invalid dates?

```
ymd(c("2010-10-10", "bananas"))
```

2. What does the `tzone` argument to `today()` do? Why is it important?

3. Use the appropriate lubridate function to parse each of the following dates:

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
```

4.7 Date-time components

Now that you know how to get date-time data into R's date-time data structures, let's explore what you can do with them. This section will focus on the accessor functions that let you get and set individual components. The next section will look at how arithmetic works with date-times.

4.7.1 Getting components

You can pull out individual parts of the date with the accessor functions `year()`, `month()`, `mday()` (day of the month), `yday()` (day of the year), `wday()` (day of the week), `hour()`, `minute()`, and `second()`.

```
datetime <- ymd_hms("2016-07-08 12:34:56")
year(datetime)
month(datetime)
mday(datetime)
yday(datetime)
wday(datetime)
```

For `month()` and `wday()` you can set `label = TRUE` to return the abbreviated name of the month or day of the week. Set `abbr = FALSE` to return the full name.

```
month(datetime, label = TRUE)
wday(datetime, label = TRUE, abbr = FALSE)
```

4.8 Time spans

Next you'll learn about how arithmetic with dates works, including subtraction, addition, and division. Along the way, you'll learn about three important classes that represent time spans:

- **durations**, which represent an exact number of seconds.
- **periods**, which represent human units like weeks and months.
- **intervals**, which represent a starting and ending point.

4.8.1 Durations

In R, when you subtract two dates, you get a difftime object:

```
# How old is Hadley?
h_age <- today() - ymd(19791014)
h_age
```

For more see R for Data Science.

Chapter 5

Manipulating and analyzing data with dplyr

5.0.1 Learning Objectives

- Describe the purpose of the **dplyr** and **tidyr** packages.
- Select certain columns in a data frame with the **dplyr** function `select`.
- Select certain rows in a data frame according to filtering conditions with the **dplyr** function `filter` .
- Link the output of one **dplyr** function to the input of another function with the 'pipe' operator `%>%`.
- Add new columns to a data frame that are functions of existing columns with `mutate`.
- Use the split-apply-combine concept for data analysis.
- Use `summarize`, `group_by`, and `count` to split a data frame into groups of observations, apply a summary statistics for each group, and then combine the results.
- Describe the concept of a wide and a long table format and for which purpose those formats are useful.
- Describe what key-value pairs are.
- Reshape a data frame from long to wide format and back with the `spread` and `gather` commands from the **tidyr** package.
- Export a data frame to a .csv file. Work with Factors in R

5.1 Data Manipulation using `dplyr` and `tidyr`

Bracket subsetting is handy, but it can be cumbersome and difficult to read, especially for complicated operations. Enter **dplyr**. **dplyr** is a package for making tabular data manipulation easier. It pairs nicely with **tidyr** which enables you to swiftly convert between different data formats for plotting and analysis.

Packages in R are basically sets of additional functions that let you do more stuff. The functions we've been using so far, like `str()` or `data.frame()`, come built into R; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent R session when you need it. You should already have installed the **tidyverse**

package. This is an "umbrella-package" that installs several packages useful for data analysis which work together well such as **tidyr**, **dplyr**, **ggplot2**, **tibble**, etc.

To load the package type:

```
## load the tidyverse packages, incl. dplyr
library("tidyverse")
```

5.2 What are dplyr and tidyr?

The package **dplyr** provides easy tools for the most common data manipulation tasks. It is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++). An additional feature is the ability to work directly with data stored in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query are returned.

This addresses a common problem with R in that all operations are conducted in-memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can connect to a database of many hundreds of GB, conduct queries on it directly, and pull back into R only what you need for analysis.

The package **tidyr** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups - like plots or aquaria. Moving back and forth between these formats is nontrivial, and **tidyr** gives you tools for this and more sophisticated data manipulation.

To learn more about **dplyr** and **tidyr** after the workshop, you may want to check out this handy data transformation with **dplyr** cheatsheet and this one about **tidyr**.

```
surveys <- read_csv("data/portal_data_joined.csv")
```

```
#> Parsed with column specification:
#> cols(
#>   record_id = col_integer(),
#>   month = col_integer(),
#>   day = col_integer(),
#>   year = col_integer(),
#>   plot_id = col_integer(),
#>   species_id = col_character(),
#>   sex = col_character(),
#>   hindfoot_length = col_integer(),
#>   weight = col_integer(),
#>   genus = col_character(),
#>   species = col_character(),
#>   taxa = col_character(),
#>   plot_type = col_character()
#> )
```

```
## inspect the data
str(surveys)

## preview the data
# View(surveys)
```

We're going to learn some of the most common **dplyr** functions:

- `select()`: subset columns
- `filter()`: subset rows on conditions
- `mutate()`: create new columns by using information from other columns
- `group_by()` and `summarize()`: create summary statisitcs on grouped data
- `arrange()`: sort results
- `count()`: count discrete values

5.3 Selecting columns and filtering rows

To select columns of a data frame, use `select()`. The first argument to this function is the data frame (`surveys`), and the subsequent arguments are the columns to keep.

```
select(surveys, plot_id, species_id, weight)
```

To select all columns *except* certain ones, put a "-" in front of the variable to exclude it.

```
select(surveys, -record_id, -species_id)
```

This will select all the variables in `surveys` except `record_id` and `species_id`.

To choose rows based on a specific criteria, use `filter()`:

```
filter(surveys, year == 1995)
```

5.4 Pipes

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

With intermediate steps, you create a temporary data frame and use that as input to the next function, like this:

```
surveys2 <- filter(surveys, weight < 5)
surveys_sml <- select(surveys2, species_id, sex, weight)
```

This is readable, but can clutter up your workspace with lots of objects that you have to name individually. With multiple steps, that can be hard to keep track of.

You can also nest functions (i.e. one function inside of another), like this:

```
surveys_sml <- select(filter(surveys, weight < 5), species_id, sex, weight)
```

This is handy, but can be difficult to read if too many functions are nested, as R evaluates the expression from the inside out (in this case, filtering, then selecting).

The last option, *pipes*, are a recent addition to R. Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in R look like `%>%` and are made available via the **`magrittr`** package, installed automatically with **`dplyr`**. If you use RStudio, you can type the pipe with Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

```
surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)
```

In the above code, we use the pipe to send the `surveys` dataset first through `filter()` to keep rows where `weight` is less than 5, then through `select()` to keep only the `species_id`, `sex`, and `weight` columns. Since

`%>%` takes the object on its left and passes it as the first argument to the function on its right, we don't need to explicitly include the data frame as an argument to the `filter()` and `select()` functions any more.

Some may find it helpful to read the pipe like the word "then". For instance, in the above example, we took the data frame `surveys`, *then* we `filter`ed for rows with `weight < 5`, *then* we `select`ed columns `species_id`, `sex`, and `weight`. The `dplyr` functions by themselves are somewhat simple, but by combining them into linear workflows with the pipe, we can accomplish more complex manipulations of data frames.

If we want to create a new object with this smaller version of the data, we can assign it a new name:

```
surveys_sml <- surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)

surveys_sml
```

Note that the final data frame is the leftmost part of this expression.

5.4.1 Challenge

Using pipes, subset the `surveys` data to include animals collected before 1995 and retain only the columns `year`, `sex`, and `weight`.

Answer

```
surveys %>%
    filter(year < 1995) %>%
    select(year, sex, weight)
```

5.4.2 Mutate

Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions, or to find the ratio of values in two columns. For this we'll use `mutate()`.

To create a new column of weight in kg:

```
surveys %>%
  mutate(weight_kg = weight / 1000)
```

You can also create a second new column based on the first new column within the same call of `mutate()`:

```
surveys %>%
  mutate(weight_kg = weight / 1000,
         weight_kg2 = weight_kg * 2)
```

If this runs off your screen and you just want to see the first few rows, you can use a pipe to view the `head()` of the data. (Pipes work with non-`dplyr` functions, too, as long as the `dplyr` or `magrittr` package is loaded).

```
surveys %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

The first few rows of the output are full of `NA`s, so if we wanted to remove those we could insert a `filter()` in the chain:

```
surveys %>%
  filter(!is.na(weight)) %>%
```

```
  mutate(weight_kg = weight / 1000) %>%
  head()
```

`is.na()` is a function that determines whether something is an `NA`. The `!` symbol negates the result, so we're asking for every row where weight *is not* an `NA`.

5.4.3 Challenge

Create a new data frame from the `surveys` data that meets the following criteria: contains only the `species_id` column and a new column called `hindfoot_half` containing values that are half the `hindfoot_length` values. In this `hindfoot_half` column, there are no `NA`s and all values are less than 30.

Hint: think about how the commands should be ordered to produce this data frame!

Answer

```
surveys_hindfoot_half <- surveys %>%
    filter(!is.na(hindfoot_length)) %>%
    mutate(hindfoot_half = hindfoot_length / 2) %>%
    filter(hindfoot_half < 30) %>%
    select(species_id, hindfoot_half)
```

5.4.4 The `summarize()` function

`group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group. `group_by()` takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics. So to compute the mean `weight` by sex:

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

You may also have noticed that the output from these calls doesn't run off the screen anymore. It's one of the advantages of `tbl_df` over data frame.

You can also group by multiple columns:

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

When grouping both by `sex` and `species_id`, the last few rows are for animals that escaped before their sex and body weights could be determined. You may notice that the last column does not contain `NA` but `NaN` (which refers to "Not a Number"). To avoid this, we can remove the missing values for weight before we attempt to calculate the summary statistics on weight. Because the missing values are removed first, we can omit `na.rm = TRUE` when computing the mean:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight))
```

Here, again, the output from these calls doesn't run off the screen anymore. If you want to display more data, you can use the `print()` function at the end of your chain with the argument `n` specifying the number of rows to display:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight)) %>%
  print(n = 15)
```

Once the data are grouped, you can also summarize multiple variables at the same time (and not necessarily on the same variable). For instance, we could add a column indicating the minimum weight for each species for each sex:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight))
```

It is sometimes useful to rearrange the result of a query to inspect the values. For instance, we can sort on `min_weight` to put the lighter species first:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(min_weight)
```

To sort in descending order, we need to add the `desc()` function. If we want to sort the results by decreasing order of mean weight:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(desc(mean_weight))
```

5.4.5 Counting

When working with data, we often want to know the number of observations found for each factor or combination of factors. For this task, **`dplyr`** provides `count()`. For example, if we wanted to count the number of rows of data for each sex, we would do:

```
surveys %>%
    count(sex)
```

The `count()` function is shorthand for something we've already seen: grouping by a variable, and summarizing it by counting the number of observations in that group. In other words, `surveys %>% count()` is equivalent to:

```
surveys %>%
    group_by(sex) %>%
    summarise(count = n())
```

For convenience, `count()` provides the `sort` argument:

```
surveys %>%
    count(sex, sort = TRUE)
```

5.4.6 Challenge

1. How many animals were caught in each `plot_type` surveyed?

Answer

```
surveys %>%
    count(plot_type)
```

2. Use `group_by()` and `summarize()` to find the mean, min, and max hindfoot length for each species (using `species_id`). Also add the number of observations (hint: see `?n`).

Answer

```
surveys %>%
    filter(!is.na(hindfoot_length)) %>%
    group_by(species_id) %>%
    summarize(
        mean_hindfoot_length = mean(hindfoot_length),
        min_hindfoot_length = min(hindfoot_length),
        max_hindfoot_length = max(hindfoot_length),
        n = n()
    )
```

3. What was the heaviest animal measured in each year? Return the columns `year`, `genus`, `species_id`, and `weight`.

Answer

```
surveys %>%
    filter(!is.na(weight)) %>%
    group_by(year) %>%
    filter(weight == max(weight)) %>%
    select(year, genus, species, weight) %>%
    arrange(year)
```

5.4.7 Reshaping with gather and spread

In the spreadsheet lesson, we discussed how to structure our data leading to the four rules defining a tidy dataset:

1. Each variable has its own column
2. Each observation has its own row
3. Each value must have its own cell
4. Each type of observational unit forms a table

Sometimes your you want to spread the observations of on variable across multiple columns.

5.4.7.1 Spreading

`spread()` takes three principal arguments:

1. the data
2. the *key* column variable whose values will become new column names.

3. the *value* column variable whose values will fill the new column variables.

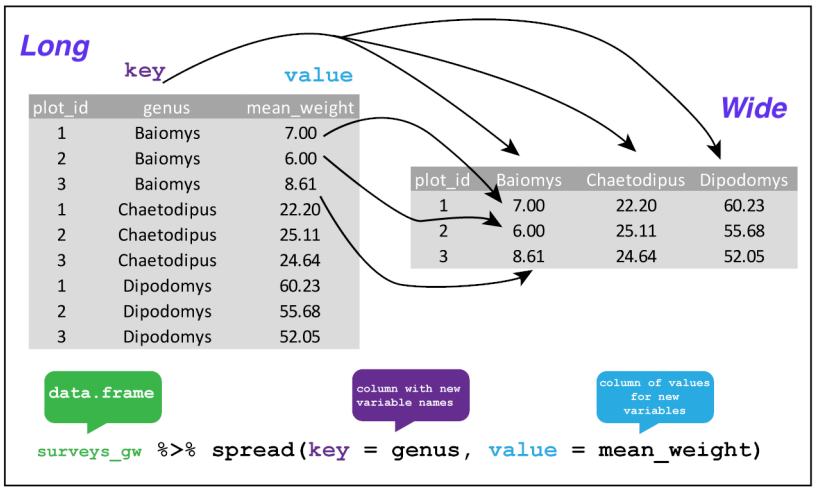


Figure 5.1:

Further arguments include `fill` which, if set, fills in missing values with the value provided.

Let's use `spread()` to transform surveys to find the mean weight of each species in each plot over the entire survey period. We use `filter()`, `group_by()` and `summarise()` to filter our observations and variables of interest, and create a new variable for the `mean_weight`. We use the pipe as before too.

```
surveys_gw <- surveys %>%
  filter(!is.na(weight)) %>%
  group_by(genus, plot_id) %>%
  summarize(mean_weight = mean(weight))

str(surveys_gw)
```

This yields `surveys_gw` where the observations for each plot are spread across multiple rows, 196 observations of 3 variables. Using `spread()` to key on `genus` with values from `mean_weight` this becomes 24 observations of 11 variables, one row for each plot. We again use pipes:

```
surveys_spread <- surveys_gw %>%
  spread(key = genus, value = mean_weight)

str(surveys_spread)
```

We could now plot comparisons between the weight of species in different plots, although we may wish to fill in the missing values first.

```
surveys_gw %>%
  spread(genus, mean_weight, fill = 0) %>%
  head()
```

5.4.7.2 Gathering

The opposing situation could occur if we had been provided with data in the form of `surveys_spread`, where the genus names are column names, but we wish to treat them as values of a genus variable instead.

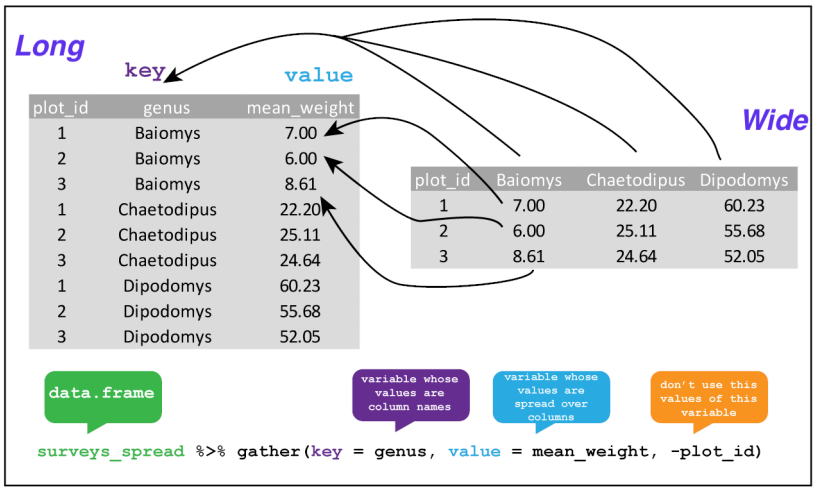


Figure 5.2:

In this situation we are gathering the column names and turning them into a pair of new variables. One variable represents the column names as values, and the other variable contains the values previously associated with the column names.

`gather()` takes four principal arguments:

1. the data
2. the *key* column variable we wish to create from column names.
3. the *values* column variable we wish to create and fill with values associated with the key.
4. the names of the columns we use to fill the key variable (or to drop).

To recreate `surveys_gw` from `surveys_spread` we would create a key called `genus` and value called `mean_weight` and use all columns except `plot_id` for the key variable. Here we drop `plot_id` column with a minus sign.

```
surveys_gather <- surveys_spread %>%
  gather(key = genus, value = mean_weight, -plot_id)

str(surveys_gather)
```

Note that now the `NA` genera are included in the re-gathered format. Spreading and then gathering can be a useful way to balance out a dataset so every replicate has the same composition.

We could also have used a specification for what columns to include. This can be useful if you have a large number of identifying columns, and it's easier to specify what to gather than what to leave alone. And if the columns are in a row, we don't even need to list them all out - just use the `:` operator!

```
surveys_spread %>%
  gather(key = genus, value = mean_weight, Baiomys:Spermophilus) %>%
  head()
```

5.4.8 Challenge

1. Spread the `surveys` data frame with `year` as columns, `plot_id` as rows, and the number of genera per plot as the values. You will need to summarize before reshaping, and use the function `n_distinct()` to get the number of unique genera within a particular chunk of data. It's a powerful function! See `?n_distinct` for more.

Answer

```
rich_time <- surveys %>%
  group_by(plot_id, year) %>%
  summarize(n_genera = n_distinct(genus)) %>%
  spread(year, n_genera)

head(rich_time)
```

2. Now take that data frame and `gather()` it again, so each row is a unique `plot_id` by `year` combination.

Answer

```
rich_time %>%
  gather(year, n_genera, -plot_id)
```

3. The `surveys` data set has two measurement columns: `hindfoot_length` and `weight`. This makes it difficult to do things like look at the relationship between mean values of each measurement per year in different plot types. Let's walk through a common solution for this type of problem. First, use `gather()` to create a dataset where we have a key column called `measurement` and a `value` column that takes on the value of either `hindfoot_length` or `weight`. *Hint*: You'll need to specify which columns are being gathered.

Answer

```
surveys_long <- surveys %>%
  gather(measurement, value, hindfoot_length, weight)
```

4. With this new data set, calculate the average of each `measurement` in each `year` for each different `plot_type`. Then `spread()` them into a data set with a column for `hindfoot_length` and `weight`. *Hint*: You only need to specify the key and value columns for `spread()`.

Answer

```
surveys_long %>%
  group_by(year, measurement, plot_type) %>%
  summarize(mean_value = mean(value, na.rm=TRUE)) %>%
  spread(measurement, mean_value)
```

5.5 Exporting data

Now that you have learned how to use **dplyr** to extract information from or summarize your raw data, you may want to export these new data sets to share them with your collaborators or for archival.

Similar to the `read_csv()` function used for reading CSV files into R, there is a `write_csv()` function that generates CSV files from data frames.

```
write_csv(surveys_spread, "data/surveys_spread.csv")
```

5.6 Factors

This section is from Jenny Bryan's course at UBC: Stat 545

Factors are the variable type that useRs love to hate. It is how we store truly categorical information in R. The values a factor can take on are called the levels. In general, the levels are friendly human-readable character strings, like "male/female" and "control/treated". But never ever ever forget that, under the hood, R is really storing integer codes 1, 2, 3, etc.

This Janus-like nature of factors means they are rich with booby traps for the unsuspecting but they are a necessary evil. I recommend you learn how to be the boss of your factors. The pros far outweigh the cons. Specifically in modelling and figure-making, factors are anticipated and accommodated by the functions and packages you will want to exploit.

The worst kind of factor is the stealth factor. The variable that you think of as character, but that is actually a factor (numeric!!). This is a classic R gotcha. Check your variable types explicitly when things seem weird. It happens to the best of us.

Where do stealth factors come from? Base R has a burning desire to turn character information into factor. The happens most commonly at data import via read.csv() and friends. But data.frame() and other functions are also eager to convert character to factor. To shut this down, use stringsAsFactors = FALSE in read.csv() and data.frame() or – even better – use the tidyverse! For data import, use readr::read_csv(), readr::read_tsv(), etc. For data frame creation, use tibble::tibble(). And so on.

See for yourself the difference in how strings are treated with the two different methods of reading a csv file.

```
surveys. <- read.csv("data/surveys.csv")
surveys_ <- read_csv("data/surveys.csv")
```

```
#> Parsed with column specification:
#> cols(
#>   record_id = col_integer(),
#>   month = col_integer(),
#>   day = col_integer(),
#>   year = col_integer(),
#>   plot_id = col_integer(),
#>   species_id = col_character(),
#>   sex = col_character(),
#>   hindfoot_length = col_integer(),
#>   weight = col_integer()
#> )
```

```
str(surveys.)
str(surveys_)
```

5.6.1 The forcats Package

forcats is a core package in the tidyverse. It is installed via `install.packages("tidyverse")` and attached via `library(tidyverse)`. You can always load it individually via `library(forcats)`. Main functions start with fct_. There really is no coherent family of base functions that forcats replaces – that's why it's such a welcome addition.

```
library(tidyverse)
library(forcats)


library(gapminder)
```

Get to know your factor before you start touching it! It's polite. Let's use gapminder$continent as our example.

```
str(gapminder$continent)
levels(gapminder$continent)
nlevels(gapminder$continent)
class(gapminder$continent)
```

To get a frequency table as a tibble, from a tibble, use dplyr::count(). To get similar from a free-range factor, use forcats::fct_count().

```
gapminder %>%
  count(continent)

fct_count(gapminder$continent)
```

5.6.1.1 Dropping unused levels:

Just because you drop all the rows corresponding to a specific factor level, the levels of the factor itself do not change. Sometimes all these unused levels can come back to haunt you later, e.g., in figure legends.

Watch what happens to the levels of country (= nothing) when we filter Gapminder to a handful of countries.

```
nlevels(gapminder$country)

h_countries <- c("Egypt", "Haiti", "Romania", "Thailand", "Venezuela")

h_gap <- gapminder %>%
  filter(country %in% h_countries)

nlevels(h_gap$country)
```

Even though `h_gap` only has data for a handful of countries, we are still schlepping around all 142 levels from the original gapminder tibble.

How can you get rid of them? The base function `droplevels()` operates on all the factors in a data frame or on a single factor. The function `forcats::fct_drop()` operates on a factor.

```
h_gap_dropped <- h_gap %>%
  droplevels()
nlevels(h_gap_dropped$country)

h_gap$country %>%
  fct_drop() %>%
  levels()
```

5.6.1.2 Change order of the levels, principled

By default, factor levels are ordered alphabetically. Which might as well be random, when you think about it! It is preferable to order the levels according to some principle:

- Frequency.Make the most common level the first and so on.
- Another variable. Order factor levels according to a summary statistic for another variable. Example: order Gapminder countries by life expectancy.

First, let's order continent by frequency, forwards and backwards. This is often a great idea for tables and figures, esp. frequency barplots.

```
## default order is alphabetical
gapminder$continent %>%
  levels()

## order by frequency
gapminder$continent %>%
  fct_infreq() %>%
  levels()

## backwards!
gapminder$continent %>%
  fct_infreq() %>%
  fct_rev() %>%
  levels()

## order countries by median life expectancy
fct_reorder(gapminder$country, gapminder$lifeExp) %>%
  levels() %>% head()
```

5.6.1.3 Change order of the levels, "because I said so"

Sometimes you just want to hoist one or more levels to the front. Why? Because I said so.

```
h_gap$country %>% levels()

h_gap$country %>% fct_relevel("Romania", "Haiti") %>% levels()
```

This might be useful if you are preparing a report for, say, the Romanian government. The reason for always putting Romania first has nothing to do with the data, it is important for external reasons and you need a way to express this.

5.6.1.4 Recode the levels

Sometimes you have better ideas about what certain levels should be. This is called recoding.

```
i_gap <- gapminder %>%
  filter(country %in% c("United States", "Sweden", "Australia")) %>%
  droplevels()

i_gap$country %>% levels()

i_gap$country %>%
  fct_recode("USA" = "United States", "Oz" = "Australia") %>% levels()
```

Exercise: Isolate the data for "Australia", "Korea, Dem. Rep.", and "Korea, Rep." in the 2000x. Revalue the country factor levels to "Oz", "North Korea", and "South Korea".

5.6.2 Learning Objectives

- Produce scatter plots, boxplots, and time series plots using ggplot.
- Set universal plot settings.
- Describe what faceting is and apply faceting in ggplot.
- Modify the aesthetics of an existing ggplot plot (including axis labels and color).
- Build complex and customized plots from data in a data frame.

Chapter 6

Data Viz: ggplot2

We start by loading the required packages. **ggplot2** is included in the **tidyverse** package.

```
library(tidyverse)
```

If not still in the workspace, load the data we saved in the previous lesson.

```
surveys_complete <- read_csv("data_output/surveys_complete.csv")
```

6.1 Plotting with ggplot2

ggplot2 is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties. Therefore, we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatterplot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

ggplot2 functions like data in the 'long' format, i.e., a column for every dimension, and a row for every observation. Well-structured data will save you lots of time when making figures with **ggplot2**

ggplot graphics are built step by step by adding new elements. Adding layers in this fashion allows for extensive flexibility and customization of plots.

To build a ggplot, we will use the following basic template that can be used for different types of plots:

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

- use the `ggplot()` function and bind the plot to a specific data frame using the `data` argument

```
ggplot(data = surveys_complete)
```

- define a mapping (using the aesthetic (`aes`) function), by selecting the variables to be plotted and specifying how to present them in the graph, e.g. as x/y positions or characteristics such as size, shape, color, etc.

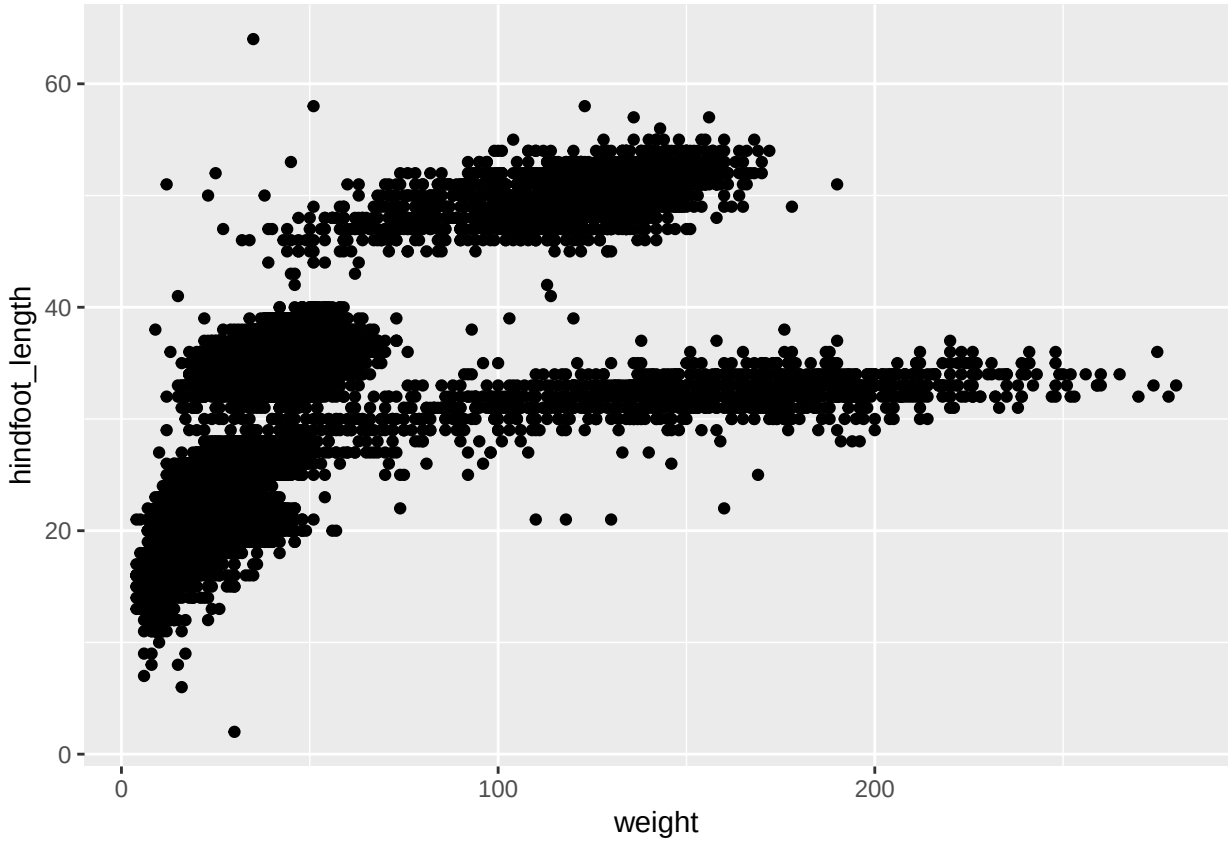
```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length))
```

- add 'geoms' – graphical representations of the data in the plot (points, lines, bars). **ggplot2** offers many different geoms; we will use some common ones today, including:
  - `geom_point()` for scatter plots, dot plots, etc.
  - `geom_boxplot()` for, well, boxplots!

- `geom_line()` for trend lines, time series, etc.

To add a geom to the plot use the `+` operator. Because we have two continuous variables, let's use `geom_point()` first:

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point()
```



The `+` in the **ggplot2** package is particularly useful because it allows you to modify existing `ggplot` objects. This means you can easily set up plot templates and conveniently explore different types of plots, so the above plot can also be generated with code like this:

```
# Assign plot to a variable
surveys_plot <- ggplot(data = surveys_complete,
                       mapping = aes(x = weight, y = hindfoot_length))

# Draw the plot
surveys_plot +
    geom_point()
```

Notes

- Anything you put in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x- and y-axis mapping you set up in `aes()`.
- You can also specify mappings for a given geom independently of the mappings defined globally in the `ggplot()` function.
- The `+` sign used to add new layers must be placed at the end of the line containing the *previous* layer. If, instead, the `+` sign is added at the beginning of the line containing the new layer, **ggplot2** will not add the new layer and will return an error message.

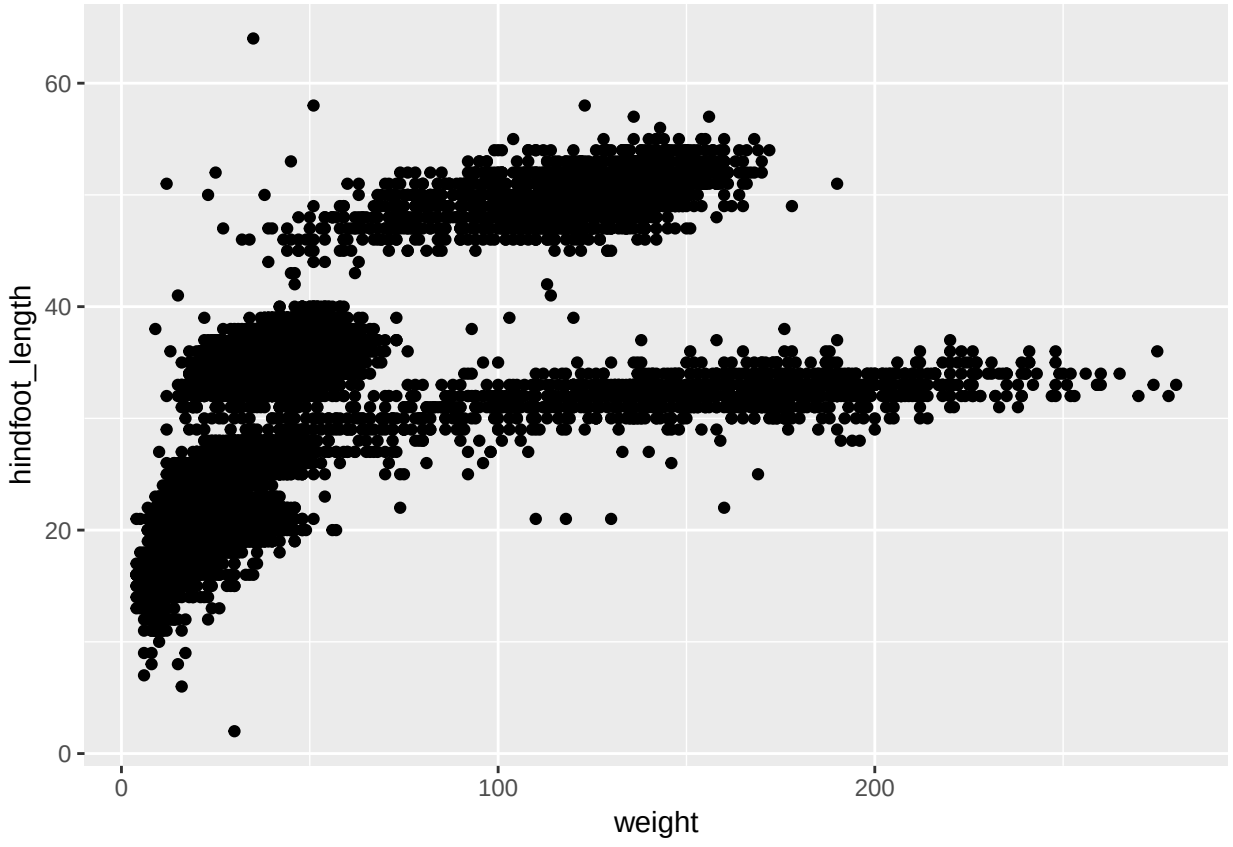
```
# This is the correct syntax for adding layers
surveys_plot +
  geom_point()

# This will not add the new layer and will return an error message
surveys_plot
  + geom_point()
```

6.2 Building your plots iteratively

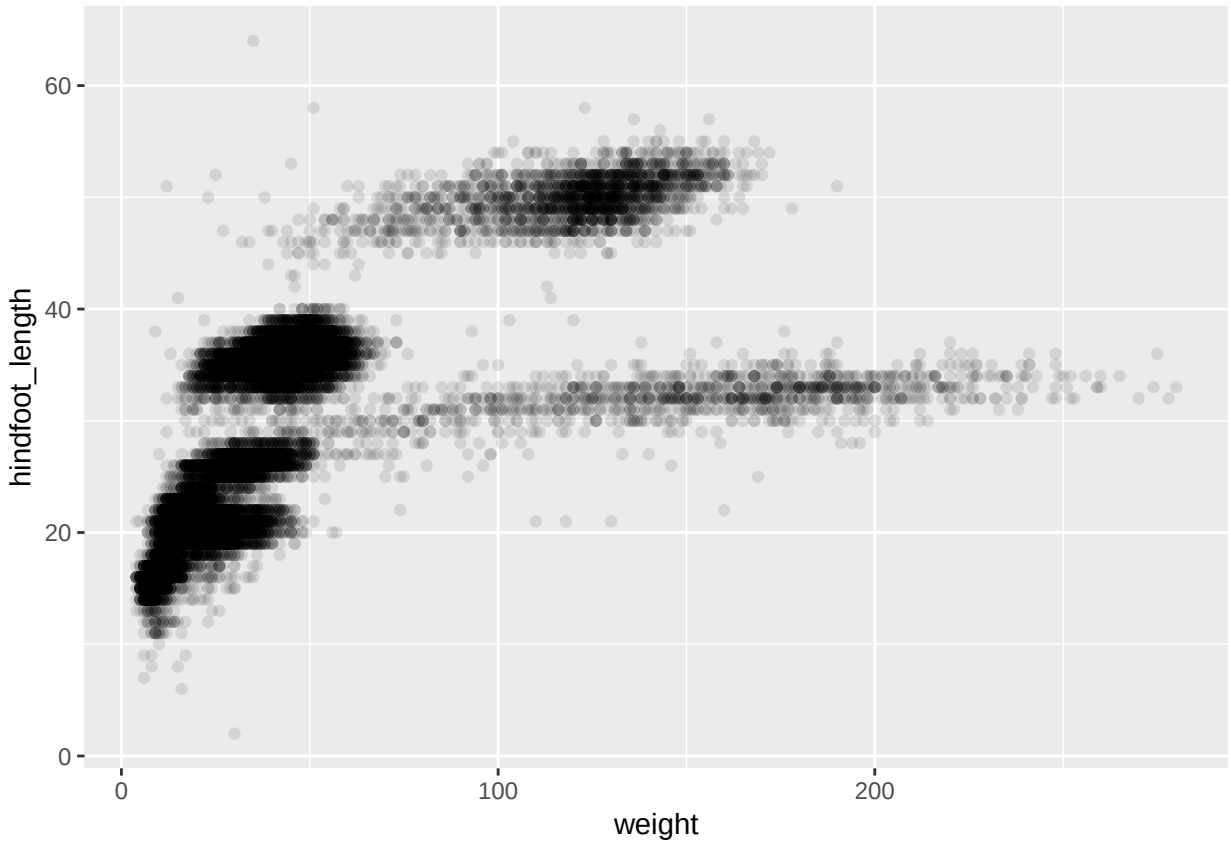
Building plots with **ggplot2** is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point()
```



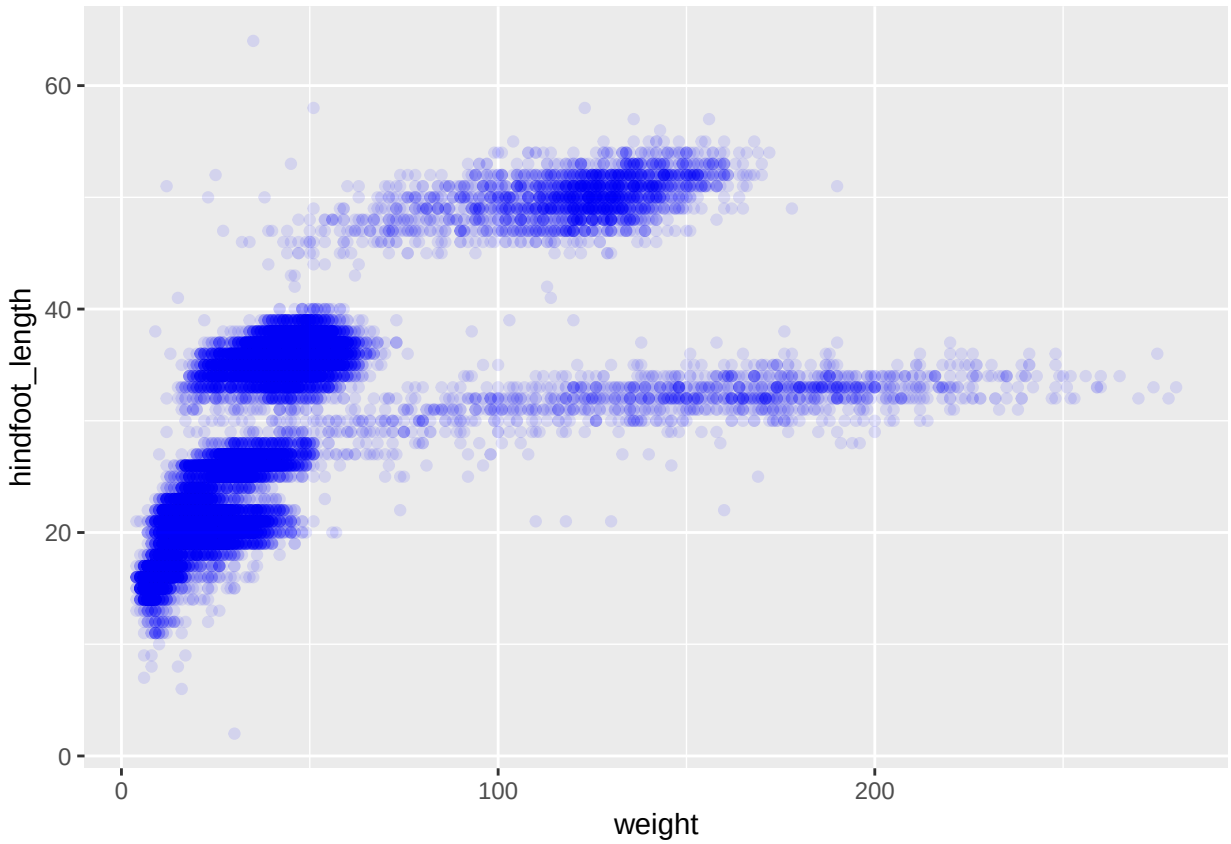
Then, we start modifying this plot to extract more information from it. For instance, we can add transparency (`alpha`) to avoid overplotting:

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1)
```



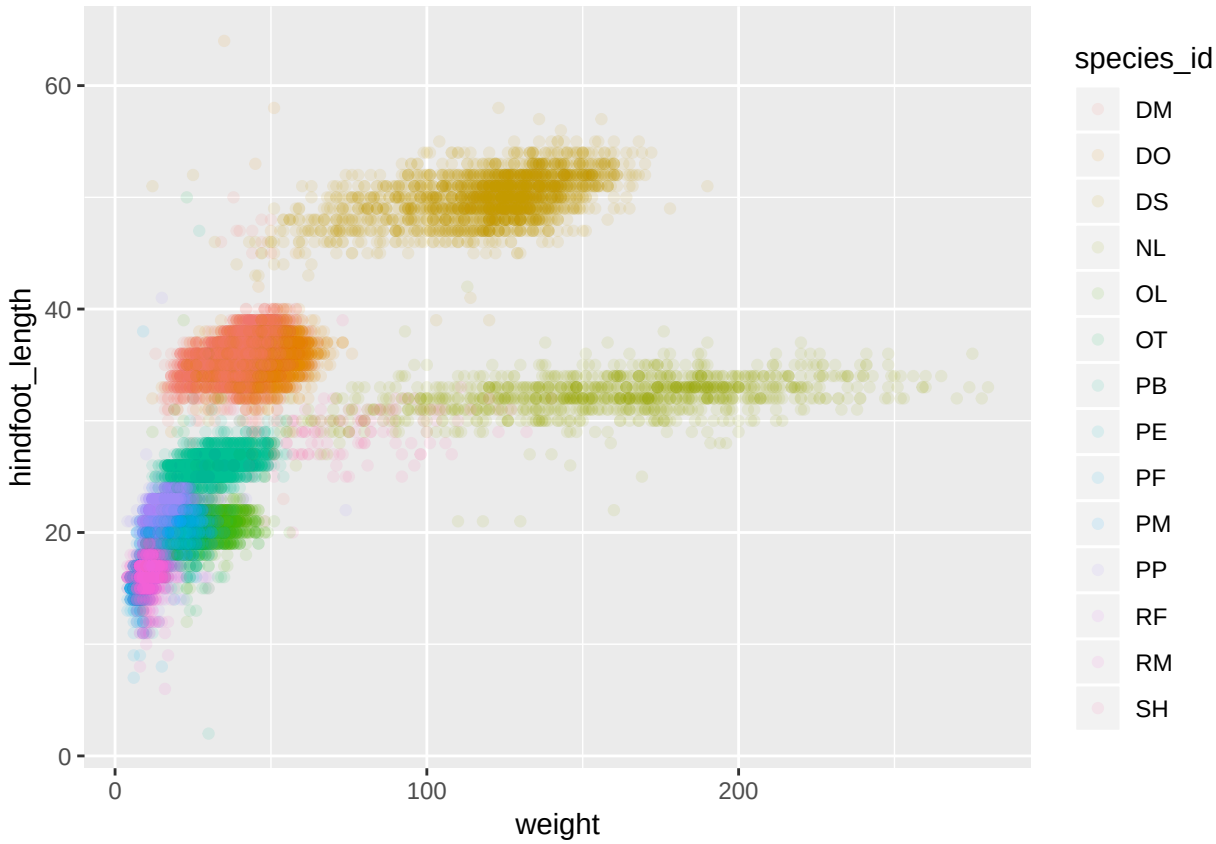
We can also add colors for all the points:

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1, color = "blue")
```



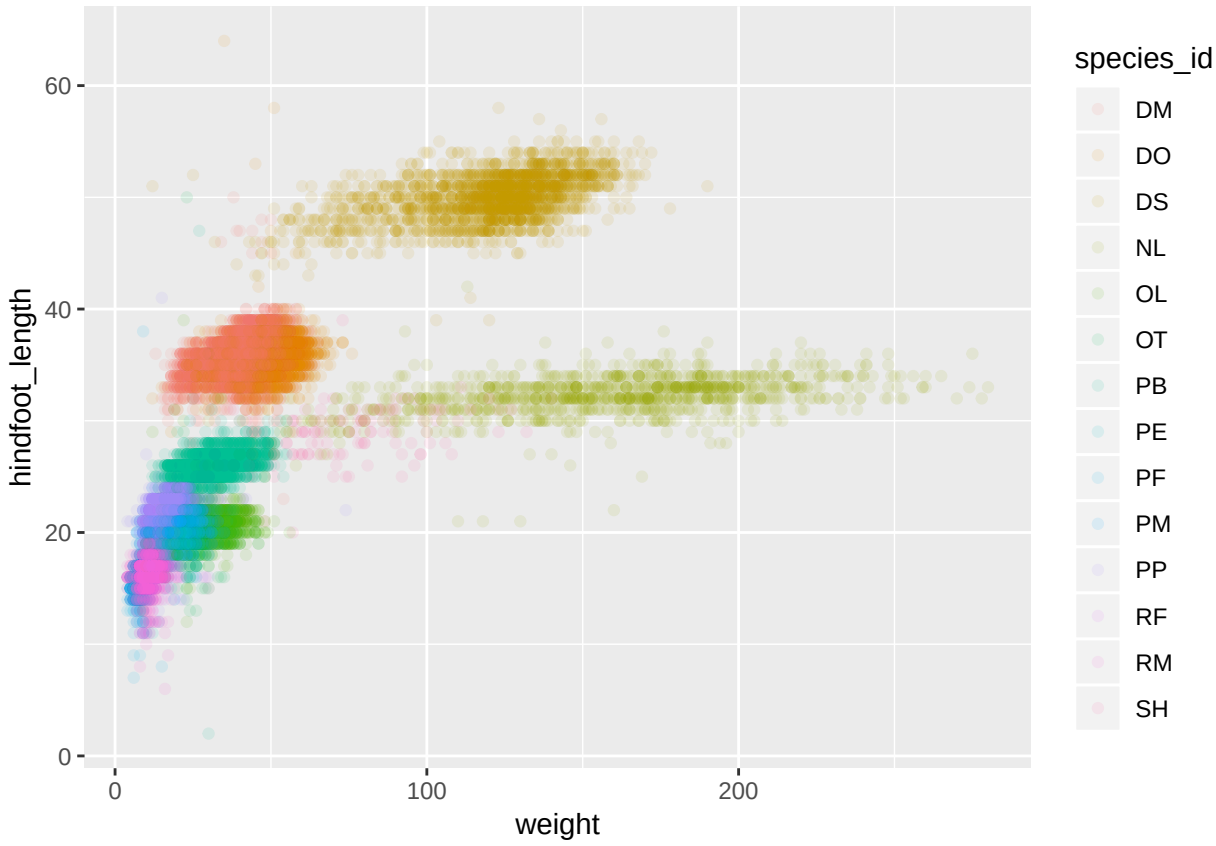
Or to color each species in the plot differently, you could use a vector as an input to the argument **color**. **ggplot2** will provide a different color corresponding to different values in the vector. Here is an example where we color with **species_id**:

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1, aes(color = species_id))
```



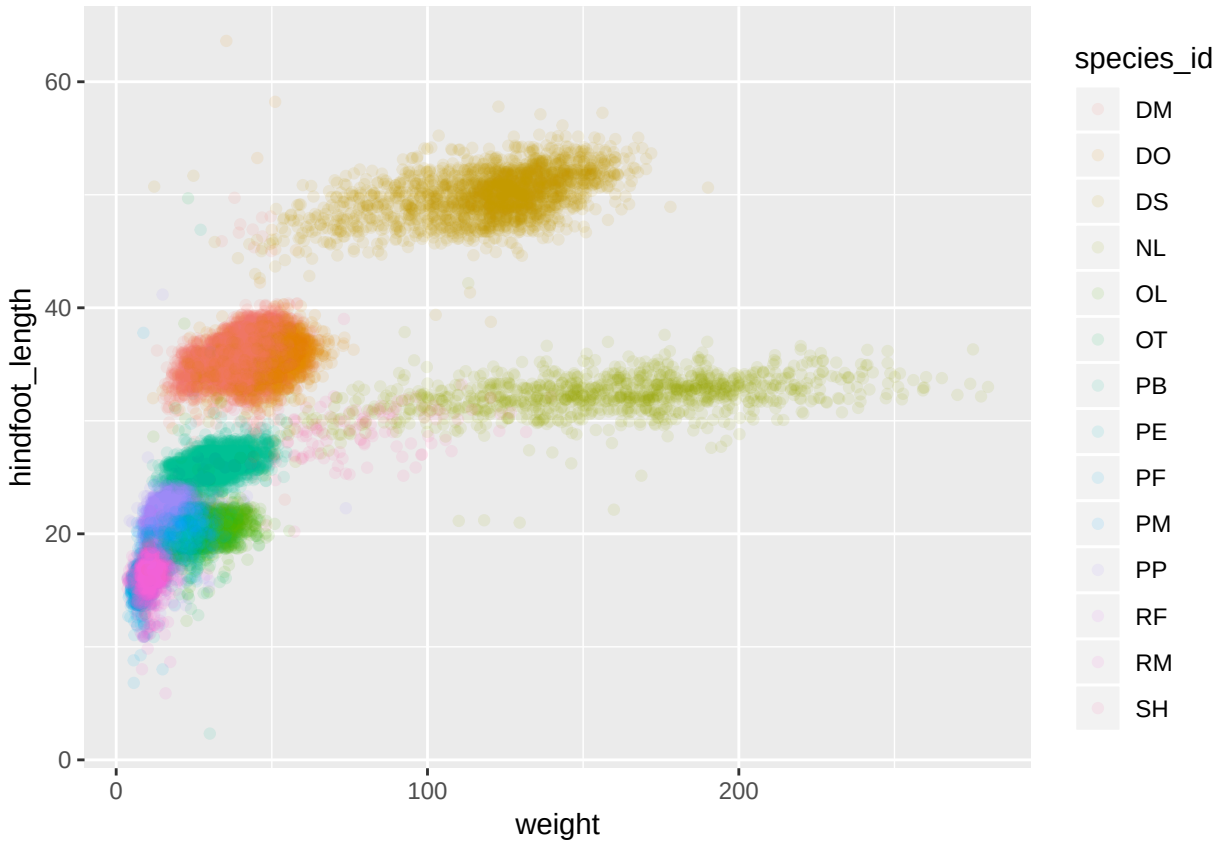
We can also specify the colors directly inside the mapping provided in the `ggplot()` function. This will be seen by any geom layers and the mapping will be determined by the x- and y-axis set up in `aes()`.

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length, color = species_id)) +
    geom_point(alpha = 0.1)
```



Notice that we can change the geom layer and colors will be still determined by **species_id**

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length, color = species_id)) +
    geom_jitter(alpha = 0.1)
```

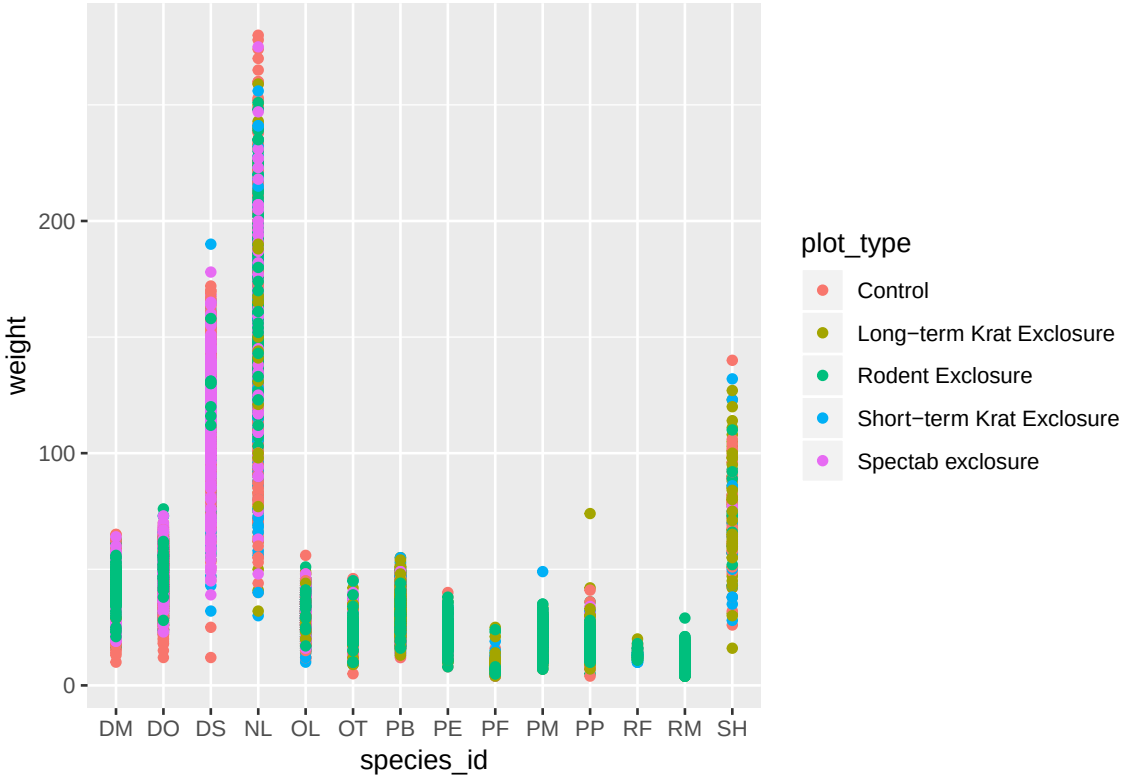


6.2.1 Challenge

Use what you just learned to create a scatter plot of `weight` over `species_id` with the plot types showing in different colors. Is this a good way to show this type of data?

Answer

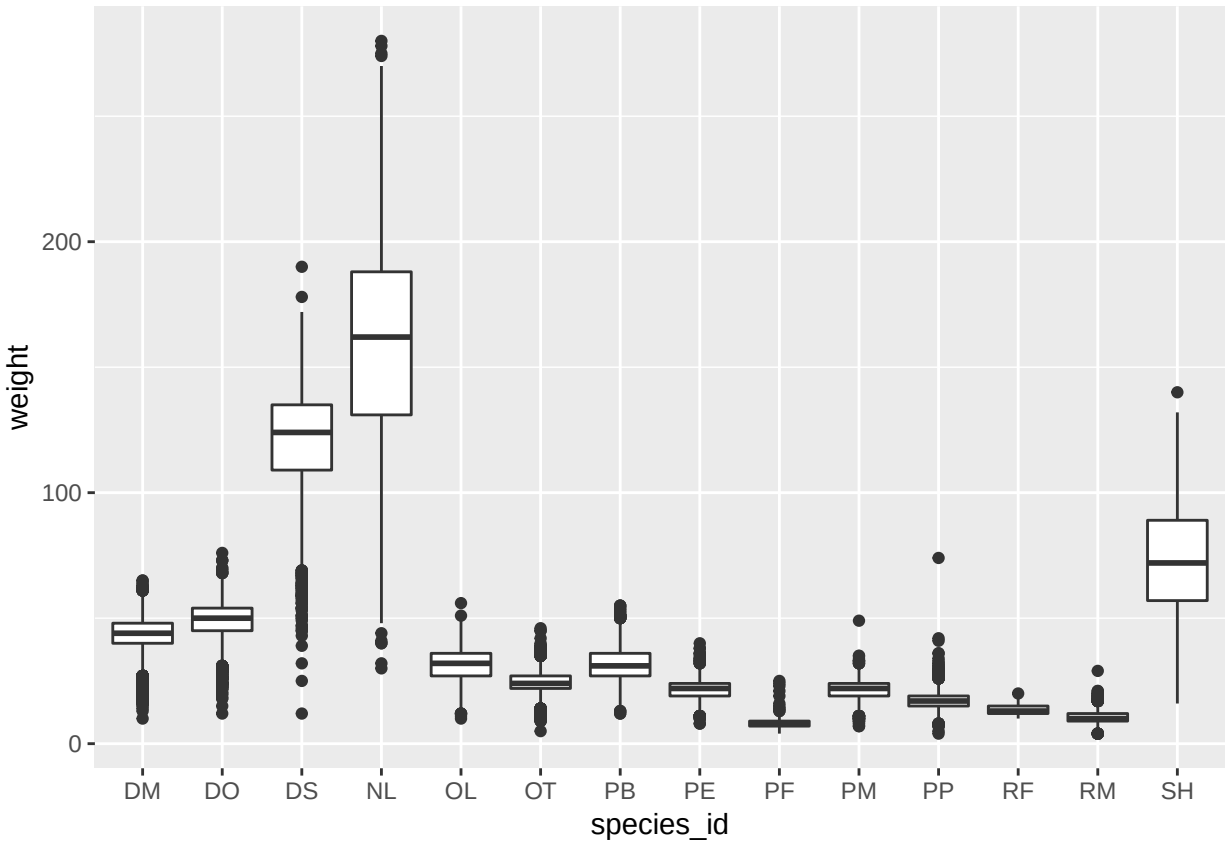
```
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
   geom_point(aes(color = plot_type))
```



6.3 Boxplot

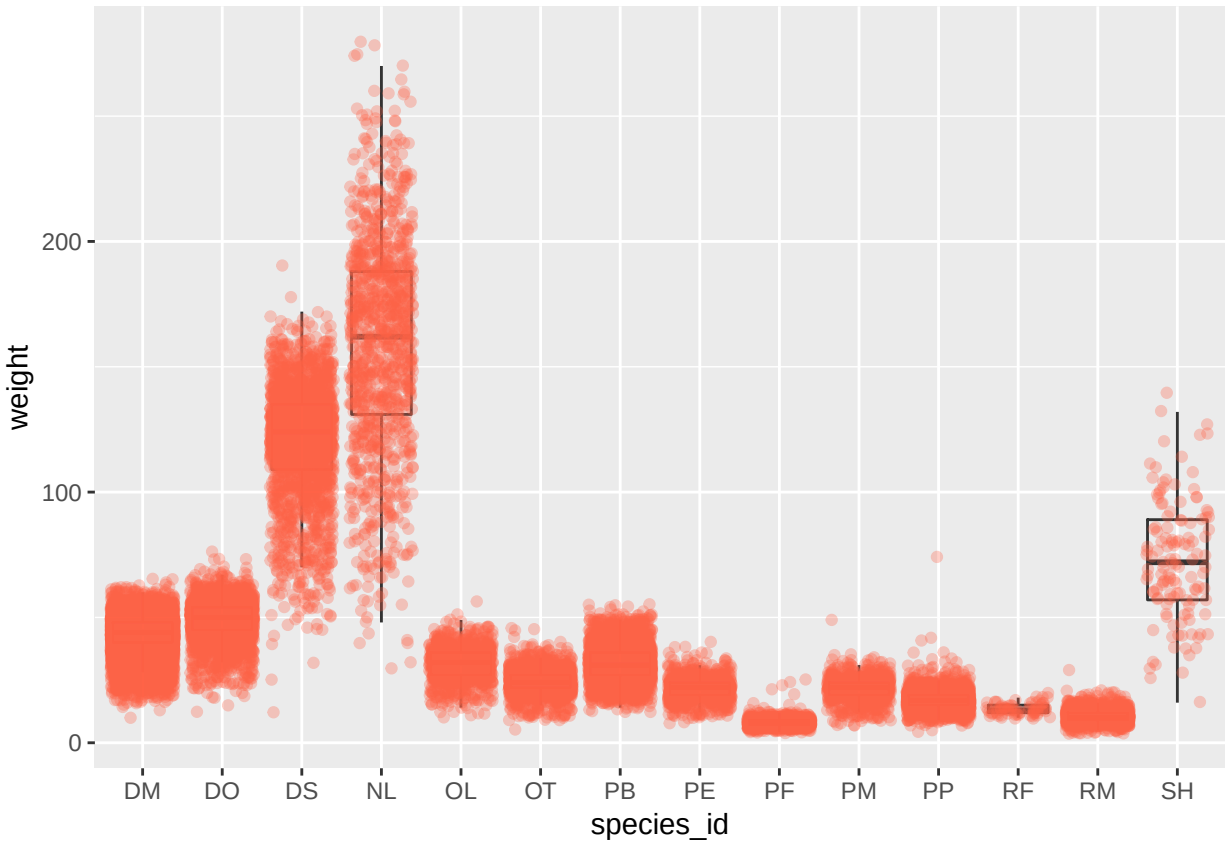
We can use boxplots to visualize the distribution of weight within each species:

```
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
    geom_boxplot()
```



By adding points to boxplot, we can have a better idea of the number of measurements and of their distribution:

```
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
    geom_boxplot(alpha = 0) +
    geom_jitter(alpha = 0.3, color = "tomato")
```



Notice how the boxplot layer is behind the jitter layer? What do you need to change in the code to put the boxplot in front of the points such that it's not hidden?

6.3.1 Challenges

Boxplots are useful summaries, but hide the *shape* of the distribution. For example, if the distribution is bimodal, we would not see it in a boxplot. An alternative to the boxplot is the violin plot, where the shape (of the density of points) is drawn.

- Replace the box plot with a violin plot; see `geom_violin()`.

In many types of data, it is important to consider the *scale* of the observations. For example, it may be worth changing the scale of the axis to better distribute the observations in the space of the plot. Changing the scale of the axes is done similarly to adding/modifying other components (i.e., by incrementally adding commands). Try making these modifications:

- Represent weight on the $\log_{10}$ scale; see `scale_y_log10()`.

So far, we've looked at the distribution of weight within species. Try making a new plot to explore the distribution of another variable within each species.

- Create a boxplot for `hindfoot_length`. Overlay the boxplot layer on a jitter layer to show actual measurements.
- Add color to the data points on your boxplot according to the plot from which the sample was taken (`plot_id`).

Hint: Check the class for `plot_id`. Consider changing the class of `plot_id` from integer to factor. Why does this change how R makes the graph?

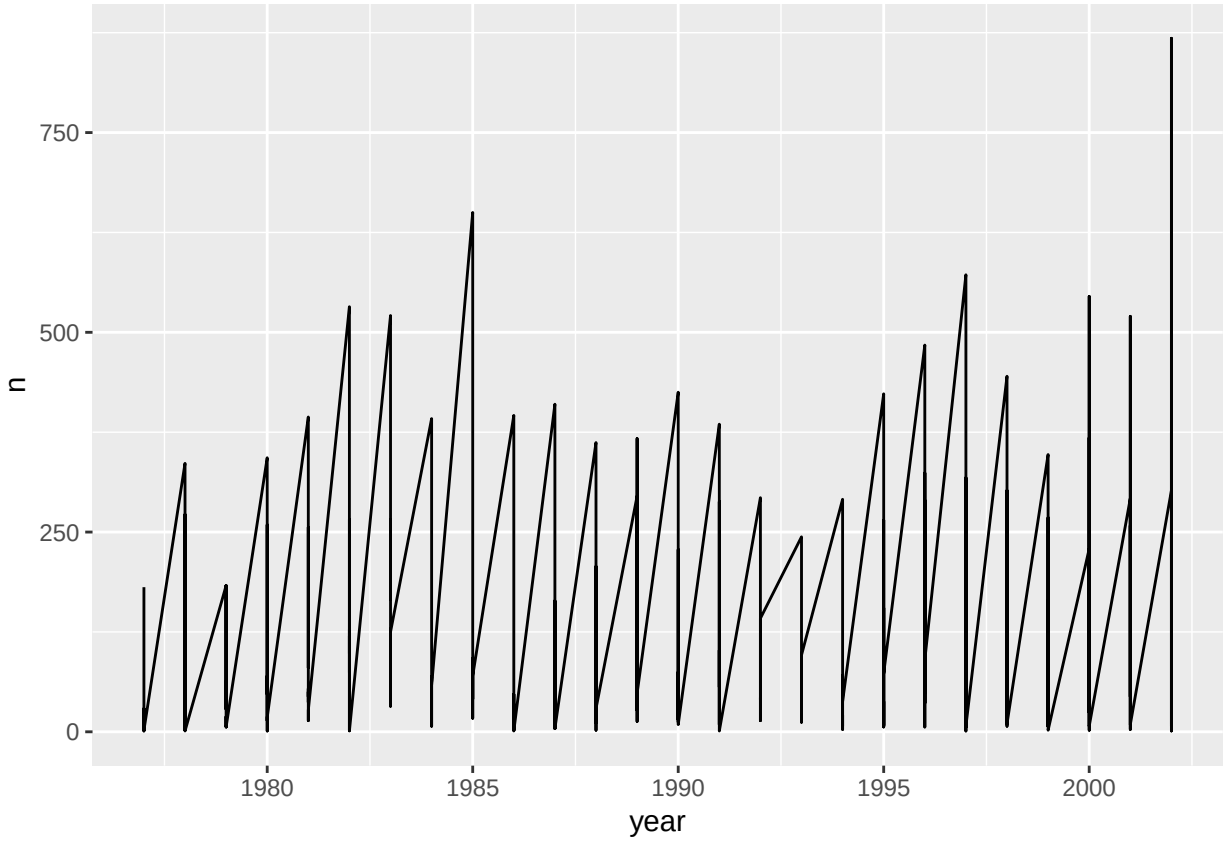
6.4 Plotting time series data

Let's calculate number of counts per year for each species. First we need to group the data and count records within each group:

```
yearly_counts <- surveys_complete %>%
                 count(year, species_id)
```

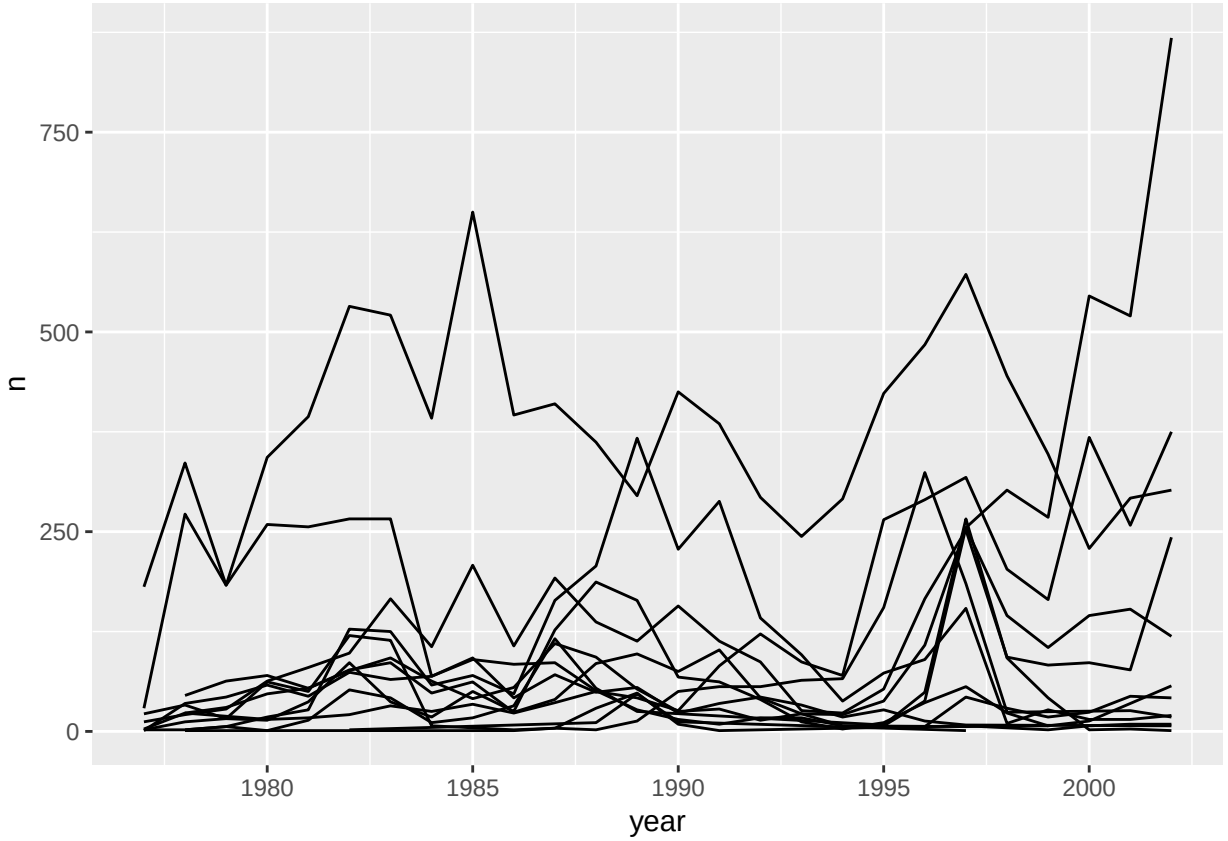
Time series data can be visualized as a line plot with years on the x axis and counts on the y axis:

```
ggplot(data = yearly_counts, mapping = aes(x = year, y = n)) +
     geom_line()
```



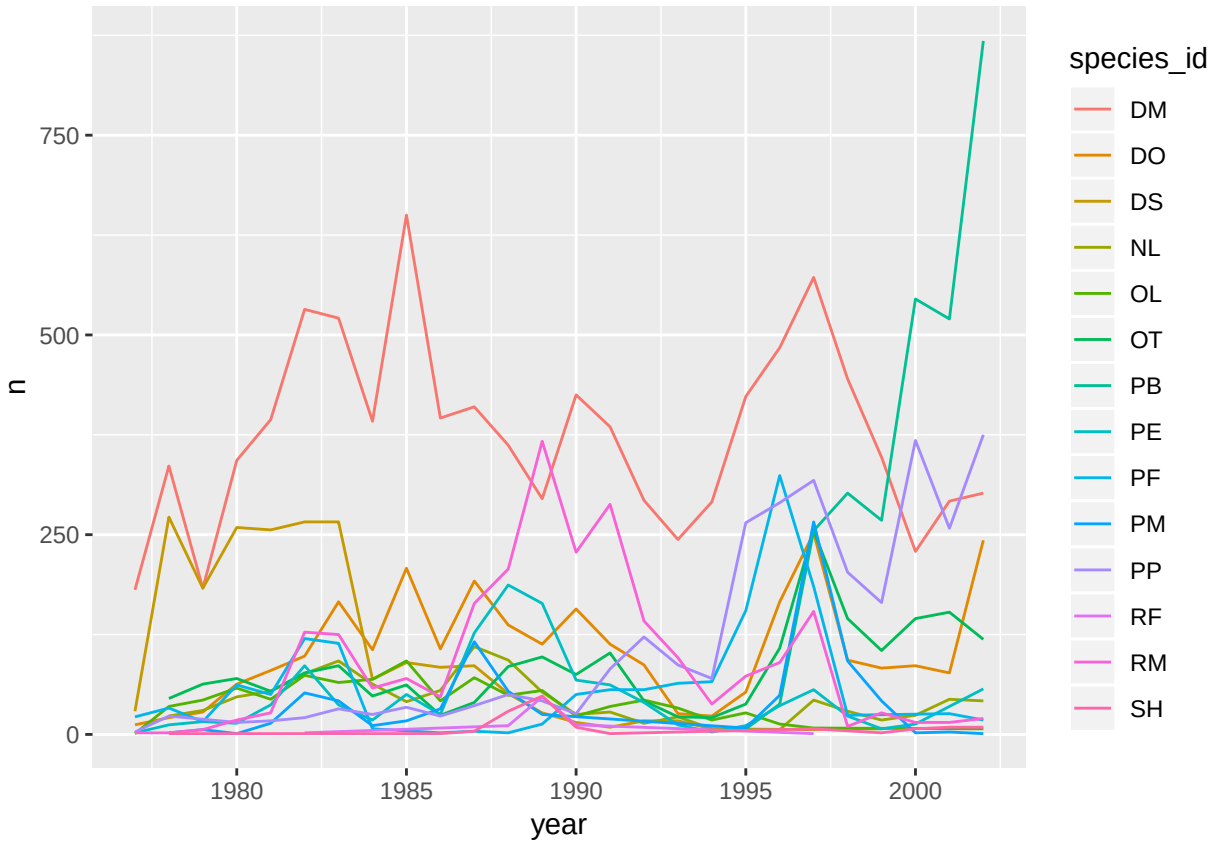
Unfortunately, this does not work because we plotted data for all the species together. We need to tell ggplot to draw a line for each species by modifying the aesthetic function to include `group = species_id`:

```
ggplot(data = yearly_counts, mapping = aes(x = year, y = n, group = species_id)) +
    geom_line()
```



We will be able to distinguish species in the plot if we add colors (using `color` also automatically groups the data):

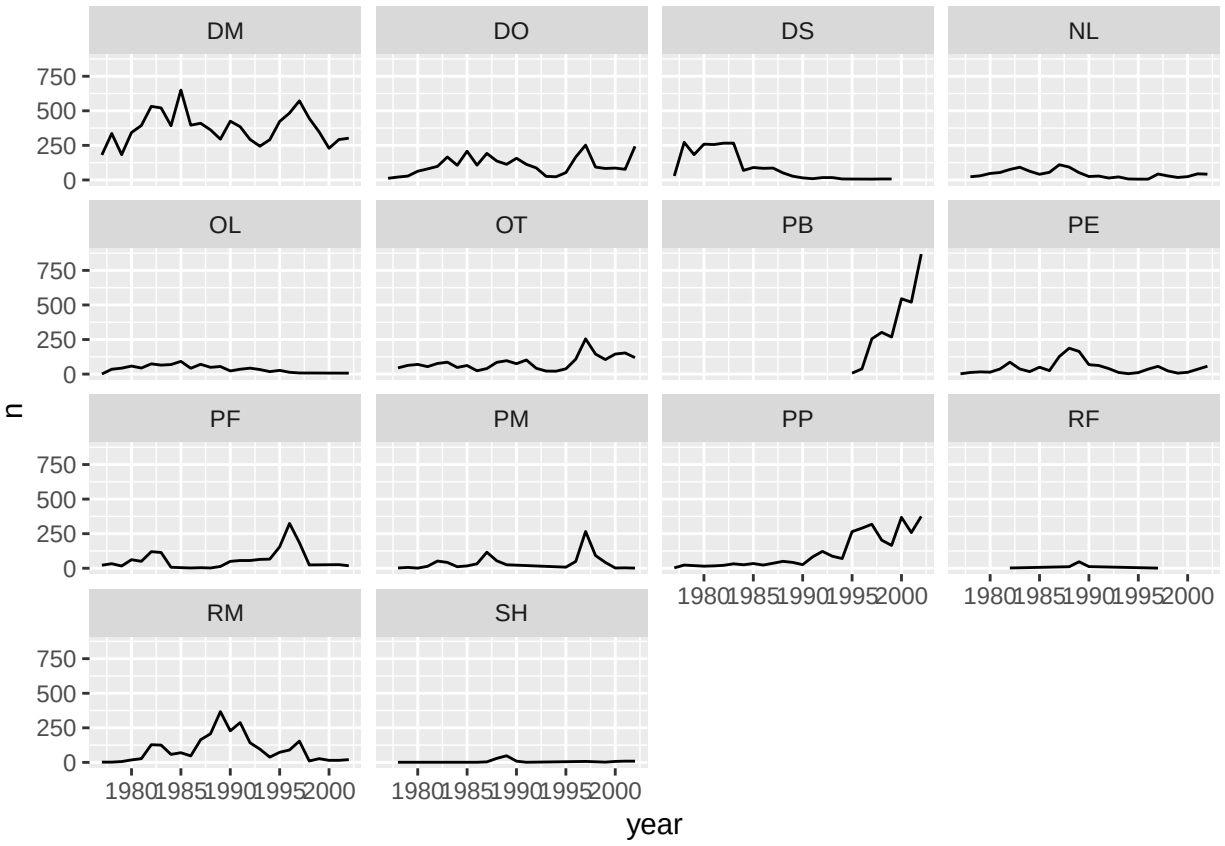
```
ggplot(data = yearly_counts, mapping = aes(x = year, y = n, color = species_id)) +
    geom_line()
```



6.5 Faceting

`ggplot2` has a special technique called *faceting* that allows the user to split one plot into multiple plots based on a factor included in the dataset. We will use it to make a time series plot for each species:

```
ggplot(data = yearly_counts, mapping = aes(x = year, y = n)) +
    geom_line() +
    facet_wrap(~ species_id)
```

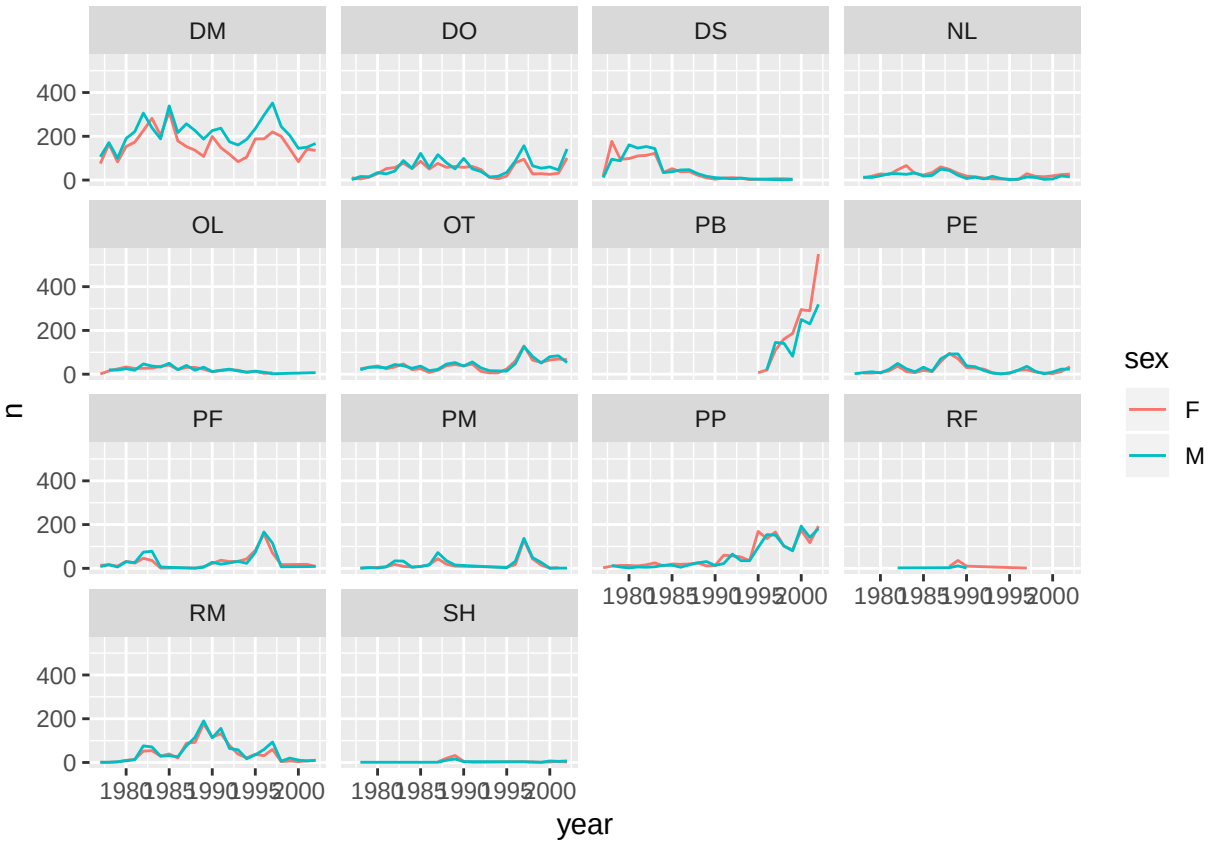


Now we would like to split the line in each plot by the sex of each individual measured. To do that we need to make counts in the data frame grouped by `year`, `species_id`, and `sex`:

```
yearly_sex_counts <- surveys_complete %>%
                     count(year, species_id, sex)
```

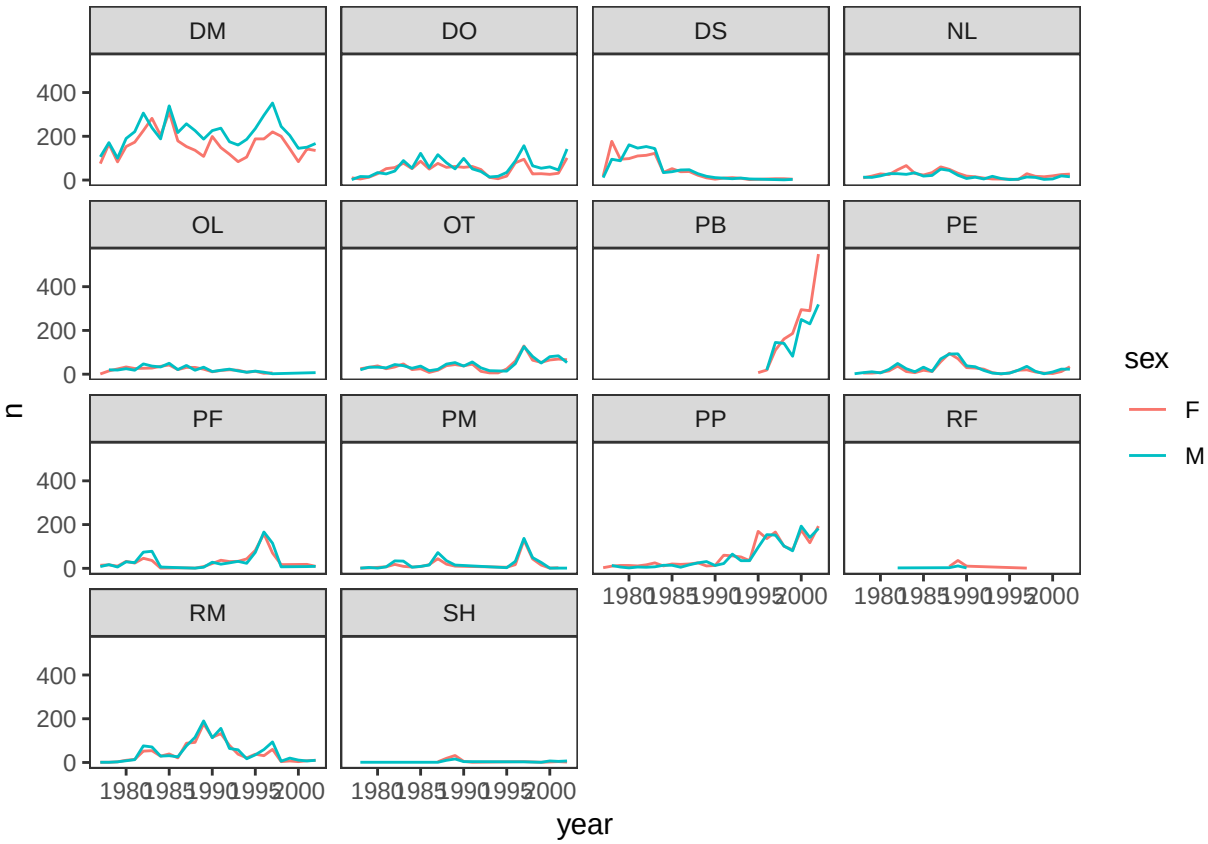
We can now make the faceted plot by splitting further by sex using `color` (within a single plot):

```
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(~ species_id)
```



Usually plots with white background look more readable when printed. We can set the background to white using the function `theme_bw()`. Additionally, you can remove the grid:

```
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(~ species_id) +
    theme_bw() +
    theme(panel.grid = element_blank())
```



6.6 ggplot2 themes

In addition to `theme_bw()`, which changes the plot background to white, **ggplot2** comes with several other themes which can be useful to quickly change the look of your visualization. The complete list of themes is available at http://docs.ggplot2.org/current/ggtheme.html. `theme_minimal()` and `theme_light()` are popular, and `theme_void()` can be useful as a starting point to create a new hand-crafted theme.

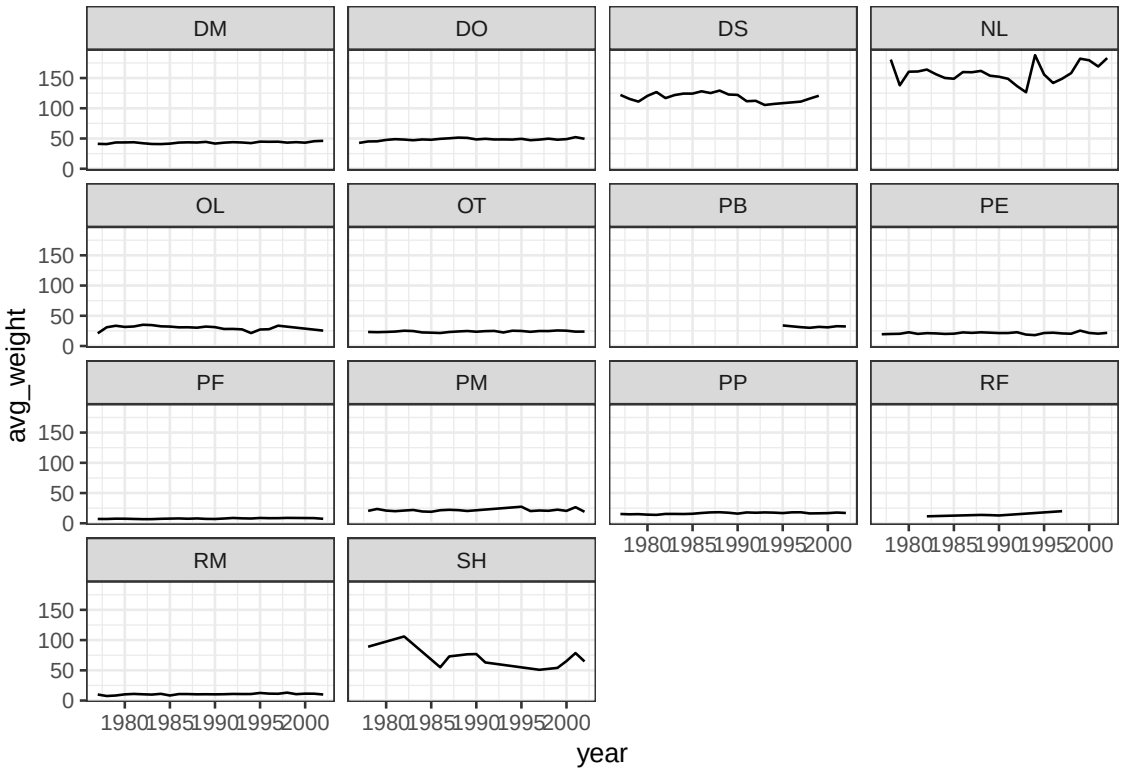
The ggthemes package provides a wide variety of options (including an Excel 2003 theme). The **ggplot2** extensions website provides a list of packages that extend the capabilities of **ggplot2**, including additional themes.

6.6.1 Challenge

Use what you just learned to create a plot that depicts how the average weight of each species changes through the years.

Answer

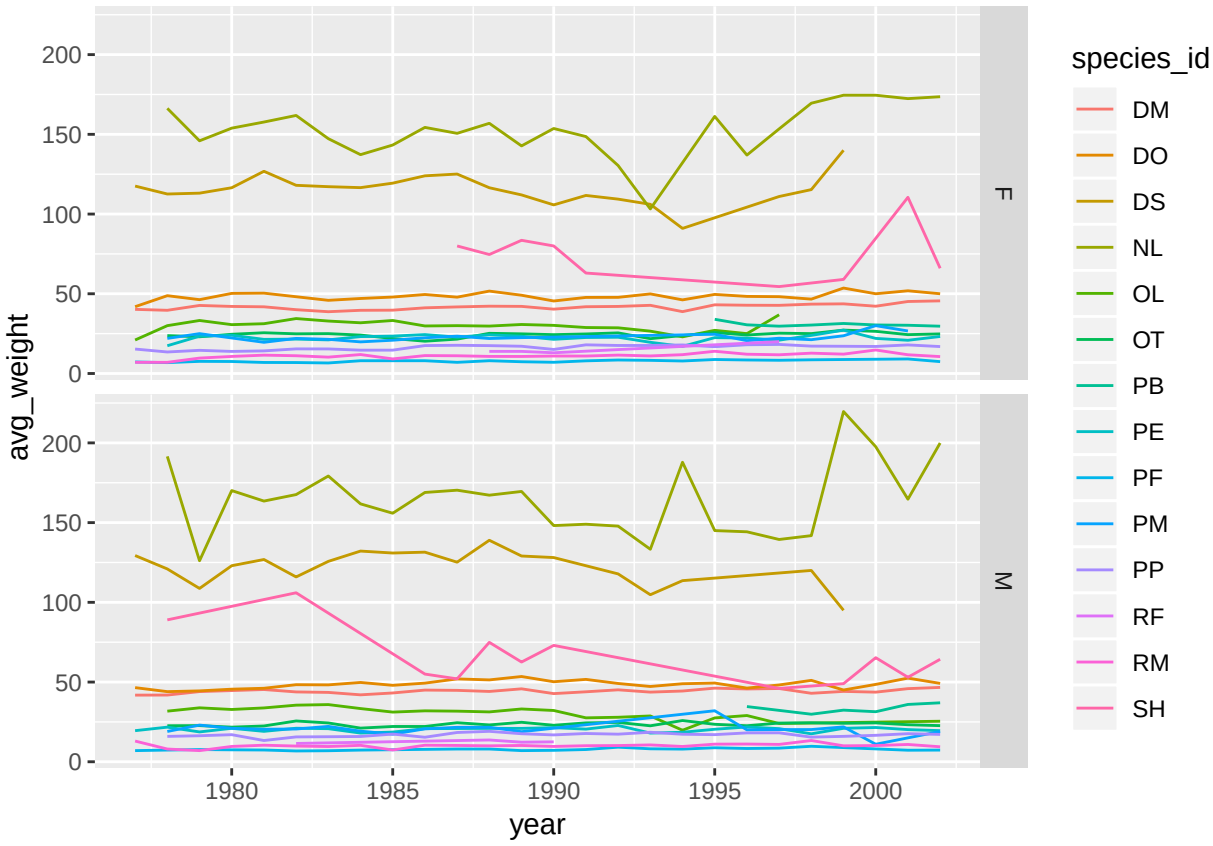
```
yearly_weight <- surveys_complete %>%
                group_by(year, species_id) %>%
                 summarize(avg_weight = mean(weight))
ggplot(data = yearly_weight, mapping = aes(x=year, y=avg_weight)) +
   geom_line() +
   facet_wrap(~ species_id) +
   theme_bw()
```



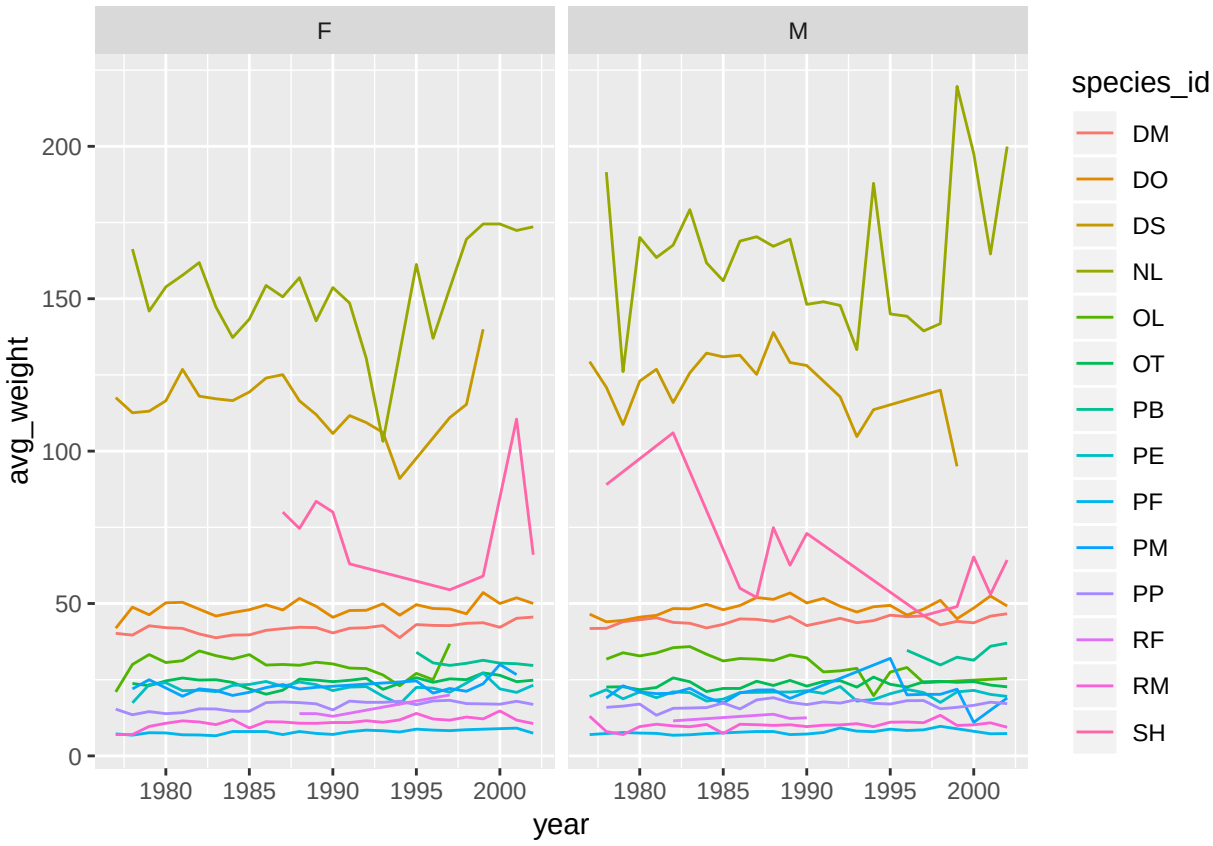
The `facet_wrap` geometry extracts plots into an arbitrary number of dimensions to allow them to cleanly fit on one page. On the other hand, the `facet_grid` geometry allows you to explicitly specify how you want your plots to be arranged via formula notation (`rows ~ columns`; a `.` can be used as a placeholder that indicates only one row or column).

Let's modify the previous plot to compare how the weights of males and females has changed through time:

```
# One column, facet by rows
yearly_sex_weight <- surveys_complete %>%
    group_by(year, sex, species_id) %>%
    summarize(avg_weight = mean(weight))
ggplot(data = yearly_sex_weight,
       mapping = aes(x = year, y = avg_weight, color = species_id)) +
    geom_line() +
    facet_grid(sex ~ .)
```




```
# One row, facet by column
ggplot(data = yearly_sex_weight,
       mapping = aes(x = year, y = avg_weight, color = species_id)) +
    geom_line() +
    facet_grid(. ~ sex)
```

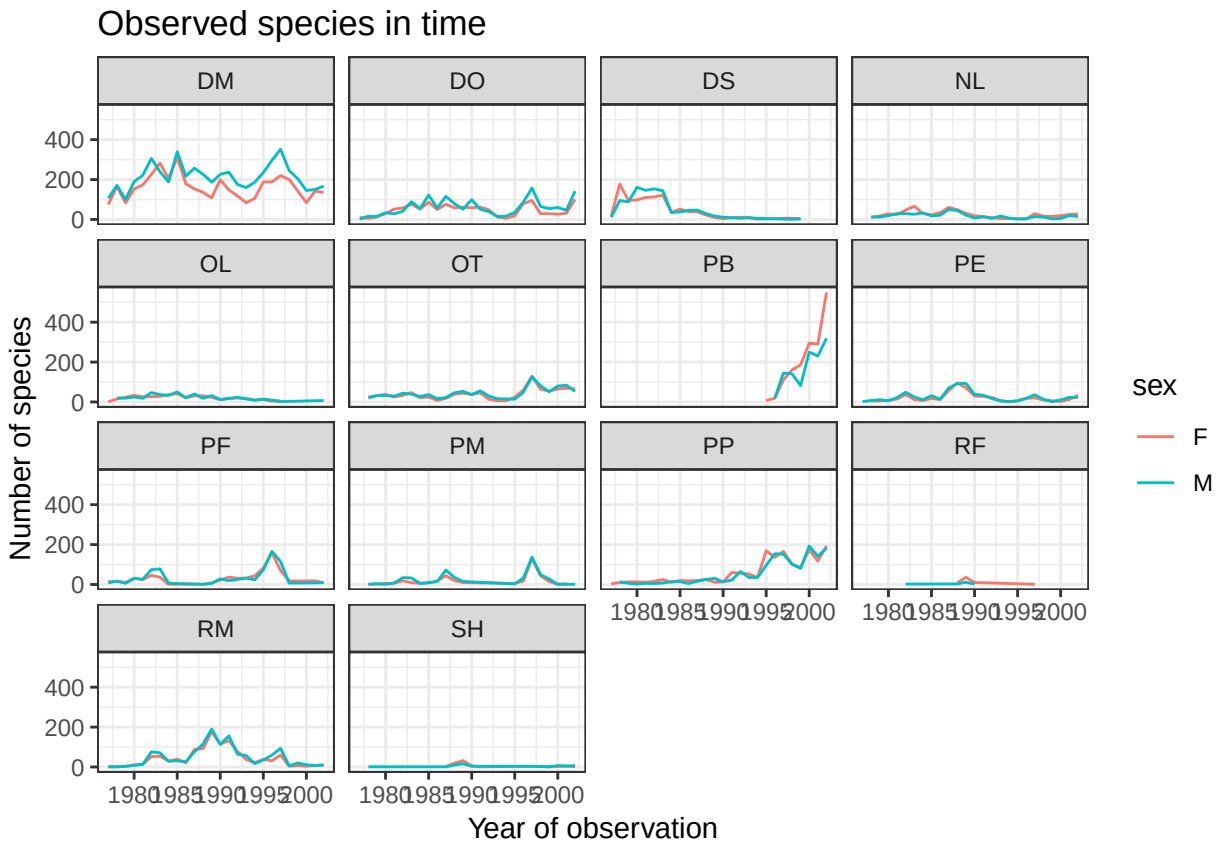


6.7 Customization

Take a look at the **ggplot2** cheat sheet, and think of ways you could improve the plot.

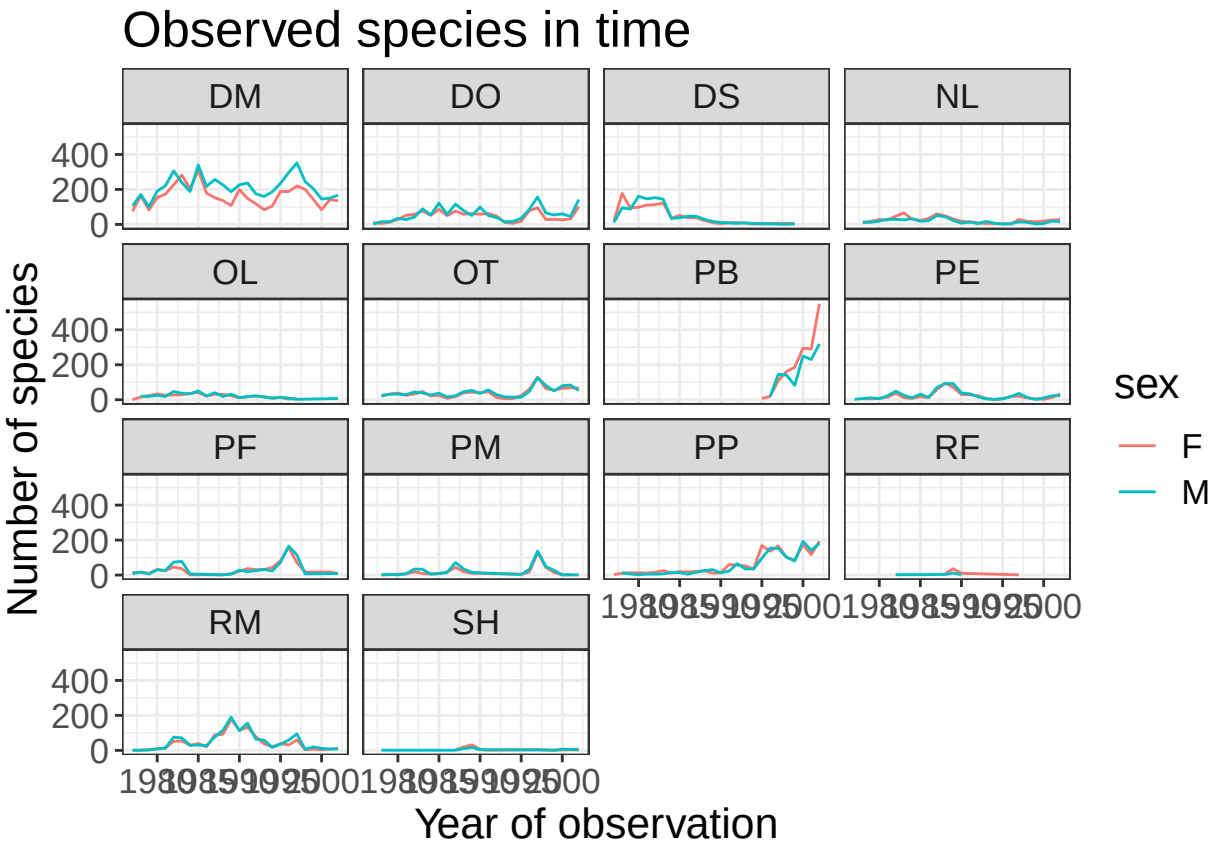
Now, let's change names of axes to something more informative than 'year' and 'n' and add a title to the figure:

```
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(~ species_id) +
    labs(title = "Observed species in time",
         x = "Year of observation",
         y = "Number of species") +
    theme_bw()
```



The axes have more informative names, but their readability can be improved by increasing the font size:

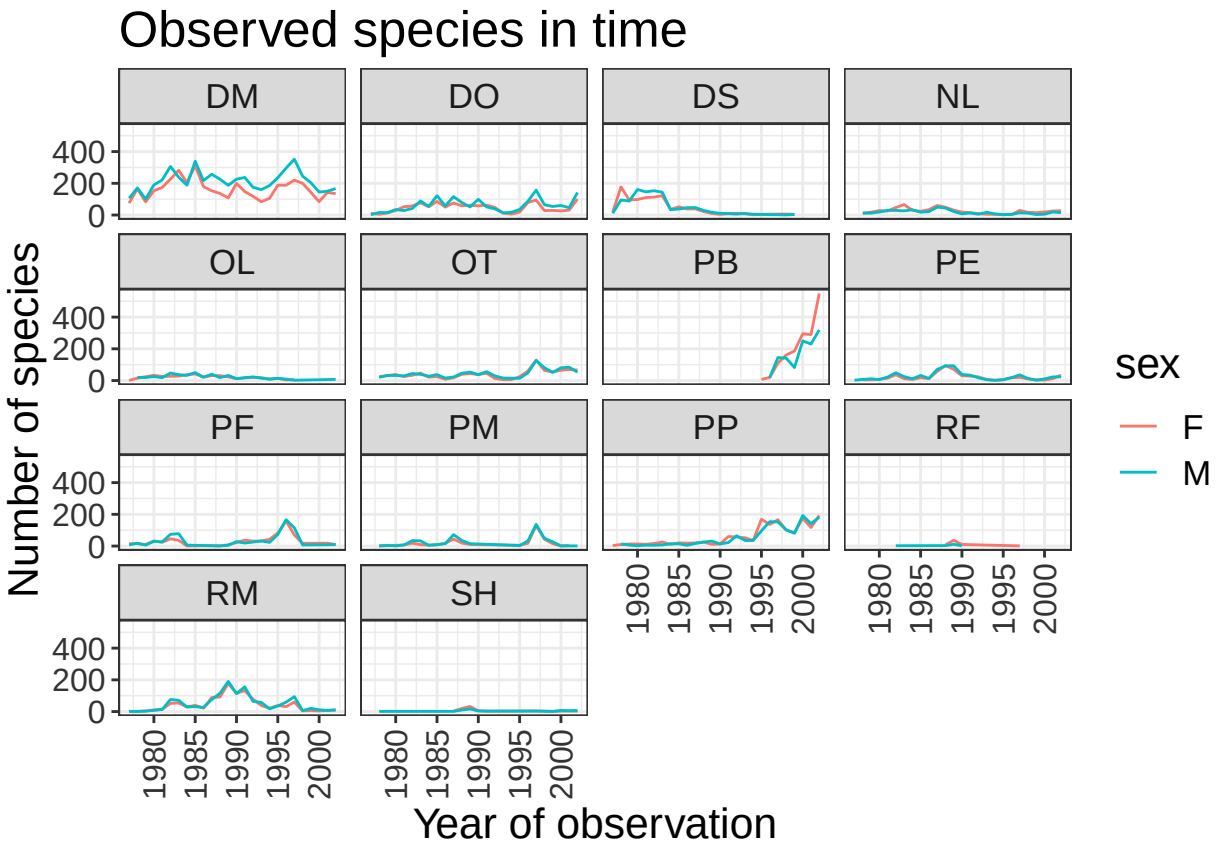
```
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(~ species_id) +
    labs(title = "Observed species in time",
        x = "Year of observation",
        y = "Number of species") +
    theme_bw() +
    theme(text=element_text(size = 16))
```



Note that it is also possible to change the fonts of your plots. If you are on Windows, you may have to install the **extrafont** package, and follow the instructions included in the README for this package.

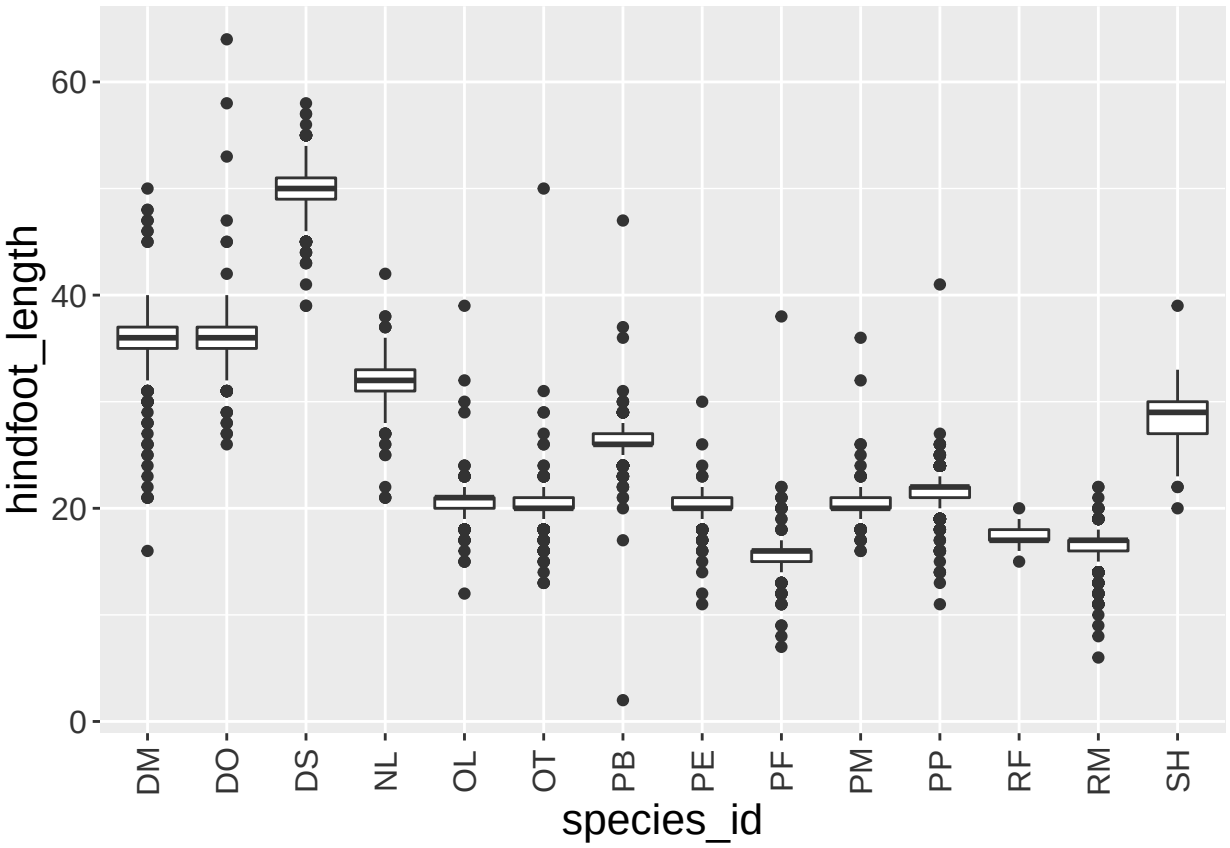
After our manipulations, you may notice that the values on the x-axis are still not properly readable. Let's change the orientation of the labels and adjust them vertically and horizontally so they don't overlap. You can use a 90-degree angle, or experiment to find the appropriate angle for diagonally oriented labels:

```
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(~ species_id) +
    labs(title = "Observed species in time",
        x = "Year of observation",
        y = "Number of species") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 90, hjust = 0.5, vjust = 0.5)
                      axis.text.y = element_text(colour = "grey20", size = 12),
          text = element_text(size = 16))
```



If you like the changes you created better than the default theme, you can save them as an object to be able to easily apply them to other plots you may create:

```
grey_theme <- theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 90, hjust = 0.5, v
                           axis.text.y = element_text(colour = "grey20", size = 12),
                           text = element_text(size = 16))
ggplot(surveys_complete, aes(x = species_id, y = hindfoot_length)) +
    geom_boxplot() +
    grey_theme
```



6.7.1 Challenge

With all of this information in hand, please take another five minutes to either improve one of the plots generated in this exercise or create a beautiful graph of your own. Use the RStudio `ggplot2` cheat sheet for inspiration. Here are some ideas:

- See if you can change the thickness of the lines.
- Can you find a way to change the name of the legend? What about its labels?
- Try using a different color palette (see http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/).

6.8 Arranging and exporting plots

Faceting is a great tool for splitting one plot into multiple plots, but sometimes you may want to produce a single figure that contains multiple plots using different variables or even different data frames. The `gridExtra` package allows us to combine separate ggplots into a single figure using `grid.arrange()`:

```
install.packages("gridExtra")
```

```
library(gridExtra)

spp_weight_boxplot <- ggplot(data = surveys_complete,
                             mapping = aes(x = species_id, y = weight)) +
  geom_boxplot() +
  xlab("Species") + ylab("Weight (g)") +
```

```
  scale_y_log10()

spp_count_plot <- ggplot(data = yearly_counts,
                         mapping = aes(x = year, y = n, color = species_id)) +
  geom_line() +
  xlab("Year") + ylab("Abundance")

grid.arrange(spp_weight_boxplot, spp_count_plot, ncol = 2, widths = c(4, 6))
```

In addition to the `ncol` and `nrow` arguments, used to make simple arrangements, there are tools for constructing more complex layouts.

After creating your plot, you can save it to a file in your favorite format. The Export tab in the **Plot** pane in RStudio will save your plots at low resolution, which will not be accepted by many journals and will not scale well for posters.

Instead, use the `ggsave()` function, which allows you easily change the dimension and resolution of your plot by adjusting the appropriate arguments (`width`, `height` and `dpi`).

Make sure you have the `fig_output/` folder in your working directory.

```
my_plot <- ggplot(data = yearly_sex_counts,
                  mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(~ species_id) +
    labs(title = "Observed species in time",
        x = "Year of observation",
        y = "Number of species") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 90, hjust = 0.5, vjust = 0.5
                      axis.text.y = element_text(colour = "grey20", size = 12),
          text=element_text(size = 16))
ggsave("fig_output/yearly_sex_counts.png", my_plot, width = 15, height = 10)

# This also works for grid.arrange() plots
combo_plot <- grid.arrange(spp_weight_boxplot, spp_count_plot, ncol = 2, widths = c(4, 6))
ggsave("fig_output/combo_plot_abun_weight.png", combo_plot, width = 10, dpi = 300)
```

Note: The parameters `width` and `height` also determine the font size in the saved plot.

Chapter 7

Hakai Data Portal API

Often your data source changes over time as new data are added, or as errors are corrected. It can be a pain to go somewhere to re-download a data file, put the data in the right place, and then re-run your code.

It is possible to download data from the Hakai EIMS Data Portal database directly from R Studio. This is accomplished by interacting with an application programming interface (API) that was developed for downloading data from Hakai's data portal.

Below is a quickstart example of how you can download some chlorophyll data. Run the code below one line at a time. When you run the `client <- ...` line a web URL will be displayed in the console. Copy and paste that URL into your browser. This should take to you a webpage that displays another web URL, this is your authentication token that permits you access to the database. Copy and paste the URL into the console in R where it tells you to do so.

```
# Run this first line only if you haven't installedt the R API before
devtools::install_github("HakaiInstitute/hakai-api-client-r", subdir='hakaiApi')

library('hakaiApi')

# Run this line independently before the rest of the code to get the API authentication
client <- hakaiApi::Client$new() # Follow stdout prompts to get an API token

# Make a data request for chlorophyll data
endpoint <- sprintf("%s/%s", client$api_root, "eims/views/output/chlorophyll?limit=50")
data <- client$get(endpoint)

# Print out the data
print(data)
```

By running this code you should see chlorophyll data in your environment. The above code can be modified to select different datasets other than chlorophyll and filter based on different logical parameters you set. This is accomplished by editing the text after the ? in `"eims/views/output/chlorophyll?limit=50"`.

The formula you set after the question mark is known as query string filtering. To learn how to filter your data read this.

To read generally about the API and how to use it for your first time go here.

If you don't want to learn how to write a querystring yourself there is an option to just copy and paste the querystring from the EIMS Data Portal. Use the portal to select the sample type, and dates and sites you'd like to download as you normally would. To copy the querystring go to the top right of the window where it says Options and click 'Display API query'. You can copy that string in to your endpoint definition in R. Just

be sure to copy that string starting from `eims/views/...`, excluding `https://hecate.hakai.org/api/` and then paste that into the definitions of your endpoint and surround that string with single quotes, ie:

```
endpoint <- sprintf("%s/%s", client$api_root, 'eims/views/output/chlorophyll?date>=2016-11-01&date<2018-1
```

Make sure to add &limit=-1 at the end of your query string so that not only the first 20 results are downloaded, but rather everything matching your query string is downloaded.

The page documenting the API usage can be found here