R Workshop UBC Hakai

Brett Johnson

1/18/2019

Contents

1 Did you do your homework?

Did you successfully install and set up all the tools necessary for this workshop? If not do not pass go, do not collect $200. See README

2 Introduction

Why am I giving this workshop?

- Facilitate the process of getting specific results from University affiliates back into the Hakai Ecological Information Management Database, and Hakai Data Catalogue.
- Reduce the amount of data processing strain on Hakai by providing training to others
- Take a team based approach to move the lab forward with a modern, collaborative, and reproducible set of methods
- Make your lives as researchers easier and more fun
- Better science less time

2.1 Filling in the gaps

Most University courses that teach statistics and data analysis focus on teaching statistical techniques but they pay little attention to the tools, workflows, and data wrangling skills required to actually conduct an analysis from start to finish. Questions that often remain un-answered include:

What questions remain for them about how to do a data analysis?

- What is an efficient workflow?
- How do I access and import data?
- How do I clean and manipulate my data into a format to analyze?
- How can I re-run my analysis in case I get new data or someone else wants to run my code?
- How can I collaborate on this analysis?
- How can I get my analysis into a format for someone to meaningfully conduct peer-review?
- How do I efficiently produce a professional report or other artifact of my analysis to communicate results?

2.2 Objectives

The objectives of this workshop are:

- Become familiar with tools to manage and analyze data efficiently
- Write code in R-Studio
- Use `tidyverse`, `dplyr`, `ggplot2` and `tidyr` R packages to analyze your data
- Use Git and GitHub version control and a changelog
- Create reproducible analyses
- You can easily re-run your code and analysis when you get new data
- Others can easily find your code and understand the steps you took to analyze your data
- Raw data are available

- Produce well-documented tidy data-sets that have excellent provenance
- Data sets contain a change log with a version history of what has changed and the steps used to process data
- All variables in the dataset are defined in a data dictionary
- Laboratory, and analytical methods are thoroughly described

2.3 Background

New ideas about what makes a good data analysis are emerging. With data being so readily available in vast quantities, analyzing data using out of date methods — such as Excel — can quickly become overwhelming, not reproducible, error-prone, and difficult to assess for reliability.

'Duke Scandal'

Science is in a reproducibility crisis.

If we want the work we do to have the greatest impact, others must be able to *easily* reproduce the work so that they can buildon it. That's what we're trying to do at Hakai.

Some important concepts in defining a good data analysis are:

1) Reproducible Research;
2) Open Science Collaboration

- An additional peer review
- Collaborate with future you
- Literate Programming
- Nothing to hide; increased reliablity or trustworthyness
- You can share your analyses in hopes that others will improve them

2.4 About me

- BSc. in Wildlife and Fisheries
- Manage the Juvenile Salmon Program at Hakai
- Work with in the IT department as well
- Competent practitioner
- Task oriented, process driven
- I love 'Aha! moments' and making researchers work easier

Throughought the course I will ask for a helper for each section. If you have some previous experience in a topic, please volunteer to help people out!

Sticky Notes!

2.5 Before We Start

2.5.1 Learning Objectives

- Describe the purpose of the RStudio Script, Console, Environment, and Plots panes.
- Organize files and directories for a set of analyses as an R Project, and understand the purpose of the working directory.

- Use the built-in RStudio help interface to search for more information on R functions.
- Demonstrate how to provide sufficient information for troubleshooting with the R user community. ————

2.6 What is R? What is RStudio?

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

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

2.7 Why learn R?

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

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

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

2.7.2 R code is great for reproducibility

2.7.3 R is interdisciplinary and extensible

Academics write packages, an extension of R. For instance, R has packages for image analysis, GIS, time series, population genetics, and a lot more.

2.7.4 R produces high-quality graphics

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

2.7.5 R has a large and welcoming community

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

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

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

2.8 Knowing your way around RStudio

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

Open up R-Studio.

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).

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.

2.9 Data Visualization Teaser

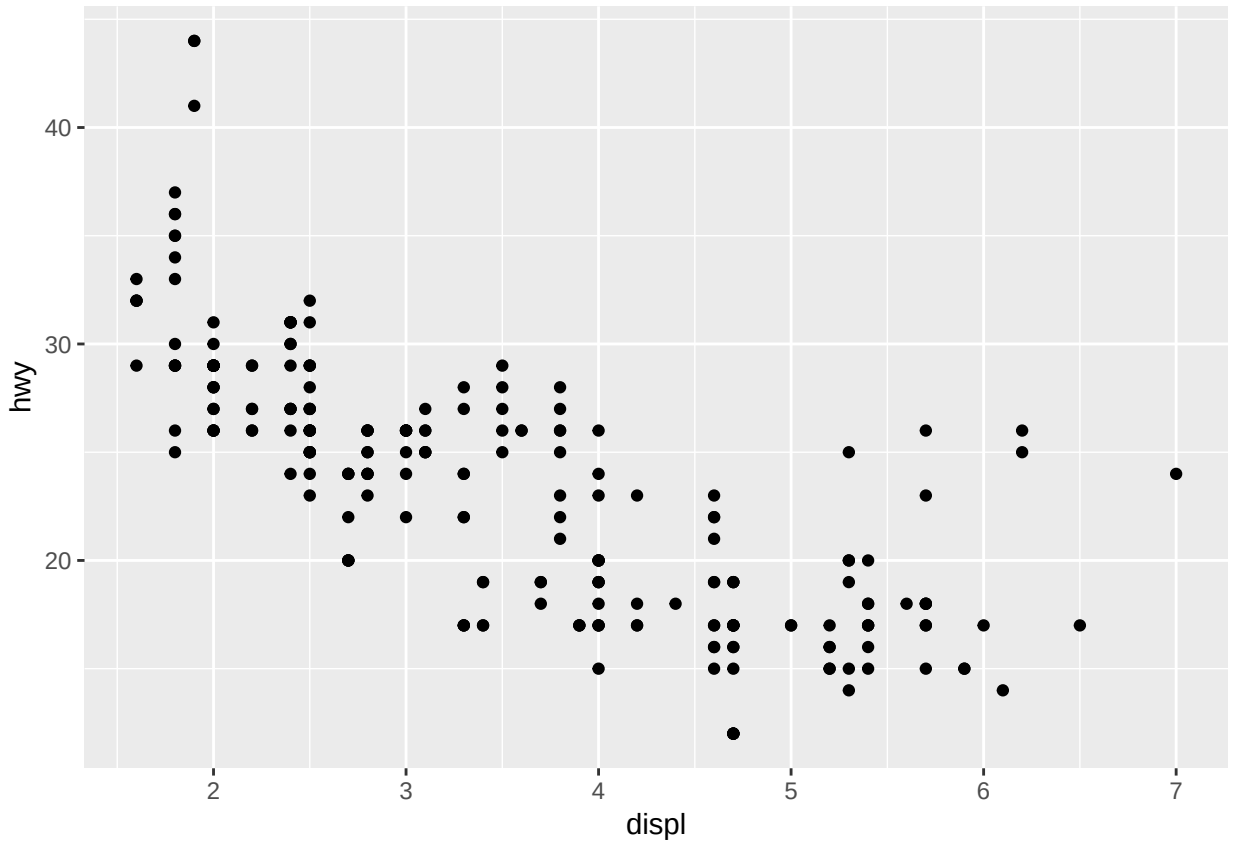
Before getting into details let's have a sneak preview to what's possible and where we are heading.

In the R-Studio console type out the following commands with me.

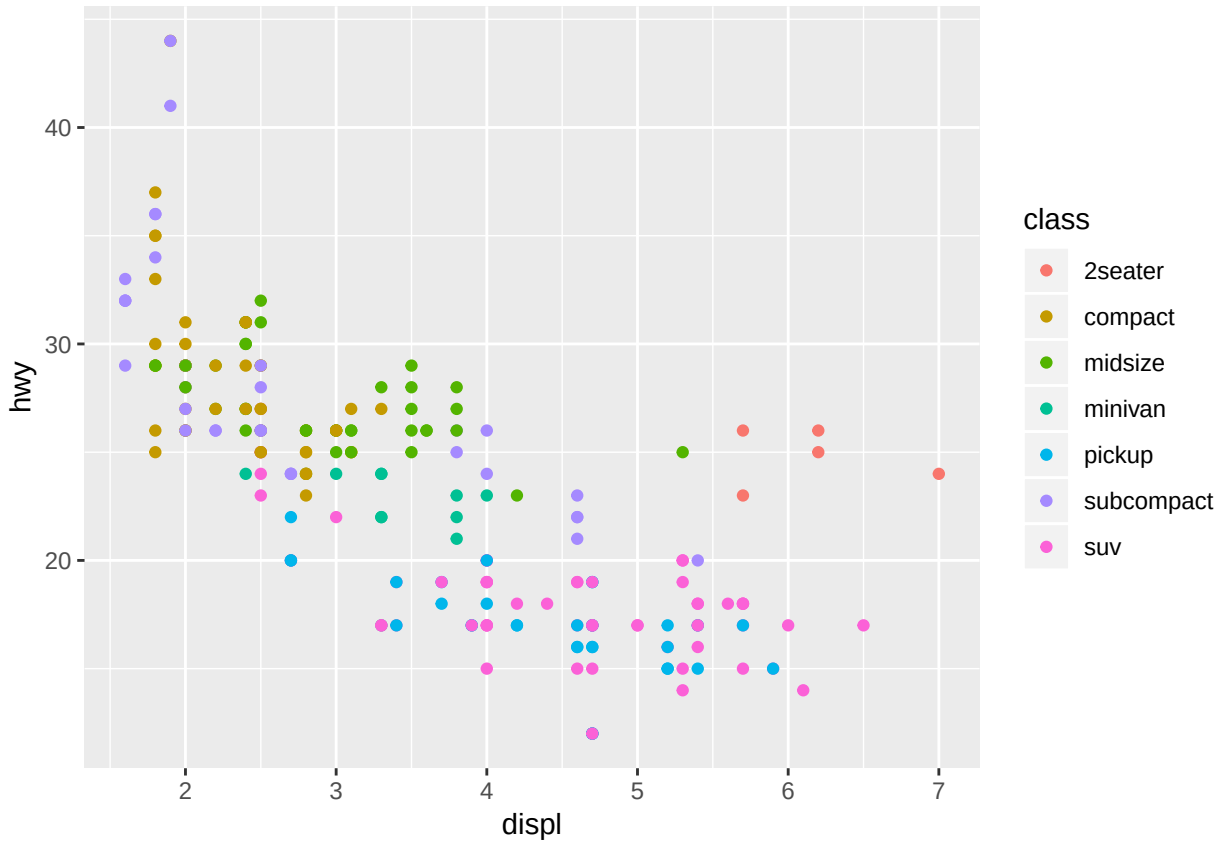
```
library(tidyverse)

# The tidyverse library has a data frame object called mpg, it's about cars.
# Check it out
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4      1.8  1999     4 auto~ f        18    29 p     comp~
##  2 audi         a4      1.8  1999     4 manu~ f        21    29 p     comp~
##  3 audi         a4      2    2008     4 manu~ f        20    31 p     comp~
##  4 audi         a4      2    2008     4 auto~ f        21    30 p     comp~
##  5 audi         a4      2.8  1999     6 auto~ f        16    26 p     comp~
##  6 audi         a4      2.8  1999     6 manu~ f        18    26 p     comp~
##  7 audi         a4      3.1  2008     6 auto~ f        18    27 p     comp~
##  8 audi         a4 q~   1.8  1999     4 manu~ 4        18    26 p     comp~
##  9 audi         a4 q~   1.8  1999     4 auto~ 4        16    25 p     comp~
## 10 audi         a4 q~   2    2008     4 manu~ 4        20    28 p     comp~
## # ... with 224 more rows
```

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```

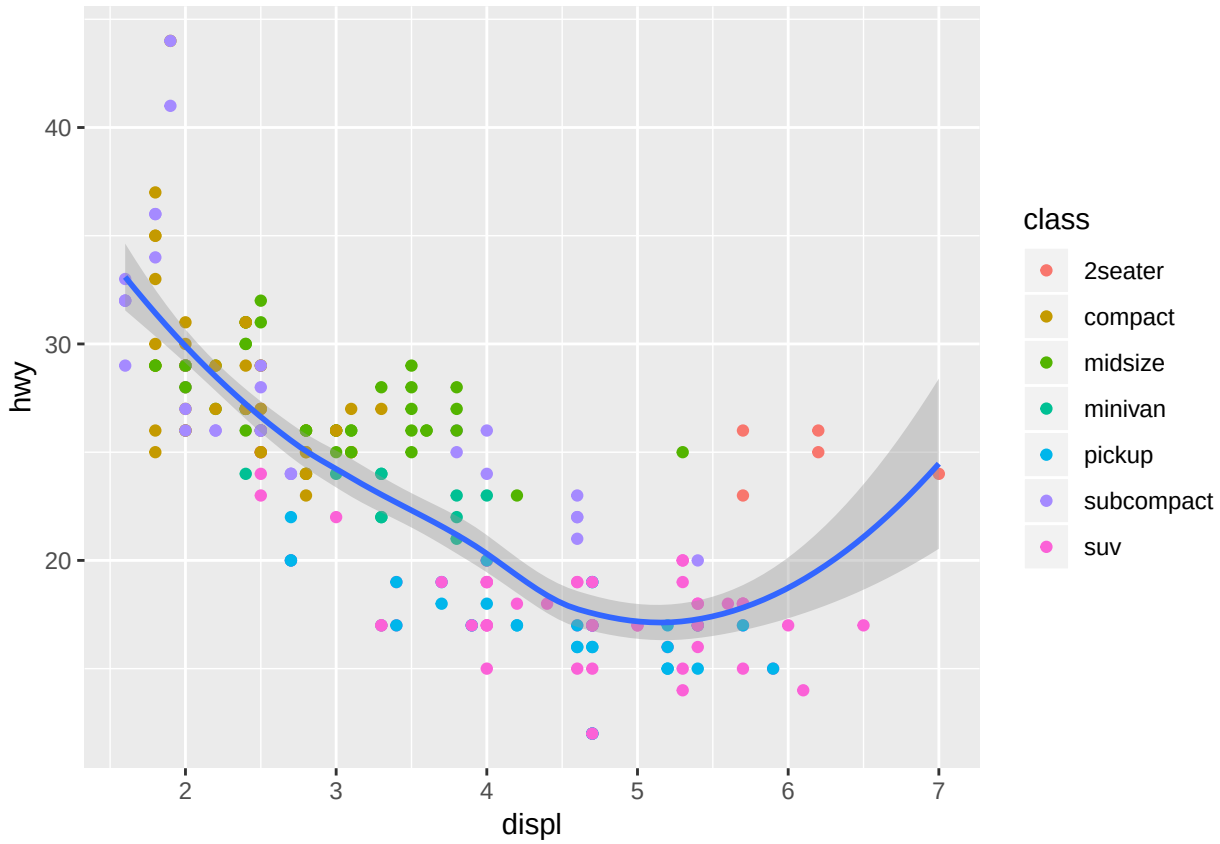



```
# What about this other column class? Maybe we want to see what type of car it is too

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```

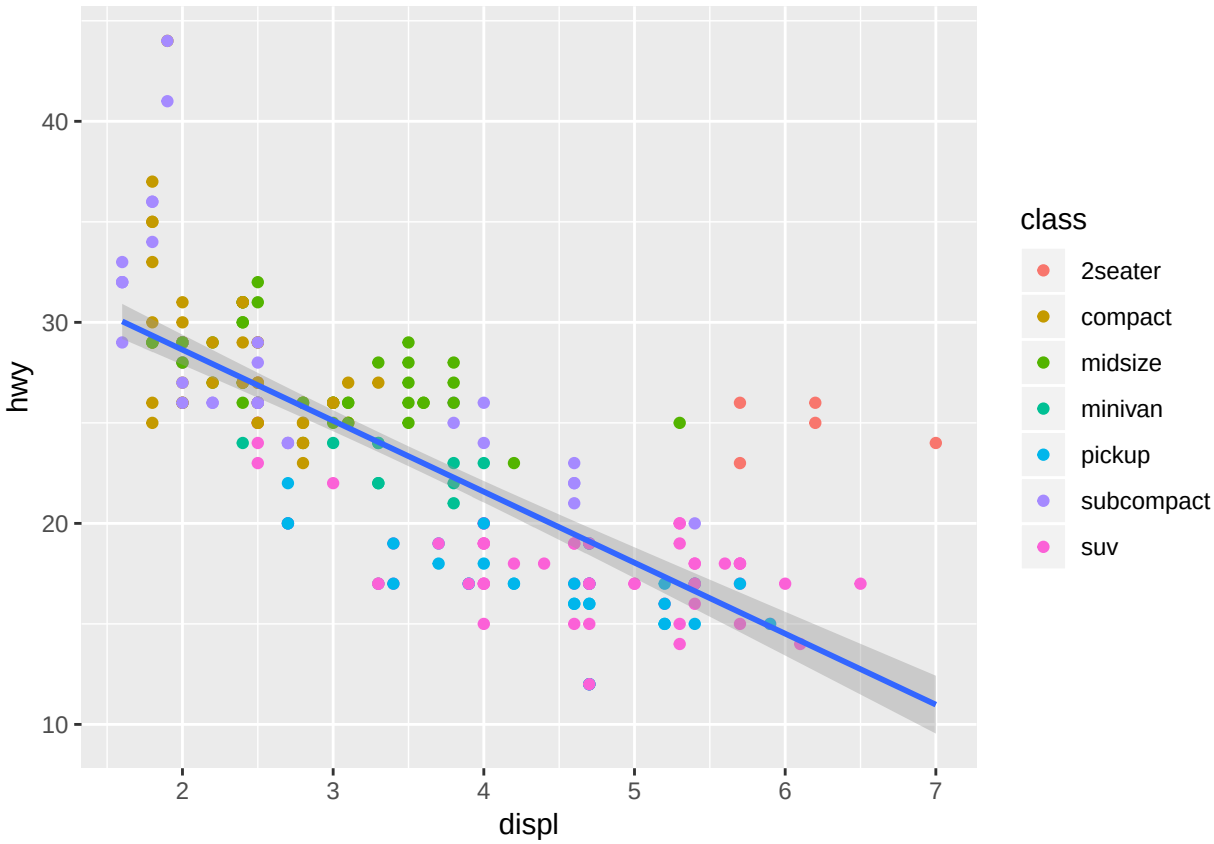



```
# It would be nice to see a trend line

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
# The default smoothing line is a loess model, which looks funny here, lets use a linear model

# It would be nice to see a trend line

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class)) +
  geom_smooth(mapping = aes(x = displ, y = hwy), method = lm)
```



3 Project Oriented Workflow

3.0.1 Learning Objectives

- Understand the benefits of version control and set up a GitHub Repository
- Describe the benefits of a Project Oriented Workflow
- Organize files and directories for a set of analyses as an R
- Project, understand the purpose of the working directory, and the `here()` package
-

3.1 Explain when to use the source editor vs. the console

3.2 Version Control

From Jenny Bryan:

"Git is a version control system. Its original purpose was to help groups of developers work collaboratively on big software projects. Git manages the evolution of a set of files – called a repository – in a sane, highly structured way. If you have no idea what I'm talking about, think of it as the "Track Changes" features from Microsoft Word on steroids."

GitHub provides a home for your Git-based projects on the internet. If you have no idea what I'm talking about, think of it as DropBox but much, much better. The remote host acts as a distribution channel or clearinghouse for your Git-managed project. It allows other people to see your stuff, sync up with you, and perhaps even make changes."

You can think of having an additional 'save' that comes from version control. This additional save, in git terms is known as a *commit*. You save your files like you normally would, but every once in a while you commit your files as an official version to be remembered. A commit can be thought of as a bullet point in the to do list of your analysis, and each commit you make must be accompanied by a message. For example; 'read in data and tidy it up', or 'remove observations from non-standard sampling event, and re-fit GLM'. Git tracks the commits you make in R-Studio locally on your own computer. When you are ready for a series of commits to be made public, you *push* your commits to your remote repository at GitHub.

Read Happy Git with R by Jenny Bryan for more details!

3.2.1 New Project, GitHub first.

Let's set up your first project.

Again fromm Jenny Bryan, edited for this use-case.

We create a new Project, with the preferred "GitHub first, then RStudio" sequence. Why do we prefer this? Because this method of copying the Project from GitHub to your computer also sets up the local Git repository for immediate pulling and pushing.

3.2.2 Make a repo on GitHub

Do this once per new project.

Go to https://github.com/pelagic-ecosystems and make sure you are logged in.

Click on "Repositories", then click the green "New" button.

Repository name: `your-study-discipline` Public
YES Initialize this repository with a README

Click the big green button "Create repository."

Copy the HTTPS clone URL to your clipboard via the green "Clone or Download" button. Or copy the SSH URL if you chose to set up SSH keys.

3.2.3 New RStudio Project

In RStudio, start a new Project:

- *File > New Project > Version Control > Git.* In the "repository URL" paste the URL of your new GitHub repository. It will be something like this `https://github.com/pelagic-ecosystems/your-study-discipline.git`.
- Create this project in a new folder you should create now called R Projects.
- Suggest you "Open in new session".
- Click "Create Project" to create a new directory, which will be all of these things:
  - a directory or "folder" on your computer
  - a Git repository, linked to a remote GitHub repository
  - an RStudio Project
- **In the absence of other constraints, I suggest that all of your R projects have exactly this set-up.**

This should download the `README.md` file that we created on GitHub in the previous step. Look in RStudio's file browser pane for the `README.md` file.

There's a big advantage to the "GitHub first, then RStudio" workflow: the remote GitHub repo is added as a remote for your local repo and your local `master` branch is now tracking `master` on GitHub. This is a technical but important point about Git. The practical implication is that you are now set up to push and pull. No need to fanny around setting up Git remotes and tracking branches on the command line.

3.2.4 Make local changes, save, commit

Do this every time you finish a valuable chunk of work, probably many times a day.

From RStudio, modify the `README.md` file, e.g., by adding the line "This is a line from RStudio". Save your changes.

Commit these changes to your local repo. How?

- Click the "Git" tab in upper right pane
- Check "Staged" box for any files whose existence or modifications you want to commit.
  - To see more detail on what's changed in file since the last commit, click on "Diff" for a Git pop-up
- If you're not already in the Git pop-up, click "Commit"
- Type a message in "Commit message", such as "Commit from RStudio".
- Click "Commit"

3.2.5 Push your local changes to GitHub

Do this a few times a day, but possibly less often than you commit.

You have new work in your local Git repository, but the changes are not online yet.

This will seem counterintuitive, but first let's stop and pull from GitHub.

Why? Establish this habit for the future! If you make changes to the repo in the browser or from another machine or (one day) a collaborator has pushed, you will be happier if you pull those changes in before you attempt to push.

Click the blue "Pull" button in the "Git" tab in RStudio. I doubt anything will happen, i.e. you'll get the message "Already up-to-date." This is just to establish a habit.

Click the green "Push" button to send your local changes to GitHub. You should see some message along these lines.

```
[master dc671f0] blah
 3 files changed, 22 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 myrepo.Rproj
```

3.2.6 Confirm the local change propagated to the GitHub remote

Go back to the browser. I assume we're still viewing your new GitHub repo.

Refresh.

You should see the new "This is a line from RStudio" in the README.

If you click on "commits," you should see one with the message "Commit from RStudio".

3.2.7 Make a change on GitHub

Click on README.md in the file listing on GitHub.

In the upper right corner, click on the pencil for "Edit this file".

Add a line to this file, such as "Line added from GitHub."

Edit the commit message in "Commit changes" or accept the default.

Click the big green button "Commit changes."

3.2.8 Pull from GitHub

Back in RStudio locally ...

Inspect your README.md. It should NOT have the line "Line added from GitHub". It should be as you left it. Verify that.

Click the blue Pull button.

Look at README.md again. You should now see the new line there.

Now just ... repeat. Do work somewhere. Commit it. Push it or pull it* depending on where you did it, but get local and remote "synced up". Repeat.

* Note that in general (and especially in future when collaborating with other developers) you will usually need to pull changes from the remote (GitHub) before pushing the local changes you have made. For this reason, it's a good idea to try and get into the habit of pulling before you attempt to push.

Questions?

3.3 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.

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'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.'

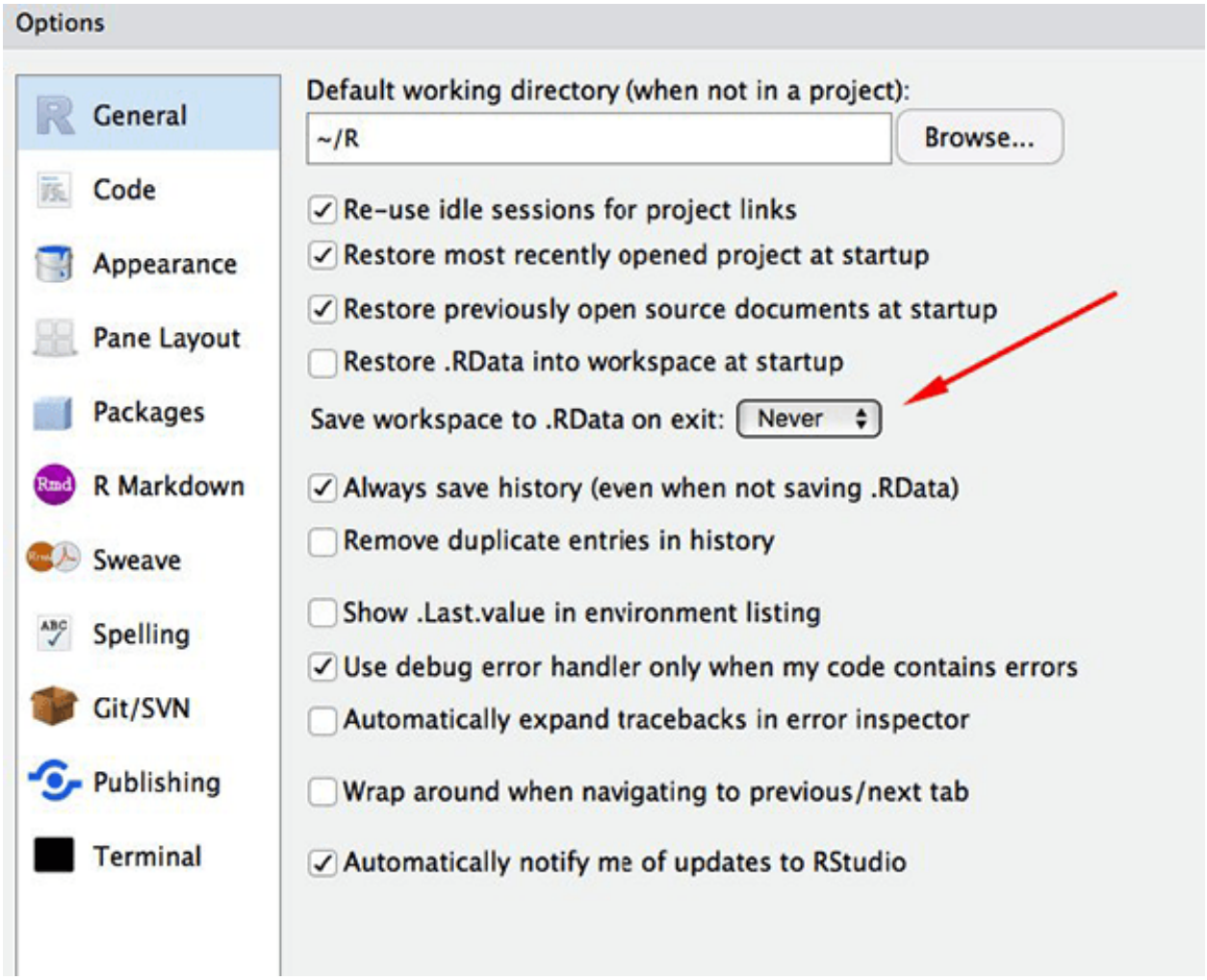


Figure 1: Set 'Save workspace to .RData on exit' to 'Never'

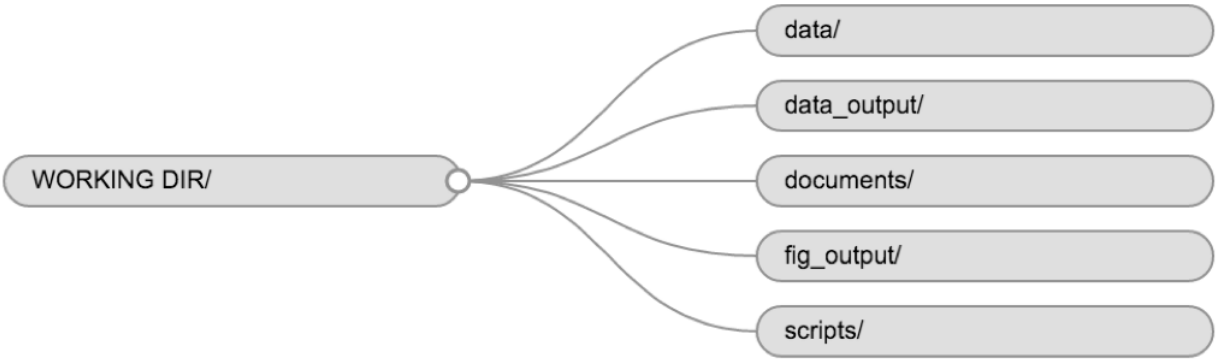


Figure 2: Example of a working directory structure.

3.3.1 Template folder structure

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**, **raw_data**, **processed_data**, **figs**, and **documents**. You will also wante to creat a CHANGELOG file, which will track the major versions of your data, scripts, and figures.

- `raw_data/` Use this folder to store your raw data and don't ever change your raw data!. For the sake of transparency and 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 in excel which you can't reproduce) as possible. Separating raw data from processed data is critical.
- `processed_data/` Save derived data sets here after cleaning or summarizing.
- `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).
- `figs/` Programatically save the output of to `figs/` using `ggsave()`.

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

For this workshop, we will need a `raw_data/` folder to store our raw data, and we will use `processed_data/` for when we learn how to export data as CSV files, and `figs/` 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/raw_data`). (Alternatively, type `dir.create("data")` at your R console.) Repeat these operations to create a `processed_data/` and a `figs/` folders.
- Again under the `Files` tab, click on `New File` and create a new `Text File`. Type #CHANGELOG on the first line and save the new file as CHANGELOG.TXT in the root of your working directory.

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 mine.

3.3.2 The `here()` package

To avoid having to set your working directory completely, a recommended method to work with relative file paths is using the `here()` package in conjunction with R-Studio projects. When you create a new R-Studio project, a .Rproj file is automatically created in the new folder that you created for the project. The `here()` package will automatically set your working directory to wherever your .Rproj file is saved. That means you can save a file like this: `write_csv(file_name, here("data", "file_name.csv"))`. Using `here()` means that if you access your collaborators folder where the .Rproj file is and they have been using relative paths using `here()`, the scripts should all just work—no chaning working directories or absolute file paths.

Questions?

3.4 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 *running* those commands. We use *functions* that are part of *packages*.

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.

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.

3.5 R Markdown

Create a new R Markdown File.

File > New > R Markdown

3.5.1 What's Markdown?

Markdown is a lightweight markup language, like html but way simpler:

Single hashtags heading 1, 3 for heading 3: "###",

3.5.2 Heading 3 Example

Use '*' around words for bold or italics.

Bullet points also with a single '*'

Markdown cheatsheet in help > cheatsheets.

R Markdown is a file format that allows you to organize your notes, code, and results in a single document. It's a great tool for "literate programming" – the idea that your code should be readable by humans as well as computers! It also keeps your writing and results together, so if you collect some new data or change how you clean the data, you just have to re-compile the document and you're up to date!

What your final data product is going to be will dictate what your final scripts will be. R Markdown files formats that have pre-made templates and solutions that are easy to modify to suit your needs include:

- Analysis report templates (html, .pdf, or .doc outputs);

- A Manuscript;
- A Book;
- A Dissertation;
- A Research Compendium;
- A Slideshow;
- An interactive dashboard;
- An R Package
- A website

As a baseline I recommend .Rmd as the final format because this gives you a lot of flexibility in terms of polished data products.

3.5.3 Knitting your document

Knit the document!

4 Intro to R

4.0.1 Learning Objectives

- Define the following terms as they relate to R: object, assign, call, function, arguments, options.
- Assign values to objects in R.
- Learn how to *name* objects
- Use comments to inform script.
- Solve simple arithmetic operations in R.
- Call functions and use arguments to change their default options.
-

4.1 Inspect the content of vectors and manipulate their content.

4.2 Creating objects in R

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

```
3 + 5
```

```
## [1] 8
```

```
12 / 7
```

```
## [1] 1.714286
```

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`.

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`).

Avoid using dots, use underscores to separate words.

Using a consistent coding style makes your code clearer to read for your future self and your collaborators. Use the tidyverse's. The tidyverse's is very comprehensive and may seem overwhelming at first. You can install the **`stylr`** package to automatically check for issues in the styling of your code.

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

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

```
## [1] 126.5
```

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?

4.2.1 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.

4.2.2 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.

4.3 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
```

```
## [1] 50 60 65 82
```

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

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

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`).

4.4 Starting with Data

4.4.1 Learning Objectives

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

4.4.2 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

Column	Description
weight	weight of the animal in grams
genus	genus of animal
species	species of animal
taxon	e.g. Rodent, Reptile, Bird, Rabbit
plot_type	type of plot

We are going to use the R function `download.file()` to download the CSV file that contains the survey data from figshare, and we will use `read_csv()` to load into memory the content of the CSV file as an object of class `data.frame`. Inside the download.file command, the first entry is a character string with the source URL ("https://ndownloader.figshare.com/files/2292169"). This source URL downloads a CSV file from figshare. The text after the comma ("data/portal_data_joined.csv") is the destination of the file on your local machine. You'll need to have a folder on your machine called "raw_data" where you'll download the file. So this command downloads a file from figshare, names it "portal_data_joined.csv," and adds it to a preexisting folder named "data."

```
library(tidyverse)

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

You are now ready to load the data:

```
surveys <- read_csv(here("raw_data", "portal_data_joined.csv"))
```

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

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

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

```
head(surveys)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id species_id sex   hindfoot_length
##       <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
## 1         1     7    16  1977       2 NL         M                  32
## 2        72     8    19  1977       2 NL         M                  31
## 3       224     9    13  1977       2 NL         <NA>               NA
```

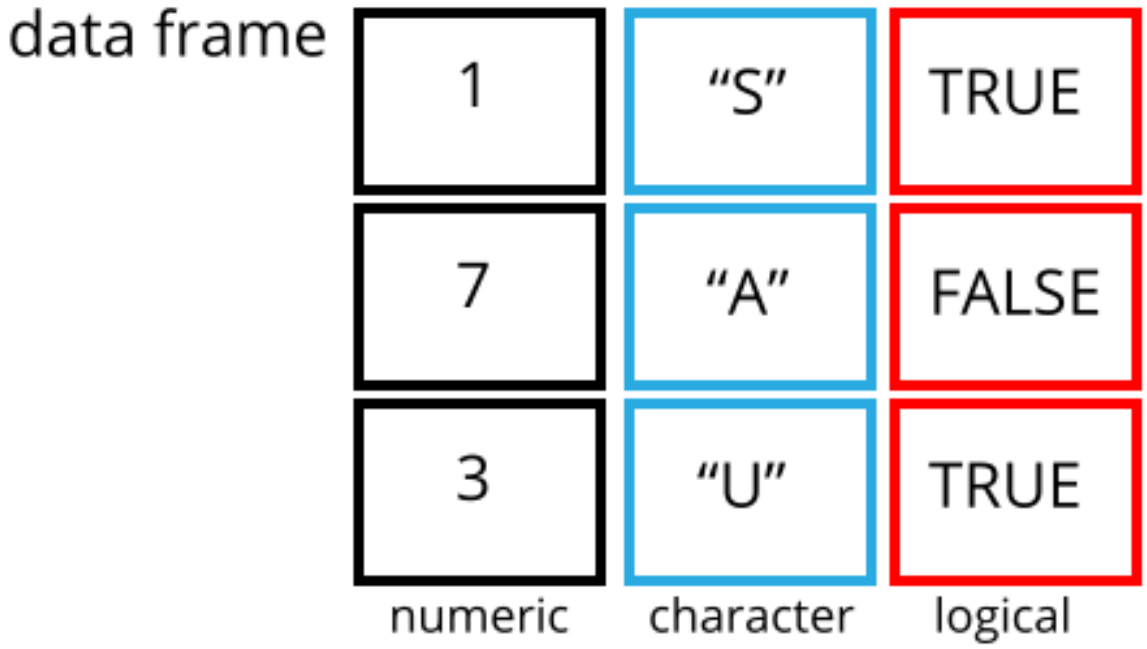


Figure 3:

```
## 4       266    10    16  1977       2 NL         <NA>              NA
## 5       349    11    12  1977       2 NL         <NA>              NA
## 6       363    11    12  1977       2 NL         <NA>              NA
## # ... with 5 more variables: weight <int>, genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>
```

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

4.4.3 What are data frames?

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

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

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

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

```
str(surveys)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    34786 obs. of  13 variables:
##  $ record_id      : int  1 72 224 266 349 363 435 506 588 661 ...
##  $ month          : int  7 8 9 10 11 11 12 1 2 3 ...
##  $ day            : int  16 19 13 16 12 12 10 8 18 11 ...
```

```
##  $ year           : int  1977 1977 1977 1977 1977 1977 1977 1978 1978 1978 ...
##  $ plot_id        : int  2 2 2 2 2 2 2 2 2 2 ...
##  $ species_id     : chr  "NL" "NL" "NL" "NL" ...
##  $ sex            : chr  "M" "M" NA NA ...
##  $ hindfoot_length: int  32 31 NA NA NA NA NA NA NA NA ...
##  $ weight         : int  NA NA NA NA NA NA NA NA 218 NA ...
##  $ genus          : chr  "Neotoma" "Neotoma" "Neotoma" "Neotoma" ...
##  $ species        : chr  "albigula" "albigula" "albigula" "albigula" ...
##  $ taxa           : chr  "Rodent" "Rodent" "Rodent" "Rodent" ...
##  $ plot_type      : chr  "Control" "Control" "Control" "Control" ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 13
##   .. ..$ record_id      : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ month          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ day            : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ year           : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ plot_id        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ species_id     : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ sex            : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ hindfoot_length: list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ weight         : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ genus          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ species        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ taxa           : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ plot_type      : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   ..$ default: list()
##   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
##   ..- attr(*, "class")= chr "col_spec"
```

4.4.4 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
- Summary:

- `str(surveys)` - structure of the object and information about the class, length and content of each column
- `summary(surveys)` - summary statistics for each column

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

4.4.5 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]
```

```
## # A tibble: 1 x 1
##   record_id
##       <int>
## 1         1
```

```
# first element in the 6th column (as a vector)
surveys[1, 6]
```

```
## # A tibble: 1 x 1
##   species_id
##   <chr>
## 1 NL
```

```
# first column of the data frame (as a vector)
surveys[, 1]
```

```
## # A tibble: 34,786 x 1
##    record_id
##        <int>
##  1         1
##  2        72
##  3       224
##  4       266
##  5       349
##  6       363
##  7       435
##  8       506
##  9       588
## 10       661
## # ... with 34,776 more rows
```

```
# first column of the data frame (as a data.frame)
surveys[1]
```

```
## # A tibble: 34,786 x 1
##    record_id
##        <int>
##  1         1
##  2        72
##  3       224
##  4       266
```

```
##  5       349
##  6       363
##  7       435
##  8       506
##  9       588
## 10       661
## # ... with 34,776 more rows
```

```
# first three elements in the 7th column (as a vector)
surveys[1:3, 7]
```

```
## # A tibble: 3 x 1
##   sex
##   <chr>
## 1 M
## 2 M
## 3 <NA>
```

```
# the 3rd row of the data frame (as a data.frame)
surveys[3, ]
```

```
## # A tibble: 1 x 13
##   record_id month   day  year plot_id species_id sex   hindfoot_length
##       <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
## 1       224     9    13  1977       2 NL         <NA>               NA
## # ... with 5 more variables: weight <int>, genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>
```

```
# 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
```

```
## # A tibble: 34,786 x 12
##    month   day  year plot_id species_id sex   hindfoot_length weight genus
##    <int> <int> <int>   <int> <chr>      <chr>           <int>  <int> <chr>
##  1     7    16  1977       2 NL         M                  32     NA Neot~
##  2     8    19  1977       2 NL         M                  31     NA Neot~
##  3     9    13  1977       2 NL         <NA>               NA     NA Neot~
##  4    10    16  1977       2 NL         <NA>               NA     NA Neot~
##  5    11    12  1977       2 NL         <NA>               NA     NA Neot~
##  6    11    12  1977       2 NL         <NA>               NA     NA Neot~
##  7    12    10  1977       2 NL         <NA>               NA     NA Neot~
##  8     1     8  1978       2 NL         <NA>               NA     NA Neot~
##  9     2    18  1978       2 NL         M                  NA    218 Neot~
## 10     3    11  1978       2 NL         <NA>               NA     NA Neot~
## # ... with 34,776 more rows, and 3 more variables: species <chr>,
## #   taxa <chr>, plot_type <chr>
```

```
surveys[-c(7:34786), ] # Equivalent to head(surveys)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id species_id sex   hindfoot_length
```

```
##       <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
## 1         1     7    16  1977       2 NL         M                  32
## 2        72     8    19  1977       2 NL         M                  31
## 3       224     9    13  1977       2 NL         <NA>               NA
## 4       266    10    16  1977       2 NL         <NA>               NA
## 5       349    11    12  1977       2 NL         <NA>               NA
## 6       363    11    12  1977       2 NL         <NA>               NA
## # ... with 5 more variables: weight <int>, genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>
```

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

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

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

4.5 Data Manipulation

4.5.1 Learning Objectives

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

4.5.2 dplyr and tidyr

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

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

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

To load the package, type:

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

The package **dplyr** provides easy tools for the most common data manipulation tasks. It is built to work directly with data frames.

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

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

```
surveys <- read_csv(here("raw_data", "portal_data_joined.csv"))
```

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

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

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

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

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

4.5.3 Selecting columns and filtering rows

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

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

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

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

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

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

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

```
## Warning: package 'bindrcpp' was built under R version 3.4.4
```

```
## # A tibble: 1,180 x 13
##    record_id month   day  year plot_id species_id sex   hindfoot_length
##        <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
##  1     22314     6     7  1995       2 NL         M                  34
##  2     22728     9    23  1995       2 NL         F                  32
##  3     22899    10    28  1995       2 NL         F                  32
##  4     23032    12     2  1995       2 NL         F                  33
##  5     22003     1    11  1995       2 DM         M                  37
##  6     22042     2     4  1995       2 DM         F                  36
##  7     22044     2     4  1995       2 DM         M                  37
##  8     22105     3     4  1995       2 DM         F                  37
##  9     22109     3     4  1995       2 DM         M                  37
## 10     22168     4     1  1995       2 DM         M                  36
## # ... with 1,170 more rows, and 5 more variables: weight <int>,
## #   genus <chr>, species <chr>, taxa <chr>, plot_type <chr>
```

4.5.4 Pipes

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

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

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

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

The a sometimes better 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)
```

```
## # A tibble: 17 x 3
##    species_id sex   weight
##    <chr>      <chr>  <int>
##  1 PF         F          4
##  2 PF         F          4
##  3 PF         M          4
##  4 RM         F          4
##  5 RM         M          4
##  6 PF         <NA>       4
##  7 PP         M          4
##  8 RM         M          4
##  9 RM         M          4
## 10 RM         M          4
## 11 PF         M          4
## 12 PF         F          4
## 13 RM         M          4
## 14 RM         M          4
## 15 RM         F          4
## 16 RM         M          4
## 17 RM         M          4
```

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
```

```
## # A tibble: 17 x 3
##    species_id sex   weight
##    <chr>      <chr>  <int>
##  1 PF         F          4
##  2 PF         F          4
##  3 PF         M          4
##  4 RM         F          4
##  5 RM         M          4
##  6 PF         <NA>       4
##  7 PP         M          4
##  8 RM         M          4
##  9 RM         M          4
## 10 RM         M          4
## 11 PF         M          4
## 12 PF         F          4
## 13 RM         M          4
## 14 RM         M          4
```

```
## 15 RM         F           4
## 16 RM         M           4
## 17 RM         M           4
```

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

4.5.5 Challenge

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

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

4.5.6 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)
```

```
## # A tibble: 34,786 x 14
##    record_id month   day  year plot_id species_id sex   hindfoot_length
##        <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
##  1         1     7    16  1977       2 NL         M                  32
##  2        72     8    19  1977       2 NL         M                  31
##  3       224     9    13  1977       2 NL         <NA>               NA
##  4       266    10    16  1977       2 NL         <NA>               NA
##  5       349    11    12  1977       2 NL         <NA>               NA
##  6       363    11    12  1977       2 NL         <NA>               NA
##  7       435    12    10  1977       2 NL         <NA>               NA
##  8       506     1     8  1978       2 NL         <NA>               NA
##  9       588     2    18  1978       2 NL         M                  NA
## 10       661     3    11  1978       2 NL         <NA>               NA
## # ... with 34,776 more rows, and 6 more variables: weight <int>,
## #   genus <chr>, species <chr>, taxa <chr>, plot_type <chr>,
## #   weight_kg <dbl>
```

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)
```

```
## # A tibble: 34,786 x 15
##    record_id month   day  year plot_id species_id sex   hindfoot_length
##        <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
##  1         1     7    16  1977       2 NL         M                  32
##  2        72     8    19  1977       2 NL         M                  31
##  3       224     9    13  1977       2 NL         <NA>               NA
##  4       266    10    16  1977       2 NL         <NA>               NA
##  5       349    11    12  1977       2 NL         <NA>               NA
```

```
##  6       363    11    12  1977       2 NL         <NA>              NA
##  7       435    12    10  1977       2 NL         <NA>              NA
##  8       506     1     8  1978       2 NL         <NA>              NA
##  9       588     2    18  1978       2 NL         M                 NA
## 10       661     3    11  1978       2 NL         <NA>              NA
## # ... with 34,776 more rows, and 7 more variables: weight <int>,
## #   genus <chr>, species <chr>, taxa <chr>, plot_type <chr>,
## #   weight_kg <dbl>, weight_kg2 <dbl>
```

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()
```

```
## # A tibble: 6 x 14
##   record_id month   day  year plot_id species_id sex   hindfoot_length
##       <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
## 1         1     7    16  1977       2 NL         M                  32
## 2        72     8    19  1977       2 NL         M                  31
## 3       224     9    13  1977       2 NL         <NA>               NA
## 4       266    10    16  1977       2 NL         <NA>               NA
## 5       349    11    12  1977       2 NL         <NA>               NA
## 6       363    11    12  1977       2 NL         <NA>               NA
## # ... with 6 more variables: weight <int>, genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>, weight_kg <dbl>
```

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

```
## # A tibble: 6 x 14
##   record_id month   day  year plot_id species_id sex   hindfoot_length
##       <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
## 1       588     2    18  1978       2 NL         M                  NA
## 2       845     5     6  1978       2 NL         M                  32
## 3       990     6     9  1978       2 NL         M                  NA
## 4      1164     8     5  1978       2 NL         M                  34
## 5      1261     9     4  1978       2 NL         M                  32
## 6      1453    11     5  1978       2 NL         M                  NA
## # ... with 6 more variables: weight <int>, genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>, weight_kg <dbl>
```

`drop_na()` is a function that determines whether something is an `NA`, and will drop the whole row if there is an NA in the column you specified (weight in this case). Using `drop_na()` with no column named will look for an NA in any column and drop the whole row if there are any NAs

4.5.7 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!

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

4.5.8 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))
```

```
## # A tibble: 3 x 2
##   sex   mean_weight
##   <chr>       <dbl>
## 1 F            42.2
## 2 M            43.0
## 3 <NA>         64.7
```

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))
```

```
## # A tibble: 92 x 3
## # Groups:   sex [?]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     BA                9.16
##  2 F     DM               41.6
##  3 F     DO               48.5
##  4 F     DS              118.
##  5 F     NL              154.
##  6 F     OL               31.1
##  7 F     OT               24.8
##  8 F     OX               21
##  9 F     PB               30.2
## 10 F     PE               22.8
## # ... with 82 more rows
```

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 %>%
  drop_na(weight) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight))
```

```
## # A tibble: 64 x 3
## # Groups:   sex [?]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     BA                9.16
##  2 F     DM               41.6
##  3 F     DO               48.5
##  4 F     DS              118.
##  5 F     NL              154.
##  6 F     OL               31.1
##  7 F     OT               24.8
##  8 F     OX               21
##  9 F     PB               30.2
## 10 F     PE               22.8
## # ... with 54 more rows
```

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)
```

```
## # A tibble: 64 x 3
## # Groups:   sex [?]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     BA                9.16
##  2 F     DM               41.6
##  3 F     DO               48.5
##  4 F     DS              118.
##  5 F     NL              154.
##  6 F     OL               31.1
##  7 F     OT               24.8
##  8 F     OX               21
##  9 F     PB               30.2
## 10 F     PE               22.8
## 11 F     PF                7.97
## 12 F     PH               30.8
## 13 F     PL               19.3
## 14 F     PM               22.1
## 15 F     PP               17.2
## # ... with 49 more rows
```

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))
```

```
## # A tibble: 64 x 4
## # Groups:   sex [?]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 F     BA                9.16          6
##  2 F     DM               41.6          10
##  3 F     DO               48.5          12
##  4 F     DS              118.           45
##  5 F     NL              154.           32
##  6 F     OL               31.1          10
##  7 F     OT               24.8           5
##  8 F     OX               21            20
##  9 F     PB               30.2          12
## 10 F     PE               22.8          11
## # ... with 54 more rows
```

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)
```

```
## # A tibble: 64 x 4
## # Groups:   sex [3]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 F     PF                7.97          4
##  2 F     RM               11.1           4
##  3 M     PF                7.89          4
##  4 M     PP               17.2           4
##  5 M     RM               10.1           4
##  6 <NA>  PF                6             4
##  7 F     OT               24.8           5
##  8 F     PP               17.2           5
##  9 F     BA                9.16          6
## 10 M     BA                7.36          6
## # ... with 54 more rows
```

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))
```

```
## # A tibble: 64 x 4
## # Groups:   sex [3]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 <NA>  NL               168.          83
##  2 M     NL               166.          30
##  3 F     NL               154.          32
##  4 M     SS               130          130
##  5 <NA>  SH               130          130
##  6 M     DS               122.          12
##  7 <NA>  DS               120           78
##  8 F     DS               118.          45
##  9 F     SH                78.8         30
## 10 F     SF                69           46
## # ... with 54 more rows
```

4.5.9 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)
```

```
## # A tibble: 3 x 2
##   sex       n
##   <chr> <int>
## 1 F     15690
## 2 M     17348
## 3 <NA>   1748
```

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())
```

```
## # A tibble: 3 x 2
##   sex   count
##   <chr> <int>
## 1 F     15690
## 2 M     17348
## 3 <NA>   1748
```

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

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

```
## # A tibble: 3 x 2
##   sex       n
##   <chr> <int>
## 1 M     17348
## 2 F     15690
## 3 <NA>   1748
```

4.5.10 Challenge

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

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

```
## # A tibble: 5 x 2
##   plot_type                     n
##   <chr>                     <int>
## 1 Control                   15611
## 2 Long-term Krat Exclosure   5118
## 3 Rodent Exclosure           4233
## 4 Short-term Krat Exclosure  5906
## 5 Spectab exclosure          3918
```

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`).

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

```
## # A tibble: 25 x 5
##    species_id mean_hindfoot_len~ min_hindfoot_leng~ max_hindfoot_len~     n
##    <chr>                   <dbl>              <dbl>             <dbl> <int>
##  1 AH                       33                   31                35     2
##  2 BA                       13                    6                16    45
##  3 DM                       36.0                 16                50  9972
##  4 DO                       35.6                 26                64  2887
##  5 DS                       49.9                 39                58  2132
##  6 NL                       32.3                 21                70  1074
##  7 OL                       20.5                 12                39   920
##  8 OT                       20.3                 13                50  2139
##  9 OX                       19.1                 13                21     8
## 10 PB                       26.1                  2                47  2864
## # ... with 15 more rows
```

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

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

```
    group_by(year) %>%
    filter(weight == max(weight)) %>%
    select(year, genus, species, weight) %>%
    arrange(year)
```

```
## # A tibble: 27 x 4
## # Groups:   year [26]
##     year genus     species     weight
##    <int> <chr>     <chr>        <int>
##  1  1977 Dipodomys spectabilis    149
##  2  1978 Neotoma   albigula       232
##  3  1978 Neotoma   albigula       232
##  4  1979 Neotoma   albigula       274
##  5  1980 Neotoma   albigula       243
##  6  1981 Neotoma   albigula       264
##  7  1982 Neotoma   albigula       252
##  8  1983 Neotoma   albigula       256
##  9  1984 Neotoma   albigula       259
## 10  1985 Neotoma   albigula       225
## # ... with 17 more rows
```

5 END OF DAY 1

6 Tidy Data in Spreadsheets

6.1 Principles of tidy data

Tidy data

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

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

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()`.

6.1.1 Spreading

`spread()` takes three principal arguments:

1. the data
2. the *key* column variable whose values will become new column names.
3. the *value* column variable whose values will fill the new column variables.

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)
```

```
## Classes 'grouped_df', 'tbl_df', 'tbl' and 'data.frame':  196 obs. of  3 variables:
##  $ genus      : chr  "Baiomys" "Baiomys" "Baiomys" "Baiomys" ...
##  $ plot_id    : int  1 2 3 5 18 19 20 21 1 2 ...
##  $ mean_weight: num  7 6 8.61 7.75 9.5 ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 13
##   .. ..$ record_id      : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ month          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ day            : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ year           : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ plot_id        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ species_id     : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ sex            : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ hindfoot_length: list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ weight         : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ genus          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ species        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ taxa           : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ plot_type      : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   ..$ default: list()
##   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
```

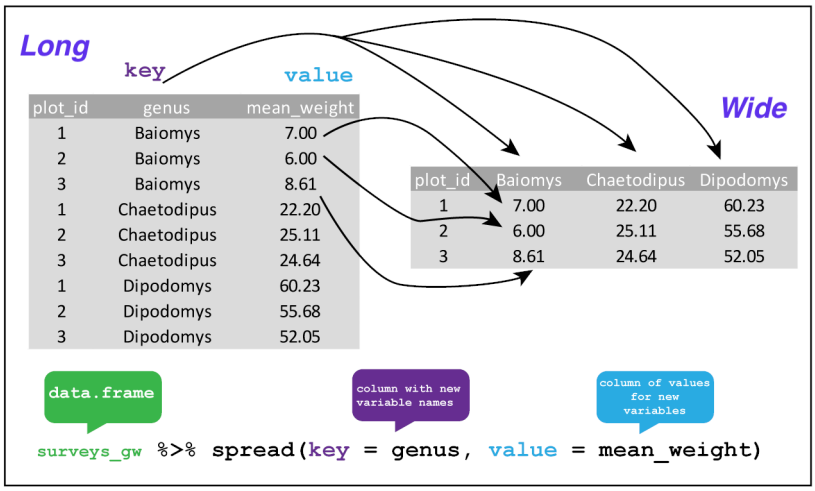


Figure 4:

```
##   ..- attr(*, "class")= chr "col_spec"
##  - attr(*, "vars")= chr "genus"
##  - attr(*, "drop")= logi TRUE
```

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)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    24 obs. of  11 variables:
##  $ plot_id        : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Baiomys        : num  7 6 8.61 NA 7.75 ...
##  $ Chaetodipus    : num  22.2 25.1 24.6 23 18 ...
##  $ Dipodomys      : num  60.2 55.7 52 57.5 51.1 ...
##  $ Neotoma        : num  156 169 158 164 190 ...
##  $ Onychomys      : num  27.7 26.9 26 28.1 27 ...
##  $ Perognathus    : num  9.62 6.95 7.51 7.82 8.66 ...
##  $ Peromyscus     : num  22.2 22.3 21.4 22.6 21.2 ...
##  $ Reithrodontomys: num  11.4 10.7 10.5 10.3 11.2 ...
##  $ Sigmodon       : num  NA 70.9 65.6 82 82.7 ...
##  $ Spermophilus   : num  NA NA NA NA NA NA NA NA NA NA ...
```

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()
```

```
## # A tibble: 6 x 11
##   plot_id Baiomys Chaetodipus Dipodomys Neotoma Onychomys Perognathus
```

```
##     <int>   <dbl>       <dbl>     <dbl>   <dbl>     <dbl>       <dbl>
## 1       1    7           22.2      60.2    156.      27.7        9.62
## 2       2    6           25.1      55.7    169.      26.9        6.95
## 3       3    8.61        24.6      52.0    158.      26.0        7.51
## 4       4    0           23.0      57.5    164.      28.1        7.82
## 5       5    7.75        18.0      51.1    190.      27.0        8.66
## 6       6    0           24.9      58.6    180.      25.9        7.81
## # ... with 4 more variables: Peromyscus <dbl>, Reithrodontomys <dbl>,
## #   Sigmodon <dbl>, Spermophilus <dbl>
```

6.1.2 Gathering

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

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)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    240 obs. of  3 variables:
##  $ plot_id    : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ genus      : chr  "Baiomys" "Baiomys" "Baiomys" "Baiomys" ...
##  $ mean_weight: num  7 6 8.61 NA 7.75 ...
```

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()
```

```
## # A tibble: 6 x 3
##   plot_id genus   mean_weight
##     <int> <chr>         <dbl>
## 1       1 Baiomys        7
## 2       2 Baiomys        6
## 3       3 Baiomys        8.61
## 4       4 Baiomys       NA
```

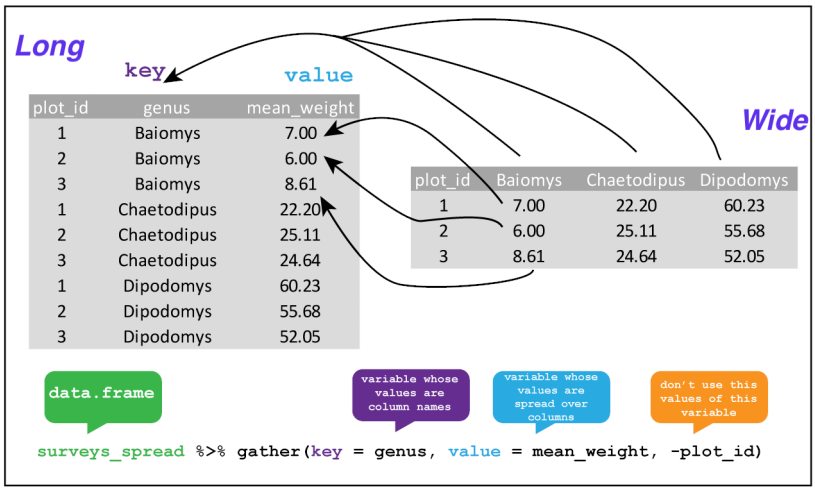


Figure 5:

```
## 5       5 Baiomys        7.75
## 6       6 Baiomys          NA
```

6.1.3 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.

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

head(rich_time)
```

```
## # A tibble: 6 x 27
## # Groups:   plot_id [6]
##   plot_id `1977` `1978` `1979` `1980` `1981` `1982` `1983` `1984` `1985`
##     <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>
## 1       1      2      3      4      7      5      6      7      6      4
## 2       2      6      6      6      8      5      9      9      9      6
## 3       3      5      6      4      6      6      8     10     11      7
## 4       4      4      4      3      4      5      4      6      3      4
## 5       5      4      3      2      5      4      6      7      7      3
## 6       6      3      4      3      4      5      9      9      7      5
## # ... with 17 more variables: `1986` <int>, `1987` <int>, `1988` <int>,
## #   `1989` <int>, `1990` <int>, `1991` <int>, `1992` <int>, `1993` <int>,
## #   `1994` <int>, `1995` <int>, `1996` <int>, `1997` <int>, `1998` <int>,
## #   `1999` <int>, `2000` <int>, `2001` <int>, `2002` <int>
```

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

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

```
## # A tibble: 624 x 3
## # Groups:   plot_id [24]
##    plot_id year  n_genera
##      <int> <chr>    <int>
##  1       1 1977         2
##  2       2 1977         6
##  3       3 1977         5
##  4       4 1977         4
##  5       5 1977         4
##  6       6 1977         3
##  7       7 1977         3
##  8       8 1977         2
##  9       9 1977         3
## 10      10 1977         1
## # ... with 614 more rows
```

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.

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

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

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

```
## # A tibble: 130 x 4
## # Groups:   year [26]
##     year plot_type                hindfoot_length weight
##    <int> <chr>                              <dbl>  <dbl>
##  1  1977 Control                             36.1   50.4
##  2  1977 Long-term Krat Exclosure            33.7   34.8
##  3  1977 Rodent Exclosure                    39.1   48.2
##  4  1977 Short-term Krat Exclosure           35.8   41.3
##  5  1977 Spectab exclosure                   37.2   47.1
##  6  1978 Control                             38.1   70.8
##  7  1978 Long-term Krat Exclosure            22.6   35.9
##  8  1978 Rodent Exclosure                    37.8   67.3
##  9  1978 Short-term Krat Exclosure           36.9   63.8
## 10  1978 Spectab exclosure                   42.3   80.1
## # ... with 120 more rows
```

6.2 Exporting data and ensuring reprocibility

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

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

```
write_csv(surveys_spread, here("processed_data", "surveys_spread.csv"))
```

- Knit your document
- Commit, Pull, Push

Pretend we just completed a significant chapter of our work:

- Update CHANGELOG
- Make a release

Questions?

7 Analyzing Data

7.1 Importing Data

7.1.1 From Hakai Data Portal API

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

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

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

library('hakaiApi')

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

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

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

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

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

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

If you don't want to learn how to write a querystring yourself there is an option to just copy and paste the querystring from the EIMS Data Portal. Use the portal to select the sample type, and dates and sites you'd like to download as you normally would. To copy the querystring go to the top right of the window where it says Options and click 'Display API query'. You can copy that string in to your endpoint definition in R. Just be sure to copy that string starting from `eims/views/...`, excluding `https://hecate.hakai.org/api/` and then paste that into the definitions of your endpoint and surround that string with single quotes ie: `endpoint <- sprintf("%s/%s", client$api_root, 'eims/views/output/chlorophyll?date>=2016-11-01&date<2018-11-20&work_area&&{"CALVERT"}&site_id&&{"KC13"`

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

The page documenting the API usage can be found here

Once you're happy with the formatting and filtering you've applied to your data make sure to write a new data file that you can read in from your separate analysis script.

In the Console:!!

```
write_csv(here("raw_data", "my_data.csv"))
```

7.2 Annoying things that will get you

7.2.1 Factors

7.2.2 Dates

7.2.3 Joining data

7.3 Advanced Visualization

7.4 CHANGELOG

Having commit messages, and a history of what you've done is great. But if you think of your commit messages as if they are bullet points in your to-do list for your analysis, sometimes you might want a broader overview of what all those bullet points amount to. That's where a changelog comes in.

A changelog will keep track of the major versions of your files. So when you make a major modification, or complete a major component of your analysis you should give it a version number and summarize what changed since your last version, in your CHANGELOG file.

Keep a changelog

Always add newest versions to the top of the file.

Tag your commit message with a release version on GitHub, so that the version of your repository matches the version of your changelog. More on that later.

7.5 Data Packaging

8 Bonus Material

8.1 Collaboration with GitHub