Working with data - Answers

This lesson is adapted from the Data Analysis and visualisation in R for Ecologists 4 lesson. (https://datacarpentry.github.io/R-ecology-lesson/working-with-data.html)

Questions

- How do you manipulate tabular data in R?

Objectives

- Import CSV data into R.
- Understand the difference between base R and `tidyverse` approaches.
- Subset rows and columns of data.frames.
- Use pipes to link steps together into pipelines.
- Create new data.frame columns using existing columns.
- Utilize the concept of split-apply-combine data analysis.
- Reshape data between wide and long formats.
- Export data to a CSV file.

This is an R Markdown (http://rmarkdown.rstudio.com) Notebook. When you execute code within the notebook, the results appear beneath the code.

You can execute code chunks by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing `Ctrl + Shift + Enter` or OS X: `Cmd + Shift + Enter`. To execute a specific line within a code chunk, place your cursor on that line and press `Ctrl + Enter` or OS X: `Cmd + Enter`.

```
# try running this code chunk

3 + 2
```

```
## [1] 5
```

```
3 * 2
```

```
## [1] 6
```

AYou can add a new chunk of code in your language of choice by clicking the *Insert Chunk* button on the toolbar or by pressing `Ctrl + Alt + I` or OS X: `Cmd + Option + I`.

Tip - View source or visual mode

The R Markdown notebook (e.g. *'working_with_data_answers.Rmd'*) can be viewed or edited in either in *'Source'* mode which shows the raw Markdown commands, or in the *'Visual'* mode which shows how the Markdown will look in its formatted form. You can switch between the *'Source'* and *'Visual'* modes at any time using the tabs at the top of the notebook.

Tip - Preview

When you save the notebook, an HTML file containing the code and *output* will be saved alongside it. To preview the HTML file in the 'Viewer' tab click the *Preview* button or press `Ctrl + Shift + K` or OS X: `Cmd + Shift + K`. The preview shows you a rendered HTML copy of the current contents of the editor. Consequently, *Preview* does not run any R code chunks. Instead, the output of the chunk from when it was last run in the editor is displayed. There are other more sophisticated ways to display your notebook, such as using the *Knit* command.

1. Setup

We start by loading the **`tidyverse`** package that you installed during the setup.

If you do not have the **`tidyverse`** installed, you can run `install.packages("tidyverse")` in the console.

It is a good practice not to put `install.packages()` into a script. This is because every time you run that whole script, the package will be reinstalled, which is typically unnecessary. You want to install the package to your computer once, and then load it with `library()` in each script where you need to use it.

```
# load tidyverse

library(tidyverse)
```

1.1 Importing data

In the previous lessons, we have been working with the `complete_old` dataframe contained in the `ratdat` package. However, you typically won't access data from an R package; it is much more common to access data files stored somewhere on your computer. We are going to download a different dataset (the surveys data) from a CSV file stored on the internet to our computer, which we will then read into R.

Click this link to download the file: https://datacarpentry.org/R-ecology-lesson/data/cleaned/surveys_complete_77_89.csv (https://datacarpentry.org/R-ecology-lesson/data/cleaned/surveys_complete_77_89.csv).

You will be prompted to save the file on your computer somewhere. Save it inside the `raw` data folder, which is in the `data` folder in your `CIDS_Carpentries_R_materials` folder. Once it's inside our project, we will be able to point R towards it.

Tip: `download.file()` function

Alternately, you could use the R function `download.file()` to download the CSV file. Inside the download.file command, the first entry is a character string with the source URL ("https://datacarpentry.org/R-ecology-lesson/data/raw/surveys_complete_77_89.csv (https://datacarpentry.org/R-ecology-lesson/data/raw/surveys_complete_77_89.csv)"). This source URL downloads the CSV file from the Data Carpentries website. The text after the comma ("data/raw/surveys_complete_77_89.csv") is the destination of the file on your local machine. You'll need to have a folder in your `CIDS_Carpentries_R_materials` folder called "data/raw", where you'll download the file. So this command downloads a file from the Data Carpentries website, names it "surveys_complete_77_89.csv" and adds it to a preexisting folder named "data/raw".

```
# download file

download.file(url = "https://datacarpentry.org/R-ecology-lesson/data/cleaned/surveys_
complete_77_89.csv",
              destfile = "data/raw/surveys_complete_77_89.csv")
```

1.2 File paths

When we reference other files from an R script, we need to give R precise instructions on where those files are. We do that using something called a **file path**. It looks something like this: `"Documents/Manuscripts/Chapter_2.txt"`. This path would tell your computer how to get from whatever folder contains the `Documents` folder all the way to the `.txt` file.

There are two kinds of paths: **absolute** and **relative**. Absolute paths are specific to a particular computer, whereas relative paths are relative to a certain folder. Because we are keeping all of our work in the `R-Ecology-Workshop` folder, all of our paths can be relative to this folder.

Now, let's read our CSV file into R and store it in an object named `surveys`. We will use the `read_csv` function from the `tidyverse`'s `readr` package, and the argument we give will be the **relative path** to the CSV file.

```
# read in the csv

surveys <- read_csv("data/raw/surveys_complete_77_89.csv")
```

```
## Rows: 16878 Columns: 13
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ","
## chr (6): species_id, sex, genus, species, taxa, plot_type
## dbl (7): record_id, month, day, year, plot_id, hindfoot_length, weight
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Tip: keyboard shortcuts

Typing out paths can be error prone, but we can utilise a keyboard shortcut. Inside the parentheses of `read_csv()`, type out a pair of quotes and put your cursor between them. Then hit `Tab`. A small menu showing your folders and files should show up. You can use the `↑` and `↓` keys to move through the options, or start typing to narrow them down. You can hit `Enter` to select a file or folder, and hit `Tab` again to continue building the file path. This might take a bit of getting used to, but once you get the hang of it, it will speed up writing file paths and reduce the number of mistakes you make.

You may have noticed a bit of feedback from R when you ran the last line of code. We got some useful information about the CSV file we read in. We can see:

- the number of rows and columns
- the **delimiter** of the file, which is how values are separated, ie a comma `","`
- a set of columns that were **parsed** as various vector types
    - the file has 6 character columns and 7 numeric columns
    - we can see the names of the columns for each type

2. Manipulating data

2.1 The `survey` data schema

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

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

When working with the output of a new function, it’s often a good idea to check the `class()`:

```
# check the class

class(surveys)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

Whoa! What is this thing? It has multiple classes? Well, it’s called a `tibble`, and it is the `tidyverse` version of a data.frame. It *is* a data.frame, but with some added perks. It prints out a little more nicely, it highlights `NA` values and negative values in red, and it will generally communicate with you more (in terms of warnings and errors, which is a good thing).

Tip: `tidyverse` vs base R

As we begin to delve more deeply into the `tidyverse`, we should briefly pause to mention some of the reasons for focusing on the `tidyverse` set of tools. In R, there are often many ways to get a job done, and there are other approaches that can accomplish tasks similar to the `tidyverse`.

The phrase **base R** is used to refer to approaches that utilize functions contained in R’s default packages. We have already used some base R functions, such as `str()`, and `head()`, and we will be using more scattered throughout this lesson. However, there are some key base R approaches we will not be teaching. These include square bracket subsetting and base plotting. You may come across code written by other people that looks like `surveys[1:10, 2]` or `plot(surveys$weight, surveys$hindfoot_length)`, which are base R commands. If you’re interested in learning more about these approaches, you can check out other Carpentries lessons like the Software Carpentry Programming with R (https://swcarpentry.github.io/r-novice-inflammation/) lesson.

We choose to teach the `tidyverse` set of packages because they share a similar syntax and philosophy, making them consistent and producing highly readable code. They are also very flexible and powerful, with a growing number of packages designed according to similar principles and to work well with the rest of the `tidyverse` packages. The `tidyverse` packages tend to have very clear documentation and wide array of learning materials that tend to be written with novice users in mind. Finally, the `tidyverse` has only continued to grow, and has strong support from RStudio, which implies that these approaches will be relevant into the future.
One of the most important skills for working with data in R is the ability to manipulate, modify, and reshape data. The `dplyr` and `tidyr` packages in the `tidyverse` provide a series of powerful functions for many common data manipulation tasks.

We're now 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 `summarise()` : create summary statistics on grouped data
- `arrange()` : sort results

We'll start off with two of the most commonly used `dplyr` functions: `select()` , which selects certain columns of a data.frame, and `filter()` , which filters out rows according to certain criteria.

Tip: `select()` vs `filter()`

Between `select()` and `filter()` , it can be hard to remember which operates on columns and which operates on rows. `select()` has a **c** for **c**olumns and `filter()` has an **r** for **r**ows.

2.2 The `select()` function

To use the `select()` function, the first argument is the name of the data.frame, and the rest of the arguments are *unquoted* names of the columns you want:

```
# select the columns plot_id, species_id, hindfoot_length

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

plot_id	species_id	hindfoot_length
<dbl>	<chr>	<dbl>
2	NL	32
3	NL	33
2	DM	37
7	DM	36
3	DM	35
1	PF	14
2	PE	*NA*
1	DM	37
1	DM	34
6	PF	20

1-10 of 10,000 rows Previous **1** 2 3 4 5 6 … 1000 Next

The columns that are output are arranged in the order we specified inside `select()`.

To select all columns except specific columns, put a `-` in front of the column you want to exclude:

```
# exclude the columns -record_id, -year

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

m… <dbl>	d.. <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei… <dbl>	genus <chr>	species <chr>	tax <ch
7	16	2	NL	M	32	*NA*	Neotoma	albigula	Ro
7	16	3	NL	M	33	*NA*	Neotoma	albigula	Ro
7	16	2	DM	F	37	*NA*	Dipodomys	merriami	Ro
7	16	7	DM	M	36	*NA*	Dipodomys	merriami	Ro
7	16	3	DM	M	35	*NA*	Dipodomys	merriami	Ro
7	16	1	PF	M	14	*NA*	Perognathus	flavus	Ro
7	16	2	PE	F	*NA*	*NA*	Peromyscus	eremicus	Ro
7	16	1	DM	M	37	*NA*	Dipodomys	merriami	Ro
7	16	1	DM	F	34	*NA*	Dipodomys	merriami	Ro
7	16	6	PF	F	20	*NA*	Perognathus	flavus	Ro

1-10 of 10,000 rows | 1-10 of 11 columns Previous **1** 2 3 4 5 6 … 1000 Next

`select()` also works with numeric vectors for the order of the columns. To select the 3rd, 4th, 5th, and 10th columns, we could run the following code:

```
# select columns by position

select(surveys, c(3:5, 10))
```

day <dbl>	year <dbl>	plot_id <dbl>	genus <chr>
16	1977	2	Neotoma
16	1977	3	Neotoma
16	1977	2	Dipodomys
16	1977	7	Dipodomys
16	1977	3	Dipodomys
16	1977	1	Perognathus
16	1977	2	Peromyscus
16	1977	1	Dipodomys

day	year	plot_id	genus
<dbl>	<dbl>	<dbl>	<chr>
16	1977	1	Dipodomys
16	1977	6	Perognathus

1-10 of 10,000 rows Previous **1** 2 3 4 5 6 … 1000 Next

You should be careful when using this method, since you are being less explicit about which columns you want. However, it can be useful if you have a data.frame with many columns and you don't want to type out too many names.

Finally, you can select columns based on whether they match a certain criteria by using the `where()` function. If we want all numeric columns, we can ask to `select` all the columns `where` the class `is numeric`:

```
# select all numeric columns

select(surveys, where(is.numeric))
```

record_id	month	day	year	plot_id	hindfoot_length	weight
<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	7	16	1977	2	32	*NA*
2	7	16	1977	3	33	*NA*
3	7	16	1977	2	37	*NA*
4	7	16	1977	7	36	*NA*
5	7	16	1977	3	35	*NA*
6	7	16	1977	1	14	*NA*
7	7	16	1977	2	*NA*	*NA*
8	7	16	1977	1	37	*NA*
9	7	16	1977	1	34	*NA*
10	7	16	1977	6	20	*NA*

1-10 of 10,000 rows Previous **1** 2 3 4 5 6 … 1000 Next

Instead of giving names or positions of columns, we instead pass the `where()` function with the name of another function inside it, in this case `is.numeric()`, and we get all the columns for which that function returns `TRUE`.

We can use this to select any columns that have any `NA` values in them:

```
# select columns containing 'NA' values

select(surveys, where(anyNA))
```

species_id <chr>	s… <chr>	hindfoot_length <dbl>	weight <dbl>	genus <chr>	species <chr>	taxa <chr>
NL	M	32	*NA*	Neotoma	albigula	Rodent
NL	M	33	*NA*	Neotoma	albigula	Rodent
DM	F	37	*NA*	Dipodomys	merriami	Rodent
DM	M	36	*NA*	Dipodomys	merriami	Rodent
DM	M	35	*NA*	Dipodomys	merriami	Rodent
PF	M	14	*NA*	Perognathus	flavus	Rodent
PE	F	*NA*	*NA*	Peromyscus	eremicus	Rodent
DM	M	37	*NA*	Dipodomys	merriami	Rodent
DM	F	34	*NA*	Dipodomys	merriami	Rodent
PF	F	20	*NA*	Perognathus	flavus	Rodent

1-10 of 10,000 rows Previous **1** 2 3 4 5 6 … 1000 Next

2.3 The `filter()` function

The `filter()` function is used to select rows that meet certain criteria. To get all the rows where the value of `year` is equal to 1985, we would run the following:

```
# filter rows with year = 1985

filter(surveys, year == 1985)
```

record_id <dbl>	m… <dbl>	d.. <dbl>	y… <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei… <dbl>	genus <chr>
9790	1	19	1985	16	RM	F	16	4	Reithrodontomys
9791	1	19	1985	17	OT	F	20	16	Onychomys
9792	1	19	1985	6	DO	M	35	48	Dipodomys
9793	1	19	1985	12	DO	F	35	40	Dipodomys
9794	1	19	1985	24	RM	M	16	4	Reithrodontomys
9795	1	19	1985	12	DO	M	34	48	Dipodomys
9796	1	19	1985	6	DM	F	37	35	Dipodomys
9797	1	19	1985	14	DM	M	36	45	Dipodomys
9798	1	19	1985	6	DM	F	36	38	Dipodomys
9799	1	19	1985	19	RM	M	16	4	Reithrodontomys

1-10 of 1,438 rows | 1-10 of 13 columns Previous **1** 2 3 4 5 6 … 144 Next

The `==` sign means "is equal to". There are several other operators we can use: >, >=, <, <=, and != (not equal to). Another useful operator is `%in%`, which asks if the value on the lefthand side is found anywhere in the vector on the righthand side. For example, to get rows with specific `species_id` values, we could run:

```
# use the `%in%` operator

filter(surveys, species_id %in% c("RM", "DO"))
```

record_id <dbl>	m... <dbl>	d.. <dbl>	y... <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei... <dbl>	genus <chr>
68	8	19	1977	8	DO	F	32	52	Dipodomys
292	10	17	1977	3	DO	F	36	33	Dipodomys
294	10	17	1977	3	DO	F	37	50	Dipodomys
311	10	17	1977	19	RM	M	18	13	Reithrodontomys
317	10	17	1977	17	DO	F	32	48	Dipodomys
323	10	17	1977	17	DO	F	33	31	Dipodomys
337	10	18	1977	8	DO	F	35	41	Dipodomys
356	11	12	1977	1	DO	F	32	44	Dipodomys
378	11	12	1977	1	DO	M	33	48	Dipodomys
397	11	13	1977	17	RM	F	16	7	Reithrodontomys

1-10 of 2,835 rows | 1-10 of 13 columns Previous **1** 2 3 4 5 6 … 284 Next

We can also use multiple conditions in one `filter()` statement. Here we will get rows with a year less than or equal to 1988 and whose hindfoot length values are not `NA`. The `!` before the `is.na()` function means "not".

```
# filter on multiple conditions

filter(surveys, year <= 1988 & !is.na(hindfoot_length))
```

record_id <dbl>	m... <dbl>	d.. <dbl>	y... <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei... <dbl>	genus <chr>
1	7	16	1977	2	NL	M	32	*NA*	Neotoma
2	7	16	1977	3	NL	M	33	*NA*	Neotoma
3	7	16	1977	2	DM	F	37	*NA*	Dipodomys
4	7	16	1977	7	DM	M	36	*NA*	Dipodomys
5	7	16	1977	3	DM	M	35	*NA*	Dipodomys
6	7	16	1977	1	PF	M	14	*NA*	Perognathus
8	7	16	1977	1	DM	M	37	*NA*	Dipodomys
9	7	16	1977	1	DM	F	34	*NA*	Dipodomys
10	7	16	1977	6	PF	F	20	*NA*	Perognathus

record_id <dbl>	m... <dbl>	d.. <dbl>	y... <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei... <dbl>	genus <chr>
11	7	16	1977	5	DS	F	53	*NA*	Dipodomys

1-10 of 10,000 rows | 1-10 of 13 columns Previous **1** 2 3 4 5 6 … 1000 Next

Challenge 1.a: Filtering and selecting

1. Use the surveys data to make a data.frame that has only data with years from 1980 to 1985.

```
surveys_filtered <- filter(surveys, year >= 1980 & year <= 1985)
```

Challenge 1.b: Use the surveys data to make a data.frame that has only the following columns, in order: `year`, `month`, `species_id`, `plot_id`.

```
surveys_selected <- select(surveys, year, month, species_id, plot_id)
surveys_selected
```

year <dbl>	month <dbl>	species_id <chr>	plot_id <dbl>
1977	7	NL	2
1977	7	NL	3
1977	7	DM	2
1977	7	DM	7
1977	7	DM	3
1977	7	PF	1
1977	7	PE	2
1977	7	DM	1
1977	7	DM	1
1977	7	PF	6

1-10 of 10,000 rows Previous **1** 2 3 4 5 6 … 1000 Next

2.4 The pipe `%>%` operator

What happens if we want to both `select()` and `filter()` our data? We have a couple options. First, we could use **nested** functions:

```
# select and filter

filter(select(surveys, -day), month >= 7)
```

record_id	m...	y...	plot_id	species_id	s..	hindfoot_length	wei...	genus	speci
<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>	<dbl>	<dbl>	<chr>	<chr>
1	7	1977	2	NL	M	32	*NA*	Neotoma	albigu
2	7	1977	3	NL	M	33	*NA*	Neotoma	albigu
3	7	1977	2	DM	F	37	*NA*	Dipodomys	merria
4	7	1977	7	DM	M	36	*NA*	Dipodomys	merria
5	7	1977	3	DM	M	35	*NA*	Dipodomys	merria
6	7	1977	1	PF	M	14	*NA*	Perognathus	flavus
7	7	1977	2	PE	F	*NA*	*NA*	Peromyscus	eremi
8	7	1977	1	DM	M	37	*NA*	Dipodomys	merria
9	7	1977	1	DM	F	34	*NA*	Dipodomys	merria
10	7	1977	6	PF	F	20	*NA*	Perognathus	flavus

1-10 of 8,244 rows | 1-10 of 12 columns Previous **1** 2 3 4 5 6 … 825 Next

R will evaluate statements from the inside out. First, `select()` will operate on the `surveys` data.frame, removing the column `day`. The resulting data.frame is then used as the first argument for `filter()`, which selects rows with a month greater than or equal to 7.

Nested functions can be very difficult to read with only a few functions, and nearly impossible when many functions are done at once. An alternative approach is to create **intermediate** objects:

```
# select then filter

surveys_noday <- select(surveys, -day)
filter(surveys_noday, month >= 7)
```

record_id	m...	y...	plot_id	species_id	s..	hindfoot_length	wei...	genus	speci
<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>	<dbl>	<dbl>	<chr>	<chr>
1	7	1977	2	NL	M	32	*NA*	Neotoma	albigu
2	7	1977	3	NL	M	33	*NA*	Neotoma	albigu
3	7	1977	2	DM	F	37	*NA*	Dipodomys	merria
4	7	1977	7	DM	M	36	*NA*	Dipodomys	merria
5	7	1977	3	DM	M	35	*NA*	Dipodomys	merria
6	7	1977	1	PF	M	14	*NA*	Perognathus	flavus
7	7	1977	2	PE	F	*NA*	*NA*	Peromyscus	eremi
8	7	1977	1	DM	M	37	*NA*	Dipodomys	merria
9	7	1977	1	DM	F	34	*NA*	Dipodomys	merria
10	7	1977	6	PF	F	20	*NA*	Perognathus	flavus

1-10 of 8,244 rows | 1-10 of 12 columns Previous **1** 2 3 4 5 6 … 825 Next

This approach is easier to read, since we can see the steps in order, but after enough steps, we are left with a cluttered mess of intermediate objects, often with confusing names. For large data files this also can use up a lot of computer storage).

An elegant solution to this problem is an operator called the **pipe**, which looks like `%>%`. You can insert it by using the keyboard shortcut `Shift+Ctrl+M` or OSX: `Shift+Cmd+M`. Here's how you could use a pipe to select and filter in one step:

```
# use pipes instead

surveys %>%
  select(-day) %>%
  filter(month >= 7)
```

record_id <dbl>	m… <dbl>	y… <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei… <dbl>	genus <chr>	speci <chr>
1	7	1977	2	NL	M	32	*NA*	Neotoma	albigu
2	7	1977	3	NL	M	33	*NA*	Neotoma	albigu
3	7	1977	2	DM	F	37	*NA*	Dipodomys	merria
4	7	1977	7	DM	M	36	*NA*	Dipodomys	merria
5	7	1977	3	DM	M	35	*NA*	Dipodomys	merria
6	7	1977	1	PF	M	14	*NA*	Perognathus	flavus
7	7	1977	2	PE	F	*NA*	*NA*	Peromyscus	eremi
8	7	1977	1	DM	M	37	*NA*	Dipodomys	merria
9	7	1977	1	DM	F	34	*NA*	Dipodomys	merria
10	7	1977	6	PF	F	20	*NA*	Perognathus	flavus

1-10 of 8,244 rows | 1-10 of 12 columns Previous **1** 2 3 4 5 6 … 825 Next

What it does is take the thing on the lefthand side and insert it as the first argument of the function on the righthand side. By putting each of our functions onto a new line, we can build a nice, readable **pipeline**. It can be useful to think of this as a little assembly line for our data. It starts at the top and gets piped into a `select()` function, and it comes out modified somewhat. It then gets sent into the `filter()` function, where it is further modified, and then the final product gets printed out to our console. It can also be helpful to think of `%>%` as meaning "and then". Since many `tidyverse` functions have verbs for names, a pipeline can be read like a sentence.

Tip: running a pipeline

You can run a **pipeline** without highlighting the whole thing. If your cursor is on any line of a pipeline, running that line will run the whole thing.

If you highlight a section of a pipeline, you can run only the selected steps of it.

If we want to store this final product as an object, we use an assignment arrow at the start:

```
# store an object

surveys_sub <- surveys %>%
  select(-day) %>%
  filter(month >= 7)
```

Tip - Assignment operator

In RStudio, typing `Alt + -` will write `<-` in a single keystroke or OS X: `Option + -`

A good approach is to build a pipeline step by step prior to assignment. You add functions to the pipeline as you go, with the results printing in the console for you to view. Once you're satisfied with your final result, go back and add the assignment arrow statement at the start. This approach is very interactive, allowing you to see the results of each step as you build the pipeline, and produces nicely readable code.

Challenge 2: Using pipes

Use the surveys data to make a data.frame that has the columns `record_id` , `month` , and `species_id` , with data from the year 1988. Use a pipe between the function calls.

```
surveys_1988 <- surveys %>%
  filter(year == 1988) %>%
  select(record_id, month, species_id)
```

In this example, to get the desired output you need to make sure to `filter()` before you `select()` . You need to use the `year` column for filtering rows, but it is discarded in the `select()` step. You also need to make sure to use `==` instead of `=` when you are filtering rows where `year` is equal to 1988.

2.5 The `mutate()` function

Another common task is creating a new column based on values in existing columns. For example, we could add a new column that has the weight in kilograms instead of grams, using the `mutate()` function.

```
# create a new column called weight_kg

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

record_id <dbl>	m… <dbl>	d.. <dbl>	y… <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei… <dbl>	genus <chr>
1	7	16	1977	2	NL	M	32	*NA*	Neotoma
2	7	16	1977	3	NL	M	33	*NA*	Neotoma
3	7	16	1977	2	DM	F	37	*NA*	Dipodomys
4	7	16	1977	7	DM	M	36	*NA*	Dipodomys
5	7	16	1977	3	DM	M	35	*NA*	Dipodomys
6	7	16	1977	1	PF	M	14	*NA*	Perognathus
7	7	16	1977	2	PE	F	*NA*	*NA*	Peromyscus
8	7	16	1977	1	DM	M	37	*NA*	Dipodomys

record_id	m...	d..	y...	plot_id	species_id	s..	hindfoot_length	wei...	genus
<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>	<dbl>	<dbl>	<chr>
9	7	16	1977	1	DM	F	34	*NA*	Dipodomys
10	7	16	1977	6	PF	F	20	*NA*	Perognathus

1-10 of 10,000 rows | 1-10 of 14 columns Previous **1** 2 3 4 5 6 … 1000 Next

You can create multiple columns in one `mutate()` call, and they will get created in the order you write them. This means you can even reference the first new column in the second new column:

```
# create mutliple columns

surveys %>%
  mutate(weight_kg = weight / 1000,
         weight_lbs = weight_kg * 2.2)
```

record_id	m...	d..	y...	plot_id	species_id	s..	hindfoot_length	wei...	genus
<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>	<dbl>	<dbl>	<chr>
1	7	16	1977	2	NL	M	32	*NA*	Neotoma
2	7	16	1977	3	NL	M	33	*NA*	Neotoma
3	7	16	1977	2	DM	F	37	*NA*	Dipodomys
4	7	16	1977	7	DM	M	36	*NA*	Dipodomys
5	7	16	1977	3	DM	M	35	*NA*	Dipodomys
6	7	16	1977	1	PF	M	14	*NA*	Perognathus
7	7	16	1977	2	PE	F	*NA*	*NA*	Peromyscus
8	7	16	1977	1	DM	M	37	*NA*	Dipodomys
9	7	16	1977	1	DM	F	34	*NA*	Dipodomys
10	7	16	1977	6	PF	F	20	*NA*	Perognathus

1-10 of 10,000 rows | 1-10 of 15 columns Previous **1** 2 3 4 5 6 … 1000 Next

We can also use multiple columns to create a single column. For example, it's often good practice to keep the components of a date in separate columns until necessary, as we've done here. This is because programs like Excel can automatically do unexpected things with dates in a way that is not reproducible and sometimes hard to notice. However, now that we are working in R, we can safely put together a date column.

To put together the columns into something that looks like a date, we can use the `paste()` function, which takes arguments of the items to paste together, as well as the argument `sep` , which is the character used to separate the items.

```
# create a single date column

surveys %>%
  mutate(date = paste(year, month, day, sep = "-"))
```

record_id <dbl>	m… <dbl>	d.. <dbl>	y… <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei… <dbl>	genus <chr>
1	7	16	1977	2	NL	M	32	*NA*	Neotoma
2	7	16	1977	3	NL	M	33	*NA*	Neotoma
3	7	16	1977	2	DM	F	37	*NA*	Dipodomys
4	7	16	1977	7	DM	M	36	*NA*	Dipodomys
5	7	16	1977	3	DM	M	35	*NA*	Dipodomys
6	7	16	1977	1	PF	M	14	*NA*	Perognathus
7	7	16	1977	2	PE	F	*NA*	*NA*	Peromyscus
8	7	16	1977	1	DM	M	37	*NA*	Dipodomys
9	7	16	1977	1	DM	F	34	*NA*	Dipodomys
10	7	16	1977	6	PF	F	20	*NA*	Perognathus

1-10 of 10,000 rows | 1-10 of 14 columns Previous **1** 2 3 4 5 6 … 1000 Next

Since our new column gets moved all the way to the end, it doesn't end up printed out on the first page of the output. To move the `year` column we can use the `relocate()` function to put it after our `year` column:

```
# move the date column

surveys %>%
  mutate(date = paste(year, month, day, sep = "-")) %>%
  relocate(date, .after = year)
```

record_id <dbl>	mo… <dbl>	d.. <dbl>	y… <dbl>	date <chr>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	weight <dbl>
1	7	16	1977	1977-7-16	2	NL	M	32	*NA*
2	7	16	1977	1977-7-16	3	NL	M	33	*NA*
3	7	16	1977	1977-7-16	2	DM	F	37	*NA*
4	7	16	1977	1977-7-16	7	DM	M	36	*NA*
5	7	16	1977	1977-7-16	3	DM	M	35	*NA*
6	7	16	1977	1977-7-16	1	PF	M	14	*NA*
7	7	16	1977	1977-7-16	2	PE	F	*NA*	*NA*
8	7	16	1977	1977-7-16	1	DM	M	37	*NA*
9	7	16	1977	1977-7-16	1	DM	F	34	*NA*
10	7	16	1977	1977-7-16	6	PF	F	20	*NA*

1-10 of 10,000 rows | 1-10 of 14 columns Previous **1** 2 3 4 5 6 … 1000 Next

Now we can see that we have a character column that contains our date string. However, it's not truly a date column. Dates are a type of numeric variable with a defined, ordered scale. To turn this column into a proper date, we will use a function from the `tidyverse`'s `lubridate` package, which has lots of useful functions for

working with dates. The function `ymd()` will parse a date string that has the order year-month-day. The `lubridate` package is already loaded as part of the `tidyverse`. We'll use the function `ymd()` to convert the `date` column from a character to a date type.

```
surveys %>%
  mutate(date = paste(year, month, day, sep = "-"),
         date = ymd(date)) %>%
  relocate(date, .after = year)
```

record_id <dbl>	mo... <dbl>	d.. <dbl>	y... <dbl>	date <date>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	weight <dbl>
1	7	16	1977	1977-07-16	2	NL	M	32	*NA*
2	7	16	1977	1977-07-16	3	NL	M	33	*NA*
3	7	16	1977	1977-07-16	2	DM	F	37	*NA*
4	7	16	1977	1977-07-16	7	DM	M	36	*NA*
5	7	16	1977	1977-07-16	3	DM	M	35	*NA*
6	7	16	1977	1977-07-16	1	PF	M	14	*NA*
7	7	16	1977	1977-07-16	2	PE	F	*NA*	*NA*
8	7	16	1977	1977-07-16	1	DM	M	37	*NA*
9	7	16	1977	1977-07-16	1	DM	F	34	*NA*
10	7	16	1977	1977-07-16	6	PF	F	20	*NA*

1-10 of 10,000 rows | 1-10 of 14 columns Previous **1** 2 3 4 5 6 … 1000 Next

```
surveys %>%
  mutate(date = paste(year, month, day, sep = "-"),
         date = as.Date(date)) %>%
  relocate(date, .after = year)
```

record_id <dbl>	mo... <dbl>	d.. <dbl>	y... <dbl>	date <date>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	weight <dbl>
1	7	16	1977	1977-07-16	2	NL	M	32	*NA*
2	7	16	1977	1977-07-16	3	NL	M	33	*NA*
3	7	16	1977	1977-07-16	2	DM	F	37	*NA*
4	7	16	1977	1977-07-16	7	DM	M	36	*NA*
5	7	16	1977	1977-07-16	3	DM	M	35	*NA*
6	7	16	1977	1977-07-16	1	PF	M	14	*NA*
7	7	16	1977	1977-07-16	2	PE	F	*NA*	*NA*
8	7	16	1977	1977-07-16	1	DM	M	37	*NA*
9	7	16	1977	1977-07-16	1	DM	F	34	*NA*
10	7	16	1977	1977-07-16	6	PF	F	20	*NA*

1-10 of 10,000 rows | 1-10 of 14 columns Previous **1** 2 3 4 5 6 … 1000 Next

Now we can see that our `date` column has the type `date` as well. In this example, we created our column with two separate lines in `mutate()`, but we can combine them into one:

```
# using nested functions
surveys %>%
  mutate(date = ymd(paste(year, month, day, sep = "-"))) %>%
  relocate(date, .after = year)
```

record_id <dbl>	mo... <dbl>	d.. <dbl>	y... <dbl>	date <date>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	weight <dbl>
1	7	16	1977	1977-07-16	2	NL	M	32	*NA*
2	7	16	1977	1977-07-16	3	NL	M	33	*NA*
3	7	16	1977	1977-07-16	2	DM	F	37	*NA*
4	7	16	1977	1977-07-16	7	DM	M	36	*NA*
5	7	16	1977	1977-07-16	3	DM	M	35	*NA*
6	7	16	1977	1977-07-16	1	PF	M	14	*NA*
7	7	16	1977	1977-07-16	2	PE	F	*NA*	*NA*
8	7	16	1977	1977-07-16	1	DM	M	37	*NA*
9	7	16	1977	1977-07-16	1	DM	F	34	*NA*
10	7	16	1977	1977-07-16	6	PF	F	20	*NA*

1-10 of 10,000 rows | 1-10 of 14 columns Previous **1** 2 3 4 5 6 … 1000 Next

```
# using a pipe *inside* mutate()
surveys %>%
  mutate(date = paste(year, month, day,
                      sep = "-") %>% ymd()) %>%
  relocate(date, .after = year)
```

record_id <dbl>	mo... <dbl>	d.. <dbl>	y... <dbl>	date <date>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	weight <dbl>
1	7	16	1977	1977-07-16	2	NL	M	32	*NA*
2	7	16	1977	1977-07-16	3	NL	M	33	*NA*
3	7	16	1977	1977-07-16	2	DM	F	37	*NA*
4	7	16	1977	1977-07-16	7	DM	M	36	*NA*
5	7	16	1977	1977-07-16	3	DM	M	35	*NA*
6	7	16	1977	1977-07-16	1	PF	M	14	*NA*
7	7	16	1977	1977-07-16	2	PE	F	*NA*	*NA*
8	7	16	1977	1977-07-16	1	DM	M	37	*NA*

record_id <dbl>	mo… <dbl>	d.. <dbl>	y… <dbl>	date <date>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	weight <dbl>
9	7	16	1977	1977-07-16	1	DM	F	34	*NA*
10	7	16	1977	1977-07-16	6	PF	F	20	*NA*

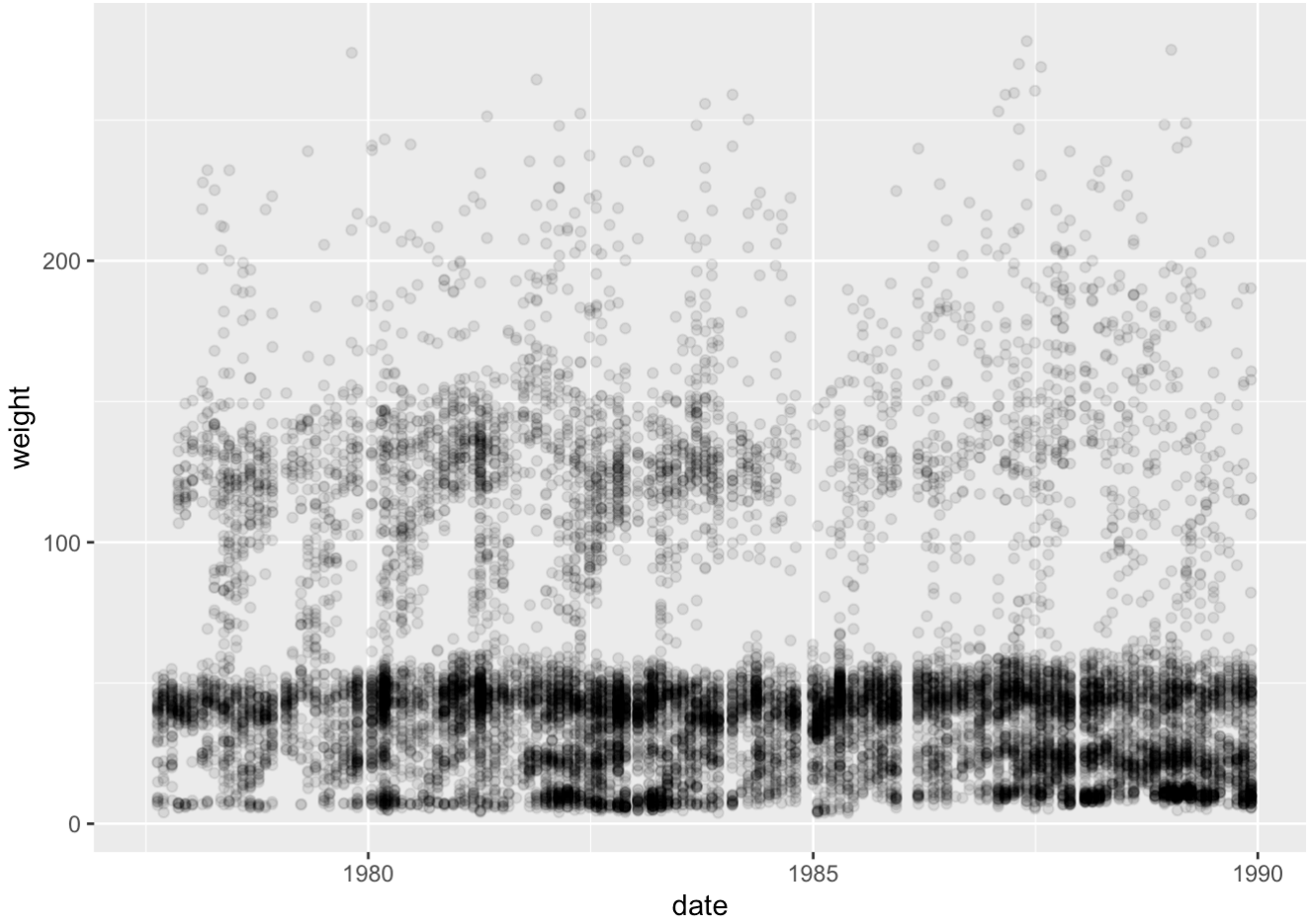
1-10 of 10,000 rows | 1-10 of 14 columns Previous **1** 2 3 4 5 6 … 1000 Next

Challenge 3: Plotting date

Because the `ggplot()` function takes the data as its first argument, you can actually pipe data straight into `ggplot()`. Try building a pipeline that creates the date column and plots weight across date.

```
surveys %>%
  mutate(date = ymd(paste(year, month, day, sep = "-"))) %>%
  ggplot(aes(x = date, y = weight)) +
  geom_jitter(alpha = 0.1)
```

```
## Warning: Removed 1692 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



This isn't necessarily the most useful plot, but we will learn some techniques that will help produce nice time series plots

2.6 The split-apply-combine approach using `group_by()` and

summarise()

Many data analysis tasks can be achieved using the split-apply-combine approach: you **split** the data into groups, **apply** some analysis to each group, and **combine** the results in some way. `dplyr` has a few convenient functions to enable this approach, the main two being `group_by()` and `summarise()`.

`group_by()` takes a data.frame and the name of one or more columns with categorical values that define the groups. `summarise()` then collapses each group into a one-row summary of the group, giving you back a data.frame with one row per group. The syntax for `summarise()` is similar to `mutate()`, where you define new columns based on values of other columns. Let's try calculating the mean weight of all our animals by sex.

```
# calculate mean weight by sex

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

sex <chr>	mean_weight <dbl>
F	53.11447
M	53.16446
NA	74.03636

3 rows

You can see that the mean weight for males is slightly higher than for females, but that animals whose sex is unknown have much higher weights. This is probably due to small sample size, but we should check to be sure. Like `mutate()`, we can define multiple columns in one `summarise()` call. The function `n()` will count the number of rows in each group.

```
# multiple columns in summarise

surveys %>%
  group_by(sex) %>%
  summarise(mean_weight = mean(weight, na.rm = T),
            n = n())
```

sex <chr>	mean_weight <dbl>	n <int>
F	53.11447	7318
M	53.16446	8260
NA	74.03636	1300

3 rows

You will often want to create groups based on multiple columns. For example, we might be interested in the mean weight of every species + sex combination. All we have to do is add another column to our `group_by()` call.

```
# group by multiple columns

surveys %>%
  group_by(species_id, sex) %>%
  summarise(mean_weight = mean(weight, na.rm = T),
            n = n())
```

```
## `summarise()` has grouped output by 'species_id'. You can override using the
## `.groups` argument.
```

species_id <chr>	sex <chr>	mean_weight <dbl>	n <int>
AB	*NA*	NaN	223
AH	*NA*	NaN	136
BA	M	7.000000	3
CB	*NA*	NaN	23
CM	*NA*	NaN	13
CQ	*NA*	NaN	16
CS	*NA*	NaN	1
CV	*NA*	NaN	1
DM	F	40.724797	2522
DM	M	44.003983	3108

1-10 of 67 rows Previous **1** 2 3 4 5 6 7 Next

Our resulting data.frame is much larger, since we have a greater number of groups. We also see a strange value showing up in our `mean_weight` column: `NaN`. This stands for "Not a Number", and it often results from trying to do an operation a vector with zero entries. How can a vector have zero entries? Well, if a particular group (like the AB species ID + `NA` sex group) has **only** `NA` values for weight, then the `na.rm = T` argument in `mean()` will remove **all** the values prior to calculating the mean. The result will be a value of `NaN`. Since we are not particularly interested in these values, let's add a step to our pipeline to remove rows where weight is `NA` **before** doing any other steps. This means that any groups with only `NA` values will disappear from our data.frame before we formally create the groups with `group_by()`.

```
# remove `NA` weights

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

```
## `summarise()` has grouped output by 'species_id'. You can override using the
## `.groups` argument.
```

species_id <chr>	sex <chr>	mean_weight <dbl>	n <int>
BA	M	7.000000	3
DM	F	40.724797	2460
DM	M	44.003983	3013
DM	*NA*	37.000000	8
DO	F	48.368189	679
DO	M	49.330214	748
DO	*NA*	44.000000	1
DS	F	118.051185	1055
DS	M	122.510135	1184
DS	*NA*	120.750000	16

1-10 of 46 rows Previous **1** 2 3 4 5 Next

That looks better! It's often useful to take a look at the results in some order, like the lowest mean weight to highest. We can use the `arrange()` function for that:

```
# order by mean_weight

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

```
## `summarise()` has grouped output by 'species_id'. You can override using the
## `.groups` argument.
```

species_id <chr>	sex <chr>	mean_weight <dbl>	n <int>
PF	*NA*	6.000000	2
BA	M	7.000000	3
PF	F	7.093023	215
PF	M	7.097973	296
RM	M	9.921829	678
RM	*NA*	10.428571	7
RM	F	10.739269	629
RF	M	12.437500	16
RF	F	13.695652	46

species_id	sex	mean_weight	n
<chr>	<chr>	<dbl>	<int>
PP	*NA*	15.000000	2

1-10 of 46 rows Previous **1** 2 3 4 5 Next

If we want to reverse the order, we can wrap the column name in `desc()`:

```
# order by mean_weight in descending order

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

```
## `summarise()` has grouped output by 'species_id'. You can override using the
## `.groups` argument.
```

species_id	sex	mean_weight	n
<chr>	<chr>	<dbl>	<int>
NL	M	167.935211	355
NL	*NA*	164.333333	9
NL	F	150.997826	460
SS	M	130.000000	1
DS	M	122.510135	1184
DS	*NA*	120.750000	16
DS	F	118.051185	1055
SH	F	79.163934	61
SH	M	67.588235	34
SF	F	58.333333	3

1-10 of 46 rows Previous **1** 2 3 4 5 Next

You may have seen several messages saying `summarise() has grouped output by 'species_id'. You can override using the .groups argument.` These are warning you that your resulting data.frame has retained some group structure, which means any subsequent operations on that data.frame will happen at the group level. If you look at the resulting data.frame output, you will see these lines:

```
# A tibble: 46 × 4
# Groups:   species_id [18]
```

They tell us we have a data.frame with 46 rows, 4 columns, and a group variable `species_id`, for which there are 18 groups that contain at least one non `NA` value in the column `weight`. We will see something similar if we use `group_by()` alone, but with more rows as we are not summarising the data:

```
# group byspecies_id and sex

surveys %>%
  group_by(species_id, sex)
```

record_id <dbl>	m... <dbl>	d.. <dbl>	y... <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei... <dbl>	genus <chr>
1	7	16	1977	2	NL	M	32	*NA*	Neotoma
2	7	16	1977	3	NL	M	33	*NA*	Neotoma
3	7	16	1977	2	DM	F	37	*NA*	Dipodomys
4	7	16	1977	7	DM	M	36	*NA*	Dipodomys
5	7	16	1977	3	DM	M	35	*NA*	Dipodomys
6	7	16	1977	1	PF	M	14	*NA*	Perognathus
7	7	16	1977	2	PE	F	*NA*	*NA*	Peromyscus
8	7	16	1977	1	DM	M	37	*NA*	Dipodomys
9	7	16	1977	1	DM	F	34	*NA*	Dipodomys
10	7	16	1977	6	PF	F	20	*NA*	Perognathus

1-10 of 10,000 rows | 1-10 of 13 columns Previous **1** 2 3 4 5 6 … 1000 Next

What we get back is the entire `surveys` data.frame, but with the grouping variables added to the tibble: 67 groups of `species_id + sex` combinations. Groups are often maintained throughout a pipeline, and if you assign the resulting data.frame to a new object, it will also have those groups. This can lead to confusing results if you forget about the grouping and want to carry out operations on the whole data.frame, not by group. Therefore, it is a good habit to remove the groups at the end of a pipeline containing `group_by()` by using the `ungroup()` function:

```
# ungroup at the end of the pipeline

surveys %>%
  filter(!is.na(weight)) %>%
  group_by(species_id, sex) %>%
  summarise(mean_weight = mean(weight),
            n = n()) %>%
  arrange(desc(mean_weight)) %>%
  ungroup()
```

```
## `summarise()` has grouped output by 'species_id'. You can override using the
## `.groups` argument.
```

species_id <chr>	sex <chr>	mean_weight <dbl>	n <int>
NL	M	167.935211	355
NL	*NA*	164.333333	9
NL	F	150.997826	460

species_id <chr>	sex <chr>	mean_weight <dbl>	n <int>
SS	M	130.000000	1
DS	M	122.510135	1184
DS	*NA*	120.750000	16
DS	F	118.051185	1055
SH	F	79.163934	61
SH	M	67.588235	34
SF	F	58.333333	3

1-10 of 46 rows Previous **1** 2 3 4 5 Next

Now our data.frame just says `# A tibble: 46 × 4` at the top, with no groups.

While it is common that you will want to get the one-row-per-group summary that `summarise()` provides, there are times where you want to calculate a per-group value but keep all the rows in your data.frame. For example, we might want to know the mean weight for each species ID + sex combination, and then we might want to know how far from that mean value each observation in the group is. For this, we can use `group_by()` and `mutate()` together:

```
# group_by species_id and sex and create some new columns

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

record_id <dbl>	m… <dbl>	d.. <dbl>	y… <dbl>	plot_id <dbl>	species_id <chr>	s.. <chr>	hindfoot_length <dbl>	wei… <dbl>	genus <chr>
63	8	19	1977	3	DM	M	35	40	Dipodomys
64	8	19	1977	7	DM	M	37	48	Dipodomys
65	8	19	1977	4	DM	F	34	29	Dipodomys
66	8	19	1977	4	DM	F	35	46	Dipodomys
67	8	19	1977	7	DM	M	35	36	Dipodomys
68	8	19	1977	8	DO	F	32	52	Dipodomys
69	8	19	1977	2	PF	M	15	8	Perognathus
70	8	19	1977	3	OX	F	21	22	Onychomys
71	8	19	1977	7	DM	F	36	35	Dipodomys
74	8	19	1977	8	PF	M	12	7	Perognathus

1-10 of 10,000 rows | 1-10 of 15 columns Previous **1** 2 3 4 5 6 … 1000 Next

Since we get all our columns back, the new columns are at the very end and don't print out in the console. Let's use `select()` to just look at the columns of interest. Inside `select()` we can use the `contains()` function to get any column containing the word "weight" in the name:

```
#  use the `contains()` function to select any column with the word 'weight' in the n
ame

surveys %>%
  filter(!is.na(weight)) %>%
  group_by(species_id, sex) %>%
  mutate(mean_weight = mean(weight),
           weight_diff = weight - mean_weight) %>%
  select(species_id, sex, contains("weight"))
```

species_id <chr>	sex <chr>	weight <dbl>	mean_weight <dbl>	weight_diff <dbl>
DM	M	40	44.003983	-4.003983e+00
DM	M	48	44.003983	3.996017e+00
DM	F	29	40.724797	-1.172480e+01
DM	F	46	40.724797	5.275203e+00
DM	M	36	44.003983	-8.003983e+00
DO	F	52	48.368189	3.631811e+00
PF	M	8	7.097973	9.020270e-01
OX	F	22	21.000000	1.000000e+00
DM	F	35	40.724797	-5.724797e+00
PF	M	7	7.097973	-9.797297e-02

1-10 of 10,000 rows Previous **1** 2 3 4 5 6 … 1000 Next

What happens with the `group_by() + mutate()` combination is similar to using `summarise()` : for each group, the mean weight is calculated. However, instead of reporting only one row per group, the mean weight for each group is added to each row in that group. For each row in a group (like DM species ID + M sex), you will see the same value in `mean_weight` .

Challenge 4.a: Making a time series

Use the **split-apply-combine** approach to make a `data.frame` that counts the total number of animals of each sex caught on each day in the `surveys` data.

```
surveys_daily_counts <- surveys %>%
  mutate(date = ymd(paste(year, month, day, sep = "-"))) %>%
  group_by(date, sex) %>%
  summarise(n = n())
```

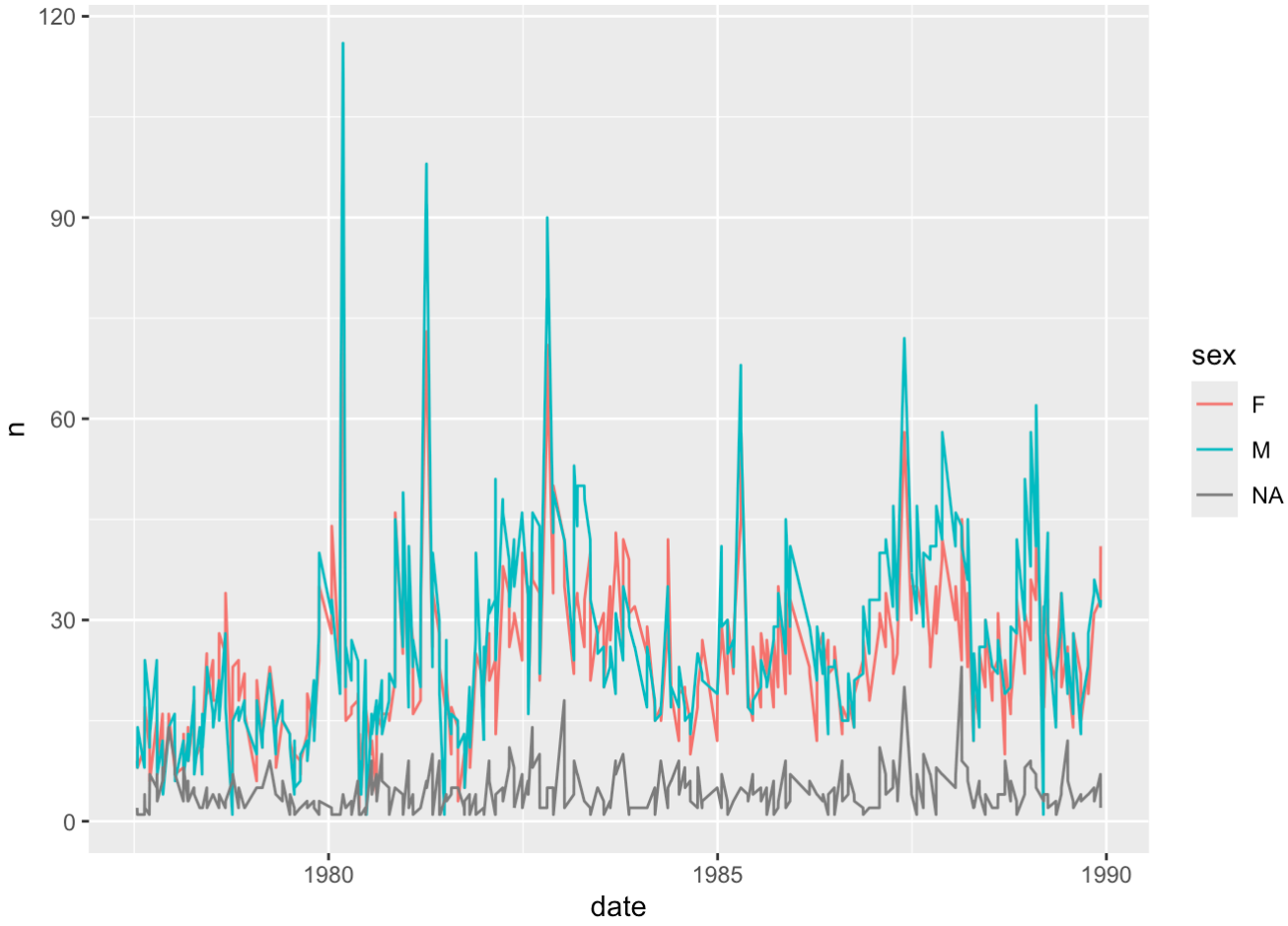
```
## `summarise()` has grouped output by 'date'. You can override using the
## `.groups` argument.
```

```
# shorter approach using count()
surveys_daily_counts <- surveys %>%
  mutate(date = ymd(paste(year, month, day, sep = "-"))) %>%
  count(date, sex)
```

Challenge 4.b: Making a time series

Now use the data.frame you just made to plot the daily number of animals of each sex caught over time. It's up to you what `geom` to use, but a `line` plot might be a good choice. You should also think about how to differentiate which data corresponds to which sex.

```
surveys_daily_counts %>%
  ggplot(aes(x = date, y = n, color = sex)) +
  geom_line()
```



2.7 Reshaping data with `tidyr`

Let's say we are interested in comparing the mean weights of each species across our different plots. We can begin this process using the `group_by() + summarise()` approach:

```
# compare the mean weights of each species across the different plots

sp_by_plot <- surveys %>%
  filter(!is.na(weight)) %>%
  group_by(species_id, plot_id) %>%
  summarise(mean_weight = mean(weight)) %>%
  arrange(species_id, plot_id)
```

```
## `summarise()` has grouped output by 'species_id'. You can override using the
## `.groups` argument.
```

```
sp_by_plot
```

species_id <chr>	plot_id <dbl>	mean_weight <dbl>
BA	3	8.000000
BA	21	6.500000
DM	1	42.658470
DM	2	42.625000
DM	3	41.234043
DM	4	41.909953
DM	5	42.567775
DM	6	42.149038
DM	7	43.181818
DM	8	43.405983

1-10 of 300 rows Previous **1** 2 3 4 5 6 … 30 Next

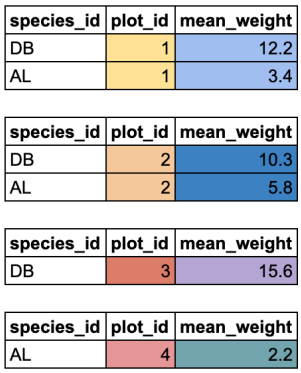
That looks great, but it is a bit difficult to compare values across plots. It would be nice if we could reshape this data.frame to make those comparisons easier. Well, the `tidyr` package from the `tidyverse` has a pair of functions that allow you to reshape data by pivoting it: `pivot_wider()` and `pivot_longer()`. `pivot_wider()` will make the data wider, which means increasing the number of columns and reducing the number of rows. `pivot_longer()` will do the opposite, reducing the number of columns and increasing the number of rows.

In this case, it might be nice to create a data.frame where each species has its own row, and each plot has its own column containing the mean weight for a given species. We will use `pivot_wider()` to reshape our data in this way. It takes 3 arguments:

1. the name of the data.frame
2. `names_from` : which column should be used to generate the names of the new columns?
3. `values_from` : which column should be used to fill in the values of the new columns?

Any columns not used for `names_from` or `values_from` will not be pivoted.

species_id	plot_id	mean_weight
DB	1	12.2
DB	2	10.3
DB	3	15.6
AL	1	3.4
AL	2	5.8
AL	4	2.2



species_id	plot_id	mean_weight
DB	1	12.2
AL	1	3.4

species_id	plot_id	mean_weight
DB	2	10.3
AL	2	5.8

species_id	plot_id	mean_weight
DB	3	15.6

species_id	plot_id	mean_weight
AL	4	2.2

species_id	1	2	3	4
DB	12.2	10.3	15.6	NA
AL	3.4	5.8	NA	2.2

In our case, we want the new columns to be named from our `plot_id` column, with the values coming from the `mean_weight` column. We can pipe our data.frame right into `pivot_wider()` and add those two arguments:

```
# pivot data to wide format

sp_by_plot_wide <- sp_by_plot %>%
  pivot_wider(names_from = plot_id,
              values_from = mean_weight)

sp_by_plot_wide
```

species_id	3	21	1	2	4	5	
<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl:
BA	8.00000	6.500000	*NA*	*NA*	*NA*	*NA*	*NA*
DM	41.23404	41.476190	42.658470	42.625000	41.90995	42.56777	42.14903
DO	42.66667	*NA*	50.131944	50.313433	46.85000	50.41975	48.96402
DS	127.64286	*NA*	128.890909	124.707965	118.08367	111.37705	114.14563
NL	171.25806	135.875000	153.648649	170.822222	164.16667	191.69231	175.72727
OL	32.10000	28.642857	35.514286	34.000000	32.96296	32.55000	31.84848
OT	24.05455	24.075000	23.685714	24.862069	26.45455	23.63636	23.51515
OX	22.00000	*NA*	*NA*	22.000000	*NA*	20.00000	*NA*
PE	22.72500	19.600000	21.625000	22.045977	*NA*	21.00000	21.60000
PF	7.12500	7.225806	6.571429	6.888889	6.75000	7.50000	7.54166

1-10 of 18 rows | 1-9 of 25 columns Previous **1** 2 Next

Now we've got our reshaped data.frame. There are a few things to notice. First, we have a new column for each `plot_id` value. There is one old column left in the data.frame: `species_id`. It wasn't used in `pivot_wider()`, so it stays, and now contains a single entry for each unique `species_id` value.

Finally, a lot of `NA`s have appeared. Some species aren't found in every plot, but because a data.frame has to have a value in every row and every column, an `NA` is inserted. We can double-check this to verify what is going on.

Looking in our new pivoted data.frame, we can see that there is an `NA` value for the species `BA` in plot `1`. Let's take our `sp_by_plot` data.frame and look for the `mean_weight` of that species + plot combination.

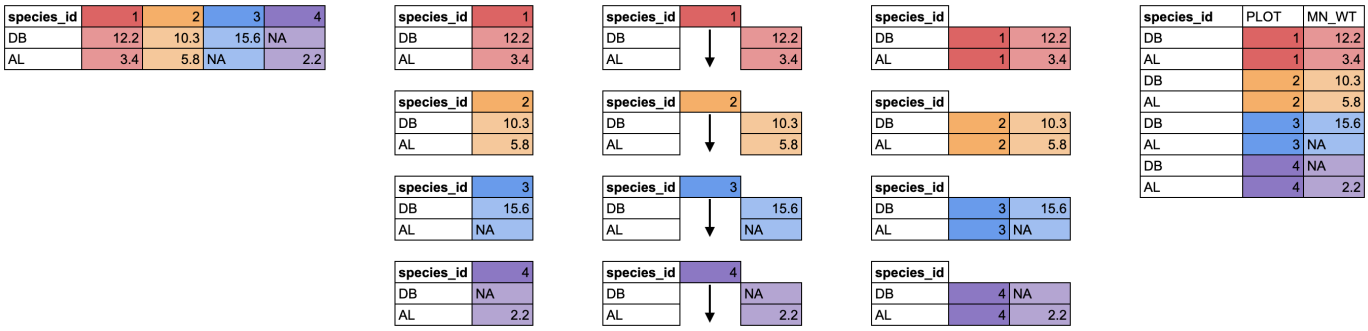
```
# confirm pivot_wider results

sp_by_plot %>%
  filter(species_id == "BA" & plot_id == 1)
```

0 rows

We get back 0 rows. There is no `mean_weight` for the species `BA` in plot `1`. This either happened because no `BA` were ever caught in plot `1`, or because every `BA` caught in plot `1` had an `NA` weight value and all the rows got removed when we used `filter(!is.na(weight))` in the process of making `sp_by_plot`.

Because there are no rows with that species + plot combination, in our pivoted data.frame, the value gets filled with `NA`.

There is another `pivot_` function that does the opposite, moving data from a wide to long format, called `pivot_longer()`. It takes 3 arguments: `cols` for the columns you want to pivot, `names_to` for the name of the new column which will contain the old column names, and `values_to` for the name of the new column which will contain the old values.



Let us now pivot our new wide data.frame to a long format using `pivot_longer()`. We want to pivot all the columns except `species_id`, and we will use `PLOT` for the new column of plot IDs, and `MEAN_WT` for the new column of mean weight values.

```
# pivot data to long format

sp_by_plot_wide %>%
  pivot_longer(cols = -species_id, names_to = "PLOT", values_to = "MEAN_WT")
```

species_id <chr>	PLOT <chr>	MEAN_WT <dbl>
BA	3	8.000000
BA	21	6.500000
BA	1	*NA*
BA	2	*NA*
BA	4	*NA*
BA	5	*NA*
BA	6	*NA*
BA	7	*NA*
BA	8	*NA*
BA	9	*NA*

1-10 of 432 rows Previous **1** 2 3 4 5 6 ... 44 Next

One thing you will notice is that all those `NA` values that got generated when we pivoted wider. However, we can filter those out, which gets us back to the same data as `sp_by_plot`, before we pivoted it wider.

```
# filter out `NA` values

sp_by_plot_wide %>%
  pivot_longer(cols = -species_id, names_to = "PLOT", values_to = "MEAN_WT") %>%
  filter(!is.na(MEAN_WT))
```

species_id <chr>	PLOT <chr>	MEAN_WT <dbl>
BA	3	8.000000
BA	21	6.500000
DM	3	41.234043
DM	21	41.476190
DM	1	42.658470
DM	2	42.625000
DM	4	41.909953
DM	5	42.567775
DM	6	42.149038
DM	7	43.181818

1-10 of 300 rows Previous **1** 2 3 4 5 6 … 30 Next

Spreadsheets are often record data in a wider format, but lots of `tidyverse` tools, especially `ggplot2`, like data in a longer format, so `pivot_longer()` is often very useful.

2.8 Exporting data

Let's say we want to send the wide version of our `sb_by_plot` data.frame to a colleague who doesn't use R. In this case, we might want to save it as a CSV file.

First, we might want to modify the names of the columns, since right now they are bare numbers, which aren't very informative. Luckily, `pivot_wider()` has an argument `names_prefix` which will allow us to add "plot_" to the start of each column.

```
# modify the column names

sp_by_plot %>%
  pivot_wider(names_from = plot_id, values_from = mean_weight,
              names_prefix = "plot_")
```

species_id <chr>	plot_3 <dbl>	plot_21 <dbl>	plot_1 <dbl>	plot_2 <dbl>	plot_4 <dbl>	plot_5 <dbl>	plot_ <dbl
BA	8.00000	6.500000	*NA*	*NA*	*NA*	*NA*	*NA*
DM	41.23404	41.476190	42.658470	42.625000	41.90995	42.56777	42.14903
DO	42.66667	*NA*	50.131944	50.313433	46.85000	50.41975	48.96402
DS	127.64286	*NA*	128.890909	124.707965	118.08367	111.37705	114.14563

species_id <chr>	plot_3 <dbl>	plot_21 <dbl>	plot_1 <dbl>	plot_2 <dbl>	plot_4 <dbl>	plot_5 <dbl>	plot_ <dbl
NL	171.25806	135.875000	153.648649	170.822222	164.16667	191.69231	175.72727
OL	32.10000	28.642857	35.514286	34.000000	32.96296	32.55000	31.84848
OT	24.05455	24.075000	23.685714	24.862069	26.45455	23.63636	23.51515
OX	22.00000	*NA*	*NA*	22.000000	*NA*	20.00000	*NA*
PE	22.72500	19.600000	21.625000	22.045977	*NA*	21.00000	21.60000
PF	7.12500	7.225806	6.571429	6.888889	6.75000	7.50000	7.54166

1-10 of 18 rows | 1-9 of 25 columns Previous **1** 2 Next

That looks better! Let's save this data.frame as a new object.

```
# save as a new object surveys_sp

surveys_sp <- sp_by_plot %>%
  pivot_wider(names_from = plot_id, values_from = mean_weight,
              names_prefix = "plot_")

surveys_sp
```

species_id <chr>	plot_3 <dbl>	plot_21 <dbl>	plot_1 <dbl>	plot_2 <dbl>	plot_4 <dbl>	plot_5 <dbl>	plot_ <dbl
BA	8.00000	6.500000	*NA*	*NA*	*NA*	*NA*	*NA*
DM	41.23404	41.476190	42.658470	42.625000	41.90995	42.56777	42.14903
DO	42.66667	*NA*	50.131944	50.313433	46.85000	50.41975	48.96402
DS	127.64286	*NA*	128.890909	124.707965	118.08367	111.37705	114.14563
NL	171.25806	135.875000	153.648649	170.822222	164.16667	191.69231	175.72727
OL	32.10000	28.642857	35.514286	34.000000	32.96296	32.55000	31.84848
OT	24.05455	24.075000	23.685714	24.862069	26.45455	23.63636	23.51515
OX	22.00000	*NA*	*NA*	22.000000	*NA*	20.00000	*NA*
PE	22.72500	19.600000	21.625000	22.045977	*NA*	21.00000	21.60000
PF	7.12500	7.225806	6.571429	6.888889	6.75000	7.50000	7.54166

1-10 of 18 rows | 1-9 of 25 columns Previous **1** 2 Next

Now we can save this data.frame to a CSV using the `write_csv()` function from the `readr` package. The first argument is the name of the data.frame, and the second is the path to the new file we want to create, including the file extension `.csv`.

```
# write data.frame to csv

write_csv(surveys_sp, "data/cleaned/surveys_meanweight_species_plot.csv")
```

If we go look into our `data/cleaned` folder, we will see this new CSV file.

Key points

- use `filter()` to subset rows and `select()` to subset columns
- build up pipelines one step at a time before assigning the result
- it is often best to keep components of dates separate until needed, then use `mutate()` to make a date column
- `group_by()` can be used with `summarise()` to collapse rows, or `mutate()` to keep the same number of rows
- `pivot_wider()` and `pivot_longer()` are powerful for reshaping data, but you should plan out how to use them thoughtfully