Before we start

Data Carpentry contributors

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.

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 (https://rstudio.com) 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.

Why learn R?

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.

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.

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.

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.

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.

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 (https://stackoverflow.com/), or on the RStudio community (https://community.rstudio.com/).

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

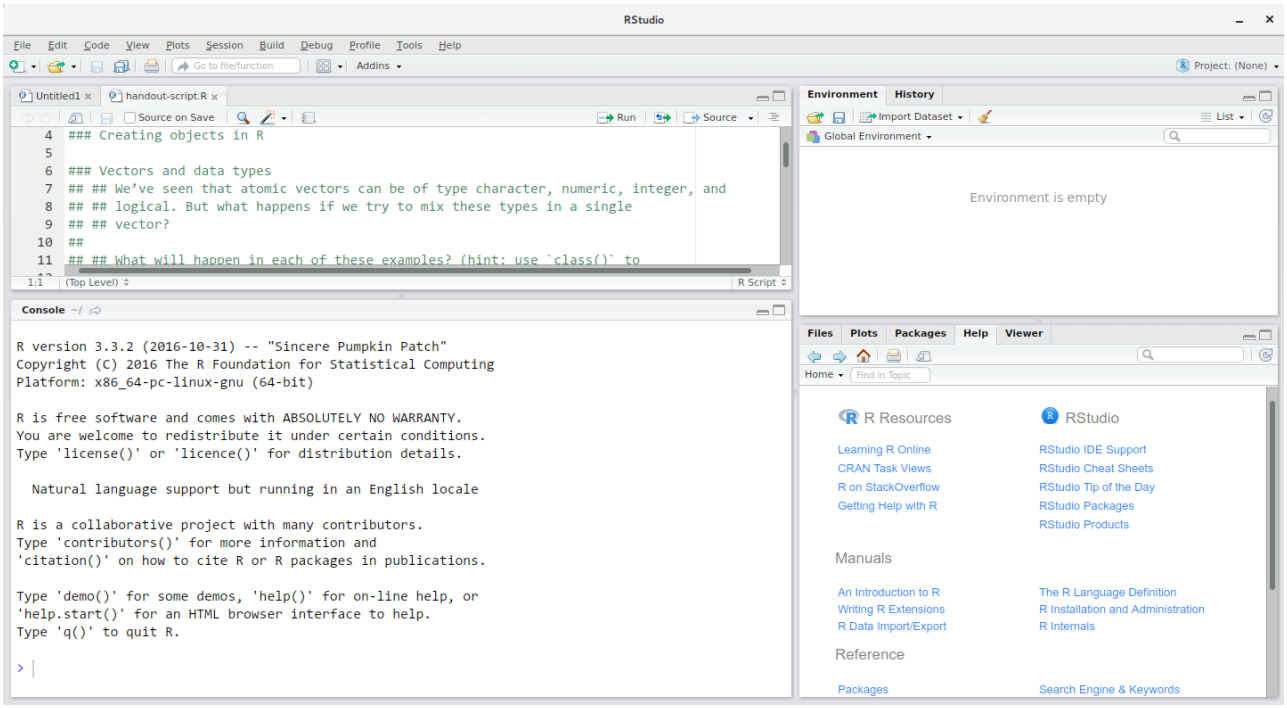
Anyone can inspect the source code to see how R works. Because of this transparency, there is less chance for mistakes, and if you (or someone else) find some, you can report and fix bugs.

Knowing your way around RStudio

Let's start by learning about RStudio (https://www.rstudio.com/), 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 (https://www.gnu.org/licenses/agpl-3.0.en.html). The RStudio IDE is also available with a commercial license and priority email support from RStudio, Inc.

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

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.

Getting set up

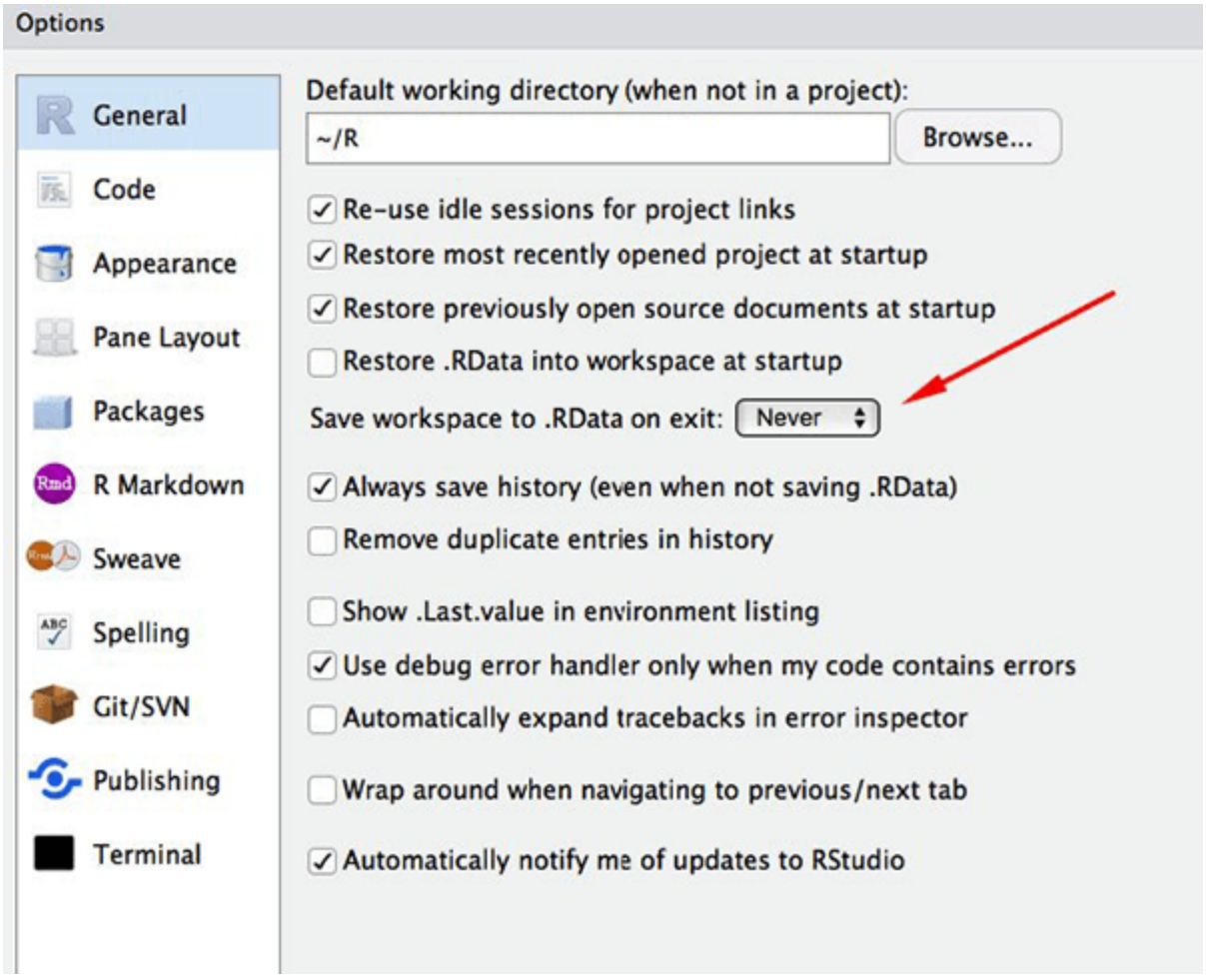
It is good practice to keep a set of related data, analyses, and text self-contained in a single folder, called the **working directory**. All of the scripts within this folder can then use *relative paths* to files that indicate where inside the project a file is located (as opposed to absolute paths, 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`.
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., `~/data-carpentry`).
4. Click on `Create Project`.
5. Download the code handout (./code-handout.R), place it in your working directory and rename it (e.g., `data-carpentry-script.R`).
6. (Optional) 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.’



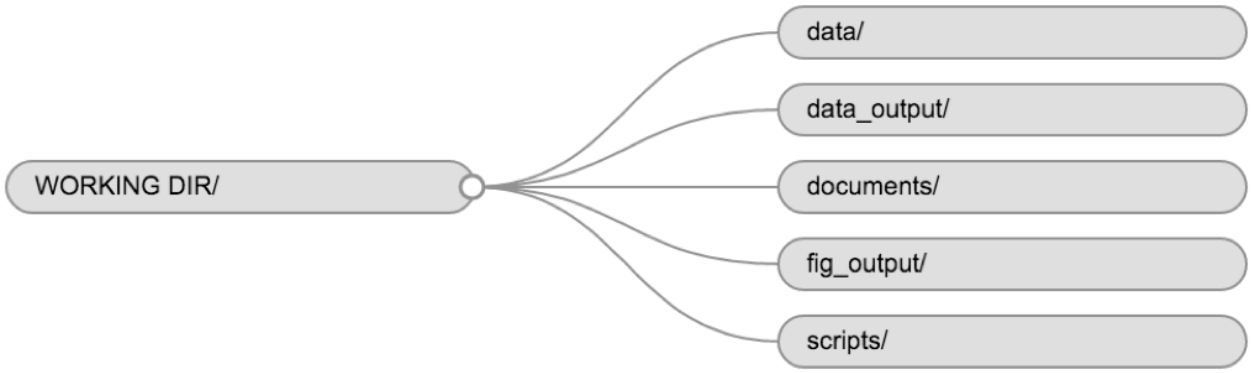
Set ‘Save workspace to .RData on exit’ to ‘Never’

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 (https://en.wikipedia.org/wiki/Provenance), you should *always* keep a 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.



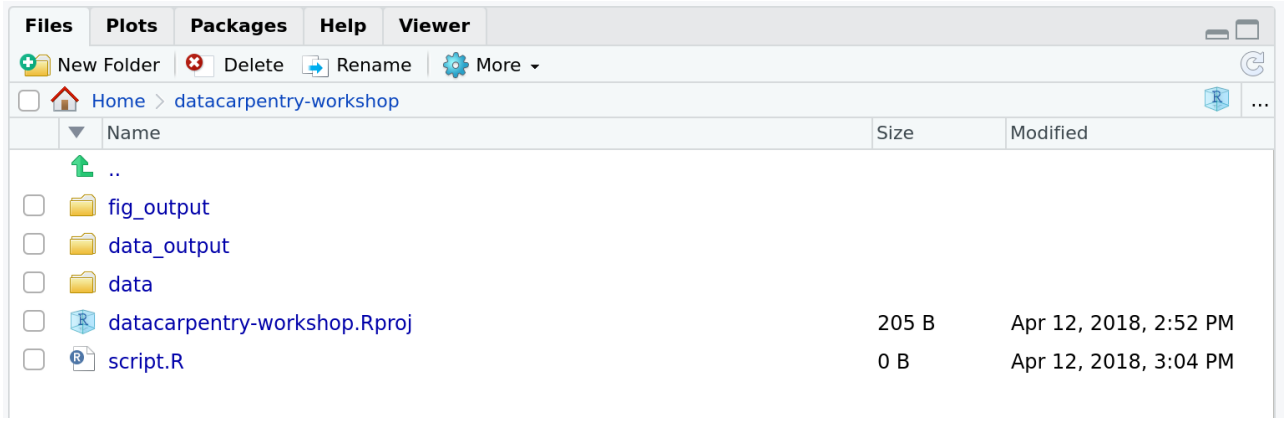
Example of a working directory structure.

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:



How it should look like at the beginning of this lesson

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.

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 (https://github.com/rstudio/cheatsheets/raw/master/rstudio-ide.pdf).

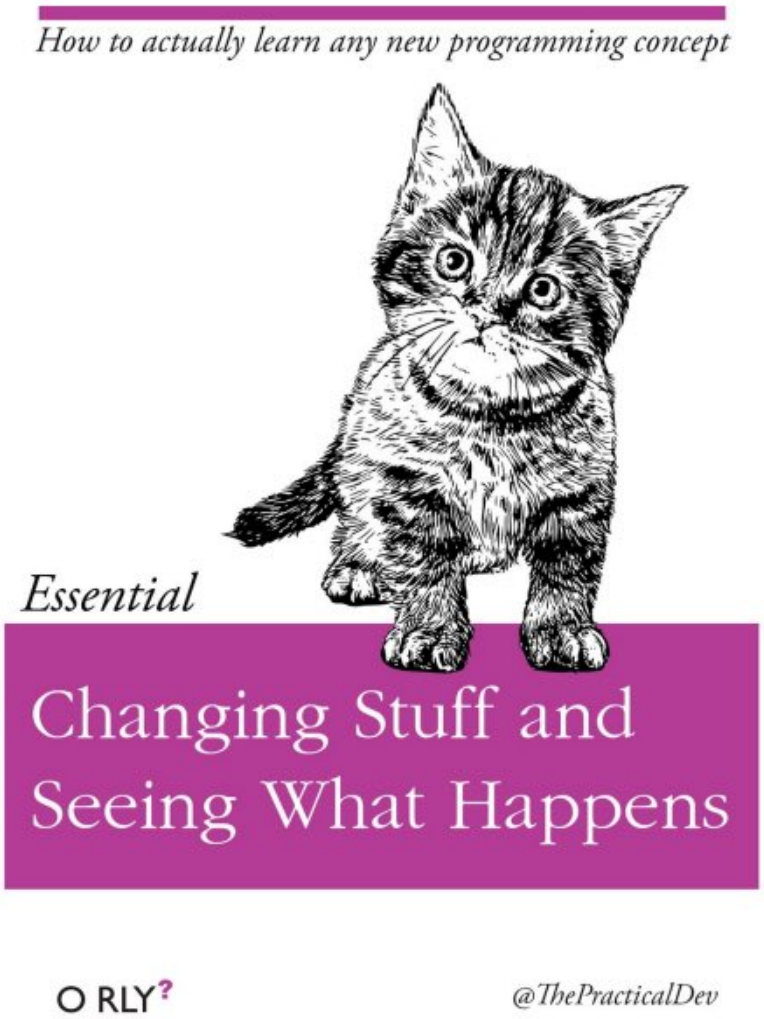
At some point in your analysis you may want to check the content of a variable or the structure of an object, without necessarily keeping a record of it in your script. You can type these commands and execute them directly in the console. RStudio provides the `Ctrl + 1` and `Ctrl + 2` shortcuts allow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

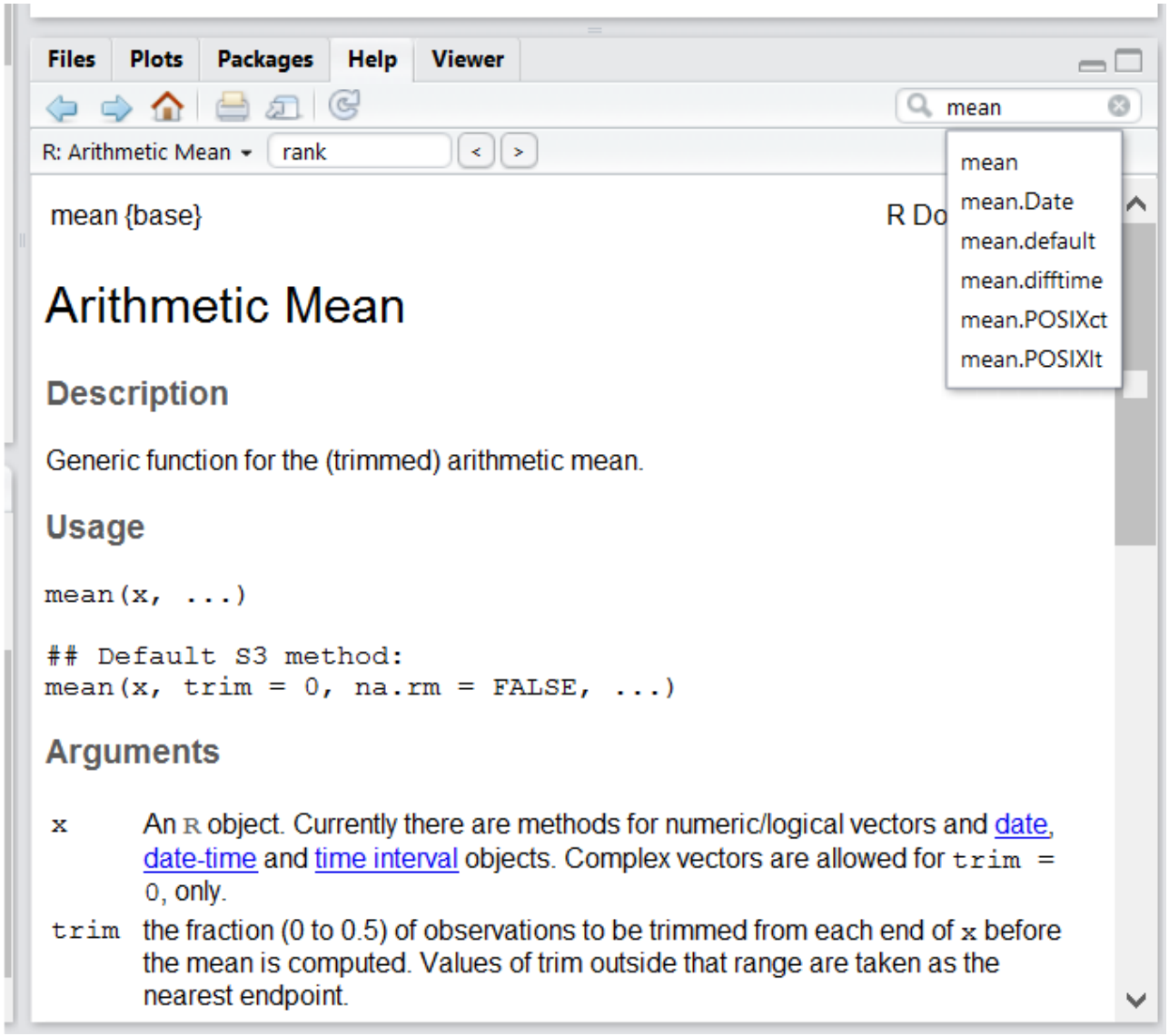
How to learn more after the workshop?

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.



Seeking help

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



RStudio help interface.

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.

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)
```

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 (http://www.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.

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 (http://stackoverflow.com/questions/tagged/r)

The Introduction to R (http://cran.r-project.org/doc/manuals/R-intro.pdf) can also be dense for people with little programming experience but it is a good place to understand the underpinnings of the R language.

The R FAQ (http://cran.r-project.org/doc/FAQ/R-FAQ.html) is dense and technical but it is full of useful information.

Asking for help

The key to receiving help from someone is for them to rapidly grasp your problem. You should make it as easy as possible to pinpoint where the issue might be.

Try to use the correct words to describe your problem. For instance, a package is not the same thing as a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. Be as precise as possible when describing your problem.

If possible, try to reduce what doesn't work to a simple *reproducible example*. If you can reproduce the problem using a very small data frame instead of your 50,000 rows and 10,000 columns one, provide the small one with the description of your problem. When appropriate, try to generalize what you are doing so even people who are not in your field can understand the question. For instance instead of using a subset of your real dataset, create a small (3 columns, 5 rows) generic one. For more information on how to write a reproducible example see this article by Hadley Wickham (http://adv-r.had.co.nz/Reproducibility.html).

To share an object with someone else, if it's relatively small, you can use the function `dput()`. It will output R code that can be used to recreate the exact same object as the one in memory:

```
dput(head(iris)) # iris is an example data frame that comes with R a
  nd head() is a function that returns the first part of the data fr
  ame
```

```
#> structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4),
#>     Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6, 3.9), Petal.Length = c
(1.4,
#>     1.4, 1.3, 1.5, 1.4, 1.7), Petal.Width = c(0.2, 0.2, 0.2,
#>     0.2, 0.2, 0.4), Species = structure(c(1L, 1L, 1L, 1L, 1L,
#>     1L), .Label = c("setosa", "versicolor", "virginica"), class =
"factor")), row.names = c(NA,
#> 6L), class = "data.frame")
```

If the object is larger, provide either the raw file (i.e., your CSV file) with your script up to the point of the error (and after removing everything that is not relevant to your issue). Alternatively, in particular if your question is not related to a data frame, you can save any R object to a file:

```
saveRDS(iris, file="/tmp/iris.rds")
```

The content of this file is however not human readable and cannot be posted directly on Stack Overflow. Instead, it can be sent to someone by email who can read it with the `readRDS()` command (here it is assumed that the downloaded file is in a `Downloads` folder in the user's home directory):

```
some_data <- readRDS(file="~/Downloads/iris.rds")
```

Last, but certainly not least, **always include the output of `sessionInfo()`** as it provides critical information about your platform, the versions of R and the packages that you are using, and other information that can be very helpful to understand your problem.

```
sessionInfo()
```

```
#> R version 3.5.2 (2017-01-27)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 14.04.5 LTS
#>
#> Matrix products: default
#> BLAS: /home/travis/R-bin/lib/R/lib/libRblas.so
#> LAPACK: /home/travis/R-bin/lib/R/lib/libRlapack.so
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
#>  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
#>  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
#>  [9] LC_ADDRESS=C               LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   b
ase
#>
#> other attached packages:
#>  [1] forcats_0.3.0   stringr_1.3.1   dplyr_0.7.8     purrr_0.3.0
#>  [5] readr_1.3.1     tidyr_0.8.2     tibble_2.0.1    ggplot2_3.1.
0
#>  [9] tidyverse_1.2.1 knitr_1.21
#>
#> loaded via a namespace (and not attached):
#>  [1] Rcpp_1.0.0       highr_0.7        cellranger_1.1.0 plyr_1.8.
4
#>  [5] pillar_1.3.1     compiler_3.5.2   bindr_0.1.1      tools_3.
5.2
#>  [9] digest_0.6.18    lubridate_1.7.4  jsonlite_1.6     evaluate_
0.12
#> [13] nlme_3.1-137     gtable_0.2.0     lattice_0.20-38  pkgconfig
_2.0.2
#> [17] rlang_0.3.1      cli_1.0.1        rstudioapi_0.9.0 yaml_2.2.
0
#> [21] haven_2.0.0      xfun_0.4         bindrcpp_0.2.2   withr_2.
1.2
#> [25] xml2_1.2.0       httr_1.4.0       hms_0.4.2        generics_
0.0.2
#> [29] grid_3.5.2       tidyselect_0.2.5 glue_1.3.0       R6_2.3.0
#> [33] readxl_1.2.0     rmarkdown_1.11   modelr_0.1.3     magrittr_
1.5
#> [37] backports_1.1.3  scales_1.0.0     htmltools_0.3.6  rvest_0.
3.2
```

```
#> [41] assertthat_0.2.0 colorspace_1.4-0 stringi_1.2.4    lazyeval_
0.2.1
#> [45] munsell_0.5.0    broom_0.5.1      crayon_1.3.4
```

Where to ask for help?

- The person sitting next to you during the workshop. Don't hesitate to talk to your neighbor during the workshop, compare your answers, and ask for help. You might also be interested in organizing regular meetings following the workshop to keep learning from each other.
- Your friendly colleagues: if you know someone with more experience than you, they might be able and willing to help you.
- Stack Overflow (http://stackoverflow.com/questions/tagged/r): if your question hasn't been answered before and is well crafted, chances are you will get an answer in less than 5 min. Remember to follow their guidelines on how to ask a good question (http://stackoverflow.com/help/how-to-ask).
- The R-help mailing list (https://stat.ethz.ch/mailman/listinfo/r-help): it is read by a lot of people (including most of the R core team), a lot of people post to it, but the tone can be pretty dry, and it is not always very welcoming to new users. If your question is valid, you are likely to get an answer very fast but don't expect that it will come with smiley faces. Also, here more than anywhere else, be sure to use correct vocabulary (otherwise you might get an answer pointing to the misuse of your words rather than answering your question). You will also have more success if your question is about a base function rather than a specific package.
- If your question is about a specific package, see if there is a mailing list for it. Usually it's included in the DESCRIPTION file of the package that can be accessed using `packageDescription("name-of-package")`. You may also want to try to email the author of the package directly, or open an issue on the code repository (e.g., GitHub).
- There are also some topic-specific mailing lists (GIS, phylogenetics, etc…), the complete list is here (http://www.r-project.org/mail.html).

More resources

- The Posting Guide (http://www.r-project.org/posting-guide.html) for the R mailing lists.
- How to ask for R help (http://blog.revolutionanalytics.com/2014/01/how-to-ask-for-r-help.html) useful guidelines
- This blog post by Jon Skeet (http://codeblog.jonskeet.uk/2010/08/29/writing-the-perfect-question/) has quite comprehensive advice on how to ask programming questions.
- The reprex (https://cran.rstudio.com/web/packages/reprex/) package is very helpful to create reproducible examples when asking for help. The [rOpenSci community call "How to ask questions so they get answered"], rOpenSci Blog (https://ropensci.org/commcalls/2017-03-07/) and video recording (https://vimeo.com/208749032) includes a presentation of the reprex package and of its philosophy.

Page built on: 2019-02-08 – 18:24:36

Introduction to R

Data Carpentry contributors

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.

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 (http://blog.revolutionanalytics.com/2008/12/use-equals-or-arrow-for-assignment.html) differences (http://r.789695.n4.nabble.com/Is-there-any-difference-between-and-tp878594p878598.html) 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 (https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html) 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, three popular style guides are Google's (https://google.github.io/styleguide/Rguide.xml), Jean Fan's (http://jef.works/R-style-guide/) and the tidyverse's (http://style.tidyverse.org/). The tidyverse's is very

comprehensive and may seem overwhelming at first. You can install the **`lintr`** (https://github.com/jimhester/lintr) package to automatically check for issues in the styling of your code.

> ## 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 (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?

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`.

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?
```

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.

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
```

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)
```

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(weight_g)
str(animals)
```

You can use the `c()` function to add other elements to your vector:

```
weight_g <- c(weight_g, 90) # add to the end of the vector
weight_g <- c(30, weight_g) # add to the beginning of the vector
weight_g
```

In the first line, we take the original vector `weight_g`, add the value `90` to the end of it, and save the result back into `weight_g`. Then we add the value `30` to the beginning, again saving the result back into `weight_g`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

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`).

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?

Answer

- 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?

Answer

- 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)
```

Answer

- 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?

Answer

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.

Conditional subsetting

Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not:

```
weight_g <- c(21, 34, 39, 54, 55)
weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
#> [1] 21 39 54
```

Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests. For instance, if you wanted to select only the values above 50:

```
weight_g > 50    # will return logicals with TRUE for the indices th
  at meet the condition
```

```
#> [1] FALSE FALSE FALSE  TRUE  TRUE
```

```
## so we can use this to select only the values above 50
weight_g[weight_g > 50]
```

```
#> [1] 54 55
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

```
weight_g[weight_g < 30 | weight_g > 50]
```

```
#> [1] 21 54 55
```

```
weight_g[weight_g >= 30 & weight_g == 21]
```

```
#> numeric(0)
```

Here, `<` stands for “less than”, `>` for “greater than”, `>=` for “greater than or equal to”, and `==` for “equal to”. The double equal sign `==` is a test for numerical equality between the left and right hand sides, and should not be confused with the single `=` sign, which performs variable assignment (similar to `<-`).

A common task is to search for certain strings in a vector. One could use the “or” operator `|` to test for equality to multiple values, but this can quickly become tedious. The function `%in%` allows you to test if any of the elements of a search vector are found:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[animals == "cat" | animals == "rat"] # returns both rat and
   cat
```

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

```
animals %in% c("rat", "cat", "dog", "duck", "goat")
```

```
#> [1] FALSE  TRUE  TRUE  TRUE
```

```
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]
```

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

Challenge (optional)

- Can you figure out why `"four" > "five"` returns `TRUE`?

Answer

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 ob
  ject is an atomic vector of type `"numeric"` (or `"double"`).
na.omit(heights)

## Extract those elements which are complete cases. The returned obj
  ect is an atomic vector of type `"numeric"` (or `"double"`).
heights[complete.cases(heights)]
```

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

Challenge

1. Using this vector of heights in inches, create a new vector, `heights_no_na` , 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.
3. Use R to figure out how many people in the set are taller than 67 inches.

Answer

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.

Page built on: 📅 2019-02-08 – 🕐 18:24:37

Starting with data

Data Carpentry contributors

Learning Objectives

- Load external data from a .csv file into a data frame.
- Describe what a data frame is.
- Summarize the contents of a data frame.
- Use indexing to subset specific portions of data frames.
- Describe what a factor is.
- Convert between strings and factors.
- Reorder and rename factors.
- Change how character strings are handled in a data frame.
- Format dates.

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 (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.”

```
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")
```

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)
```

```
#>   record_id month day year plot_id species_id sex hindfoot_length w
eight
#> 1         1     7  16 1977       2         NL   M              32
NA
#> 2        72     8  19 1977       2         NL   M              31
NA
#> 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
#>     genus  species   taxa plot_type
#> 1 Neotoma albigula Rodent   Control
#> 2 Neotoma albigula Rodent   Control
#> 3 Neotoma albigula Rodent   Control
#> 4 Neotoma albigula Rodent   Control
#> 5 Neotoma albigula Rodent   Control
#> 6 Neotoma albigula Rodent   Control
```

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

Note

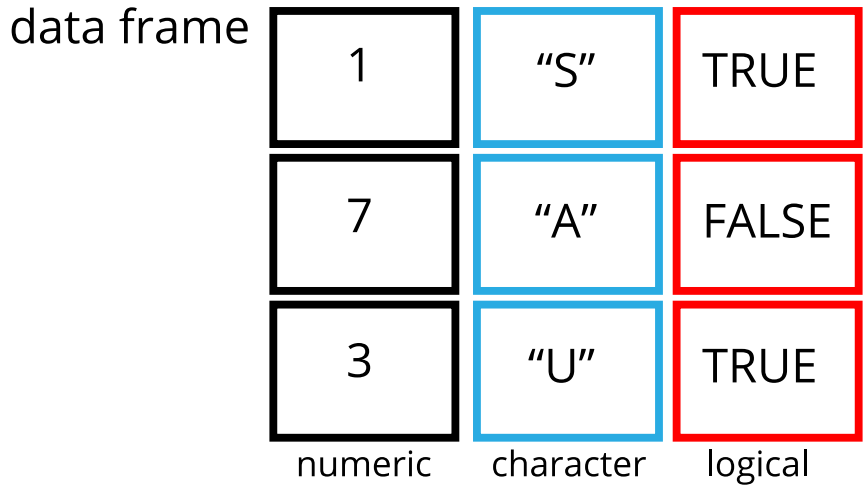
`read.csv` assumes that fields are delineated by commas, however, in several countries, the comma is used as a decimal separator and the semicolon (;) is used as a field delineator. If you want to read in this type of files in R, you can use the `read.csv2` function. It behaves exactly like `read.csv` but uses different parameters for the decimal and the field separators. If you are working with another format, they can be both specified by the user. Check out the help for `read.csv()` by typing `?read.csv` to learn more. There is also the `read.delim()` for in tab separated data files. It is important to note that all of these functions are actually wrapper functions for the main `read.table()` function with different arguments. As such, the surveys data above could have also been loaded by using `read.table()` with the separation argument as `,`. The code is as follows:
`surveys <- read.table(file="data/portal_data_joined.csv", sep=",", header=TRUE)`.
The header argument has to be set to TRUE to be able to read the headers as by default `read.table()` has the header argument set to FALSE.

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 **str**ucture of a data frame with the function `str()`:

```
str(surveys)
```

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`.

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

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.

Challenge

1. Create a `data.frame` (`surveys_200`) containing only the data in row 200 of the `surveys` dataset.
2. Notice how `nrow()` gave you the number of rows in a `data.frame`?
    - Use that number to pull out just that last row in the data frame.
    - Compare that with what you see as the last row using `tail()` to make sure it's meeting expectations.
    - Pull out that last row using `nrow()` instead of the row number.
    - Create a new data frame (`surveys_last`) from that last row.
3. Use `nrow()` to extract the row that is in the middle of the data frame. Store the content of this row in an object named `surveys_middle`.
4. Combine `nrow()` with the `-` notation above to reproduce the behavior of `head(surveys)`, keeping just the first through 6th rows of the surveys dataset.

Answer

Factors

When we did `str(surveys)` we saw that several of the columns consist of integers. The columns `genus`, `species`, `sex`, `plot_type`, … however, are of a special class called `factor`. Factors are very useful and actually contribute to making R particularly well suited to working with data. So we are going to spend a little time introducing them.

Factors represent categorical data. They are stored as integers associated with labels and they can be ordered or unordered. While factors look (and often behave) like character vectors, they are actually treated as integer vectors by R. So you need to be very careful when treating them as strings.

Once created, factors can only contain a pre-defined set of values, known as *levels*. By default, R always sorts levels in alphabetical order. For instance, if you have a factor with 2 levels:

```
sex <- factor(c("male", "female", "female", "male"))
```

R will assign `1` to the level `"female"` and `2` to the level `"male"` (because `f` comes before `m`, even though the first element in this vector is `"male"`). You can see this by using the function `levels()` and you can find the number of levels using `nlevels()`:

```
levels(sex)
nlevels(sex)
```

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., “low”, “medium”, “high”), it improves your visualization, or it is required by a particular type of analysis. Here, one way to reorder our levels in the `sex` vector would be:

```
sex # current order
```

```
#> [1] male   female female male
#> Levels: female male
```

```
sex <- factor(sex, levels = c("male", "female"))
sex # after re-ordering
```

```
#> [1] male   female female male
#> Levels: male female
```

In R’s memory, these factors are represented by integers (1, 2, 3), but are more informative than integers because factors are self describing: `"female"`, `"male"` is more descriptive than `1`, `2`. Which one is “male”? You wouldn’t be able to tell just from the integer data. Factors, on the other hand, have this information built in. It is particularly helpful when there are many levels (like the species names in our example dataset).

Converting factors

If you need to convert a factor to a character vector, you use `as.character(x)`.

```
as.character(sex)
```

In some cases, you may have to convert factors where the levels appear as numbers (such as concentration levels or years) to a numeric vector. For instance, in one part of your analysis the years might need to be encoded as factors (e.g., comparing average weights across years) but in another part of your analysis they may need to be stored as numeric values (e.g., doing math operations on the years). This conversion from factor to numeric is a little trickier. The `as.numeric()` function returns the index values of the factor, not its levels, so it will result in an entirely new (and unwanted in this case) set of numbers. One method to avoid this is to convert factors to characters, and then to numbers.

Another method is to use the `levels()` function. Compare:

```
year_fct <- factor(c(1990, 1983, 1977, 1998, 1990))
as.numeric(year_fct)               # Wrong! And there is no warning...
as.numeric(as.character(year_fct)) # Works...
as.numeric(levels(year_fct))[year_fct]    # The recommended way.
```

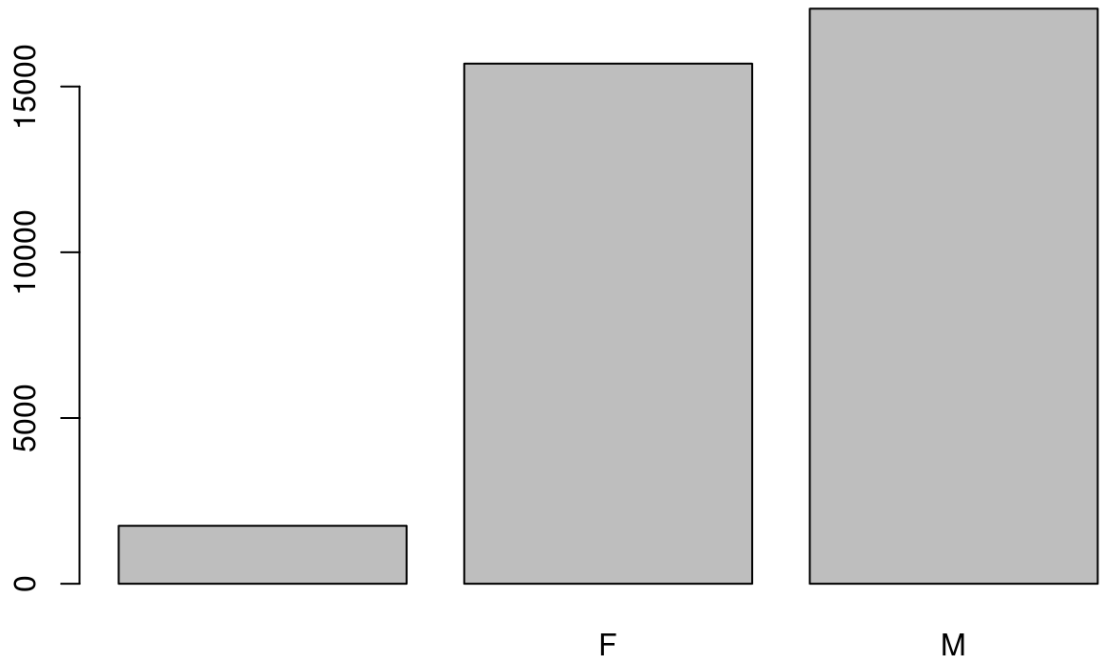
Notice that in the `levels()` approach, three important steps occur:

- We obtain all the factor levels using `levels(year_fct)`
- We convert these levels to numeric values using `as.numeric(levels(year_fct))`
- We then access these numeric values using the underlying integers of the vector `year_fct` inside the square brackets

Renaming factors

When your data is stored as a factor, you can use the `plot()` function to get a quick glance at the number of observations represented by each factor level. Let's look at the number of males and females captured over the course of the experiment:

```
## bar plot of the number of females and males captured during the exp
  eriment:
plot(surveys$sex)
```



In addition to males and females, there are about 1700 individuals for which the sex information hasn't been recorded. Additionally, for these individuals, there is no label to indicate that the information is missing or undetermined. Let's rename this label to something more meaningful. Before doing that, we're going to pull out the data on sex and work with that data, so we're not modifying the working copy of the data frame:

```
sex <- surveys$sex
head(sex)
```

```
#> [1] M M
#> Levels:  F M
```

```
levels(sex)
```

```
#> [1] ""  "F" "M"
```

```
levels(sex)[1] <- "undetermined"
levels(sex)
```

```
#> [1] "undetermined" "F"            "M"
```

```
head(sex)
```

```
#> [1] M            M            undetermined undetermined undetermine
d
#> [6] undetermined
#> Levels: undetermined F M
```

Challenge

- Rename "F" and "M" to "female" and "male" respectively.
- Now that we have renamed the factor level to "undetermined", can you recreate the barplot such that "undetermined" is last (after "male")?

Answer

Using `stringsAsFactors=FALSE`

By default, when building or importing a data frame, the columns that contain characters (i.e. text) are coerced (= converted) into factors. Depending on what you want to do with the data, you may want to keep these columns as `character`. To do so, `read.csv()` and `read.table()` have an argument called `stringsAsFactors` which can be set to `FALSE`.

In most cases, it is preferable to set `stringsAsFactors = FALSE` when importing data and to convert as a factor only the columns that require this data type.

```
## Compare the difference between our data read as `factor` vs `charac
  ter`.
surveys <- read.csv("data/portal_data_joined.csv", stringsAsFactors =
  TRUE)
str(surveys)
surveys <- read.csv("data/portal_data_joined.csv", stringsAsFactors =
  FALSE)
str(surveys)
## Convert the column "plot_type" into a factor
surveys$plot_type <- factor(surveys$plot_type)
```

Challenge

1. We have seen how data frames are created when using `read.csv()`, but they can also be created by hand with the `data.frame()` function. There are a few mistakes in this hand-crafted `data.frame`. Can you spot and fix them? Don't hesitate to experiment!

```
animal_data <- data.frame(
          animal = c(dog, cat, sea cucumber, sea urchin),
          feel = c("furry", "squishy", "spiny"),
          weight = c(45, 8 1.1, 0.8)
          )
```

2. Can you predict the class for each of the columns in the following example? Check your guesses using `str(country_climate)`:
   - Are they what you expected? Why? Why not?
   - What would have been different if we had added `stringsAsFactors = FALSE` when creating the data frame?
   - What would you need to change to ensure that each column had the accurate data type?

```
country_climate <- data.frame(
       country = c("Canada", "Panama", "South Africa", "Aust
 ralia"),
       climate = c("cold", "hot", "temperate", "hot/temperat
 e"),
       temperature = c(10, 30, 18, "15"),
       northern_hemisphere = c(TRUE, TRUE, FALSE, "FALSE"),
       has_kangaroo = c(FALSE, FALSE, FALSE, 1)
       )
```

Answer

The automatic conversion of data type is sometimes a blessing, sometimes an annoyance. Be aware that it exists, learn the rules, and double check that data you import in R are of the correct type within your data frame. If not, use it to your advantage to detect mistakes that might have been introduced during data entry (for instance, a letter in a column that should only contain numbers).

Learn more in this RStudio tutorial (https://support.rstudio.com/hc/en-us/articles/218611977-Importing-Data-with-RStudio)

Formatting Dates

One of the most common issues that new (and experienced!) R users have is converting date and time information into a variable that is appropriate and usable during analyses. As a reminder from earlier in this lesson, the best practice for dealing with date data is to ensure that each component of your date is stored as a separate variable. Using `str()`, We can confirm that our data frame has a separate column for day, month, and year, and that each contains integer values.

```
str(surveys)
```

We are going to use the `ymd()` function from the package **`lubridate`** (which belongs to the **`tidyverse`**; learn more here (https://www.tidyverse.org/)). . **`lubridate`** gets installed as part as the **`tidyverse`** installation. When you load the **`tidyverse`** (`library(tidyverse)`), the core packages (the packages used in most data analyses) get loaded. **`lubridate`** however does not belong to the core tidyverse, so you have to load it explicitly with `library(lubridate)`

Start by loading the required package:

```
library(lubridate)
```

`ymd()` takes a vector representing year, month, and day, and converts it to a `Date` vector. `Date` is a class of data recognized by R as being a date and can be manipulated as such. The argument that the function requires is flexible, but, as a best practice, is a character vector formatted as “YYYY-MM-DD”.

Let’s create a date object and inspect the structure:

```
my_date <- ymd("2015-01-01")
str(my_date)
```

Now let’s paste the year, month, and day separately - we get the same result:

```
# sep indicates the character to use to separate each component
my_date <- ymd(paste("2015", "1", "1", sep = "-"))
str(my_date)
```

Now we apply this function to the surveys dataset. Create a character vector from the `year`, `month`, and `day` columns of `surveys` using `paste()`:

```
paste(surveys$year, surveys$month, surveys$day, sep = "-")
```

This character vector can be used as the argument for `ymd()`:

```
ymd(paste(surveys$year, surveys$month, surveys$day, sep = "-"))
```

```
#> Warning: 129 failed to parse.
```

The resulting `Date` vector can be added to `surveys` as a new column called `date`:

```
surveys$date <- ymd(paste(surveys$year, surveys$month, surveys$day, se
  p = "-"))
```

```
#> Warning: 129 failed to parse.
```

```
str(surveys) # notice the new column, with 'date' as the class
```

Let's make sure everything worked correctly. One way to inspect the new column is to use `summary()`:

```
summary(surveys$date)
```

```
#>         Min.      1st Qu.       Median         Mean      3rd Qu.
#> "1977-07-16" "1984-03-12" "1990-07-22" "1990-12-15" "1997-07-29"
#>         Max.         NA's
#> "2002-12-31"        "129"
```

Something went wrong: some dates have missing values. Let's investigate where they are coming from.

We can use the functions we saw previously to deal with missing data to identify the rows in our data frame that are failing. If we combine them with what we learned about subsetting data frames earlier, we can extract the columns "year,"month","day" from the records that have `NA` in our new column `date`. We will also use `head()` so we don't clutter the output:

```
missing_dates <- surveys[is.na(surveys$date), c("year", "month", "day"
  )]

head(missing_dates)
```

```
#>      year month day
#> 3144 2000     9  31
#> 3817 2000     4  31
#> 3818 2000     4  31
#> 3819 2000     4  31
#> 3820 2000     4  31
#> 3856 2000     9  31
```

Why did these dates fail to parse? If you had to use these data for your analyses, how would you deal with this situation?

Page built on: 📅 2019-02-08 – 🕐 18:24:46

Manipulating, analyzing and exporting data with tidyverse

Data Carpentry contributors

Manipulating and analyzing data with dplyr

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 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.

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.

The **tidyverse** package tries to address 3 common issues that arise when doing data analysis with some of the functions that come with R:

1. The results from a base R function sometimes depend on the type of data.
2. Using R expressions in a non standard way, which can be confusing for new learners.

3. Hidden arguments, having default operations that new learners are not aware of.

We have seen in our previous lesson that when building or importing a data frame, the columns that contain characters (i.e., text) are coerced (=converted) into the `factor` data type. We had to set **`stringsAsFactors`** to **`FALSE`** to avoid this hidden argument to convert our data type.

This time we will use the **`tidyverse`** package to read the data and avoid having to set **`stringsAsFactors`** to **`FALSE`**

To load the package type:

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

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 (https://github.com/rstudio/cheatsheets/raw/master/data-transformation.pdf) and this one about **`tidyr`** (https://github.com/rstudio/cheatsheets/raw/master/data-import.pdf).

We'll read in our data using the `read_csv()` function, from the tidyverse package **`readr`**, instead of `read.csv()`.

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

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

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

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

Notice that the class of the data is now `tbl_df`

This is referred to as a “tibble”. Tibbles tweak some of the behaviors of the data frame objects we introduced in the previous episode. The data structure is very similar to a data frame. For our purposes the only differences are that:

1. In addition to displaying the data type of each column under its name, it only prints the first few rows of data and only as many columns as fit on one screen.
2. Columns of class `character` are never converted into factors.

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

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)
```

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.

Challenge

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

Answer

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`.

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

Split-apply-combine data analysis and the summarize() function

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, and then combine the results. **`dplyr`** makes this very easy through the use of the `group_by()` function.

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))
```

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)
```

Previous example shows the use of `count()` to count the number of rows/observations for *one* factor (i.e., `sex`). If we wanted to count *combination of factors*, such as `sex` and `species`, we would specify the first and the second factor as the arguments of `count()`:

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

With the above code, we can proceed with `arrange()` to sort the table according to a number of criteria so that we have a better comparison. For instance, we might want to arrange the table above in (i) an alphabetical order of the levels of the species and (ii) in descending order of the count:

```
surveys %>%
  count(sex, species) %>%
  arrange(species, desc(n))
```

From the table above, we may learn that, for instance, there are 75 observations of the *albigula* species that are not specified for its sex (i.e. `NA`).

Challenge

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

Answer

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

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

Answer

Reshaping with gather and spread

In the spreadsheet lesson (http://www.datacarpentry.org/spreadsheet-ecology-lesson/01-format-data/), 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

Here we examine the fourth rule: Each type of observational unit forms a table.

In `surveys`, the rows of `surveys` contain the values of variables associated with each record (the unit), values such the weight or sex of each animal associated with each record. What if instead of comparing records, we wanted to compare the different mean weight of each species between plots? (Ignoring `plot_type` for simplicity).

We'd need to create a new table where each row (the unit) is comprised of values of variables associated with each plot. In practical terms this means the values of the species in `genus` would become the names of column variables and the cells would contain the values of the mean weight observed on each plot.

Having created a new table, it is therefore straightforward to explore the relationship between the weight of different species within, and between, the plots. The key point here is that we are still following a tidy data structure, but we have **reshaped** the data according to the observations of interest: average species weight per plot instead of recordings per date.

The opposite transformation would be to transform column names into values of a variable.

We can do both these of transformations with two `tidyr` functions, `spread()` and `gather()`.

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.

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.

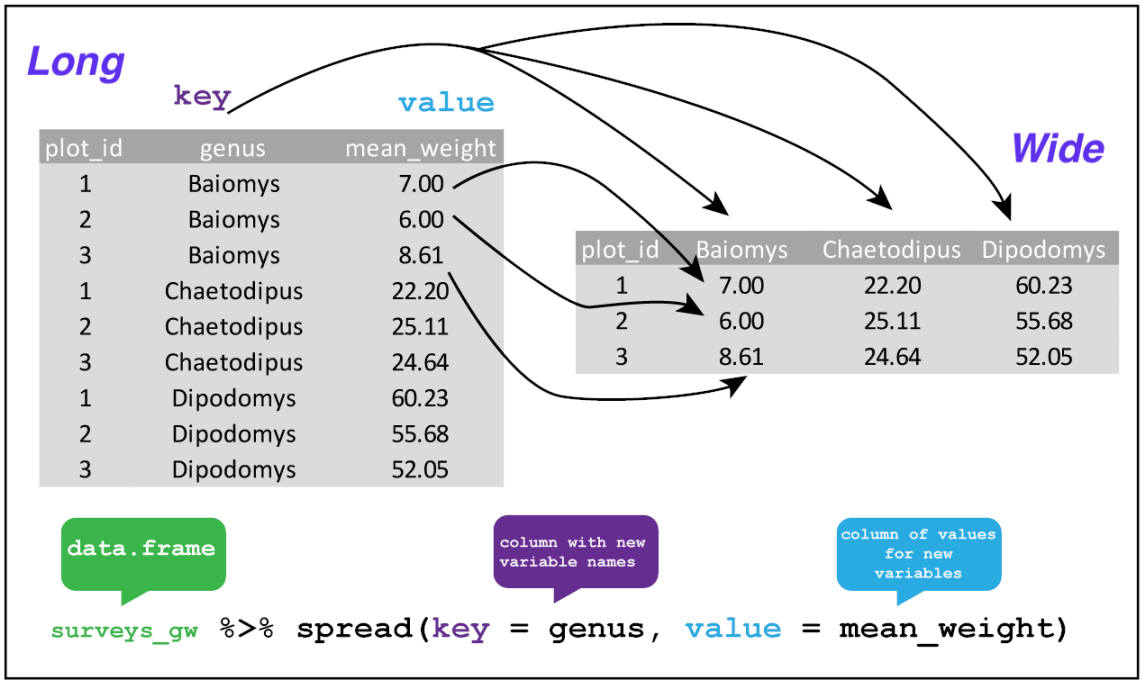
```
surveys_gw <- surveys %>%
  filter(!is.na(weight)) %>%
  group_by(genus, plot_id) %>%
  summarize(mean_weight = mean(weight))

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()
```

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.

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.

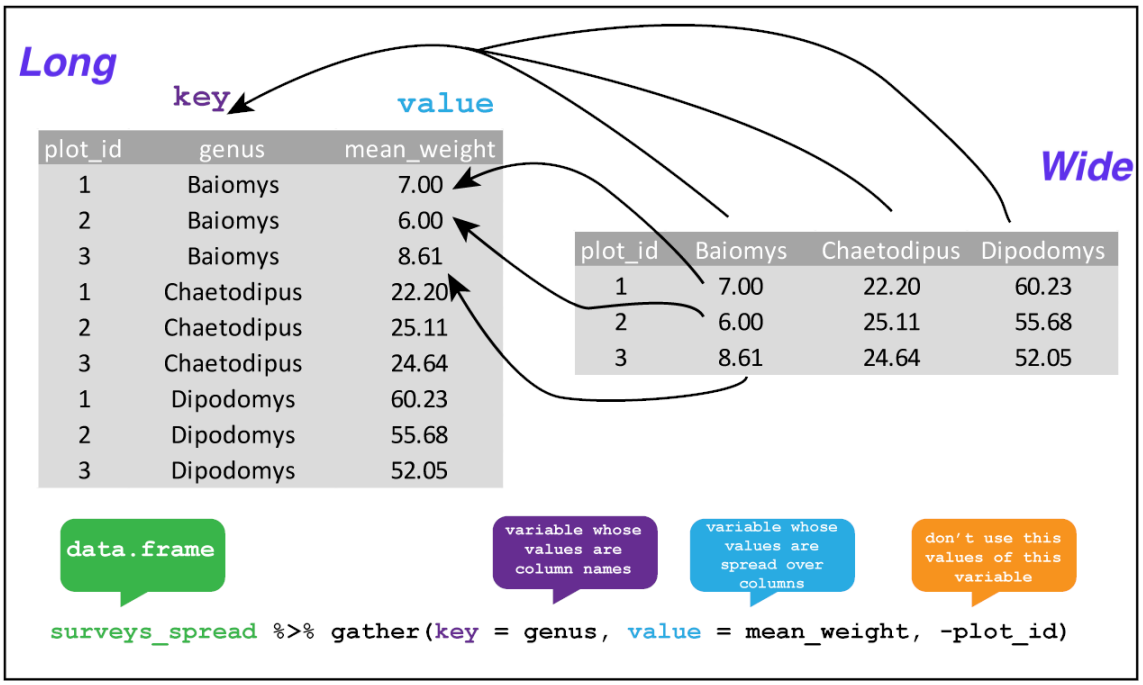
`gather()` takes four principal arguments:

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()
```

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

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

Answer

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

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

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.

Before using `write_csv()`, we are going to create a new folder, `data_output`, in our working directory that will store this generated dataset. We don't want to write generated datasets in the same directory as our raw data. It's good practice to keep them separate. The `data` folder should only contain the raw, unaltered data, and should be left alone to make sure we don't delete or modify it. In contrast, our script will generate the contents of the `data_output` directory, so even if the files it contains are deleted, we can always re-generate them.

In preparation for our next lesson on plotting, we are going to prepare a cleaned up version of the data set that doesn't include any missing data.

Let's start by removing observations of animals for which `weight` and `hindfoot_length` are missing, or the `sex` has not been determined:

```
surveys_complete <- surveys %>%
  filter(!is.na(weight),           # remove missing weight
         !is.na(hindfoot_length),  # remove missing hindfoot_length
         !is.na(sex))                # remove missing sex
```

Because we are interested in plotting how species abundances have changed through time, we are also going to remove observations for rare species (i.e., that have been observed less than 50 times). We will do this in two steps: first we are going to create a data set that counts how often each species has been observed, and filter out the rare species; then, we will extract only the observations for these more common species:

```
## Extract the most common species_id
species_counts <- surveys_complete %>%
    count(species_id) %>%
    filter(n >= 50)

## Only keep the most common species
surveys_complete <- surveys_complete %>%
  filter(species_id %in% species_counts$species_id)
```

To make sure that everyone has the same data set, check that `surveys_complete` has 30463 rows and 13 columns by typing `dim(surveys_complete)`.

Now that our data set is ready, we can save it as a CSV file in our `data_output` folder.

```
write_csv(surveys_complete, path = "data_output/surveys_complete.cs
  v")
```

Page built on: 📅 2019-02-08 – 🕐 18:24:49

Questions? Feedback? Please file an issue on GitHub (https://github.com/datacarpentry/R-ecology-lesson/issues/new).

On Twitter: @datacarpentry (https://twitter.com/datacarpentry)

Data visualization with ggplot2

Data Carpentry contributors

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.

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")
```

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.

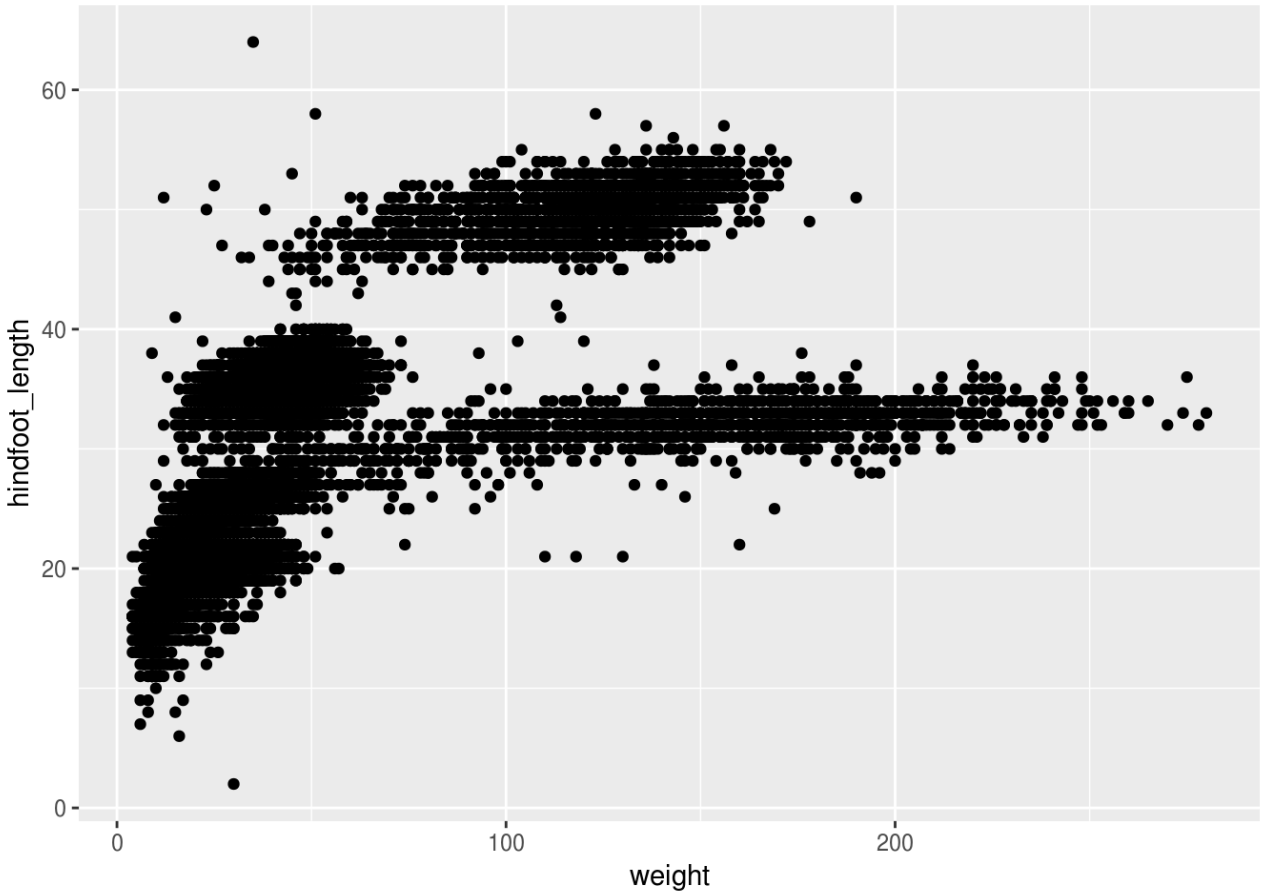
```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfo
 ot_length))
```

- 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()
```

Challenge (optional)

Scatter plots can be useful exploratory tools for small datasets. For data sets with large numbers of observations, such as the `surveys_complete` data set, overplotting of points can be a limitation of scatter plots. One strategy for handling such settings is to use hexagonal binning of observations. The plot space is tessellated into hexagons. Each hexagon is assigned a color based on the number of observations that fall within its boundaries. To use hexagonal binning with **`ggplot2`**, first install the R package `hexbin` from CRAN:

```
install.packages("hexbin")
library(hexbin)
```

Then use the `geom_hex()` function:

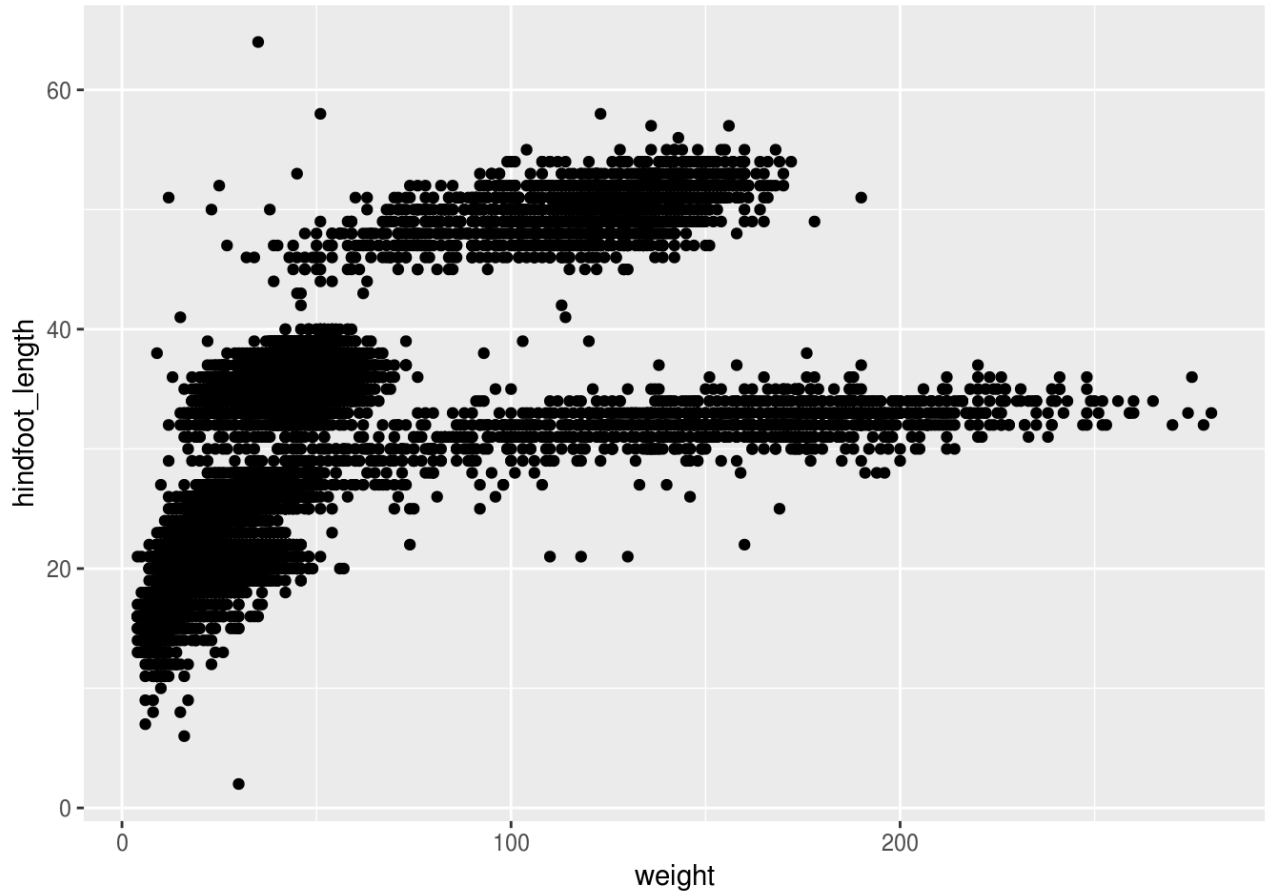
```
surveys_plot +
 geom_hex()
```

- What are the relative strengths and weaknesses of a hexagonal bin plot compared to a scatter plot? Examine the above scatter plot and compare it with the hexagonal bin plot that you created.

Building your plots iteratively

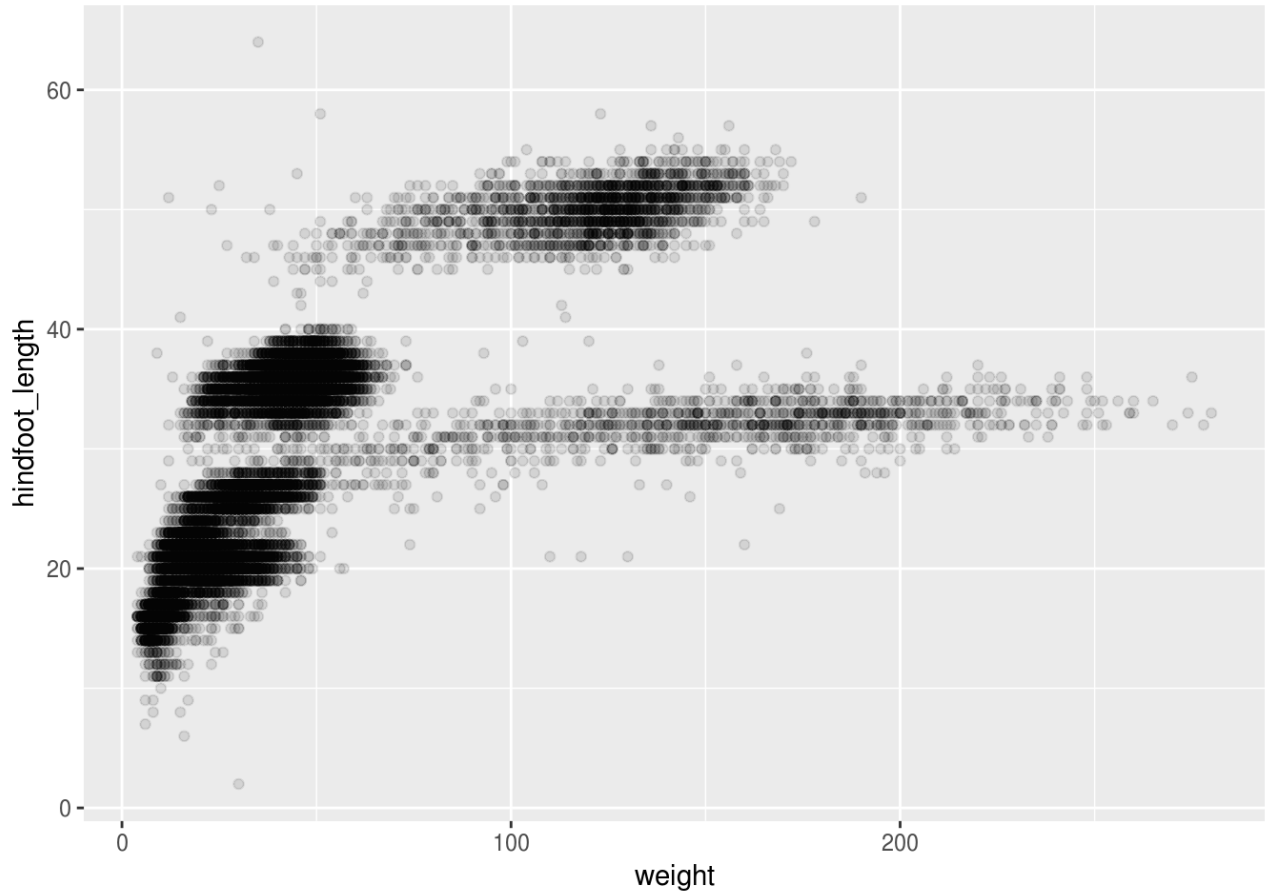
Building plots with **`ggplot2`** is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

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



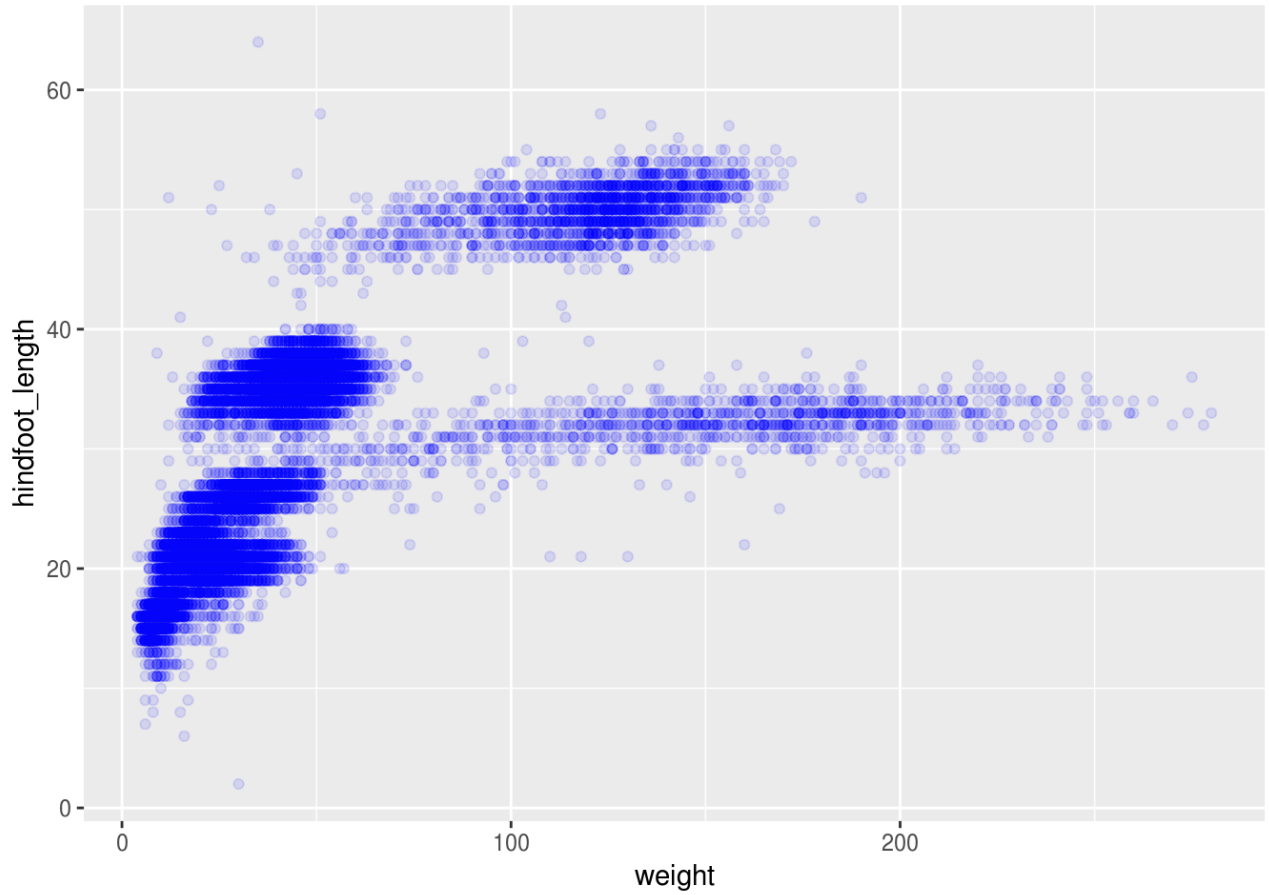
Then, we start modifying this plot to extract more information from it. For instance, we can add transparency (`alpha`) to avoid overplotting:

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



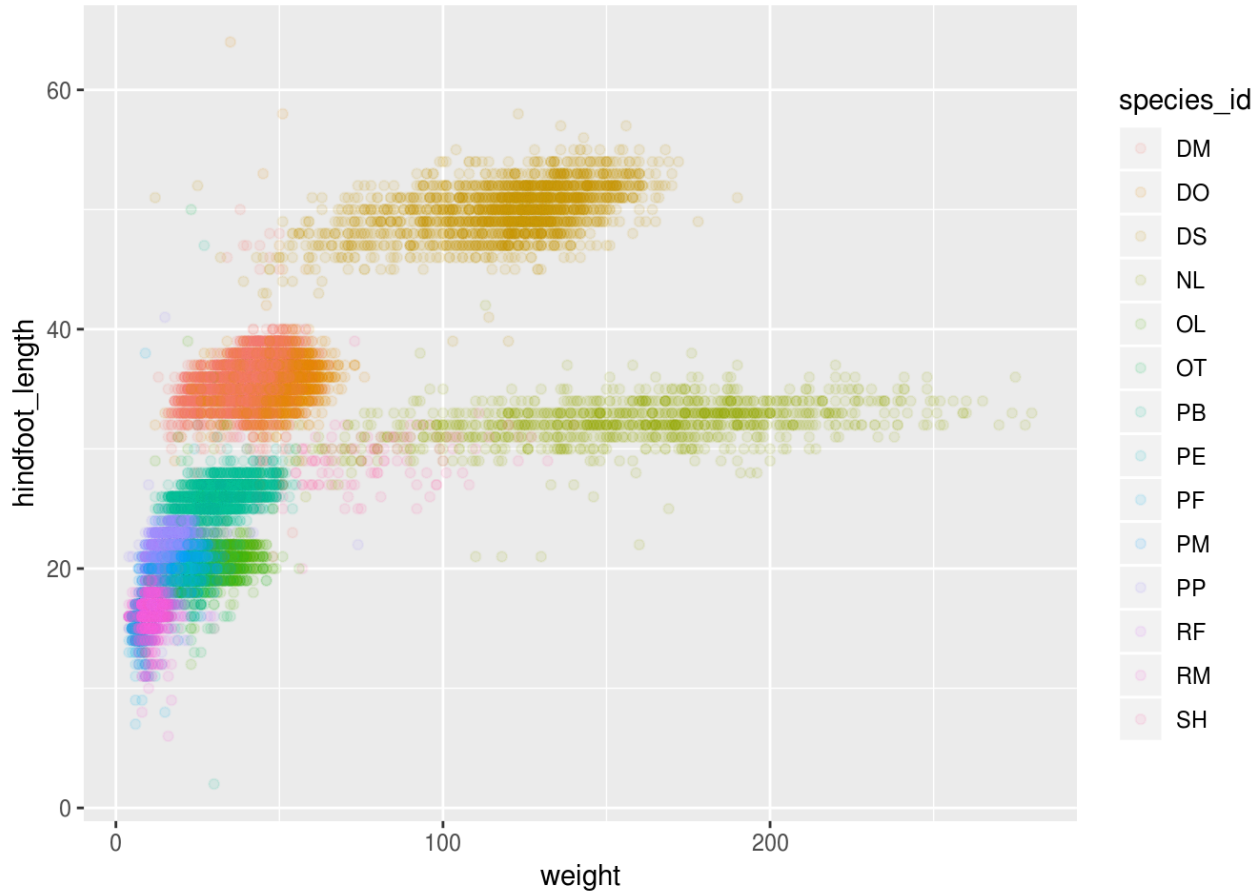
We can also add colors for all the points:

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



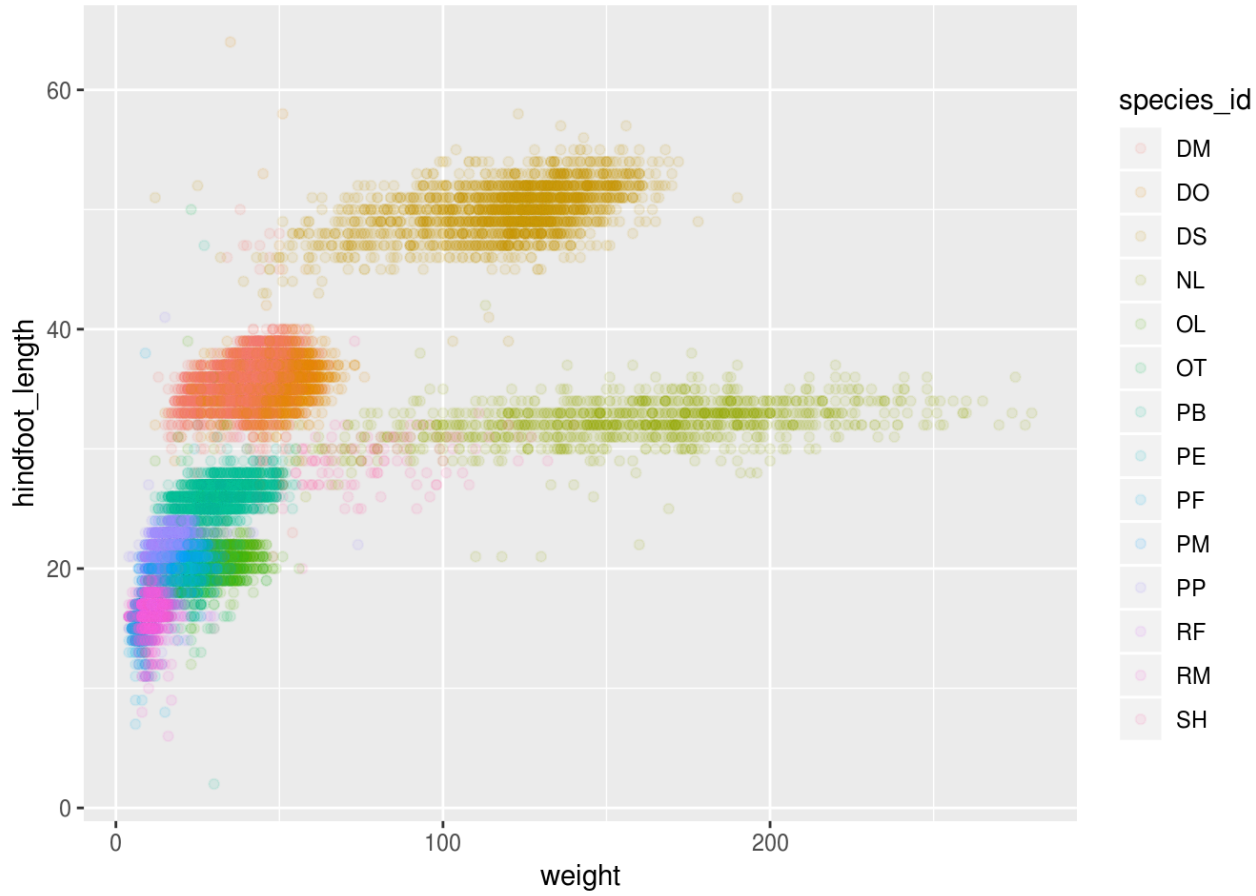
Or to color each species in the plot differently, you could use a vector as an input to the argument **color**. **ggplot2** will provide a different color corresponding to different values in the vector. Here is an example where we color with **species_id**:

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



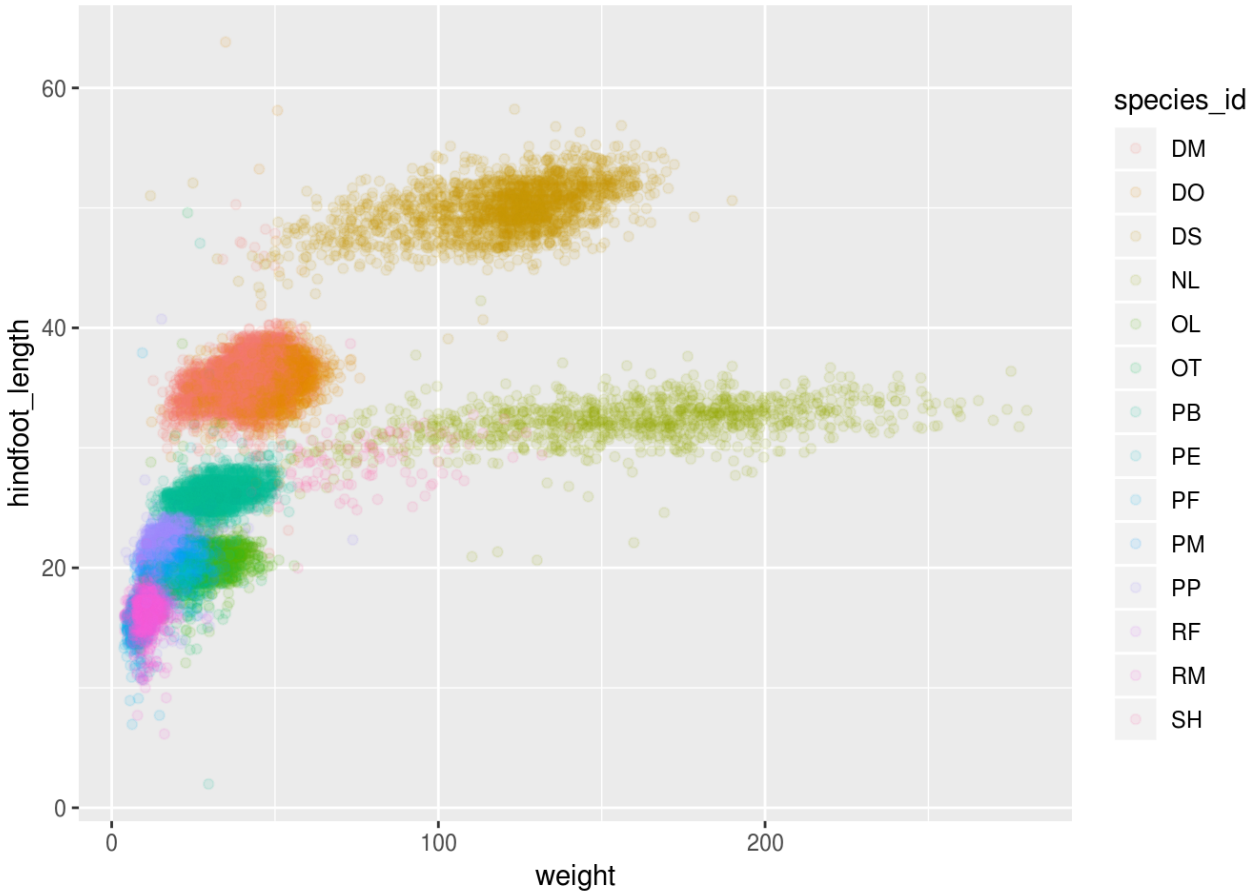
We can also specify the colors directly inside the mapping provided in the `ggplot()` function. This will be seen by any geom layers and the mapping will be determined by the x- and y-axis set up in `aes()`.

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



Challenge

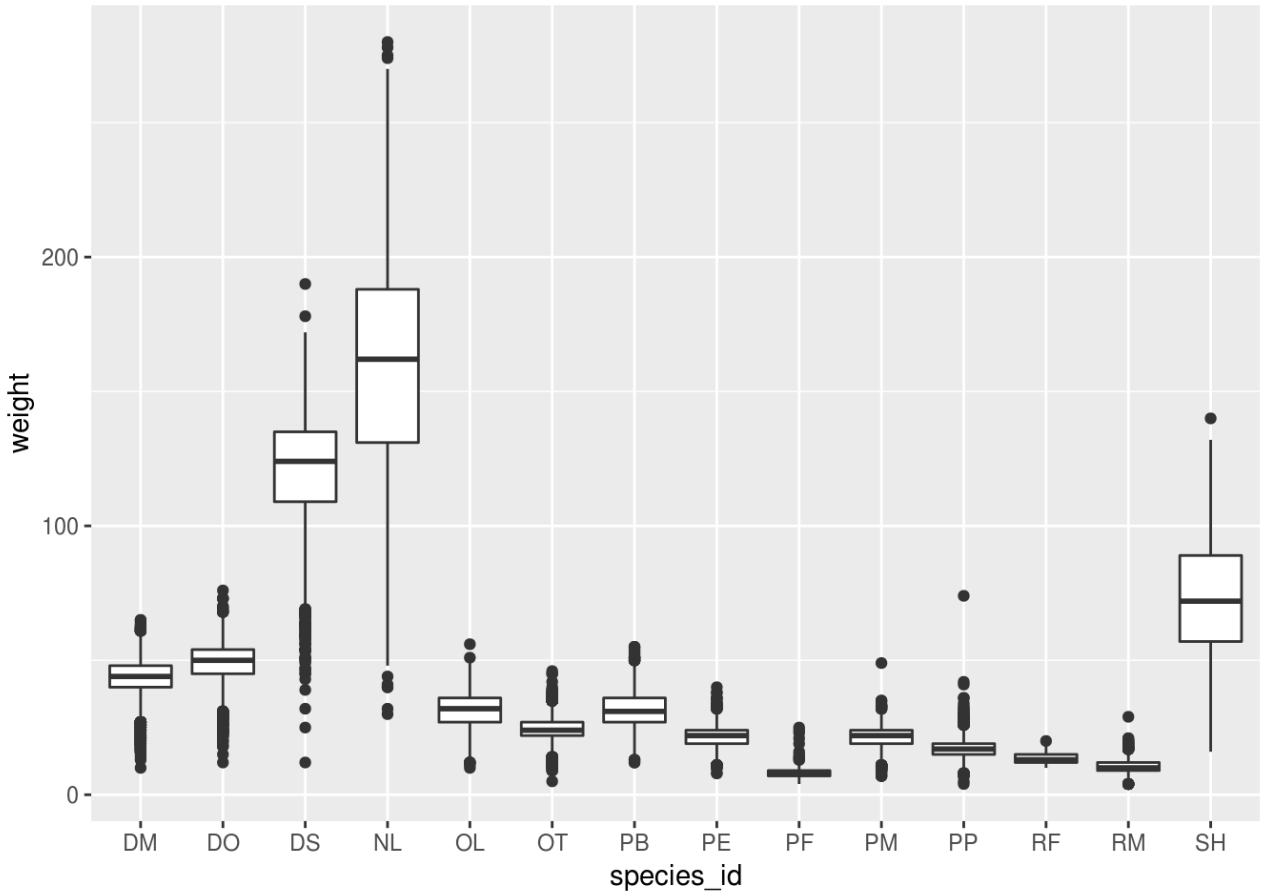
Use what you just learned to create a scatter plot of `weight` over `species_id` with the plot types showing in different colors. Is this a good way to show this type of data?

Answer

Boxplot

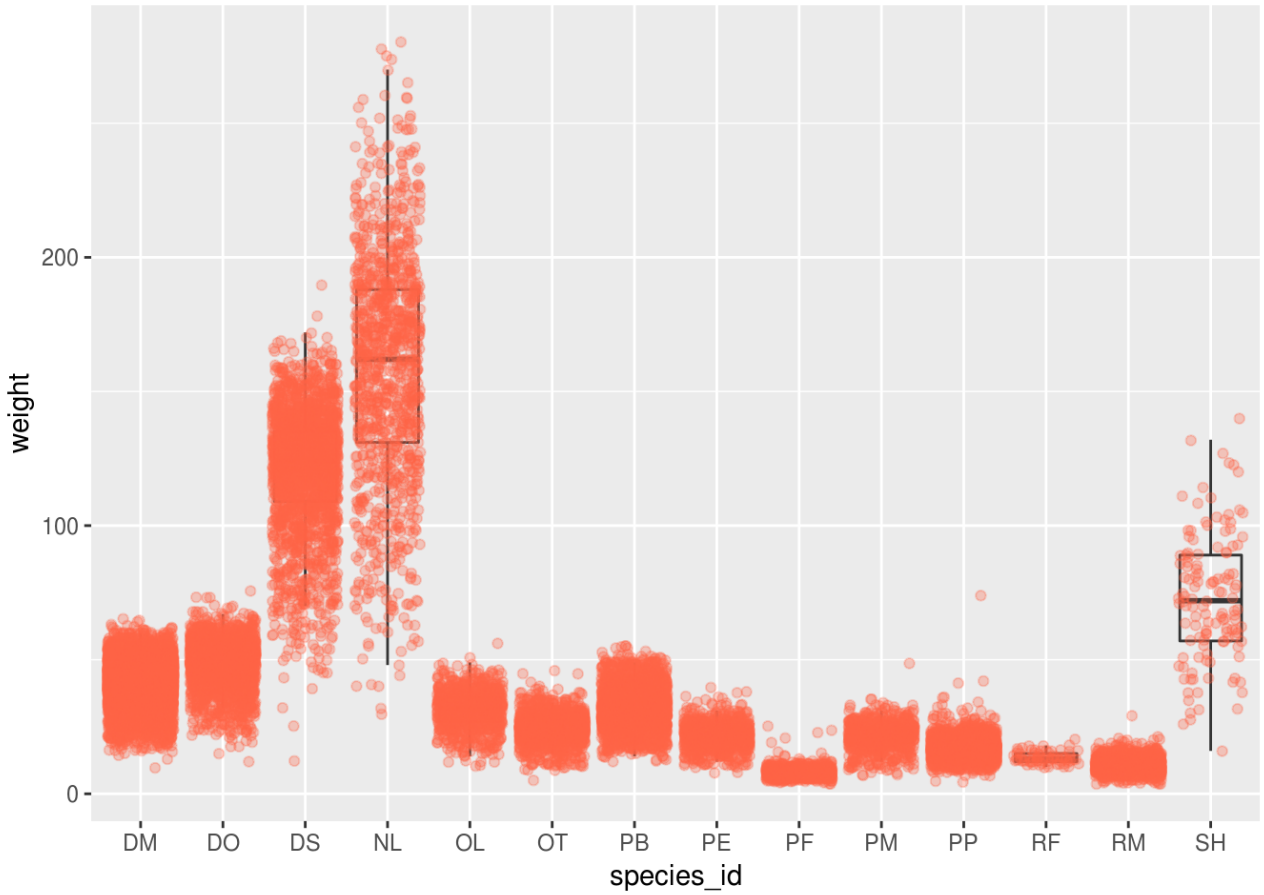
We can use boxplots to visualize the distribution of weight within each species:

```
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = we
  ight)) +
    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 = we
  ight)) +
    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?

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?

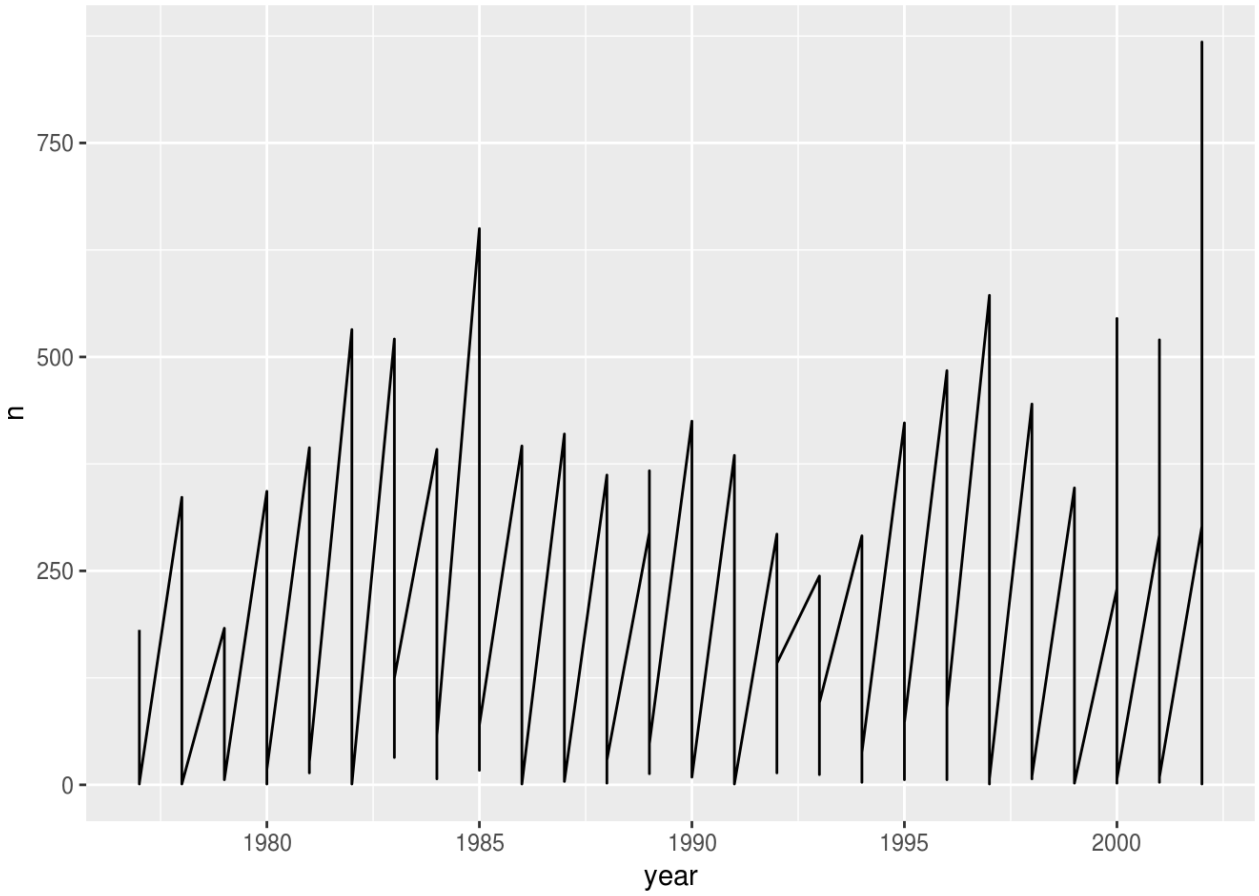
Plotting time series data

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)
```

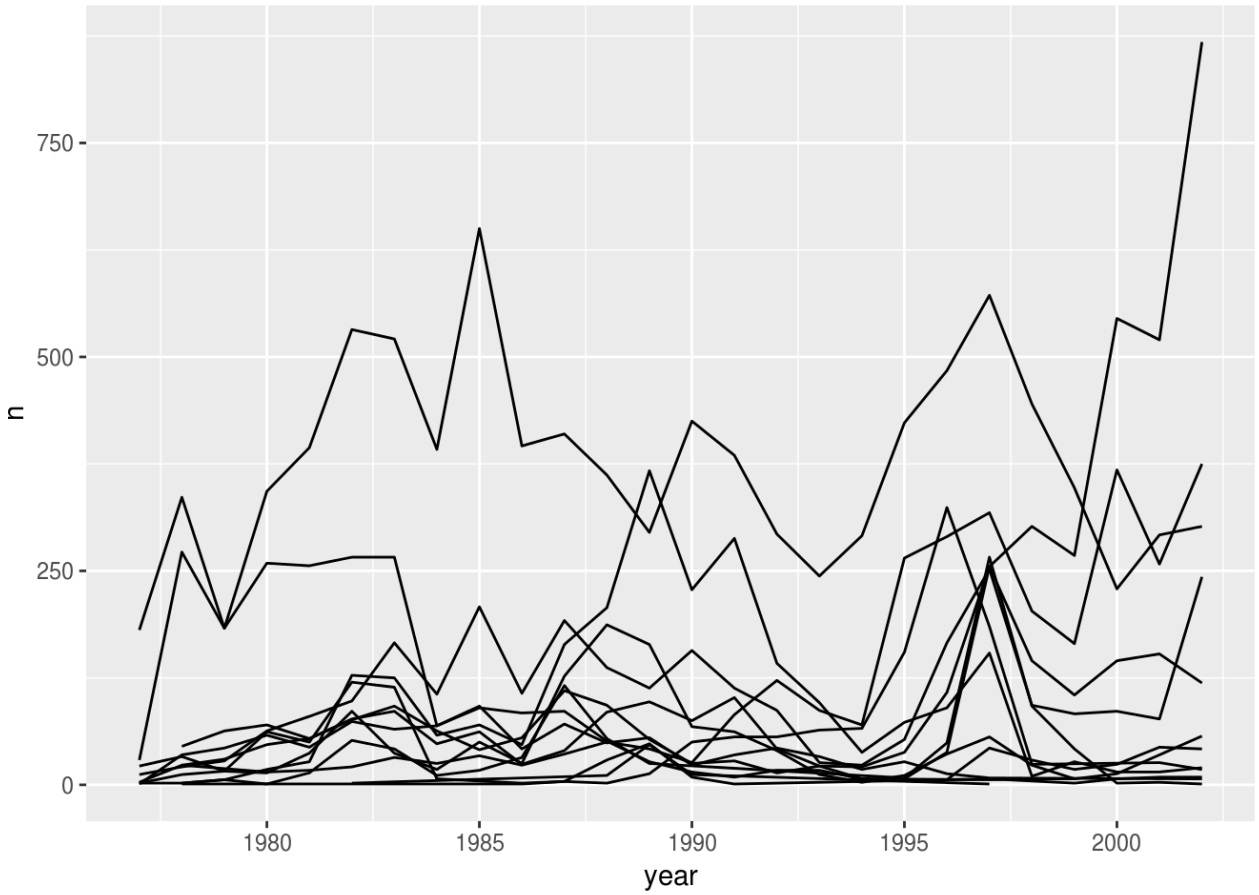
Time series data can be visualized as a line plot with years on the x axis and counts on the y axis:

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



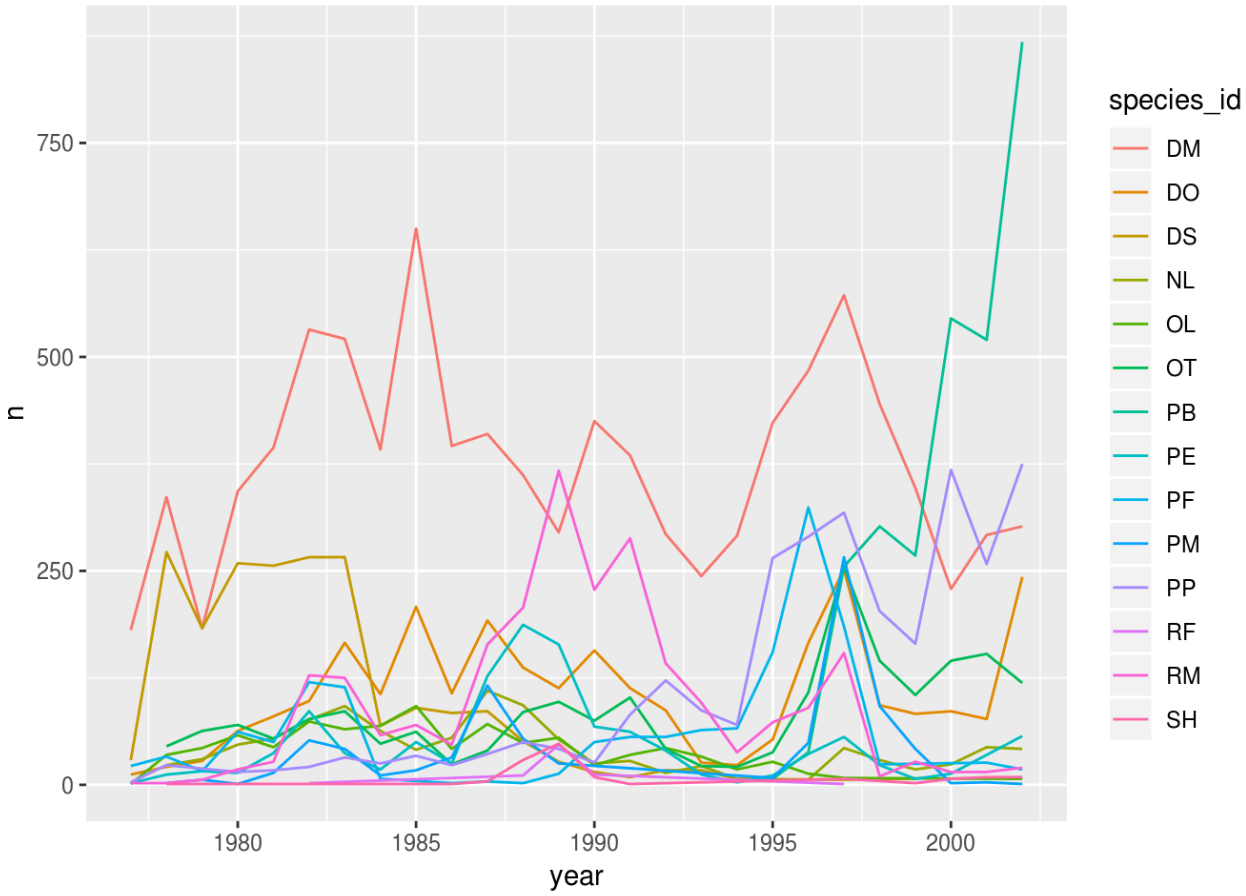
Unfortunately, this does not work because we plotted data for all the species together. We need to tell ggplot to draw a line for each species by modifying the aesthetic function to include `group = species_id`:

```
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):

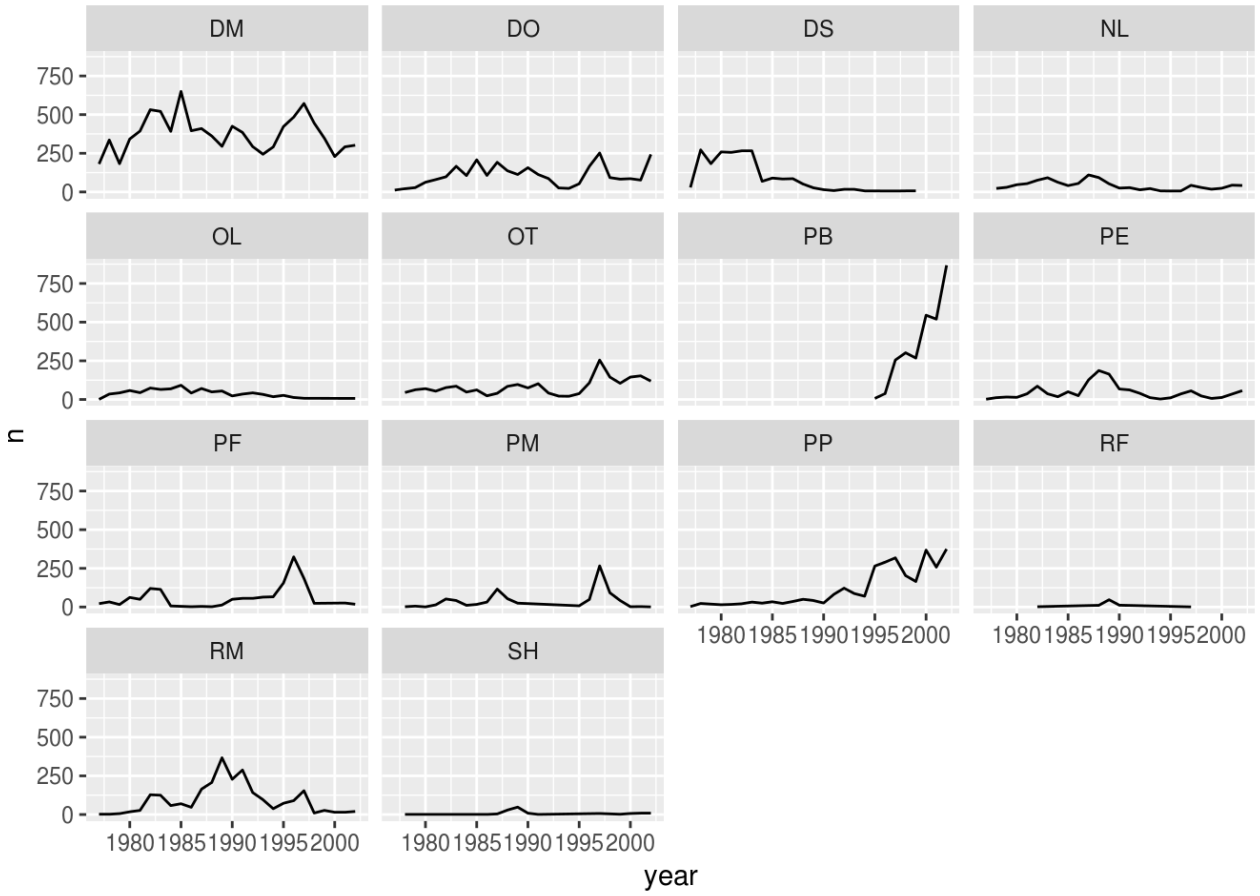
```
ggplot(data = yearly_counts, mapping = aes(x = year, y = n, color =
   species_id)) +
    geom_line()
```



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)
```

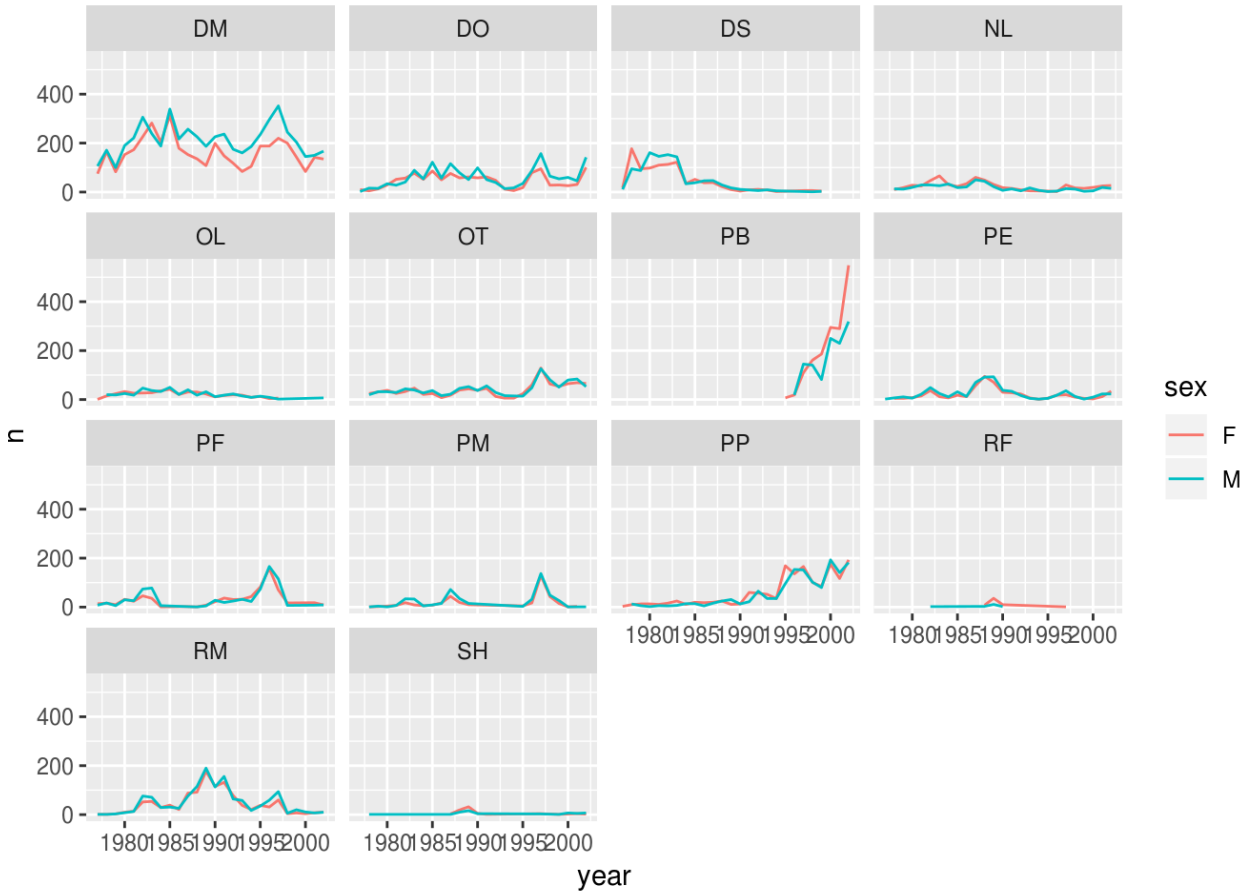


Now we would like to split the line in each plot by the sex of each individual measured. To do that we need to make counts in the data frame grouped by `year`, `species_id`, and `sex`:

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

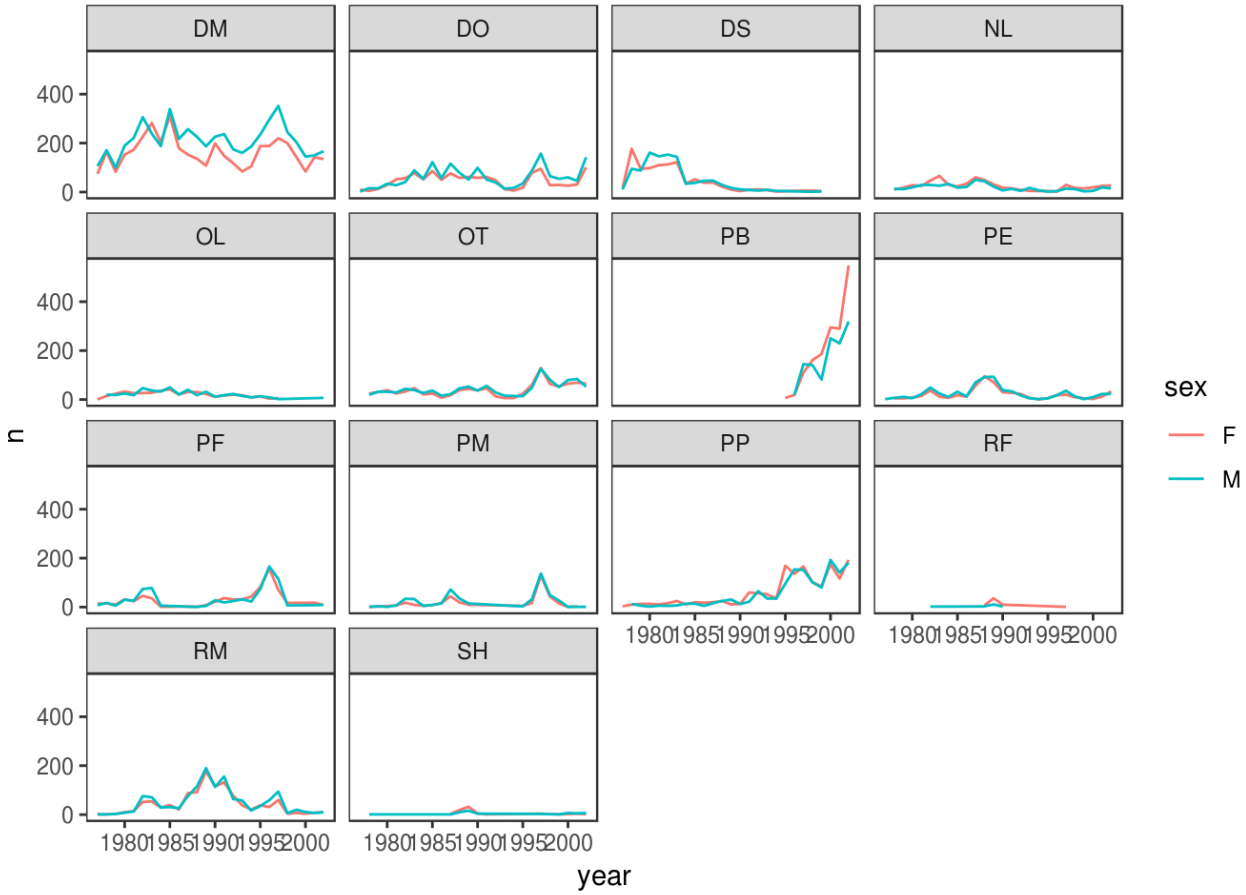
We can now make the faceted plot by splitting further by sex using `color` (within a single plot):

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



`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 (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 (https://jrnold.github.io/ggthemes/reference/index.html) package provides a wide variety of options (including an Excel 2003 theme). The **`ggplot2`** extensions website (https://www.ggplot2-exts.org) provides a list of packages that extend the capabilities of **`ggplot2`**, including additional themes.

Challenge

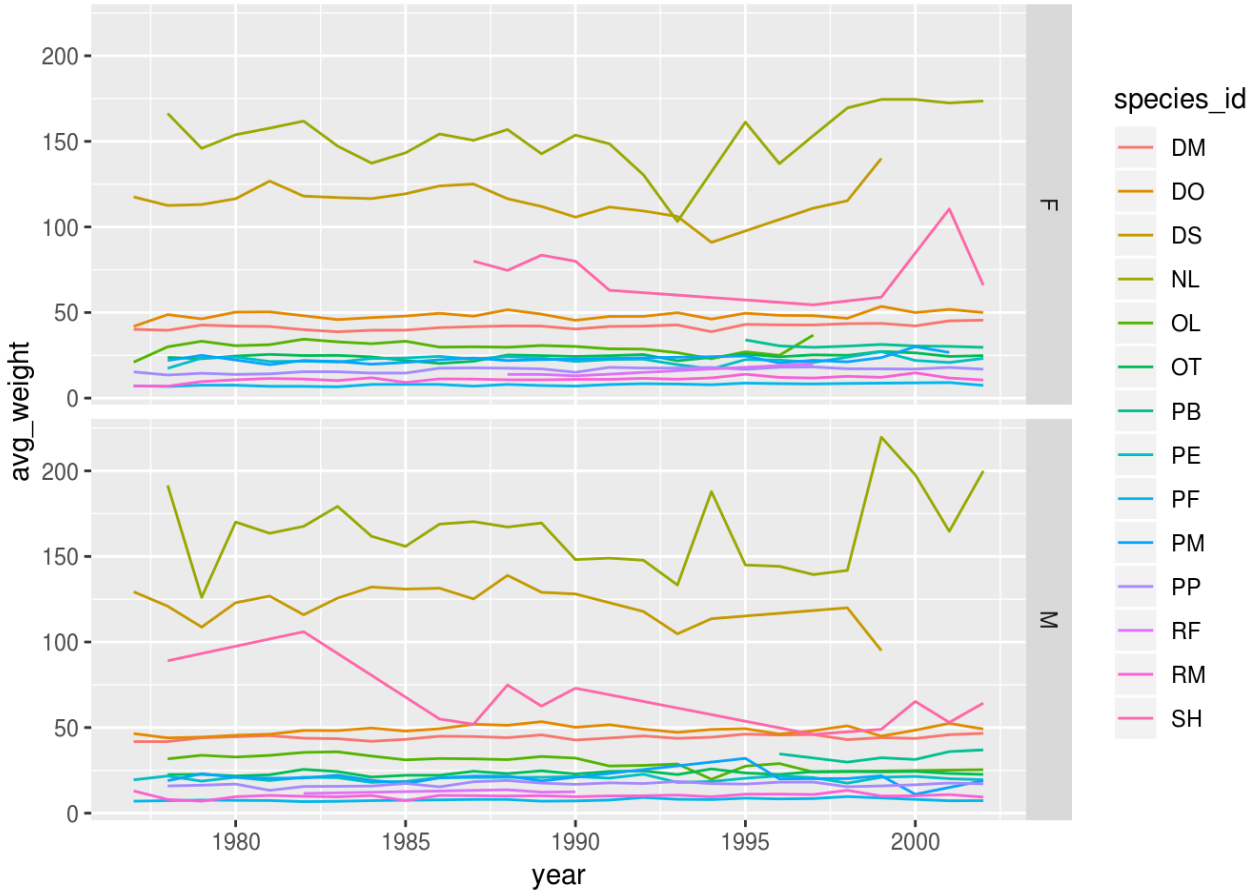
Use what you just learned to create a plot that depicts how the average weight of each species changes through the years.

Answer

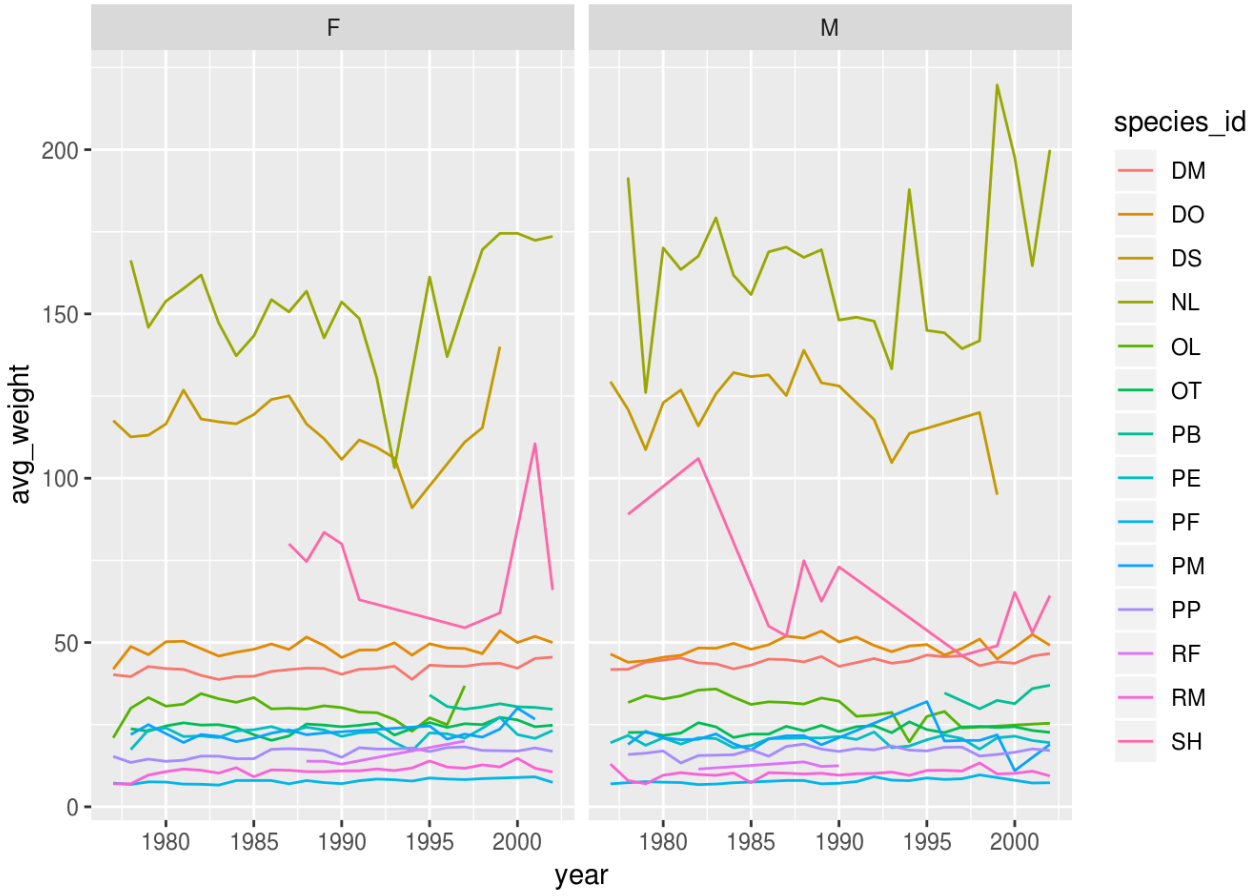
The `facet_wrap` geometry extracts plots into an arbitrary number of dimensions to allow them to cleanly fit on one page. On the other hand, the `facet_grid` geometry allows you to explicitly specify how you want your plots to be arranged via formula notation (`rows ~ columns` ; a `.` can be used as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

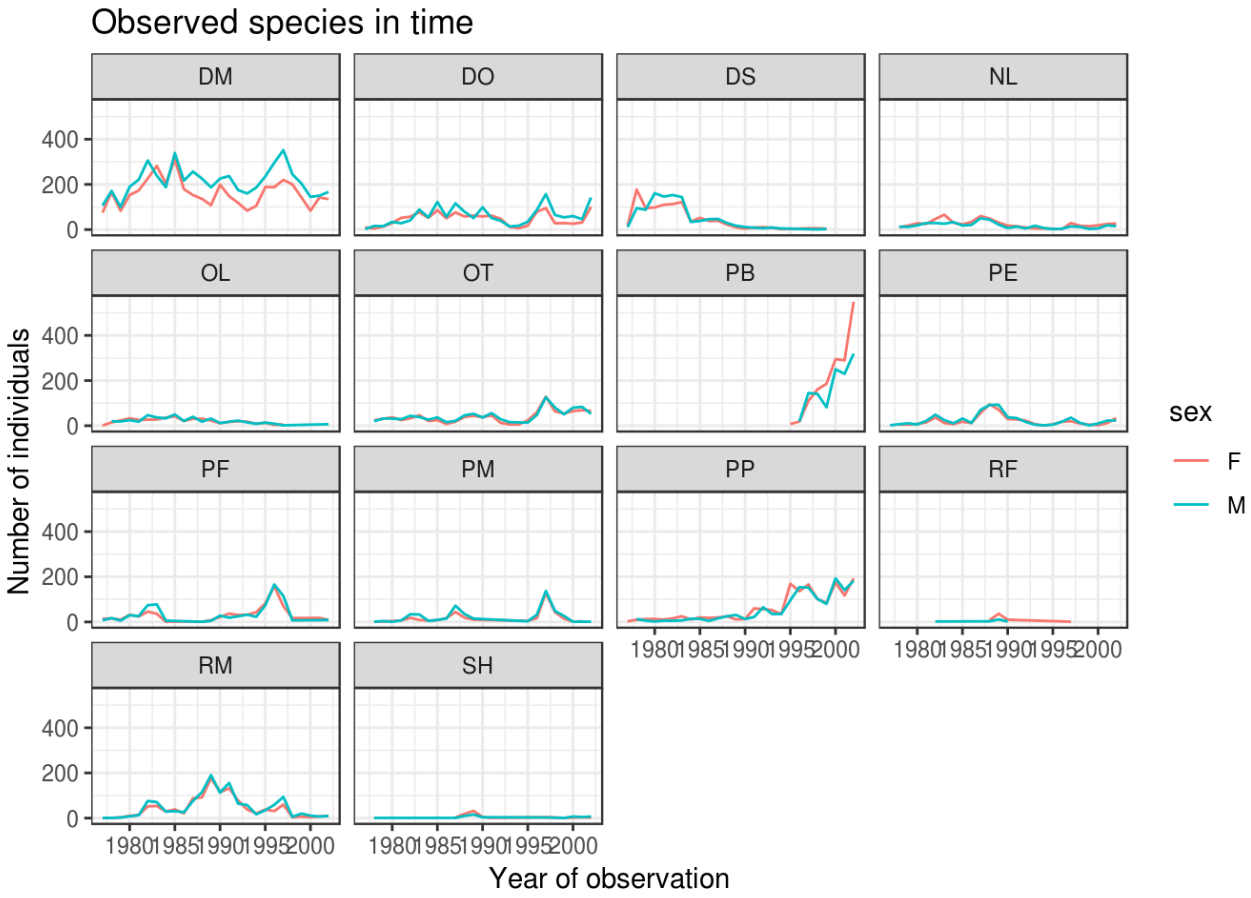


Customization

Take a look at the **ggplot2** cheat sheet (https://www.rstudio.com/wp-content/uploads/2016/11/ggplot2-cheatsheet-2.1.pdf), and think of ways you could improve the plot.

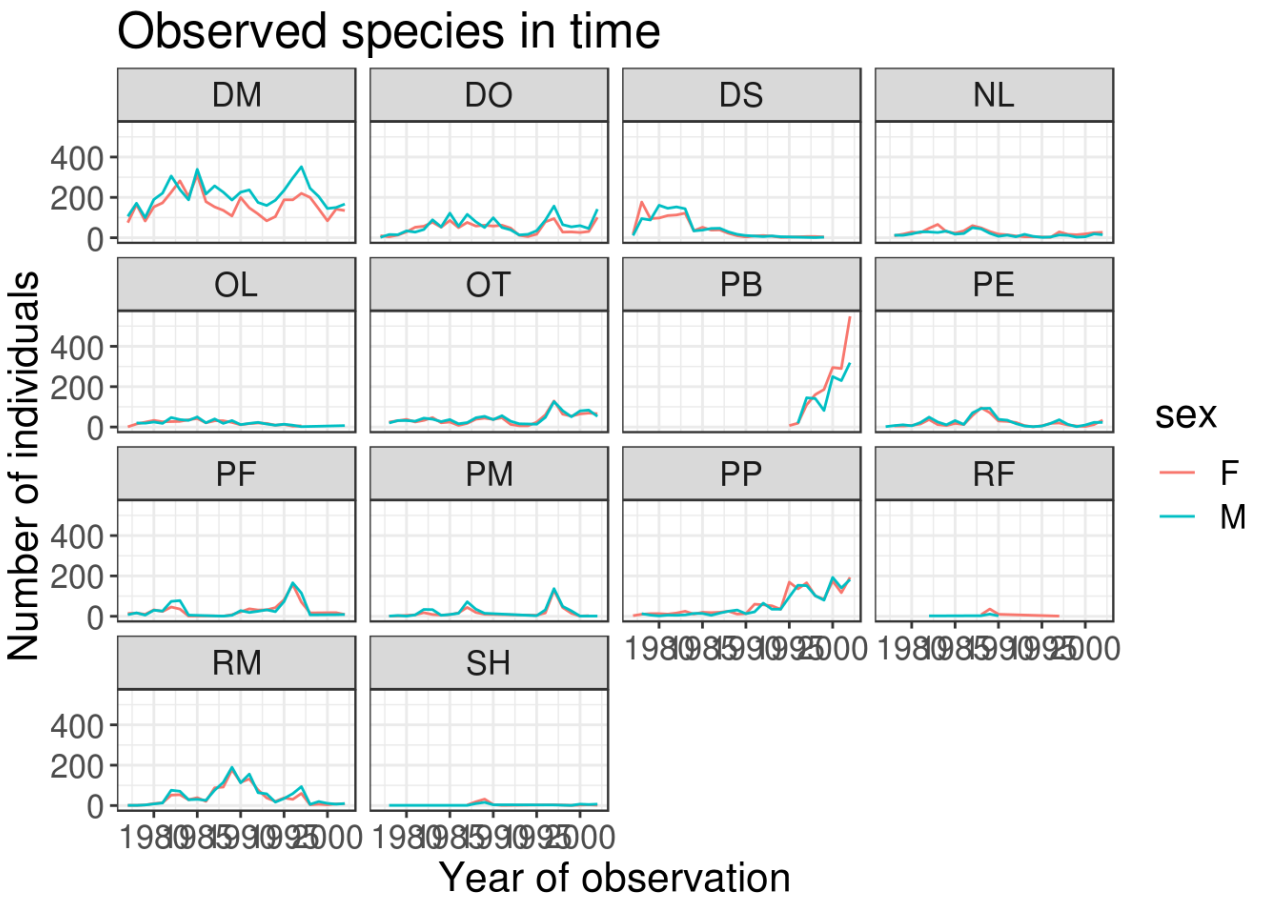
Now, let's change names of axes to something more informative than 'year' and 'n' and add a title to the figure:

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



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

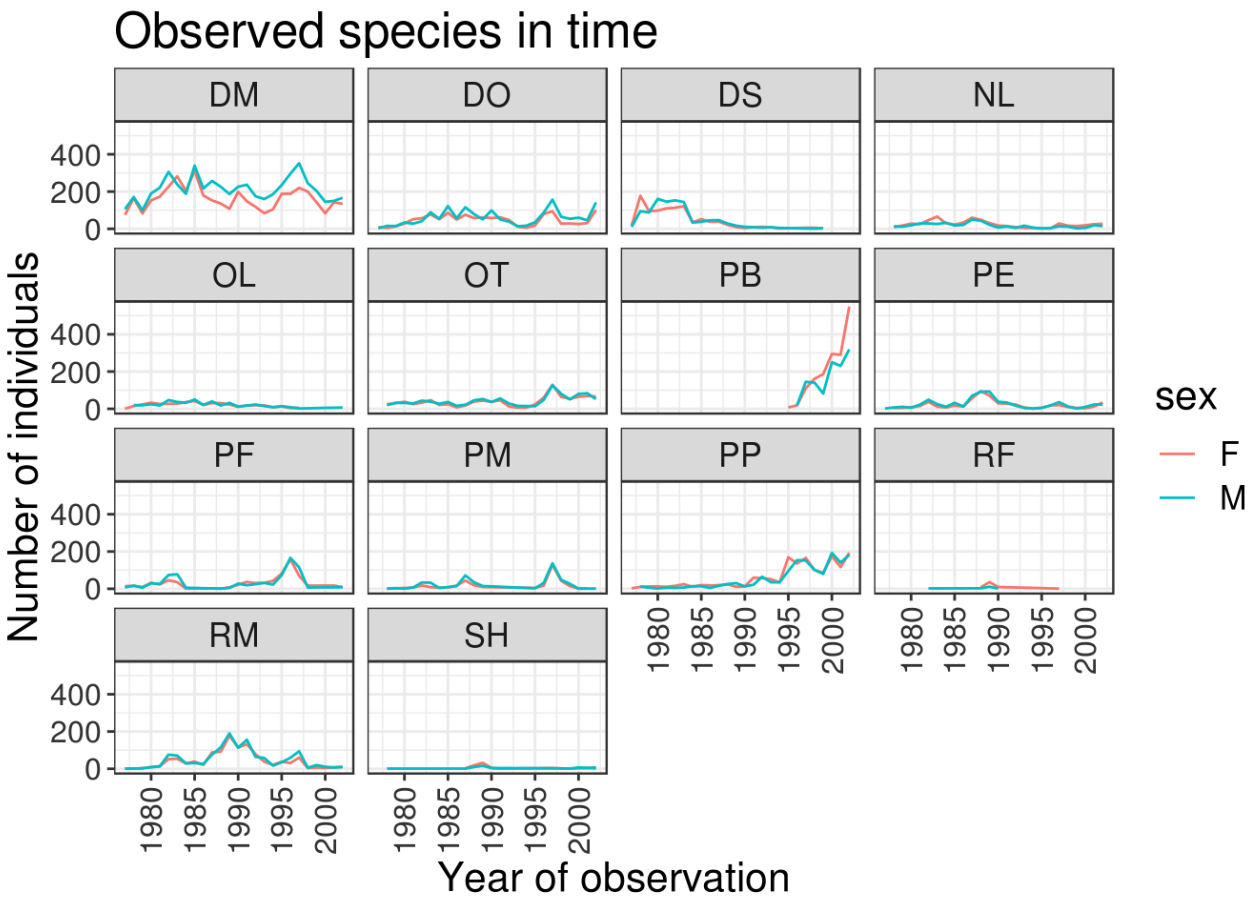
```
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, colo
  r = sex)) +
    geom_line() +
    facet_wrap(~ species_id) +
    labs(title = "Observed species in time",
        x = "Year of observation",
        y = "Number of individuals") +
    theme_bw() +
    theme(text=element_text(size = 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 (https://github.com/wch/extrafont), and follow the instructions included in the README for this package.

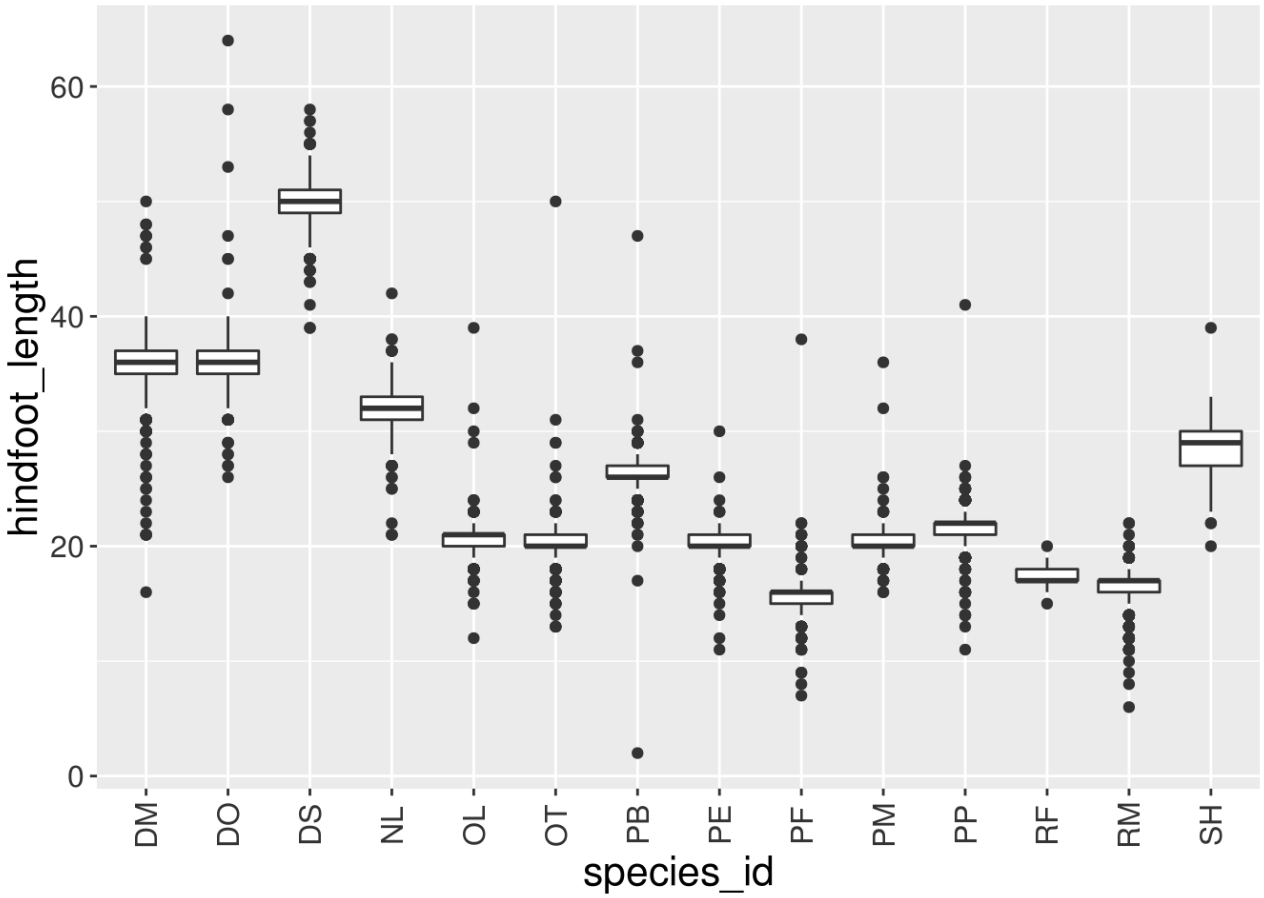
After our manipulations, you may notice that the values on the x-axis are still not properly readable. Let's change the orientation of the labels and adjust them vertically and horizontally so they don't overlap. You can use a 90-degree angle, or experiment to find the appropriate angle for diagonally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

```
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, colo
  r = sex)) +
    geom_line() +
    facet_wrap(~ species_id) +
    labs(title = "Observed species in time",
         x = "Year of observation",
         y = "Number of individuals") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, a
  ngle = 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", si
  ze = 12, angle = 90, hjust = 0.5, vjust = 0.5),
                      axis.text.y = element_text(colour = "grey2
  0", size = 12),
                      text = element_text(size = 16))
ggplot(surveys_complete, aes(x = species_id, y = hindfoot_length)) +
    geom_boxplot() +
    grey_theme
```



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 (https://www.rstudio.com/wp-content/uploads/2016/11/ggplot2-cheatsheet-2.1.pdf) 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)/ (http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/)).

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 = weigh
  t)) +
  geom_boxplot() +
  xlab("Species") + ylab("Weight (g)") +
  scale_y_log10()

spp_count_plot <- ggplot(data = yearly_counts,
                         mapping = aes(x = year, y = n, color = spec
  ies_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 (https://cran.r-project.org/web/packages/gridExtra/vignettes/arrangeGrob.html).

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 individuals") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, a
  ngle = 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, heig
  ht = 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.

Page built on: 2019-02-08 – 18:25:30

SQL databases and R

Data Carpentry contributors

Learning Objectives

- Access a database from R.
- Run SQL queries in R using **`RSQLite`** and **`dplyr`**.
- Create an SQLite database from existing .csv files.

Introduction

So far, we have dealt with small datasets that easily fit into your computer's memory. But what about datasets that are too large for your computer to handle as a whole? In this case, storing the data outside of R and organizing it in a database is helpful. Connecting to the database allows you to retrieve only the chunks needed for the current analysis.

Even better, many large datasets are already available in public or private databases. You can query them without having to download the data first.

R can connect to almost any existing database type. Most common database types have R packages that allow you to connect to them (e.g., **RSQLite**, RMySQL, etc). Furthermore, the **dplyr** (https://cran.r-project.org/web/packages/dplyr/index.html) package you used in the previous chapter, in conjunction with **dbplyr** (https://cran.r-project.org/package=dbplyr) supports connecting to the widely-used open source databases sqlite (https://sqlite.org/), mysql (https://www.mysql.com/) and postgresql (https://www.postgresql.org/), as well as Google's bigquery (https://cloud.google.com/bigquery/), and it can also be extended to other database types (a vignette (https://cran.r-project.org/web/packages/dbplyr/vignettes/new-backend.html) in the **dplyr** package explains how to do it). RStudio has created a website (http://db.rstudio.com/) that provides documentation and best practices to work on database interfaces.

Interfacing with databases using **dplyr** focuses on retrieving and analyzing datasets by generating `SELECT` SQL statements, but it doesn't modify the database itself. **dplyr** does not offer functions to `UPDATE` or `DELETE` entries. If you need these functionalities, you will need to use additional R packages (e.g., **RSQLite**). Here we will demonstrate how to interact with a database using **dplyr**, using both the **dplyr**'s verb syntax and the SQL syntax.

The portal_mammals database

We will continue to explore the `surveys` data you are already familiar with from previous lessons. First, we are going to install the **dbplyr** package:

```
install.packages(c("dbplyr", "RSQLite"))
```

The SQLite database is contained in a single file `portal_mammals.sqlite` that you generated during the SQL lesson. If you don't have it, you can download it from Figshare into the `data` subdirectory using:

```
dir.create("data", showWarnings = FALSE)
download.file(url = "https://ndownloader.figshare.com/files/2292171"
  ,
              destfile = "data/portal_mammals.sqlite", mode = "wb")
```

Connecting to databases

We can point R to this database using:

```
library(dplyr)
library(dbplyr)
```

```
#>
#> Attaching package: 'dbplyr'
```

```
#> The following objects are masked from 'package:dplyr':
#>
#>     ident, sql
```

```
mammals <- DBI::dbConnect(RSQLite::SQLite(), "data/portal_mammals.sq
  lite")
```

This command uses 2 packages that helps **dbplyr** and **dplyr** talk to the SQLite database. **DBI** is not something that you'll use directly as a user. It allows R to send commands to databases irrespective of the database management system used. The **RSQLite** package allows R to interface with SQLite databases.

This command does not load the data into the R session (as the `read_csv()` function did). Instead, it merely instructs R to connect to the `SQLite` database contained in the `portal_mammals.sqlite` file.

Using a similar approach, you could connect to many other database management systems that are supported by R including MySQL, PostgreSQL, BigQuery, etc.

Let's take a closer look at the `mammals` database we just connected to:

```
src_dbi(mammals)
```

```
#> src:  sqlite 3.22.0 [/home/travis/build/datacarpentry/R-ecology-l
esson/data/portal_mammals.sqlite]
#> tbls: plots, species, surveys
```

Just like a spreadsheet with multiple worksheets, a SQLite database can contain multiple tables. In this case three of them are listed in the `tbls` row in the output above:

- plots
- species
- surveys

Now that we know we can connect to the database, let's explore how to get the data from its tables into R.

Querying the database with the SQL syntax

To connect to tables within a database, you can use the `tbl()` function from **`dplyr`**. This function can be used to send SQL queries to the database. To demonstrate this functionality, let's select the columns "year", "species_id", and "plot_id" from the `surveys` table:

```
tbl(mammals, sql("SELECT year, species_id, plot_id FROM surveys"))
```

With this approach you can use any of the SQL queries we have seen in the database lesson.

Querying the database with the dplyr syntax

One of the strengths of **`dplyr`** is that the same operation can be done using **`dplyr`**'s verbs instead of writing SQL. First, we select the table on which to do the operations by creating the `surveys` object, and then we use the standard **`dplyr`** syntax as if it were a data frame:

```
surveys <- tbl(mammals, "surveys")
surveys %>%
    select(year, species_id, plot_id)
```

In this case, the `surveys` object behaves like a data frame. Several functions that can be used with data frames can also be used on tables from a database. For instance, the `head()` function can be used to check the first 10 rows of the table:

```
head(surveys, n = 10)
```

```
#> # Source:   lazy query [?? x 9]
#> # Database: sqlite 3.22.0
#> #   [/home/travis/build/datacarpentry/R-ecology-lesson/data/porta
l_mammals.sqlite]
#>    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         2     7    16  1977       3 NL         M
33
#>  3         3     7    16  1977       2 DM         F
37
#>  4         4     7    16  1977       7 DM         M
36
#>  5         5     7    16  1977       3 DM         M
35
#>  6         6     7    16  1977       1 PF         M
14
#>  7         7     7    16  1977       2 PE         F
NA
#>  8         8     7    16  1977       1 DM         M
37
#>  9         9     7    16  1977       1 DM         F
34
#> 10        10     7    16  1977       6 PF         F
20
#> # … with 1 more variable: weight <int>
```

This output of the `head` command looks just like a regular `data.frame`: The table has 9 columns and the `head()` command shows us the first 10 rows. Note that the columns `plot_type`, `taxa`, `genus`, and `species` are missing. These are now located in the tables `plots` and `species` which we will join together in a moment.

However, some functions don't work quite as expected. For instance, let's check how many rows there are in total using `nrow()`:

```
nrow(surveys)
```

```
#> [1] NA
```

That's strange - R doesn't know how many rows the `surveys` table contains - it returns `NA` instead. You might have already noticed that the first line of the `head()` output included `??` indicating that the number of rows wasn't known.

The reason for this behavior highlights a key difference between using **`dplyr`** on datasets in memory (e.g. loaded into your R session via `read_csv()`) and those provided by a database. To understand it, we take a closer look at how **`dplyr`** communicates with our SQLite database.

SQL translation

Relational databases typically use a special-purpose language, Structured Query Language (SQL) (https://en.wikipedia.org/wiki/SQL), to manage and query data.

For example, the following SQL query returns the first 10 rows from the `surveys` table:

```
SELECT *
FROM `surveys`
LIMIT 10
```

Behind the scenes, **`dplyr`**:

1. translates your R code into SQL
2. submits it to the database
3. translates the database's response into an R data frame

To lift the curtain, we can use **`dplyr`**'s `show_query()` function to show which SQL commands are actually sent to the database:

```
show_query(head(surveys, n = 10))
```

```
#> <SQL>
#> SELECT *
#> FROM `surveys`
#> LIMIT 10
```

The output shows the actual SQL query sent to the database; it matches our manually constructed `SELECT` statement above.

Instead of having to formulate the SQL query ourselves - and having to mentally switch back and forth between R and SQL syntax - we can delegate this translation to **`dplyr`**. (You don't even need to know SQL to interact with a database via **`dplyr`**!)

`dplyr`, in turn, doesn't do the real work of subsetting the table, either. Instead, it merely sends the query to the database, waits for its response and returns it to us.

That way, R never gets to see the full `surveys` table - and that's why it could not tell us how many rows it contains. On the bright side, this allows us to work with large datasets - even too large to fit into our computer's memory.

`dplyr` can translate many different query types into SQL allowing us to, e.g., `select()` specific columns, `filter()` rows, or join tables.

To see this in action, let's compose a few queries with **`dplyr`**.

Simple database queries

First, let's only request rows of the `surveys` table in which `weight` is less than 5 and keep only the species_id, sex, and weight columns.

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

```
#> # Source:   lazy query [?? x 3]
#> # Database: sqlite 3.22.0
#> #   [/home/travis/build/datacarpentry/R-ecology-lesson/data/porta
l_mammals.sqlite]
#>    species_id sex   weight
#>    <chr>      <chr>  <int>
#>  1 PF         M          4
#>  2 PF         F          4
#>  3 PF         <NA>       4
#>  4 PF         F          4
#>  5 PF         F          4
#>  6 RM         M          4
#>  7 RM         F          4
#>  8 RM         M          4
#>  9 RM         M          4
#> 10 RM         M          4
#> # … with more rows
```

Executing this command will return a table with 10 rows and the requested `species_id`, `sex` and `weight` columns. Great!

… but wait, why are there only 10 rows?

The last line:

```
# ... with more rows
```

indicates that there are more results that fit our filtering criterion. Why was R lazy and only retrieved 10 of them?

Laziness

Hadley Wickham, the author of **dplyr** explains (https://cran.r-project.org/web/packages/dbplyr/vignettes/dbplyr.html):

> When working with databases, **dplyr** tries to be as lazy as possible:
>
> - It never pulls data into R unless you explicitly ask for it.
> - It delays doing any work until the last possible moment - it collects together everything you want to do and then sends it to the database in one step.

When you construct a **dplyr** query, you can connect multiple verbs into a single pipeline. For example, we combined the `filter()` and `select()` verbs using the `%>%` pipe.

If we wanted to, we could add on even more steps, e.g. remove the `sex` column in an additional `select` call:

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

data_subset %>%
  select(-sex)
```

```
#> # Source:   lazy query [?? x 2]
#> # Database: sqlite 3.22.0
#> #   [/home/travis/build/datacarpentry/R-ecology-lesson/data/portal_mammals.sqlite]
#>    species_id weight
#>    <chr>       <int>
#>  1 PF              4
#>  2 PF              4
#>  3 PF              4
#>  4 PF              4
#>  5 PF              4
#>  6 RM              4
#>  7 RM              4
#>  8 RM              4
#>  9 RM              4
#> 10 RM              4
#> # … with more rows
```

Just like the first `select(species_id, sex, weight)` call, the `select(-sex)` command is not executed by R. It is sent to the database instead. Only the *final* result is retrieved and displayed to you.

Of course, we could always add on more steps, e.g., we could filter by `species_id` or minimum `weight`. That's why R doesn't retrieve the full set of results - instead it only retrieves the first 10 results from the database by default. (After all, you might want to add an additional step and get the database to do more work…)

To instruct R to stop being lazy, e.g. to retrieve all of the query results from the database, we add the `collect()` command to our pipe. It indicates that our database query is finished: time to get the *final* results and load them into the R session.

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

Now we have all 17 rows that match our query in a `data.frame` and can continue to work with them exclusively in R, without communicating with the database.

Complex database queries

`dplyr` enables database queries across one or multiple database tables, using the same single- and multiple-table verbs you encountered previously. This means you can use the same commands regardless of whether you interact with a remote database or local dataset! This is a really useful feature if you work with large datasets: you can first prototype your code on a small subset that fits into memory, and when your code is ready, you can change the input dataset to your full database without having to change the syntax.

On the other hand, being able to use SQL queries directly can be useful if your collaborators have already put together complex queries to prepare the dataset that you need for your analysis.

To illustrate how to use **`dplyr`** with these complex queries, we are going to join the `plots` and `surveys` tables. The `plots` table in the database contains information about the different plots surveyed by the researchers. To access it, we point the `tbl()` command to it:

```
plots <- tbl(mammals, "plots")
plots
```

```
#> # Source:   table<plots> [?? x 2]
#> # Database: sqlite 3.22.0
#> #   [/home/travis/build/datacarpentry/R-ecology-lesson/data/porta
l_mammals.sqlite]
#>    plot_id plot_type
#>      <int> <chr>
#>  1       1 Spectab exclosure
#>  2       2 Control
#>  3       3 Long-term Krat Exclosure
#>  4       4 Control
#>  5       5 Rodent Exclosure
#>  6       6 Short-term Krat Exclosure
#>  7       7 Rodent Exclosure
#>  8       8 Control
#>  9       9 Spectab exclosure
#> 10      10 Rodent Exclosure
#> # … with more rows
```

The `plot_id` column also features in the `surveys` table:

```
surveys
```

```
#> # Source:   table<surveys> [?? x 9]
#> # Database: sqlite 3.22.0
#> #   [/home/travis/build/datacarpentry/R-ecology-lesson/data/porta
l_mammals.sqlite]
#>    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         2     7    16  1977       3 NL         M
33
#>  3         3     7    16  1977       2 DM         F
37
#>  4         4     7    16  1977       7 DM         M
36
#>  5         5     7    16  1977       3 DM         M
35
#>  6         6     7    16  1977       1 PF         M
14
#>  7         7     7    16  1977       2 PE         F
NA
#>  8         8     7    16  1977       1 DM         M
37
#>  9         9     7    16  1977       1 DM         F
34
#> 10        10     7    16  1977       6 PF         F
20
#> # … with more rows, and 1 more variable: weight <int>
```

Because `plot_id` is listed in both tables, we can use it to look up matching records, and join the two tables.

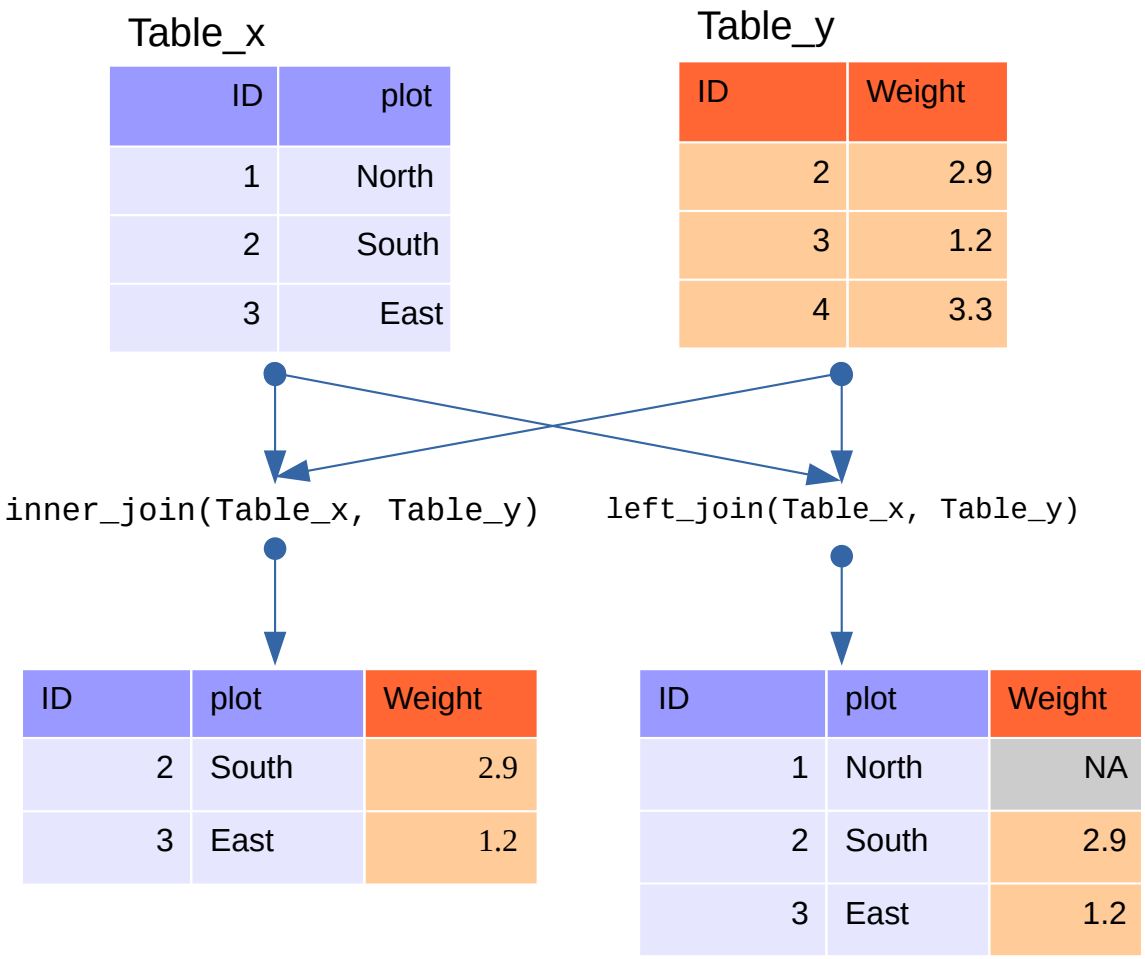


diagram illustrating inner and left joins

For example, to extract all surveys for the first plot, which has `plot_id` 1, we can do:

```
plots %>%
  filter(plot_id == 1) %>%
  inner_join(surveys) %>%
  collect()
```

```
#> Joining, by = "plot_id"
```

```
#> # A tibble: 1,995 x 10
#>    plot_id plot_type record_id month   day  year species_id sex
#>      <int> <chr>         <int> <int> <int> <int> <chr>      <chr>
#>  1       1 Spectab …         6     7    16  1977 PF         M
#>  2       1 Spectab …         8     7    16  1977 DM         M
#>  3       1 Spectab …         9     7    16  1977 DM         F
#>  4       1 Spectab …        78     8    19  1977 PF         M
#>  5       1 Spectab …        80     8    19  1977 DS         M
#>  6       1 Spectab …       218     9    13  1977 PF         M
#>  7       1 Spectab …       222     9    13  1977 DS         M
#>  8       1 Spectab …       239     9    13  1977 DS         M
#>  9       1 Spectab …       263    10    16  1977 DM         M
#> 10       1 Spectab …       270    10    16  1977 DM         F
#> # … with 1,985 more rows, and 2 more variables: hindfoot_length <
int>,
#> #   weight <int>
```

Important Note: Without the `collect()` statement, only the first 10 matching rows are returned. By adding `collect()`, the full set of 1,985 is retrieved.

Challenge

Write a query that returns the number of rodents observed in each plot in each year.

Hint: Connect to the species table and write a query that joins the species and survey tables together to exclude all non-rodents. The query should return counts of rodents by year.

Optional: Write a query in SQL that will produce the same result. You can join multiple tables together using the following syntax where foreign key refers to your unique id (e.g., `species_id`):

```
SELECT table.col, table.col
FROM table1 JOIN table2
ON table1.key = table2.key
JOIN table3 ON table2.key = table3.key
```

Answer

Challenge

Write a query that returns the total number of rodents in each genus caught in the different plot types.

Hint: Write a query that joins the species, plot, and survey tables together. The query should return counts of genus by plot type.

Answer

This is useful if we are interested in estimating the number of individuals belonging to each genus found in each plot type. But what if we were interested in the number of genera found in each plot type? Using `tally()` gives the number of individuals, instead we need to use `n_distinct()` to count the number of unique values found in a column.

```
species <- tbl(mammals, "species")
unique_genera <- left_join(surveys, plots) %>%
    left_join(species) %>%
    group_by(plot_type) %>%
    summarize(
        n_genera = n_distinct(genus)
    ) %>%
    collect()
```

```
#> Joining, by = "plot_id"
```

```
#> Joining, by = "species_id"
```

`n_distinct`, like the other **`dplyr`** functions we have used in this lesson, works not only on database connections but also on regular data frames.

Creating a new SQLite database

So far, we have used a previously prepared SQLite database. But we can also use R to create a new database, e.g. from existing `csv` files. Let's recreate the mammals database that we've been working with, in R. First let's download and read in the `csv` files. We'll import **`tidyverse`** to gain access to the `read_csv()` function.

```
download.file("https://ndownloader.figshare.com/files/3299483",
              "data/species.csv")
download.file("https://ndownloader.figshare.com/files/10717177",
              "data/surveys.csv")
download.file("https://ndownloader.figshare.com/files/3299474",
              "data/plots.csv")
library(tidyverse)
species <- read_csv("data/species.csv")
```

```
#> Parsed with column specification:
#> cols(
#>   species_id = col_character(),
#>   genus = col_character(),
#>   species = col_character(),
#>   taxa = col_character()
#> )
```

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

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

```
plots <- read_csv("data/plots.csv")
```

```
#> Parsed with column specification:
#> cols(
#>   plot_id = col_double(),
#>   plot_type = col_character()
#> )
```

Creating a new SQLite database with **dplyr** is easy. You can re-use the same command we used above to open an existing `.sqlite` file. The `create = TRUE` argument instructs R to create a new, empty database instead.

Caution: When `create = TRUE` is added, any existing database at the same location is overwritten *without warning*.

```
my_db_file <- "data_output/portal-database-output.sqlite"
my_db <- src_sqlite(my_db_file, create = TRUE)
```

Currently, our new database is empty, it doesn't contain any tables:

```
my_db
```

```
#> src:  sqlite 3.22.0 [data_output/portal-database-output.sqlite]
#> tbls:
```

To add tables, we copy the existing data.frames into the database one by one:

```
copy_to(my_db, surveys)
copy_to(my_db, plots)
my_db
```

If you check the location of our database you'll see that data is automatically being written to disk. R and **dplyr** not only provide easy ways to query existing databases, they also allows you to easily create your own databases from flat files!

Challenge

Add the remaining species table to the `my_db` database and run some of your queries from earlier in the lesson to verify that you have faithfully recreated the mammals database.

Note: In this example, we first loaded all of the data into the R session by reading the three `csv` files. Because all the data has to flow through R, this is not suitable for very large datasets.

Page built on: 2019-02-08 – 18:25:37

(https://github.com/datacarpentry/R-ecology-lesson/issues/new).
On Twitter: @datacarpentry (https://twitter.com/datacarpentry)