Data Management Workshop for the Pelagic Ecosystems Lab and Hakai Affiliates at UBC

Brett Johnson & Julian Gan

31/01/2020

Contents

1 Pre-workshop material

Did you successfully install and set up all the tools necessary for this workshop?

- Download Slack
- Update or install R
- Update or install R-Studio
- Update or install packages
- Get an account for the Hakai Data Portal from data@hakai.org
- Get Git and GitHub set up

See README

1.1 Objectives

The objectives of this workshop are:

1. 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

2. 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

3. 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

2 Day 1 AM: Intro to R for Data Management

2.1 Vectors and data types

2.1.1 Learning Objectives

- Inspect the content of vectors and manipulate their content.

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`).

2.2 Starting with data

2.2.1 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.

2.2.2 Presentation of the Survey Data

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 "read_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)
library(here)

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

You are now ready to load the data:

```
surveys <- read_csv(here("read_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()
## )
```

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 weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 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     NA
## 4       266    10    16  1977       2 NL         <NA>               NA     NA
```

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

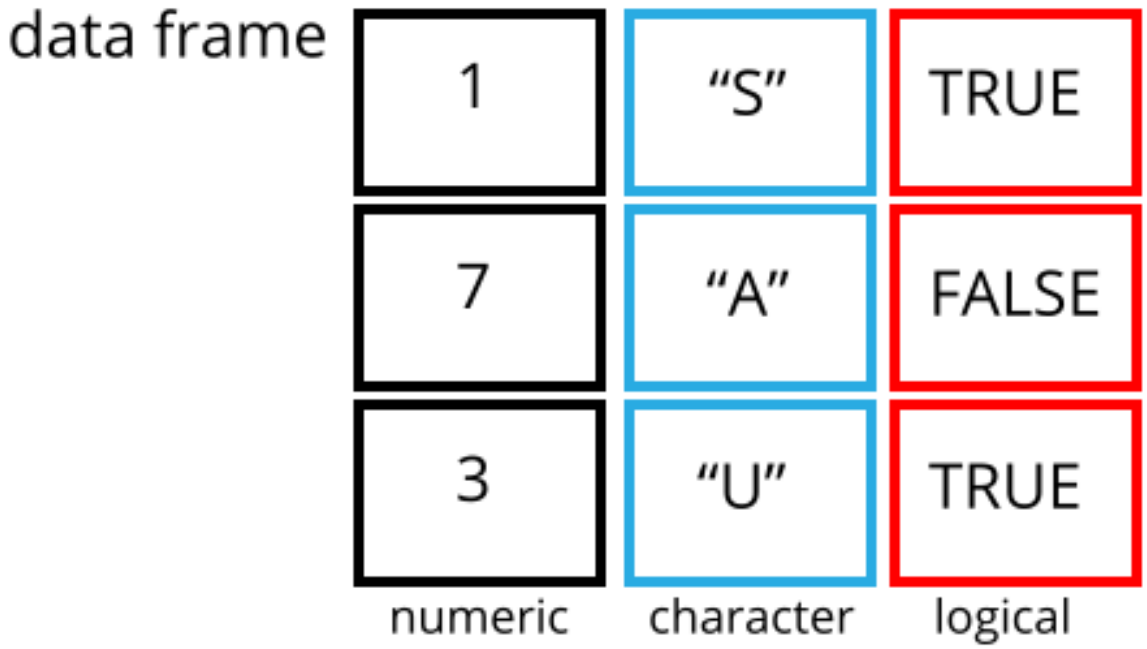
```
## Try also
## View(surveys)
```

2.2.3 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 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 34786 obs. of  13 variables:
##  $ record_id      : num  1 72 224 266 349 363 435 506 588 661 ...
##  $ month          : num  7 8 9 10 11 11 12 1 2 3 ...
##  $ day            : num  16 19 13 16 12 12 10 8 18 11 ...
##  $ year           : num  1977 1977 1977 1977 1977 ...
##  $ plot_id        : num  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: num  32 31 NA NA NA NA NA NA NA NA ...
```

```
##  $ weight         : num  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")=
##   .. 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()
##   .. )
```

2.2.4 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`.

2.2.5 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
##       <dbl>
## 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
##        <dbl>
##  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
##        <dbl>
##  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 weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
```

```
## 1       224     9    13  1977       2 NL         <NA>               NA     NA
## # ... with 4 more variables: 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
##    <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl> <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 weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 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     NA
## 4       266    10    16  1977       2 NL         <NA>               NA     NA
## 5       349    11    12  1977       2 NL         <NA>               NA     NA
## 6       363    11    12  1977       2 NL         <NA>               NA     NA
## # ... with 4 more variables: 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.

2.3 Data Manipulation

2.3.1 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.

2.3.2 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("read_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)
```

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

2.3.3 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)
```

```
## # A tibble: 1,180 x 13
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     22314     6     7  1995       2 NL         M                  34     NA
##  2     22728     9    23  1995       2 NL         F                  32    165
##  3     22899    10    28  1995       2 NL         F                  32    171
##  4     23032    12     2  1995       2 NL         F                  33     NA
##  5     22003     1    11  1995       2 DM         M                  37     41
##  6     22042     2     4  1995       2 DM         F                  36     45
##  7     22044     2     4  1995       2 DM         M                  37     46
##  8     22105     3     4  1995       2 DM         F                  37     49
##  9     22109     3     4  1995       2 DM         M                  37     46
## 10     22168     4     1  1995       2 DM         M                  36     48
## # ... with 1,170 more rows, and 4 more variables: genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>
```

2.3.4 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>  <dbl>
##  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>  <dbl>
##  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.

2.3.5 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)
```

2.3.6 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 weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  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     NA
##  4       266    10    16  1977       2 NL         <NA>               NA     NA
##  5       349    11    12  1977       2 NL         <NA>               NA     NA
##  6       363    11    12  1977       2 NL         <NA>               NA     NA
##  7       435    12    10  1977       2 NL         <NA>               NA     NA
```

```
##  8       506     1     8  1978       2 NL         <NA>               NA     NA
##  9       588     2    18  1978       2 NL         M                  NA    218
## 10       661     3    11  1978       2 NL         <NA>               NA     NA
## # ... with 34,776 more rows, and 5 more variables: 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 weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  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     NA
##  4       266    10    16  1977       2 NL         <NA>               NA     NA
##  5       349    11    12  1977       2 NL         <NA>               NA     NA
##  6       363    11    12  1977       2 NL         <NA>               NA     NA
##  7       435    12    10  1977       2 NL         <NA>               NA     NA
##  8       506     1     8  1978       2 NL         <NA>               NA     NA
##  9       588     2    18  1978       2 NL         M                  NA    218
## 10       661     3    11  1978       2 NL         <NA>               NA     NA
## # ... with 34,776 more rows, and 6 more variables: 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 weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 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     NA
## 4       266    10    16  1977       2 NL         <NA>               NA     NA
## 5       349    11    12  1977       2 NL         <NA>               NA     NA
## 6       363    11    12  1977       2 NL         <NA>               NA     NA
## # ... with 5 more variables: 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 weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
```

```
## 1       588     2    18  1978       2 NL         M                  NA    218
## 2       845     5     6  1978       2 NL         M                  32    204
## 3       990     6     9  1978       2 NL         M                  NA    200
## 4      1164     8     5  1978       2 NL         M                  34    199
## 5      1261     9     4  1978       2 NL         M                  32    197
## 6      1453    11     5  1978       2 NL         M                  NA    218
## # ... with 5 more variables: 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

2.3.7 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)
```

2.3.8 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 [3]
##    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 [3]
##    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 [3]
##    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 [3]
##    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
```

2.3.9 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
```

2.3.10 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_length min_hindfoot_length max_hindfoot_length     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
##    <dbl> <chr>     <chr>        <dbl>
##  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
```

2.4 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 processed data sets to share them with your collaborators or for archival.

Never write to `read_data/`

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_sml, here("write_data", "surveys_sml.csv"))
```

- Knit your document
- Commit, Pull, Push

Questions?

2.5 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, `pivot_wider()` and `pivot_longer()`.

2.5.1 Widening (aka. Spreading)

`pivot_wider()` makes a dataset wider by increasing the number of columns and decreasing the number of rows. It takes three principal arguments:

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

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

Let's use `pivot_wider()` 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 `summarize()` to filter our observartions 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)
```

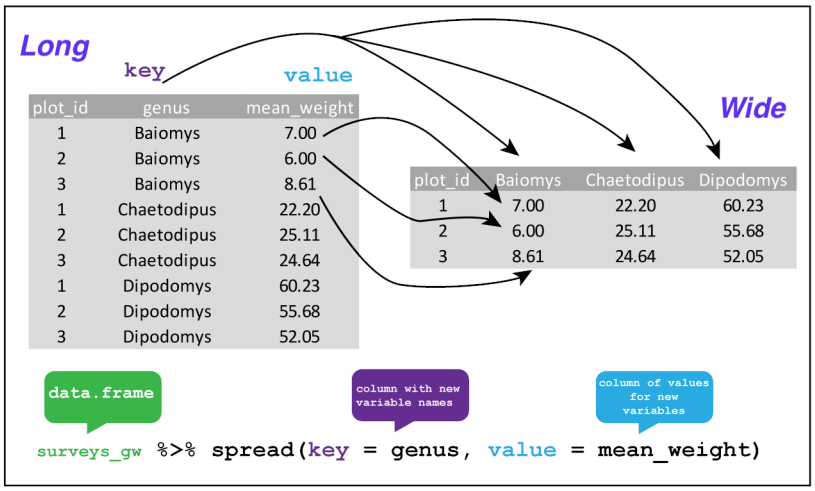
```
## Classes 'grouped_df', 'tbl_df', 'tbl' and 'data.frame':  196 obs. of  3 variables:
##  $ genus      : chr  "Baiomys" "Baiomys" "Baiomys" "Baiomys" ...
##  $ plot_id    : num  1 2 3 5 18 19 20 21 1 2 ...
##  $ mean_weight: num  7 6 8.61 7.75 9.5 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   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()
##   .. )
##  - attr(*, "groups")=Classes 'tbl_df', 'tbl' and 'data.frame':   10 obs. of  2 variables:
##   ..$ genus: chr  "Baiomys" "Chaetodipus" "Dipodomys" "Neotoma" ...
##   ..$ .rows:List of 10
##   .. ..$ : int  1 2 3 4 5 6 7 8
##   .. ..$ : int  9 10 11 12 13 14 15 16 17 18 ...
##   .. ..$ : int  33 34 35 36 37 38 39 40 41 42 ...
##   .. ..$ : int  57 58 59 60 61 62 63 64 65 66 ...
##   .. ..$ : int  81 82 83 84 85 86 87 88 89 90 ...
##   .. ..$ : int  105 106 107 108 109 110 111 112 113 114 ...
##   .. ..$ : int  128 129 130 131 132 133 134 135 136 137 ...
##   .. ..$ : int  152 153 154 155 156 157 158 159 160 161 ...
##   .. ..$ : int  176 177 178 179 180 181 182 183 184 185 ...
##   .. ..$ : int  195 196
##   ..- 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 `pivot_wider()` to names from `genus` with values from `mean_weight` this becomes 24 observations of 11 variables, one row for each plot. We again use pipes:

```
surveys_wide <- surveys_gw %>%
  pivot_wider(names_from = genus, values_from = mean_weight)

str(surveys_wide)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    24 obs. of  11 variables:
##  $ plot_id        : num  1 2 3 5 18 19 20 21 4 6 ...
##  $ Baiomys        : num  7 6 8.61 7.75 9.5 ...
##  $ Chaetodipus    : num  22.2 25.1 24.6 18 26.8 ...
##  $ Dipodomys      : num  60.2 55.7 52 51.1 61.4 ...
##  $ Neotoma        : num  156 169 158 190 149 ...
##  $ Onychomys      : num  27.7 26.9 26 27 26.6 ...
##  $ Perognathus    : num  9.62 6.95 7.51 8.66 8.62 ...
##  $ Peromyscus     : num  22.2 22.3 21.4 21.2 21.4 ...
##  $ Reithrodontomys: num  11.4 10.7 10.5 11.2 11.1 ...
##  $ Sigmodon       : num  NA 70.9 65.6 82.7 46.1 ...
##  $ Spermophilus   : num  NA NA NA NA NA NA 57 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 Peromyscus
##     <dbl>   <dbl>       <dbl>     <dbl>   <dbl>     <dbl>       <dbl>      <dbl>
## 1       1    7           22.2      60.2    156.      27.7        9.62       22.2
## 2       2    6           25.1      55.7    169.      26.9        6.95       22.3
## 3       3    8.61        24.6      52.0    158.      26.0        7.51       21.4
## 4       4    0           23.0      57.5    164.      28.1        7.82       22.6
## 5       5    7.75        18.0      51.1    190.      27.0        8.66       21.2
## 6       6    0           24.9      58.6    180.      25.9        7.81       21.8
## # ... with 3 more variables: Reithrodontomys <dbl>, Sigmodon <dbl>,
## #   Spermophilus <dbl>
```

2.5.2 Lengthening (aka. 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.

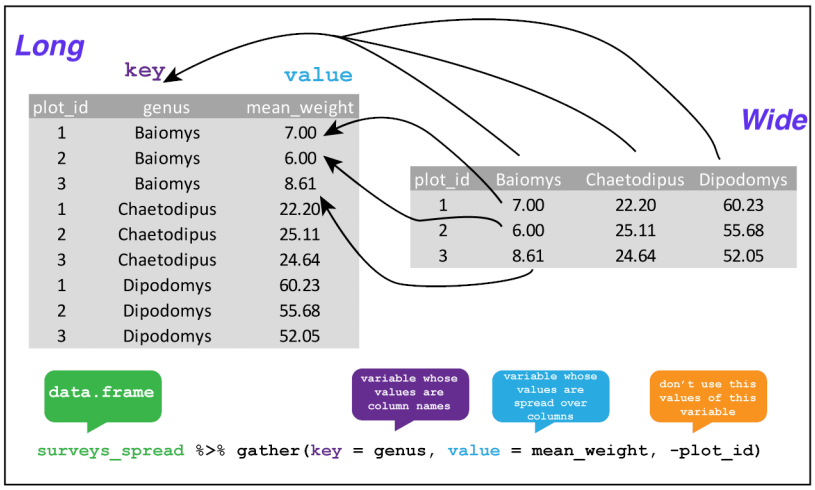
`pivot_longer()` takes four principal arguments:

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

To recreate `surveys_gw` from `surveys_wide` 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_long <- surveys_wide %>%
  pivot_longer(names_to = "genus", values_to = "mean_weight", -plot_id)

str(surveys_long)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    240 obs. of  3 variables:
##  $ plot_id    : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ genus      : chr  "Baiomys" "Chaetodipus" "Dipodomys" "Neotoma" ...
##  $ mean_weight: num  7 22.2 60.2 156.2 27.7 ...
```



Note that now the `NA` genera are included in the re-lengthened format. WIdening and then lengthening 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_wide %>%
  pivot_longer(names_to = "genus", values_to = "mean_weight", Baiomys:Spermophilus) %>%
  head()
```

```
## # A tibble: 6 x 3
##   plot_id genus       mean_weight
##     <dbl> <chr>             <dbl>
## 1       1 Baiomys            7
## 2       1 Chaetodipus       22.2
## 3       1 Dipodomys         60.2
## 4       1 Neotoma          156.
## 5       1 Onychomys         27.7
## 6       1 Perognathus        9.62
```

2.5.3 Challenge

1. Widen 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)) %>%
  pivot_wider(names_from = year, values_from = 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` `1986`
##     <dbl>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>
## 1       1      2      3      4      7      5      6      7      6      4      3
## 2       2      6      6      6      8      5      9      9      9      6      4
## 3       3      5      6      4      6      6      8     10     11      7      6
## 4       4      4      4      3      4      5      4      6      3      4      3
## 5       5      4      3      2      5      4      6      7      7      3      1
## 6       6      3      4      3      4      5      9      9      7      5      6
## # ... with 16 more variables: `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 `pivot_longer()` it again, so each row is a unique `plot_id` by `year` combination.

```
rich_time %>%
  pivot_longer(names_to = "year", values_to = "n_genera", -plot_id)
```

```
## # A tibble: 624 x 3
## # Groups:   plot_id [24]
##    plot_id year  n_genera
##      <dbl> <chr>    <int>
##  1       1 1977         2
##  2       1 1978         3
##  3       1 1979         4
##  4       1 1980         7
##  5       1 1981         5
##  6       1 1982         6
##  7       1 1983         7
##  8       1 1984         6
##  9       1 1985         4
## 10       1 1986         3
## # ... 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 `pivot_longer()` 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 %>%
  pivot_longer(names_to = "measurement", values_to = "value", cols = c(hindfoot_length, weight))
```

4. With this new data set, calculate the average of each `measurement` in each `year` for each different `plot_type`. Then `pivot_wider()` 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 `pivot_wider()`.

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

```
## # A tibble: 130 x 4
## # Groups:   year [26]
##     year plot_type                  hindfoot_length weight
##    <dbl> <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
```

3 Day 1 PM: Intermediate Topics in R

3.1 R Markdown

3.2 Data Quality Reports

3.3 Data Visualization Including Maps

3.3.1 Plotting with `ggplot2`

`ggplot2` is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties.

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

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

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

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

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

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

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

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

- add 'geoms' – graphical representations of the data in the plot (points, lines, bars). **ggplot2** offers many different geoms; we will use some common ones today, including:

  * `geom_point()` for scatter plots, dot plots, etc.
  * `geom_boxplot()` for, well, boxplots!
  * `geom_line()` for trend lines, time series, etc.

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.

3.3.2 ggplot2 themes

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

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

3.3.2.1 Faceting

- `facet_wrap()`
- `facet_grid()`

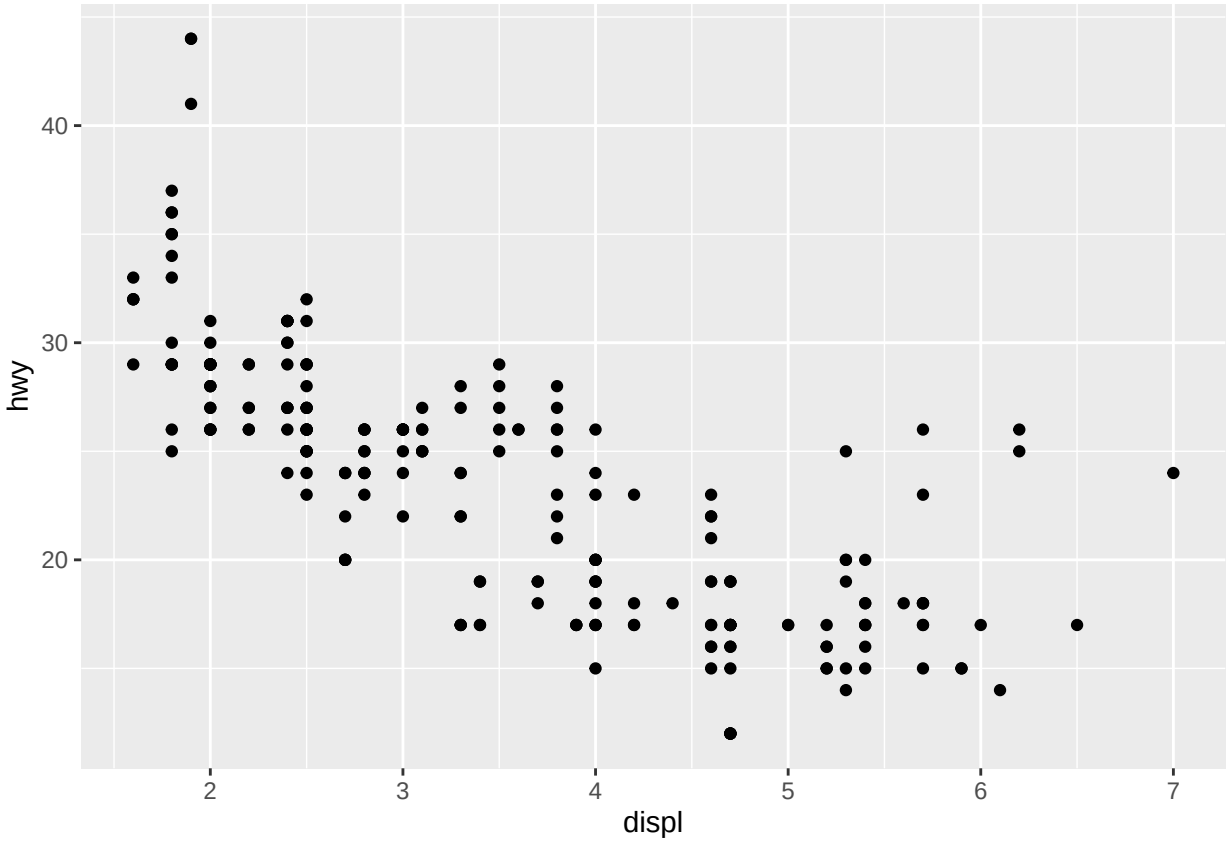
3.3.2.2 Arranging Multiple Plots

`cowplot()`

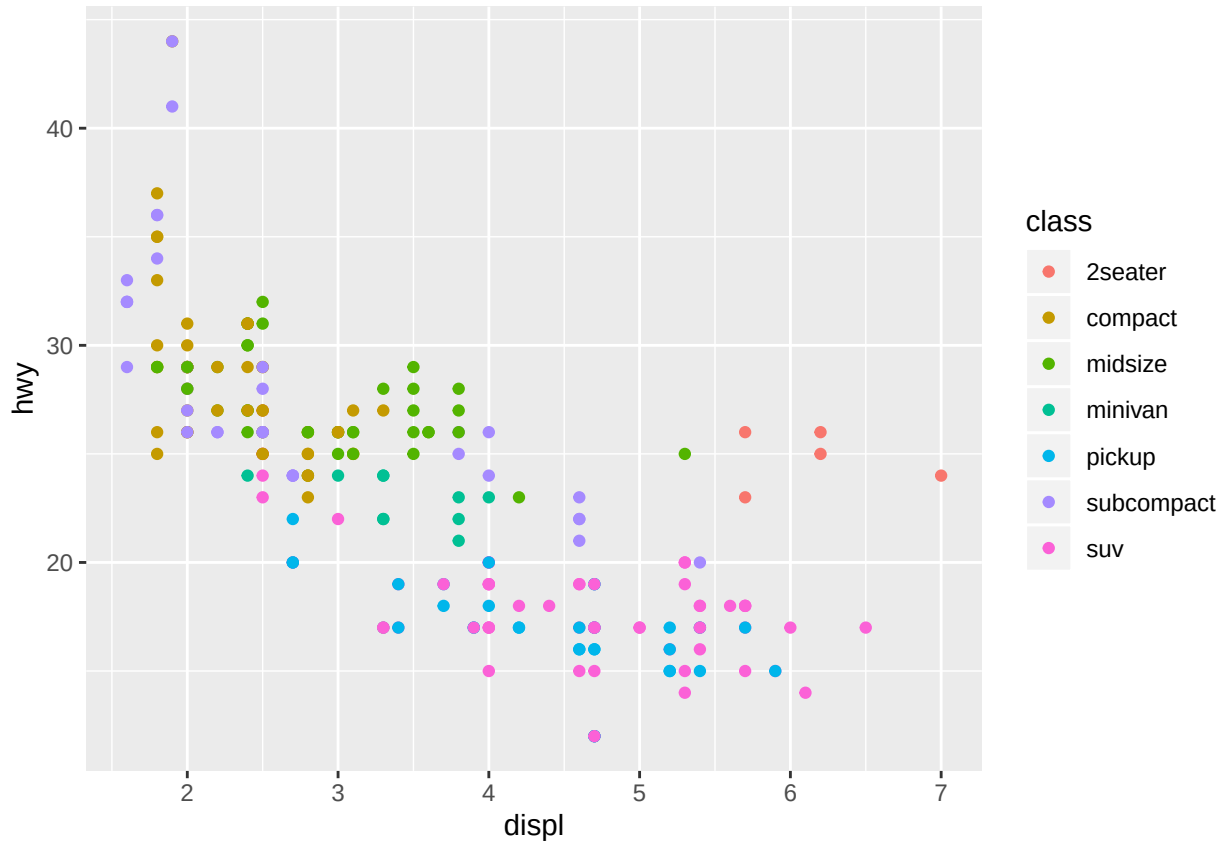
```
library(tidyverse)

# 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(l~ f        18    29 p     comp~
##  2 audi         a4         1.8  1999     4 manual~ f        21    29 p     comp~
##  3 audi         a4         2    2008     4 manual~ f        20    31 p     comp~
##  4 audi         a4         2    2008     4 auto(a~ f        21    30 p     comp~
##  5 audi         a4         2.8  1999     6 auto(l~ f        16    26 p     comp~
##  6 audi         a4         2.8  1999     6 manual~ f        18    26 p     comp~
##  7 audi         a4         3.1  2008     6 auto(a~ f        18    27 p     comp~
##  8 audi         a4 quat~   1.8  1999     4 manual~ 4        18    26 p     comp~
##  9 audi         a4 quat~   1.8  1999     4 auto(l~ 4        16    25 p     comp~
## 10 audi         a4 quat~   2    2008     4 manual~ 4        20    28 p     comp~
## # ... with 224 more rows
```

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

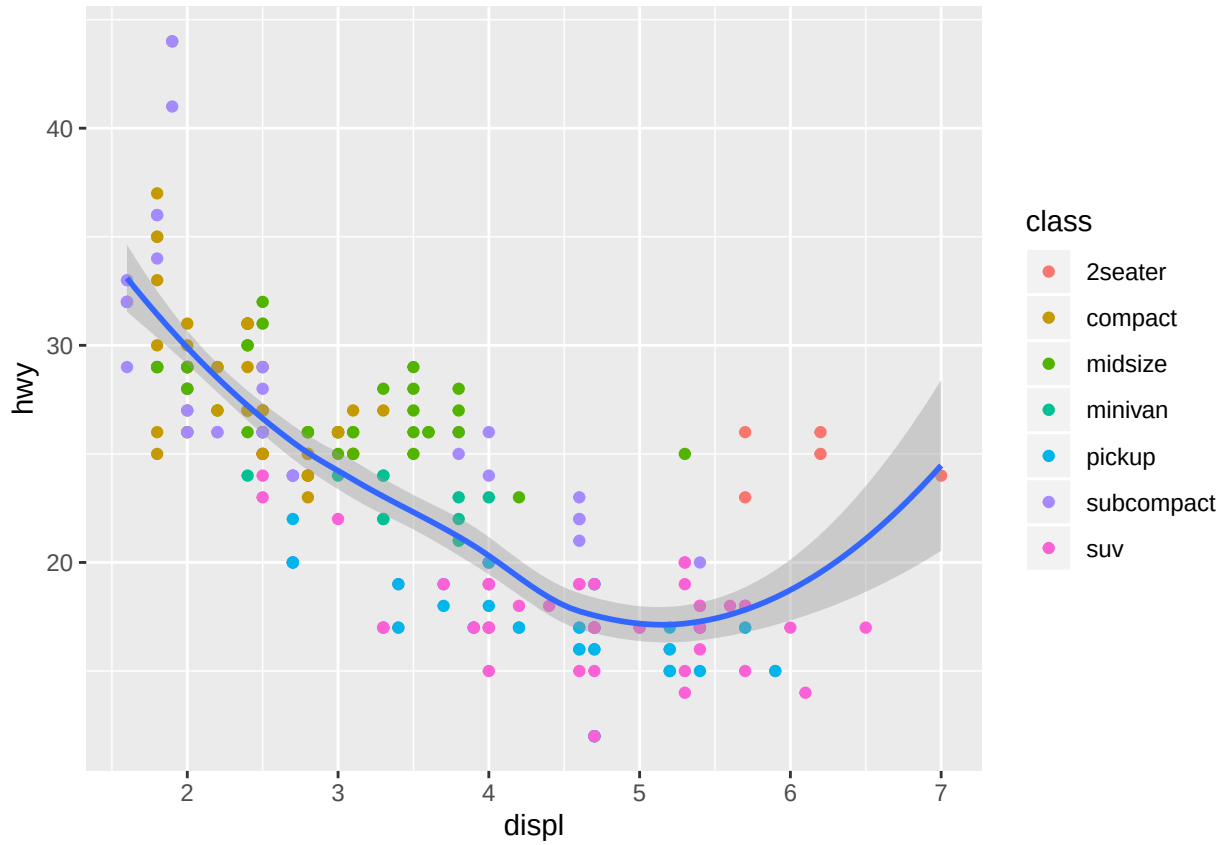



```r
# What about this other column class? Maybe we want to see what type of car it is too

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

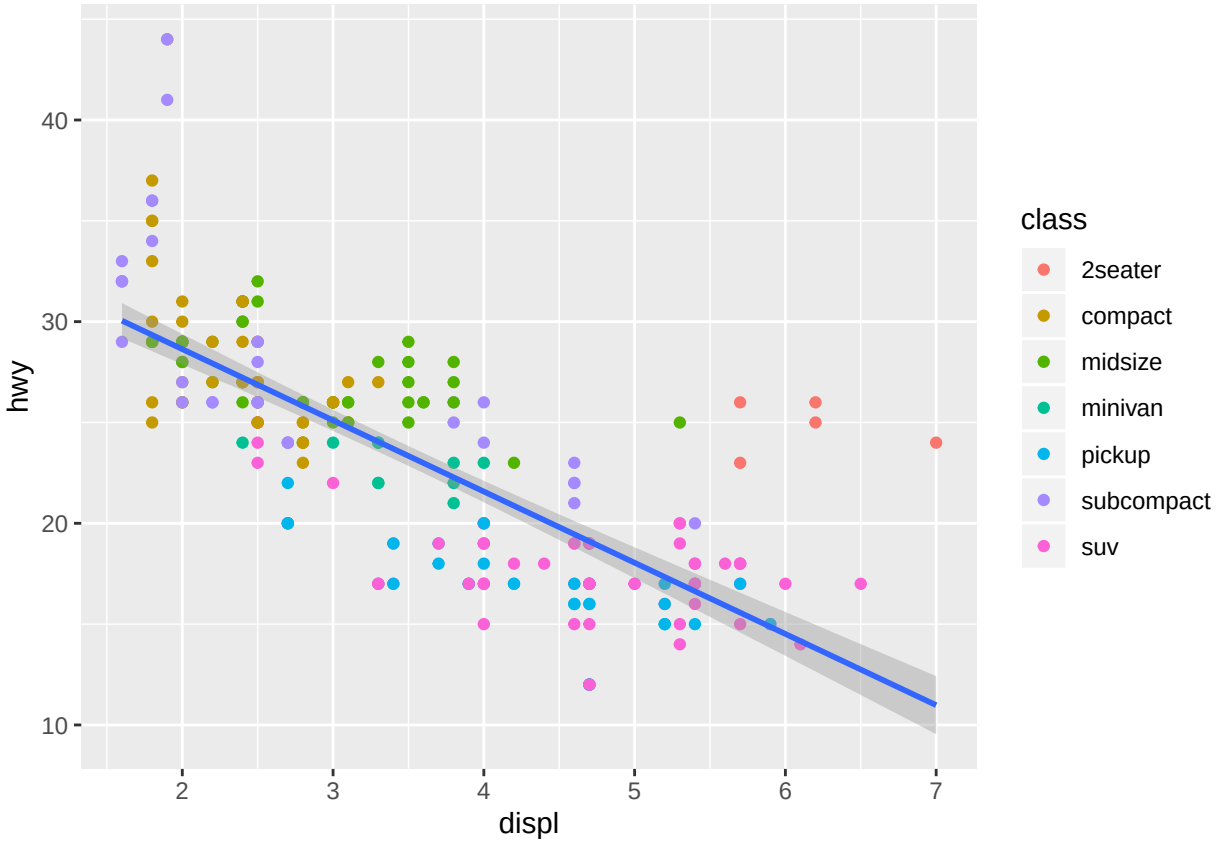



```
# It would be nice to see a trend line

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)
```




```
ggsave(here("figs", "mpg.png"))
```

```
## Saving 6.5 x 4.5 in image
```

3.3.3 Site Maps

Section Contributors:

- Dr. Daniel Okamoto
- Jenn Burt

A common task amongst most field researchers is the need to make a basic site map to describe the location of your sampling. Using Arc GIS is the most common way to produce maps at Hakai, but sometimes a simple solution that could be implemented in R is desired.

For a high resolution map with the resolution needed to see detailed coastline features you can use the following code and shapefile. To get this to work on your computer, download the shape file and put it in a R Studio project sub-folder called data.

```
knitr::opts_chunk$set(message = FALSE, warning = FALSE)
########################################################### #
###  Script to make a BC map                            ### #
###  Author:  D.K. Okamoto (modified by Jenn Burt)  ### #
########################################################### #

# Libraries needed to run this code
library(raster)
library(maps)
library(mapdata)
```

```
library(maptools)
library(mapproj)
library(rgeos)
library(rgdal)
library(ggplot2)
library(ggsn)
library(tidyverse)
library(here)
```

3.3.3.1 High Resolution Maps

The high resolution map used here requires that you download a set of ESRI shape files from this book's GitHUB repository. Those files can be downloaded from the `2_Shapefile` folder here. Put the `2_Shapefile` folder into the data folder of your R-Studio project.

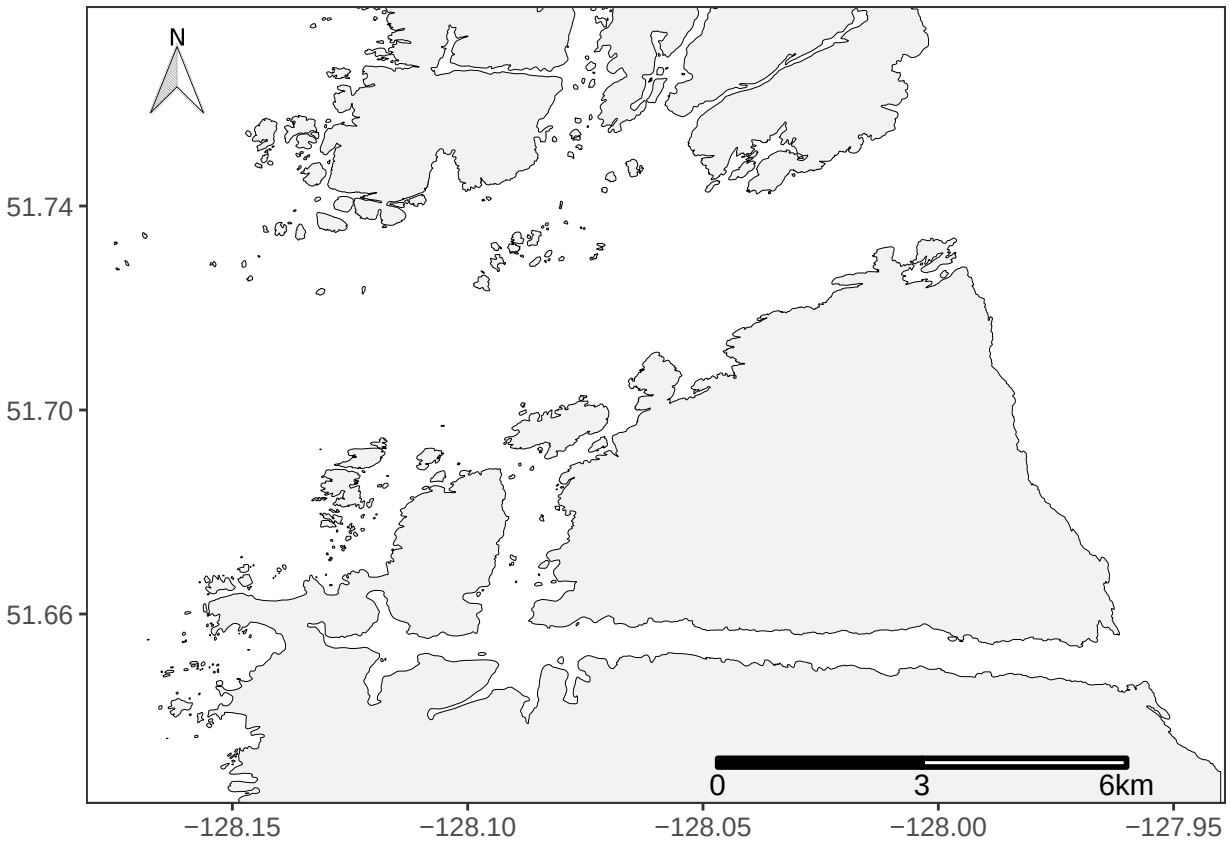
This script assumes you are using the `here()` package in conjunction with R-Studio projects to obviate setting your working directory.

```
############### Make a map with sites ##################
########## Using high resolution shapefile ################

BC.shp <- readOGR(here("read_data","2_Shapefile", "COAST_TEST2.shp"))
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\Julian\Documents\Work\Hakai\GitHub\Pelagic-Ecosystems\R-workshop\read_data\2_Shape
## with 1 features
## It has 5 fields
```

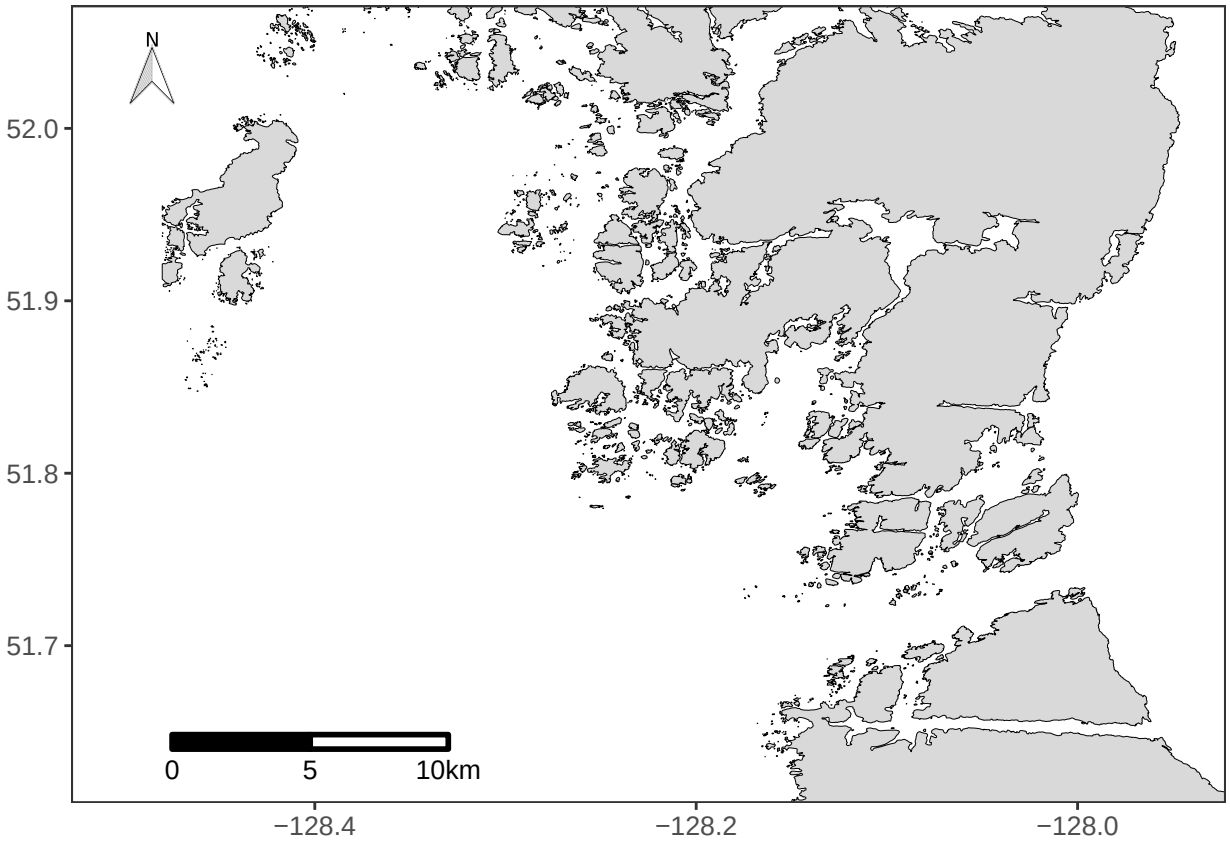
```
### chose the lat/long extent you want to show
Ncalvert <- extent(-128.18, -127.94, 51.61, 51.78)

### crop your shapefile polygons to the extent defined
# takes a moment to run (patience grasshopper)
BC.shp2 <- crop(BC.shp,Ncalvert)

### project and fortify (i.e. turn into a dataframe)
BC.df <- fortify(BC.shp2)

# (IF DESIRED) Load .csv file with your specific study site lat/longs
# this file is a dataframe with 4 columns: site_name, otterOcc(Y or N), lat, long
# EXPT
# sites <- read.csv("/Users/jennb/Dropbox/Simple_BC_map/EXPTsites.csv", header = T)

# Jenn graph
# here is where you can see the styles of north arrow (scroll to
# bottom): http://oswaldosantos.github.io/ggsn/
# the high resolution shape file works well at this scale
# as it gives lots of the coastline detail
ggplot() + theme_bw() +
  geom_polygon(
    data = BC.df,
    aes(x = long, y = lat, group = group),
    colour = "black",
    size = 0.1,
    fill = 'grey95'
```

```
) +
coord_cartesian(xlim = c(-128.17, -127.95),
                ylim = c(51.63, 51.772)) +
scalebar(
  BC.df,
  dist = 3,
  dist_unit = "km",
  st.size = 4,
  height = 0.01,
  transform = TRUE,
  model = 'WGS84',
  anchor = c(x = -127.96, y = 51.63)
) +
north(
  data = BC.df,
  scale = 0.1,
  symbol = 3,
  anchor = c(x = -128.15, y = 51.775)
) +
theme(
  panel.grid.minor = element_line(colour = NA),
  panel.grid.major = element_line(colour = NA),
  axis.title.y = element_blank(),
  axis.title.x = element_blank(),
  axis.text.y = element_text(size = 10),
  axis.text.x = element_text(size = 10)
)
```




```
### if you want to make a larger Central coast map, just change the extent selected
CCoast <- extent(-128.48, -127.9, 51.5, 52.1)
# crop the map to the new extent
CC.shp2 <- crop(BC.shp,CCoast)
# fortify
CC.df <- fortify(CC.shp2)

# Jenn graph
fig1 <- ggplot() + theme_bw() +
  geom_polygon(
    data = CC.df,
    aes(x = long, y = lat, group = group),
    colour = "black",
    size = 0.1,
    fill = 'grey85'
  ) +
  coord_cartesian(xlim = c(-128.5, -127.95),
                  ylim = c(51.63, 52.05)) +
  scale_x_continuous(breaks = c(-128.4, -128.2, -128.0)) +
  scalebar(
    CC.df,
    dist = 5,
    dist_unit = "km",
    st.size = 3.5,
    height = 0.014,
    transform = TRUE,
```

```
    model = 'WGS84',
    anchor = c(x = -128.33, y = 51.64)
  ) +
  north(
    data = CC.df,
    scale = 0.07,
    symbol = 3,
    anchor = c(x = -128.465, y = 52.056)
  ) +
  theme(
    panel.grid.minor = element_line(colour = NA),
    panel.grid.major = element_line(colour = NA),
    axis.title.y = element_blank(),
    axis.title.x = element_blank(),
    axis.text.y = element_text(size = 10),
    axis.text.x = element_text(size = 10),
    legend.position = "none"
  )
fig1
```




```
# I use this code to export a nice PDF file of specific dimensions.
cairo_pdf("Fig1.pdf", width=4, height=5)
print(fig1)
dev.off()
```

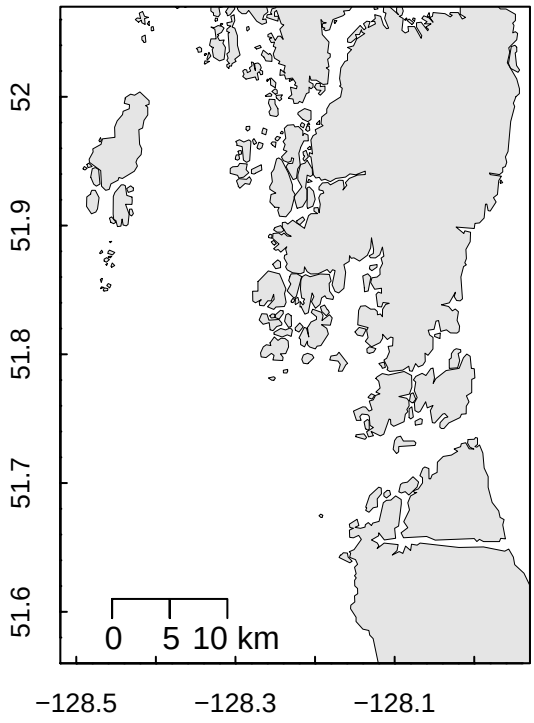
```
## pdf
```

```
##   2
```

3.3.3.2 Medium Resolution PBS Mapping Package

The Pacific Biological Station in Nanaimo has put together a mapping package that contains some medium resolution files of the Pacific Coast.

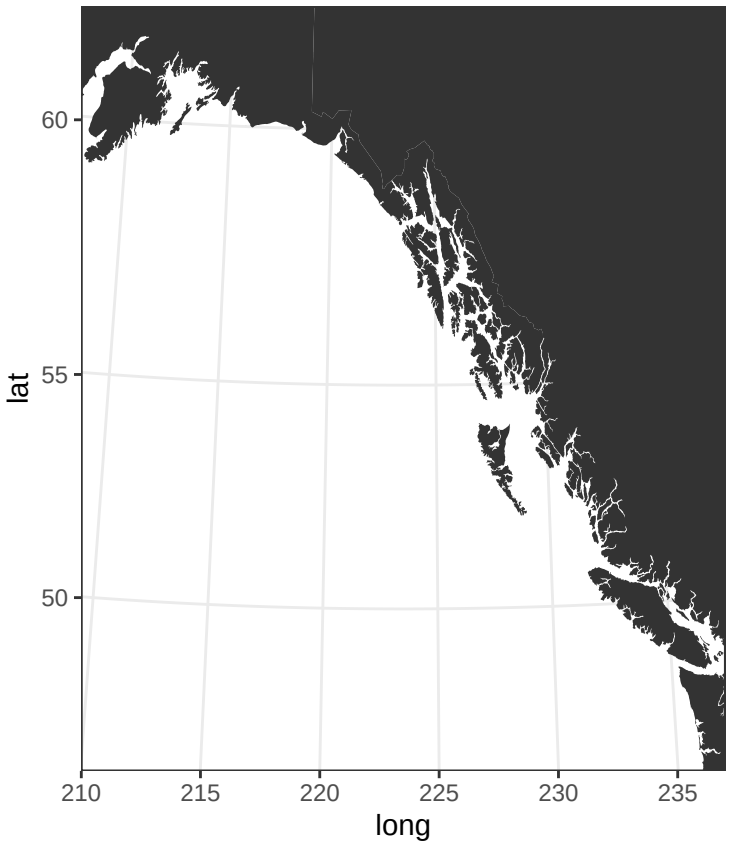
```
############## Make a map with the sites ################## #
############# Using DFO coastline data file ################

#this is lower resolution than the shapefile above

library(PBSmapping)

## Plot the map
data(nepacLLhigh)       # DFO BC Coastline data - high resolution
plotMap(
  nepacLLhigh,
  xlim = c(-128.52, -127.93),
  ylim = c(51.56, 52.07),
  col = "grey90",
  bg = "white",
  tckMinor = 0,
  xlab = "",
  ylab = "",
  lwd = 0.5
)
box()

# add a scale bar
map.scale(
  x = -128.455,
  y = 51.61,
  ratio = FALSE,
  relwidth = 0.2
)
```



3.3.3.3 Low Resolution Pacific Coast Maps

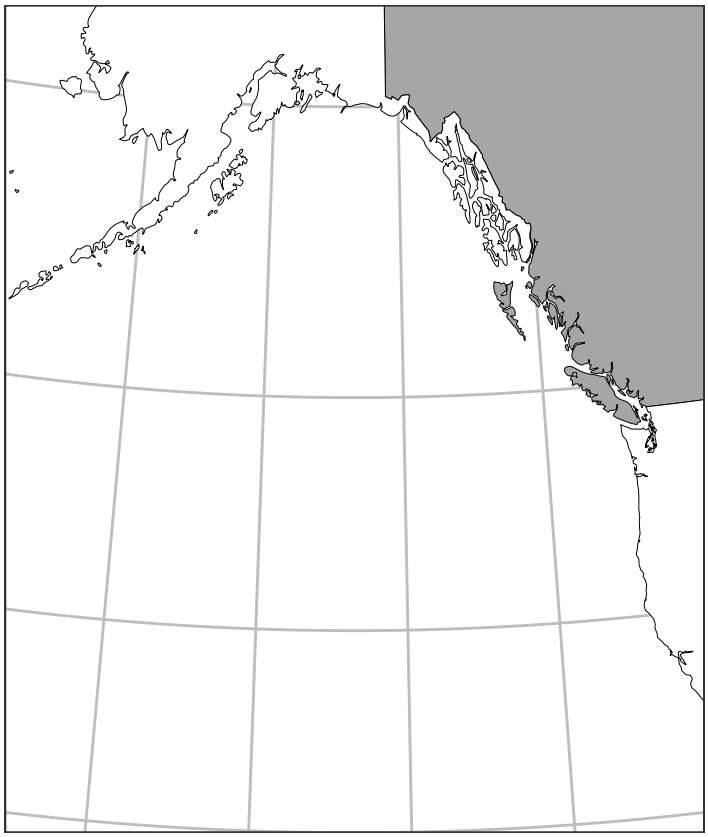
For lower resolution maps to represent larger scales you can use the maps from the `maps` and `mapdata` packages.

```
############## Pacific Coast Map ##################
################################################### #

# create a data file to make a basemap
# this database has a lower resolution (which is fine for large scale map)
m <- map_data("world", c("usa", "Canada"))

# this database has a way higher resolution
d <- map_data("worldHires", c("Canada", "usa", "Mexico"))

# make a basic map, all one colour
# play around with xlim and ylim to change the extent
ggplot() + geom_polygon(data = d, aes(x=long, y = lat, group = group)) + theme_bw()+
  coord_map("conic", lat0 = 18, xlim=c(210, 237), ylim=c(46,62))
```

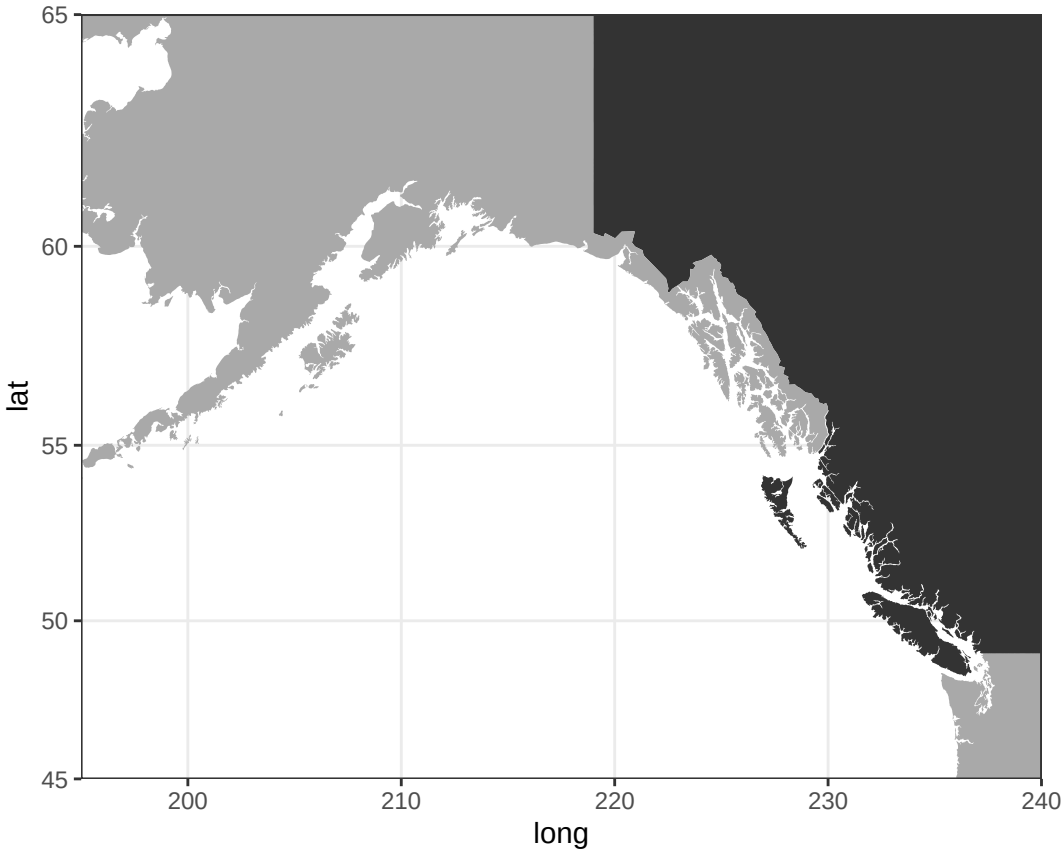



```
ggplot() +
  geom_polygon(
    data = subset(m, region == "Canada"),
    aes(x = long, y = lat, group = group),
    fill = "grey65",
    colour = "black",
    size = .1
  ) + theme_bw() +
  geom_polygon(
    data = subset(m, region == "USA"),
    aes(x = long, y = lat, group = group),
    fill = "white",
    colour = "black",
    size = .1
  ) +
  coord_map(
    "conic",
    lat0 = 18,
    xlim = c(195, 238),
    ylim = c(30, 62.5)
  ) +
  theme(
    panel.grid.minor = element_blank(),
    panel.grid.major = element_line(colour = "grey"),
    #change "grey" to NA to remove
    axis.title = element_blank(),
```

```
    axis.text = element_blank(),
    axis.ticks = element_blank()
  )
```

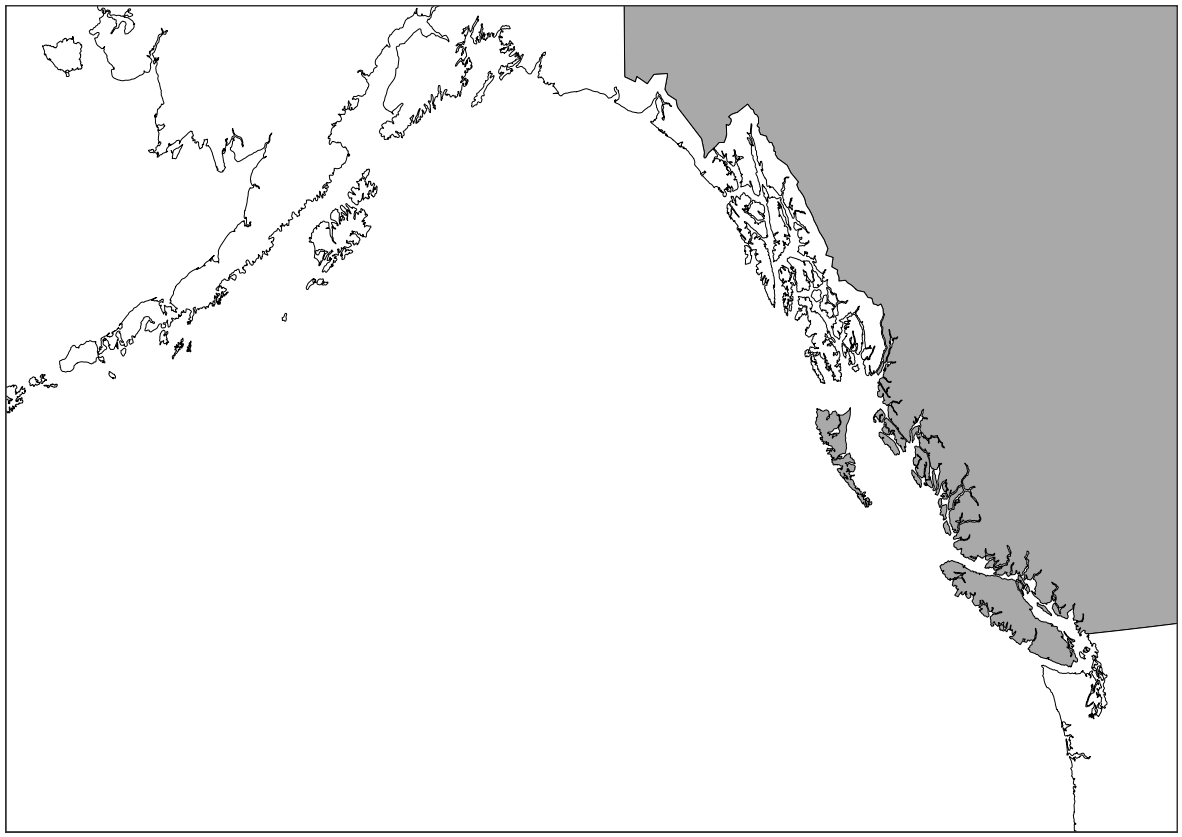


```
# playing with extent and colour
ggplot() +
  geom_polygon(data = subset(d, region == "Canada"), aes(x = long, y = lat, group = group)) +
  geom_polygon(data = subset(d, region == "USA"),
               aes(x = long, y = lat, group = group),
               fill = "darkgrey") +
  coord_map(xlim = c(195, 240), ylim = c(45, 65)) +
  theme_bw()
```




```
# here you can see that if you use the other dataframe "d" the resolution is much higher.
# this is good for smaller chunks of the BC coast, but less good for a PNW map
ggplot() +
  geom_polygon(
    data = subset(d, region == "Canada"),
    aes(x = long, y = lat, group = group),
    fill = "darkgrey",
    colour = "black",
    size = .1
  ) + theme_bw() +
  geom_polygon(
    data = subset(d, region == "USA"),
    aes(x = long, y = lat, group = group),
    fill = "white",
    colour = "black",
    size = .1
  ) +
  coord_map(
    "conic",
    lat0 = 18,
    xlim = c(195, 240),
    ylim = c(45, 61)
  ) +
  theme(
    panel.grid.minor = element_line(colour = NA),
    panel.grid.major = element_line(colour = NA),
```

```
    axis.title = element_blank(),
    axis.text = element_blank(),
    axis.ticks = element_blank()
  )
```



3.4 Dates and Times in R