February 3, 2023

The results below are generated from an R script.

```
---
title: Manipulating, analyzing and exporting data with tidyverse
author: Data Carpentry contributors
---

```{r, echo=FALSE, purl=FALSE, message = FALSE}
source("setup.R")
surveys <- read.csv("data_raw/portal_data_joined.csv")
suppressWarnings(surveys$date <- lubridate::ymd(paste(surveys$year,
                                                      surveys$month,
                                                      surveys$day,
                                                      sep = "-")))
```

### Manipulating and analyzing data with dplyr and tidyr

## Error:  <text>:2:20:  unexpected ','
## 1:  --
## 2:  title:  Manipulating,
##                         ^
```

```
> ### Learning Objectives
>
> * Describe the purpose of the **‘dplyr‘** and **‘tidyr‘** packages.
> * Select certain columns in a data frame with the **‘dplyr‘** function ‘select‘.
> * Extract certain rows in a data frame according to logical (boolean) conditions with the **‘dplyr‘**
> * Link the output of one **‘dplyr‘** function to the input of another function with the 'pipe' operato
> * Add new columns to a data frame that are functions of existing columns with ‘mutate‘.
> * Use the split-apply-combine concept for data analysis.
> * Use ‘summarize‘, ‘group_by‘, and ‘count‘ to split a data frame into groups of observations, apply su
> * Describe the concept of a wide and a long table format and for which purpose those formats are usefu
> * Describe what key-value pairs are.
> * Reshape a data frame from long to wide format and back with the ‘pivot_wider‘ and ‘pivot_longer‘ com
> * Export a data frame to a .csv file.

## Error:  <text>:2:1:  unexpected '>'
## 1:
## 2:  >
##     ^
```

```
# Data manipulation using **‘dplyr‘** and **‘tidyr‘**

Bracket subsetting is handy, but it can be cumbersome and difficult to read,
especially for complicated operations. Enter **‘dplyr‘**. **‘dplyr‘** is a package for
helping with tabular data manipulation. It pairs nicely with **‘tidyr‘** which enables you to swiftly co

The **‘tidyverse‘** package is an
"umbrella-package" that installs **‘tidyr‘**, **‘dplyr‘**, and several other useful packages for data an

The **‘tidyverse‘** package tries to address 3 common issues that arise when
doing data analysis in R:

1. The results from a base R function sometimes depend on the type of data.
2. R expressions are used in a non standard way, which can be confusing for new
   learners.
3. The existence of hidden arguments having default operations that new learners are not aware
   of.

You should already have installed and loaded the **‘tidyverse‘** package.
If you haven’t already done so, you can type ‘install.packages("tidyverse")‘ straight into the console.

## What are **‘dplyr‘** and **‘tidyr‘**?

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

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

The package **‘tidyr‘** addresses the common problem of wanting to reshape your data for
plotting and usage by different R functions. For example, sometimes we want data sets where we have one
row per measurement. Other times we want a data frame where each measurement type has its
own column, and rows are instead more aggregated groups
(e.g., a time period, an experimental unit like a plot or a batch number).
Moving back and forth between these formats is non-trivial, and **‘tidyr‘** gives you tools
for this and more sophisticated  data manipulation.

To learn more about **‘dplyr‘** and **‘tidyr‘** after the workshop, you may want to check out this
[handy data transformation with **‘dplyr‘** cheatsheet](https://raw.githubusercontent.com/rstudio/cheats
and this [one about **‘tidyr‘**](https://raw.githubusercontent.com/rstudio/cheatsheets/main/data-import.

As before, we’ll read in our data using the ‘read_csv()‘ function from the
tidyverse package **‘readr‘**.
```

```{r, results = 'hide', purl = FALSE}
surveys <- read_csv("data_raw/portal_data_joined.csv")
```

```{r, results = 'hide', purl = FALSE}
## inspect the data
str(surveys)
```

```{r, eval=FALSE, purl=FALSE}
## preview the data
view(surveys)
```

Next, we're going to learn some of the most common **`dplyr`** functions:

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

## Selecting columns and filtering rows

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

```{r, results = 'hide', purl = FALSE}
select(surveys, plot_id, species_id, weight)
```

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

```{r, results = 'hide', purl = FALSE}
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 criterion, use `filter()`:

```{r, purl = FALSE}
filter(surveys, year == 1995)
```

## Pipes

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

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

```{r, purl = FALSE}
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 individua

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

```{r, purl = FALSE}
surveys_sml <- select(filter(surveys, weight < 5), species_id, sex, weight)
```

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

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

```{r, purl = FALSE}
surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)
```

In the above code, we use the pipe to send the `surveys` dataset first through
`filter()` to keep rows where `weight` is less than 5, then through `select()`
to keep only the `species_id`, `sex`, and `weight` columns. Since `%>%` takes
the object on its left and passes it as the first argument to the function on
its right, we don't need to explicitly include the data frame as an argument
to the `filter()` and `select()` functions any more.

Some may find it helpful to read the pipe like the word "then." For instance,
in the example above, 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:

```{r, purl = FALSE}
surveys_sml <- surveys %>%
  filter(weight < 5) %>%
```
```

```
  select(species_id, sex, weight)

surveys_sml
```

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

```
> ### Challenge {.challenge}
>
>  Using pipes, subset the `surveys` data to include animals collected before
>  1995 and retain only the columns `year`, `sex`, and `weight`.
>
>
> ```{r, answer=TRUE, eval=FALSE, purl=FALSE}
> surveys %>%
>     filter(year < 1995) %>%
>     select(year, sex, weight)
> ```

```{r, eval=FALSE, purl=TRUE, echo=FALSE}
## Pipes Challenge:
##  Using pipes, subset the data to include animals collected
##  before 1995, and retain the columns `year`, `sex`, and `weight.`
```

### 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:

```{r, purl = FALSE}
surveys %>%
  mutate(weight_kg = weight / 1000)
```

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

```{r, purl = FALSE}
surveys %>%
  mutate(weight_kg = weight / 1000,
         weight_lb = weight_kg * 2.2)
```

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

```{r, purl = FALSE}
surveys %>%
  mutate(weight_kg = weight / 1000) %>%
```
```

```
  head()
```

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

```{r, purl = FALSE}
surveys %>%
  filter(!is.na(weight)) %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

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

> ### Challenge {.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_cm' containing the 'hindfoot_length' values (currently in mm)
>  converted to centimeters.
>  In this 'hindfoot_cm' column, there are no 'NA's and all values are less
>  than 3.
>
>  **Hint**: think about how the commands should be ordered to produce this data frame!
>
> ```{r, answer=TRUE, eval=FALSE, purl=FALSE}
> surveys_hindfoot_cm <- surveys %>%
>     filter(!is.na(hindfoot_length)) %>%
>     mutate(hindfoot_cm = hindfoot_length / 10) %>%
>     filter(hindfoot_cm < 3) %>%
>     select(species_id, hindfoot_cm)
> ```

```{r, eval=FALSE, purl=TRUE, echo=FALSE}
## Mutate 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_cm' containing the 'hindfoot_length' values converted to centimeters.
##  In this 'hindfoot_cm' column, there are no 'NA's and all values are less
##  than 3.

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

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

Many data analysis tasks can be approached using the *split-apply-combine*
paradigm: split the data into groups, apply some analysis to each group, and
then combine the results. Key functions of **'dplyr'** for this workflow are
'group_by()' and 'summarize()'.

```
#### The `group_by()` and `summarize()` functions

`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:

```{r, purl = FALSE}
surveys %>%
  group_by(sex) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE))
```

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

You can also group by multiple columns:

```{r, purl = FALSE}
surveys %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE)) %>%
  tail()
```

Here, we used `tail()` to look at the last six rows of our summary. Before, we had
used `head()` to look at the first six rows. We can see that the `sex` column contains
`NA` values because some animals had escaped before their sex and body weights
could be determined. The resulting `mean_weight` column does not contain `NA` but
`NaN` (which refers to "Not a Number") because `mean()` was called on a vector of
`NA` values while at the same time setting `na.rm = TRUE`. 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:

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

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

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

```
  print(n = 15)
```

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

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

It is sometimes useful to rearrange the result of a query to inspect the values. For instance, we can so

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

To sort in descending order, we need to add the `desc()` function. If we want to sort the results by dec

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

#### *Counting*

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

```{r, purl = FALSE}
surveys %>%
    count(sex)
```

The `count()` function is shorthand for something we've already seen: grouping by a variable, and summar

```{r, purl = FALSE}
surveys %>%
    group_by(sex) %>%
    summarise(count = n())
```

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

```{r, purl = FALSE}
surveys %>%
    count(sex, sort = TRUE)
```

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

```{r purl = FALSE}
surveys %>%
  count(sex, species)
```

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

```{r purl = FALSE}
surveys %>%
  count(sex, species) %>%
  arrange(species, desc(n))
```

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

> ### Challenge {.challenge}
>
> 1. How many animals were caught in each `plot_type` surveyed?
>
> ```{r, answer=TRUE, purl=FALSE}
> surveys %>%
>     count(plot_type)
> ```
>
> 2. Use `group_by()` and `summarize()` to find the mean, min, and max hindfoot
> length for each species (using `species_id`). Also add the number of
> observations (hint: see `?n`).
>
> ```{r, answer=TRUE, purl=FALSE}
> surveys %>%
>     filter(!is.na(hindfoot_length)) %>%

```
>     group_by(species_id) %>%
>     summarize(
>         mean_hindfoot_length = mean(hindfoot_length),
>         min_hindfoot_length = min(hindfoot_length),
>         max_hindfoot_length = max(hindfoot_length),
>         n = n()
>     )
> ```
>
> 3. What was the heaviest animal measured in each year? Return the columns `year`,
> `genus`, `species_id`, and `weight`.
>
> ```{r, answer=TRUE, purl=FALSE}
> surveys %>%
>     filter(!is.na(weight)) %>%
>     group_by(year) %>%
>     filter(weight == max(weight)) %>%
>     select(year, genus, species, weight) %>%
>     arrange(year)
> ```


```{r, eval=FALSE, purl=TRUE, echo=FALSE}
## Count Challenges:
##  1. How many animals were caught in each `plot_type` surveyed?

##  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`).

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

### Reshaping with pivot_longer and pivot_wider

In the [spreadsheet
lesson](https://datacarpentry.org/spreadsheet-ecology-lesson/01-format-data/),
we discussed how to structure our data leading to the four rules defining a tidy
dataset:

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

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

In `surveys`, the rows of `surveys` contain the values of variables associated
with each record (the unit), values such as 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 genus between plots? (Ignoring `plot_type` for simpl
```

We'd need to create a new table where each row (the unit) is comprised of values of variables associated
in `genus` would become the names of column variables and the cells would contain the values of the mean

Having created a new table, it is therefore straightforward to explore the
relationship between the weight of different genera 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 genus 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()`.

These may sound like dramatically different data layouts, but there are some tools that make transitions



#### *Pivoting from long to wide format*

`pivot_wider()` takes three principal arguments:

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

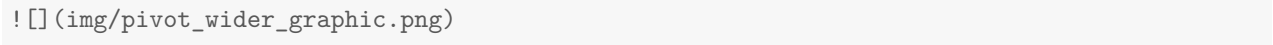
```{r, purl=FALSE}
surveys_gw <- surveys %>%
  filter(!is.na(weight)) %>%
  group_by(plot_id, genus) %>%
  summarize(mean_weight = mean(weight))

str(surveys_gw)
```

This yields `surveys_gw` where the observations for each plot are distributed across
multiple rows, 196 observations of 3 variables.
Using `pivot_wider()` with the names from `genus` and with values from `mean_weight` this becomes
24 observations of 11 variables, one row for each plot.

```{r, purl=FALSE}
surveys_wide <- surveys_gw %>%
  pivot_wider(names_from = genus, values_from = mean_weight)
```

```
str(surveys_wide)
```



We could now plot comparisons between the weight of genera (one is called a genus, multiple are called g
although we may wish to fill in the missing values first.

```{r, purl=FALSE}
surveys_gw %>%
  pivot_wider(names_from = genus, values_from = mean_weight, values_fill = 0) %>%
  head()
```

#### *Pivoting from wide to long format*

The opposing situation could occur if we had been provided with data in the
form of `surveys_wide`, 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 reshaping 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:

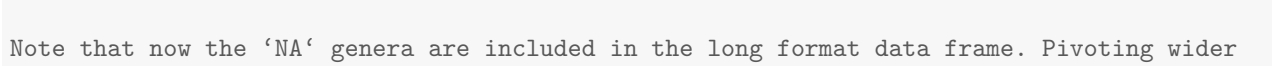
1. the data
2. the *names_to* column variable we wish to create from column names.
3. the *values_to* column variable we wish to create and fill with values.
4. *cols* are the name of the columns we use to make this pivot (or to drop).

To recreate `surveys_gw` from `surveys_wide` we would create a names variable called
`genus` and value variable called `mean_weight`.

In pivoting longer, we also need to specify what columns to reshape. If the columns are directly adjacer

```{r, purl=FALSE}
surveys_long <- surveys_wide %>%
  pivot_longer(names_to = "genus", values_to = "mean_weight", cols = -plot_id)

str(surveys_long)
```



Note that now the `NA` genera are included in the long format data frame. Pivoting wider
and then longer can be a useful way to balance out a dataset so that every
replicate has the same composition

We could also have used a specification for what columns to exclude. In this example,
we will use all columns _except_ `plot_id` for the names variable. By using the minus sign in the `cols`
we omit `plot_id` from being reshaped

```{r, purl=FALSE}
surveys_wide %>%
  pivot_longer(names_to = "genus", values_to = "mean_weight", cols = -plot_id) %>%
  head()
```

> ### Challenge {.challenge}
>
> 1. Reshape 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.
>
> ```{r, answer=TRUE, purl=FALSE}
> surveys_wide_genera <- surveys %>%
>   group_by(plot_id, year) %>%
>   summarize(n_genera = n_distinct(genus)) %>%
>   pivot_wider(names_from = year, values_from = n_genera)
>
> head(surveys_wide_genera)
> ```
>
> 2. Now take that data frame and `pivot_longer()` it, so each row is a unique
>    `plot_id` by `year` combination.
>
> ```{r, answer=TRUE, purl=FALSE}
> surveys_wide_genera %>%
>   pivot_longer(names_to = "year", values_to = "n_genera", cols = -plot_id)
> ```
>
> 3. The `surveys` data set has
>    two measurement columns: `hindfoot_length` and `weight`. This makes it
>    difficult to do things like look at the relationship between mean values of
>    each measurement per year in different plot types. Let's walk through a
>    common solution for this type of problem. First, use `pivot_longer()` to create a
>     dataset where we have a names 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 will be part of the reshape.

>
> ```{r, answer=TRUE, purl=FALSE}
> 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 names and values
>    columns for `pivot_wider()`.
>

```
> ```{r, answer=TRUE, purl=FALSE}
> 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)
> ```

```{r, eval=FALSE, purl=TRUE, echo=FALSE}
## Reshaping challenges

## 1. Reshape the `surveys` data frame with `year` as columns, `plot_id` as rows, and the number of gene

## 2. Now take that data frame and `pivot_longer()` it, so each row is a unique `plot_id` by `year` comb

## 3. The `surveys` data set has two measurement columns: `hindfoot_length` and `weight`. This makes it

## 4. With this new data set, calculate the average of each `measurement` in each `year` for each diffe
```

# Exporting data

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

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

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

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

Let's start by removing observations of animals for which `weight` and `hindfoot_length` are missing, or

```{r, purl=FALSE}
surveys_complete <- surveys %>%
  filter(!is.na(weight),           # remove missing weight
         !is.na(hindfoot_length),  # remove missing hindfoot_length
         !is.na(sex))                # remove missing sex
```

Because we are interested in plotting how species abundances have changed
through time, we are also going to remove observations for rare species (i.e.,
that have been observed less than 50 times). We will do this in two steps: first
```

we are going to create a data set that counts how often each species has been observed, and filter out the rare species; then, we will extract only the observations for these more common species:

```{r, purl=FALSE}
## Extract the most common species_id
species_counts <- surveys_complete %>%
    count(species_id) %>%
    filter(n >= 50)

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

```{r, eval=FALSE, purl=TRUE, echo=FALSE}
### Create the dataset for exporting:
##  Start by removing observations for which the `species_id`, `weight`,
##  `hindfoot_length`, or `sex` data are missing:
surveys_complete <- surveys %>%
    filter(species_id != "",        # remove missing species_id
           !is.na(weight),                 # remove missing weight
           !is.na(hindfoot_length),        # remove missing hindfoot_length
           sex != "")                      # remove missing sex

##  Now remove rare species in two steps. First, make a list of species which
##  appear at least 50 times in our dataset:
species_counts <- surveys_complete %>%
    count(species_id) %>%
    filter(n >= 50) %>%
    select(species_id)

##  Second, keep only those species:
surveys_complete <- surveys_complete %>%
    filter(species_id %in% species_counts$species_id)
```

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

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

```{r, purl=FALSE, eval=FALSE}
write_csv(surveys_complete, file = "data/surveys_complete.csv")
```

```{r, purl=FALSE, eval=TRUE, echo=FALSE}
if (!dir.exists("data")) dir.create("data")
write_csv(surveys_complete, file = "data/surveys_complete.csv")
```

```
```{r, child="_page_built_on.Rmd"}
```

```
## Error:  <text>:4:9:  unexpected symbol
## 3:
## 4:  Bracket subsetting
##             ^
```

The R session information (including the OS info, R version and all packages used):

```
sessionInfo()
```

```
## R version 4.2.1 (2022-06-23 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19045)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_New Zealand.utf8  LC_CTYPE=English_New Zealand.utf8
## [3] LC_MONETARY=English_New Zealand.utf8 LC_NUMERIC=C
## [5] LC_TIME=English_New Zealand.utf8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] ggrepel_0.9.2      ggthemes_4.2.4     sf_1.0-9           leaflet_2.1.1
##  [5] fpp_0.5            tseries_0.10-53    lmtest_0.9-40      zoo_1.8-11
##  [9] expsmooth_2.3      fma_2.4            forecast_8.20      seasonal_1.9.0
## [13] lubridate_1.9.1    timetk_2.8.2       yardstick_1.1.0    workflowsets_1.0.0
## [17] workflows_1.1.2    tune_1.0.1         rsample_1.1.1      recipes_1.0.4
## [21] parsnip_1.0.3      modeldata_1.1.0    infer_1.0.4        dials_1.1.0
## [25] scales_1.2.1       broom_1.0.3        tidymodels_1.0.0   modeltime_1.2.4
## [29] flextable_0.8.5    XKCDdata_0.1.0     forcats_1.0.0      stringr_1.5.0
## [33] dplyr_1.1.0        purrr_1.0.1        readr_2.1.3        tidyr_1.3.0
## [37] tibble_3.1.8       ggplot2_3.4.0      tidyverse_1.3.2    knitr_1.42
##
## loaded via a namespace (and not attached):
##   [1] utf8_1.2.3           tidyselect_1.2.0     htmlwidgets_1.6.1    grid_4.2.1
##   [5] munsell_0.5.0        units_0.8-1          codetools_0.2-18     xgboost_1.7.3.1
##   [9] future_1.31.0        withr_2.5.0          colorspace_2.1-0     highr_0.10
##  [13] uuid_1.1-0           rstudioapi_0.14      stats4_4.2.1         wk_0.7.1
##  [17] officer_0.5.2        TTR_0.24.3           listenv_0.9.0        labeling_0.4.2
##  [21] rstan_2.21.8         DiceDesign_1.9       farver_2.1.1         parallelly_1.34.0
##  [25] vctrs_0.5.2          generics_0.1.3       ipred_0.9-13         xfun_0.37
##  [29] timechange_0.2.0     R6_2.5.1             lhs_1.1.6            cachem_1.0.6
##  [33] assertthat_0.2.1     promises_1.2.0.1     nnet_7.3-17          googlesheets4_1.0.1
##  [37] gtable_0.3.1         globals_0.16.2       processx_3.8.0       timeDate_4022.108
##  [41] rlang_1.0.6          systemfonts_1.0.4    splines_4.2.1        lazyeval_0.2.2
##  [45] gargle_1.3.0         inline_0.3.19        s2_1.1.2             yaml_2.3.7
##  [49] modelr_0.1.10        crosstalk_1.2.0      backports_1.4.1      httpuv_1.6.8
##  [53] quantmod_0.4.20      tools_4.2.1          lava_1.7.1           ellipsis_0.3.2
##  [57] proxy_0.4-27         Rcpp_1.0.10          base64enc_0.1-3      classInt_0.4-8
```
```

```
##  [61] ps_1.7.2             prettyunits_1.1.1    rpart_4.1.16         openssl_2.0.5
##  [65] fracdiff_1.5-2       haven_2.5.1          fs_1.6.0             tinytex_0.44
##  [69] furrr_0.3.1          crul_1.3             magrittr_2.0.3       data.table_1.14.6
##  [73] reprex_2.0.2         GPfit_1.0-8          googledrive_2.0.0    x13binary_1.1.57-3
##  [77] matrixStats_0.63.0   hms_1.1.2            mime_0.12            evaluate_0.20
##  [81] xtable_1.8-4         readxl_1.4.1         gridExtra_2.3        compiler_4.2.1
##  [85] KernSmooth_2.23-20   crayon_1.5.2         StanHeaders_2.21.0-7 htmltools_0.5.4
##  [89] later_1.3.0          tzdb_0.3.0           RcppParallel_5.1.6   DBI_1.1.3
##  [93] dbplyr_2.3.0         MASS_7.3-57          Matrix_1.5-3         cli_3.6.0
##  [97] quadprog_1.5-8       parallel_4.2.1       gower_1.0.1          pkgconfig_2.0.3
## [101] plotly_4.10.1        xml2_1.3.3           foreach_1.5.2        hardhat_1.2.0
## [105] prodlim_2019.11.13   rvest_1.0.3          snakecase_0.11.0     callr_3.7.3
## [109] digest_0.6.31        janitor_2.2.0        httpcode_0.3.0       rmarkdown_2.20
## [113] cellranger_1.1.0     gdtools_0.3.0        curl_5.0.0           shiny_1.7.4
## [117] urca_1.3-3           lifecycle_1.0.3      nlme_3.1-157         jsonlite_1.8.4
## [121] viridisLite_0.4.1    askpass_1.1          fansi_1.0.4          pillar_1.8.1
## [125] lattice_0.20-45      loo_2.5.1            fastmap_1.1.0        httr_1.4.4
## [129] pkgbuild_1.4.0       survival_3.3-1       glue_1.6.2           xts_0.12.2
## [133] zip_2.2.2            iterators_1.0.14     class_7.3-20         stringi_1.7.12
## [137] prophet_1.0          gfonts_0.2.0         memoise_2.0.1        e1071_1.7-13
## [141] future.apply_1.10.0
```

```
Sys.time()
```

```
## [1] "2023-02-03 13:25:43 NZDT"
```